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

Benedikt Bünz[1] and Jessica Chen[1]

New York University

Abstract. An important part in proving machine computation is to prove the correctness of the read and write operations performed from the memory, which we term *memory-proving*. Previous methodologies required proving Merkle Tree openings or multi-set hashes, resulting in relatively large proof circuits. We construct an efficient memory-proving Incrementally Verifiable Computation (IVC) scheme from accumulation, which is particularly useful for machine computations with large memories and deterministic steps. In our scheme, the IVC prover $\mathsf{P}_{\mathsf{IVC}}$ has cost entirely independent of the memory size T and only needs to commit to approximately 15 field elements per read/write operation, marking a more than 100X improvement over prior work. We further reduce this cost by employing a modified, accumulation-friendly version of the GKR protocol. In the optimized version, $\mathsf{P}_{\mathsf{IVC}}$ only needs to commit to 6 small memory-table elements per read/write. If the table stores 32-bit values, then this is equivalent to committing to less than one single field element per read and write. Our modified GKR protocol is also valuable for proving other deterministic computations within the context of IVC. Our memory-proving protocol can be extended to support key-value stores.

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

1 Introduction

Consider the scenario where one or multiple clients outsource a large computation, possibly of infinite steps, to an untrusted server. For example, clients might want to continuously verify that all transactions in a blockchain are valid. Naturally, the clients would like the server to provide a certificate, which would allow the clients to verify that all the computation steps run up to that point were correct and even to continue the computation from that point onwards. The efficiency of verification necessitates that the size of the proof and the complexity of its verification be independent of the length of the computation. Moreover, since the computation can be long or even unbounded, it would be ideal if the server can provide the current state and a certificate upon request from the client *at any point.* This is achieved by maintaining a running certificate or proof that can be efficiently updated with each computation step. A system that achieves these properties is called an *incrementally verifiable computation* (IVC) system[Val08] [1].

IVC enables the server/prover to produce an output z_{IVC}, along with a proof π_{IVC} upon request from the client/verifier without requiring *a priori* knowledge of an upper bound on the number of computation steps. With a valid π_{IVC}, a client/verifier can be convinced z_{IVC} is the output of the correct execution of a (potentially non-deterministic) machine computation up to this point, and can even continue the computation. Recent developments have demonstrated that IVC can be constructed from simple public-coin interactive protocols featuring algebraic verifiers, such as protocols where the prover simply sends the witness. This is achieved through the use of accumulation[2] or folding schemes [BGH19; BCMS20; BCLMS21; BDFG21; KST22; BC23; EG23]. The resulting IVC has essentially the same computational overhead as the accumulation scheme. The cost of the resulting IVC prover depends on two main factors:

1. The size of the recursive circuit, predominantly comprising the accumulation verifier $\mathsf{V}_{\mathsf{acc}}$. Since the size of $\mathsf{V}_{\mathsf{acc}}$ only depends on the algebraic degree of the verifier and the number of rounds in the underlying protocol (rather than the communication or verification complexity), minimizing these two factors are crucial for reducing the cost of the IVC prover.
2. The cost of the accumulation prover $\mathsf{P}_{\mathsf{acc}}$, which is mainly influenced by the commitment cost to the prover messages, and thus is dependent on the number of elements in the prover messages of the underlying interactive protocol.

The general paradigm of using IVC to prove machine computations involves first proving the correctness of computation under the assumption that memory accesses were executed correctly, recording all the read/write operations in the circuit, and then proving the correctness of the recorded read/write operations. The primary challenge lies in the latter, i.e. efficiently proving the correctness of memory accesses, which we will refer to as *memory-proving* in this work. With the above-mentioned recent advancements in IVC construction, **we need only to design a public-coin interactive protocol for memory-proving with an algebraic verifier while ensuring that following three parameters remain small: the number of rounds, the verifier degree, and the number of elements in the prover messages** (ideally independent of T). These parameters are the only factors on which the cost of $\mathsf{P}_{\mathsf{IVC}}$ depends. Then, by applying existing accumulation compilers (e.g., the ProtoStar compiler [BC23]) to this interactive protocol, we can obtain an efficient accumulation scheme for memory-proving, and finally derive an efficient memory-proving IVC scheme from accumulation by utilizing existing IVC compilers (e.g., [BCLMS21]).

[1] The literary application of IVC is the machine Deep Thought from the Hitchhiker's 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.

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

The most rudimentary method of performing memory-proving involves unrolling the entire memory into a circuit. However, since a circuit is at least as large as its inputs, this circuit would be of size $O(\ell T)$, which is prohibitively large even medium-sized memories. An alternative approach is for the prover to simulate memory-checking internally and prove that the memory accesses would have been accepted by the memory-checking verifier, who only keeps a small local state. In all previous works, using this approach, the prover's cost is dependent on the memory size T and/or hashing is required within the circuit. For instance, Spice [SAGL18] employed offline memory-checking, requiring approximately 1500 constraints[3] per read and write operation. Since the prover needs to transmit at least one proof element per constraint, this results in 1500 elements in the prover message per read/write operation, and thus 1500 commitments per accumulation step for $\mathsf{P}_{\mathsf{IVC}}$. In contrast, ProtoStar recently showed a memory-proving protocol for static read-only memory that utilises the LogUp argument [Hab22], in which the prover only performs two group scalar multiplications per read instruction [BC23]. The prover's cost in ProtoStar is independent of the memory size T and does not involve multi-set hashing. However, their approach does not support writes into a dynamic memory[BC23].

In this paper, we present an interactive protocol for memory-proving inspired by the LogUp argument [Hab22]. Using accumulation techniques, we obtain an IVC scheme for memory-proving with minimal prover overhead. We then show an optimization of our scheme which employs an accumulation-friendly version of the GKR protocol to further reduce the prover overhead. We note that this adapted GKR protocol has other applications beyond improving our memory-proving protocol.

$O(\ell)$ memory-proving $\mathsf{P}_{\mathsf{IVC}}$ ProtoStar has previously demonstrated that LogUp is well suited for accumulation and, thus, IVC. It can be used to verify the existence of a set of witness values in a static table of values [BC23]. We design LogUp-styled arguments to support reading from and writing to a fully dynamic table.

One key challenge we address is proving only $O(\ell)$ table values were altered in a table of size $T >> \ell$, while ensuring that the cost of $\mathsf{P}_{\mathsf{IVC}}$ remains independent of T. Our memory-proving protocol is public-coin with an algebraic verifier, featuring 2 rounds of communication[4], verifier degree 3, and only $O(\ell)$ elements in prover messages, where ℓ denotes the number of reads and writes performed in each computation step. This means it can be turned into an efficient accumulation scheme using existing accumulation compiler (eg. [BC23]). The resulting $\mathsf{P}_{\mathsf{IVC}}$ only needs to commit to $O(\ell)$ elements, which is independent of the memory size T.

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

[4] Each round consists of a prover message and is possibly followed by a verifier challenge

This significantly improves on prior work, which either relied on Merkle trees requiring $\log T$ hashes per memory access or required multi-set hashes [SAGL18]. These prior methods are particularly costly in the context of memory-proving, where the hashes result in large proving circuits. In contrast, our resulting memory-proving scheme is practically efficient with the prover only having to commit to 15 field values per memory access. In addition, since the prover cost is completely independent of the memory size T, our protocol can be extended to the setting of key-value store. We describe this extension in detail at the end of Section 5.1.

Optimizing memory-proving with GKR. One limitation of the scheme is that the prover needs to commit to 6 large field elements per memory access, i.e. each of size λ bits, even if the memory entries themselves are small. This is because in the memory-proving interactive protocol, the prover needs to send 6 vectors consisting of inverses of the form $\frac{1}{r+t_i}$ where r is a constant, and each t_i is a small table entry. To resolve this overhead, we draw inspiration from [STW23; PH23] and compute this sum using formal fractions and a modified GKR protocol. Our modified protocol retains GKR's ability to prove deterministic layered computations without committing to the intermediate values. Specifically, the protocol relies on bivariate sumcheck instead of multilinear sumcheck, reducing the number of rounds per layer to 3. In the context of the ProtoStar accumulation compiler, this reduction in number of rounds significantly lowers the recursive overhead in IVC.

With the power of GKR, the memory-proving protocol no longer requires computing and committing to the large inverses. Thus, if we read/write ℓ s-bit values from memory, the number of group operations decreases from $O(\lambda\ell)$ to $O(s\ell)$, i.e., the actual size of the data that is read/written. We provide a brief overview of the resulting efficiency of our protocol in Table 1. Most importantly the prover only needs to commit to 6 elements that are as large as the table entries for each read/ write. If the table contains 32-bit entries then this is equivalent to committing to 192bits per read/write or less than a single 256bit field element. We also introduce several optimizations for our GKR-powered memory-proving protocol, which further reduce the number of GKR rounds.

Table 1. Efficiency Table for our Memory-Proving Protocol. T is the memory size, and ℓ is the number of read/write operations. $\mathbb{T}$ is the set of table entries, which might only contain small field elements. 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	$O(\log T)\mathbb{G}$

IVC for deterministic computations. GKR has numerous other applications in accumulation beyond enhancing our memory-proving protocol. In fact, for proving *any* low-depth deterministic computations, GKR only requires committing to the inputs and outputs, not the intermediate values. We demonstrate the utility of this by describing an accumulation-friendly GKR protocol for computing group scalar multiplications, which is the dominant cost within the recursive circuit.

1.1 Related Work

IVC and Accumulation. Valiant [Val08] introduced incrementally verifiable computation (IVC) and showed that IVC can be built from Succinct Non-interactive ARguments of Knowledge (SNARKs). The core concept involves the prover generating a SNARK at each computation step, certifying both the current step and the verification of the SNARK from the previous step. The latter part is commonly referred to as the *recursive circuit.* Subsequent to Valiant's work, an important line of research [BCCT13; BCTV14; COS20] has enhanced the practicality of IVC, studied its generalization to arbitrary graphs (Proof-Carrying Data, PCD), and advanced its theoretical foundations.

Halo [BGH19] showed that IVC can be constructed from simpler assumptions, sparking research on accumulation [BCMS20; BCLMS21; BDFG21; KST22; BC23; EG23]. The idea is to construct IVC by simply accumulating or batching the verification of non-interactive arguments, postponing verification to the end of each IVC step. In essence, in each accumulation round, the prover produces a new argument for the current step and proves its correct accumulation into the existing accumulator. The accumulation step can be as straightforward as taking a random linear combination between two vector commitments, and verifying the accumulation step can be significantly cheaper than verifying the proof. The more computationally intensive final verification, which is called the *decision* step in accumulation, is executed only at the end of IVC step to verify the correctness of the accumulated commitment. A valid accumulator implies that all the accumulated proofs were valid.

Recently, ProtoStar introduced a new recipe for constructing accumulation schemes and IVC [BC23] from *any* interactive public-coin protocol Π with an algebraic verifier. The resulting accumulation verifier $\mathsf{V}_{\mathsf{acc}}$ depends only on the number of rounds and the verifier degree in the underlying interactive protocol Π, and the resulting accumulation prover $\mathsf{P}_{\mathsf{acc}}$'s main cost is committing to all the prover messages in Π. Using the [BCLMS21] compiler, an accumulation scheme for NP directly yields an IVC, where $\mathsf{P}_{\mathsf{IVC}}$'s cost for computing the predicate is proportional to the cost of $\mathsf{P}_{\mathsf{acc}}$ and the recursive circuit consisting of $\mathsf{V}_{\mathsf{acc}}$.

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

Memory-checking and lookup arguments. Memory-checking [BEGKN91] enables an untrusted server to convince a client that a set of read/write operations is consistent with a memory without having to send the entire memory to the client. Each entry of read/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 v with the timestamp t, unless there was another write to a in the meantime. We briefly highlight two constructions and their limitations here, and refer to Appendix B of Jolt [AST23] for an excellent overview of memory-checking techniques.

One approach stores the memory in a Merkle Tree [BFRSBW13; BCTV14]. 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 is correctly updated. The verification for either step requires $O(\log T)$ hashes, and the prover's computational work is also $O(\log T)$, where T is the size of the memory. When this technique is used within IVC, the memory-checking verifier is part of the proving circuit, and $\log(T)$ hashes per read and write operation become a significant overhead.

The other common approach, dating back to [BEGKN91] and later refined in [CDvGS03; DNRV09; SAGL18], relies on proving that the constructed sets of reads and writes form a permutation. The state-of-the-art work Spice [SAGL18] employs multi-set hashes and proves that the hash was evaluated correctly, which results in over 1500 constraints per read/write operation, two orders of magnitude more than our approach. The approach also requires a linear scan of the memory at the end of the computation, but similar as in our construction this can be deferred to a final decider.

Recently, there has been increased attention to a related primitive called lookup arguments. Lookup arguments can be used to verify read operation in a static, possibly preprocessed memory. A recent line of work [ZBKMNS22; PK22; GK22; ZGKMR22; EFG22] showed 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 LogUp [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 compiler [BC23], which we describe in Theorem 1 in Section 2.5. It gives a recipe for constructing accumulation schemes and IVC [BC23] from *any* interactive public-coin protocol with an algebraic verifier. We summarize the recipe here into five steps:

1. Begin with *any* k-round interactive public-coin protocol featuring L verification checks of maximum degree d, and prover messages comprising n nonzero elements.

2. Compress the communication by using a homomorphic vector commitment (e.g. Pedersen commitment) to commit to each vector in the prover messages.
3. Make the protocol non-interactive through the Fiat-Shamir transformation.
4. Use the ProtoStar compiler to convert the non-interactive protocol into an accumulation scheme. The accumulation scheme 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 the cross error terms resulted from the accumulation.
5. If the underlying protocol can prove NP-complete relations, such as circuits, then the [BCLMS21] IVC compiler can be applied to construct an IVC scheme from the accumulation scheme for any function F. The compiler ensures the correct execution of the accumulation verifier alongside proving F.

Following this recipe, we design special-sound, algebraic protocols for memory-proving and GKR. One important design goal is to keep the complexity of the accumulation verifier $\mathsf{V}_{\mathsf{acc}}$ low, as $\mathsf{V}_{\mathsf{acc}}$ is the dominant component in the recursive circuit. Notably, the complexity of $\mathsf{V}_{\mathsf{acc}}$ relies solely on k and d, without any dependence on n or L whatsoever. Another design goal is to minimize the commitment cost of the accumulation prover $\mathsf{P}_{\mathsf{acc}}$, which is directly contingent on the number of nonzero elements in prover messages, as committing to 0 is free in Pedersen commitment. Therefore, to leverage the ProtoStar compiler to design an efficient IVC scheme where $\mathsf{P}_{\mathsf{IVC}}$ cost is independent of the memory size T, **we need to design an interactive, algebraic memory-proving protocol with small values for number of rounds k, verifier degree d and number of nonzero elements in prover messages n.** This implies that n should be independent of T, since otherwise the cost of $\mathsf{P}_{\mathsf{acc}}$ will be $O(T)$ even if the number of memory accesses is much smaller than T.

Constructing Read List and Write List We assume the list of "reads" and the list of "writes" were constructed similarly to the way in the classic offline memory-checking process [BEGKN91; CDvGS03; SAGL18]. Each entry in the lists is in the form of a tuple (address, value, timestamp), with the local timestamp incremented after each write operation. The specific construction is described in Section 3.

If all memory accesses were performed correctly, the constructed lists should satisfy three properties: 1) the read list and the write list should be permutations of each other; 2) the initial reads are consistent with the starting/old memory; and 3) the new memory is updated only at the addresses written to and with the correct amount. Note that the third memory-update (or mem-update for short) property requires examining all T addresses, not only the ones touched by memory accesses but also the ones untouched. Therefore, the main challenge in designing an efficient memory-proving protocol lies in proving correct mem-update in time independent of T.

LogUp based mem-update. The starting point of our construction is the *LogUp* lookup argument [Hab22] which uses the fact that the set of values in $\mathbf{w} = [\mathbf{w}_i]_{i=1}^{\ell}$ is contained in a table $\mathbf{t} = [\mathbf{t}_i]_{i=1}^{T}$ if and only if

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

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. ProtoStar [BC23] showed that the LogUp argument can be efficiently accumulated. Importantly, it observes that the prover messages in the protocol for LogUp argument, e.g. $\boldsymbol{m} = [\boldsymbol{m}_i]_{i=1}^{T}$, only contains ℓ nonzero entries. This means, in the context of the ProtoStar compiler and IVC, the accumulation prover $\mathsf{P}_{\mathsf{acc}}$ and thus the IVC prover $\mathsf{P}_{\mathsf{IVC}}$ only needs to do $O(\ell)$ work. However, the LogUp argument only supports read operations and not write operations.

We attempt to modify the LogUp argument to use it for mem-update. Assume, $\mathbf{w}$ corresponds to the ℓ-sized update vector (the difference between the final written value and the initial read value from each address), and $\mathbf{t}$ corresponds to the T-sized vector Δ that represents the difference between the new memory and the old memory, i.e. $\Delta := \mathsf{NM} - \mathsf{OM}$. However, the LogUp argument only cares about the membership of the $\mathbf{w}$ values but not their positions in Δ; in other words, the argument only indicates that some memory value is changed by $\mathbf{w}_j$, but does not constrain the change to any specific address. In addition, it is not immediately clear how to update the memory or compute the right hand side with Δ in a manner that does not require a linear scan.

To resolve the first issue, we add the address vector to random linear combination in the denominators. That is,

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

holds if and only if $\mathbf{w}_j = \Delta_{\boldsymbol{b}_j}$ for every $j \in [\ell]$, where $\boldsymbol{b} = [\boldsymbol{b}_j]_{j=1}^{\ell}$ is an address vector and Y is another random variable. Note that this is an *indexed* LogUp argument where we ensure not only the membership of the values but also their precise indices in the lookup table. In this indexed lookup argument, m_i only takes on the values 0 or 1. Still, this modified lookup argument is not sufficient, as it does not ensure that Δ is 0 at the positions for which there had been no read or write operation. This is an important criteria for correct mem-update, since an adversarial prover may use non-zero values in Δ to change the memory state arbitrarily.

We make a key observation that the correct Δ should simply be a T-sized sparse representation of $\mathbf{w}$. To ensure that Δ is 0 at unmodified addresses, we set the numerators to $\mathbf{w}_j, \Delta_i$ instead of $1, \boldsymbol{m}_i$. Namely,

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

Note that the ith fraction on the right hand side is 0 if and only if the ith value is unmodified by any write operation, and is equal to the left hand side if and only if the ith value is modified by the correct amount. Only ℓ out of all T fractions on the right hand side are nonzero, which implies an honest prover only need to do $O(\ell)$ work, resolving the second issue aforementioned. Section 4.3 shows that this LogUp-style mem-update argument is secure and indeed leads to a protocol with prover complexity independent of T.

LogUp powered memory-proving. The mem-update argument can be used to show that the memory is updated strictly at the addresses written to and with the correct amount. We can then use a homomorphic commitment to Δ to efficiently update our commitment to the memory state. In addition, we can use the indexed LogUp argument demonstrated in the intermediate step above to show that all the values initially read are consistent with the old memory. Nevertheless, merely checking these two properties (property 2 and 3) only suffice in a system where all write operations happen synchronously at the end of the computation step. Without additional checks, we would need to first update the memory whenever we read from an address that has been previously written to. This requires an expensive homomorphic commitment operation to be executed by the verifier as part of the recursive circuit. To resolve this we employ the classic permutation-based offline memory-checking idea [BEGKN91] and add a check for property 1 in our memory-proving protocol.

All three subprotocols are based on the LogUp argument and are described in Section 4. Section 5 discusses the overall memory-proving protocol and its efficiency.

Accumulation-friendly verision of GKR. Our memory-proving protocols have *almost* optimal parameters. It require committing[5] to only 15 field elements per memory access. However, 9 of these vectors consist of large field elements, i.e. $\log |\mathbb{F}|$-bit, even if the memory itself only consists of small entries. For instance, say the memory only contains 32-bit entries; using homomorphic commitments requires fields of size at least 2^{256}, which is a factor 8 blowup. Concretely, in this example the 9 large elements contribute 2300 bits and the 6 small elements only 192 bits to the prover's commitment cost.

Removing this blowup motivates the second orthogonal but highly compatible contribution of this paper: We construct an efficient accumulation scheme for the GKR protocol. GKR can be used to 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 $O(k \cdot \log n)$

[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/`.

rounds where k is the depth of the circuit and $\log n$ its width. Compilation 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 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 the benefit of using GKR. Fortunately, the idea of using a higher degree sumcheck with fewer variables 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 protocol 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.

1.3 Roadmap

In Section 2, we provide the necessary preliminaries to comprehend our construction. We describe the desired construction for lists of read/write operations in Section 3 and outline three properties that consistent read/write lists should uphold. Subsequently, in Section 4, we introduce three LogUp-style special-sound subprotocols, each tailored for proving one of the three aforementioned properties. These subprotocols are combined in parallel to form the memory-proving interactive protocol Π_{MP} in Section 5, which exhibits the desired characteristics for conversion into an efficient accumulation scheme and IVC using the ProtoStar compiler. Specifically, $\mathit{\Pi}_{\mathsf{MP}}$ has only 2 rounds and verifier degree 3, and its number of nonzero elements in prover messages is independent of T. In Section 7, we elucidate how our accumulation-friendly version of GKR (components described in Appendix 6) can be leveraged to optimize our memory-proving IVC scheme, and we highlight several other useful applications of GKR in the context of IVC. The extension of our memory-proving protocol to the setting of key-value store is described in Appendix B.1.

2 Preliminaries

Notation. For $n \in \mathbb{N}$, we use $[n]$ to denote the set $\{1, 2, \dots, n\}$. We denote λ as the security parameter and use $\mathbb{F}$ to denote a field of prime order p such that $\log(p) = \mathit{\Omega}(\lambda)$. For list of tuples $ltup = [(a_i, b_i, c_i, \dots)]_{i=1}^{k}$ 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 $\mathit{\Pi} = (\mathsf{P}, \mathsf{V})$ 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. $\mathit{\Pi}$ 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 $(a_1, \dots, a_\mu) \in \mathbb{N}^{\mu}$. An $(a_1, \dots, a_\mu)$-tree of transcript for a $(2\mu+1)$-move public-coin interactive proof $\mathit{\Pi}$ is a set of $a_1 \dot{a}_2 \dots a_\mu$ accepting transcripts arranged in a tree of depth μ and arity $a_1, \dots, a_\mu$ respectively. The nodes in the tree correspond to the prover messages and the edges to the verifier's challenges. Every internal node at depth $i-1$ $(1 \leq i \leq \mu)$ 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 $(a_1, \dots, a_\mu) \in \mathbb{N}^{\mu}$. A $(2\mu+1)$-move public-coin interactive proof $\mathit{\Pi}$ for relation $\mathcal{R}$ where the verifier samples its challenges from a set of size N is $(a_1, \dots, a_\mu)$-out-of-N special-sound if there exists a polynomial time algorithm that, on input pi and any $(a_1, \dots, a_\mu)$-tree of transcript for $\mathit{\Pi}$ 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* $i \in [\ell]$*,*

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

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

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:*

- $\mathsf{P}_{\mathsf{IVC}}(m, z_0, z_m, z_{m-1}, \mathbf{w}_{\mathsf{loc}}, \pi_{m-1}]) \to \pi_m$*. 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*.*
- $\mathsf{V}_{\mathsf{IVC}}(m, z_0, z_m, \pi_m) \to b$*. 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.*

Table 2. Efficiency of IVC schemes compiled from sps protocol

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

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 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*

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

where all inputs are linear functions in a formal variable X[7]*, and* $f_{j,i}^{\mathsf{V}_{\mathsf{sps}}}$ *is the ith* $(0 \leq i \leq \ell - 1)$ *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 Constructing Read List and Write List

In our memory-proving algorithm, we assume that the list of "reads" and the list of "writes" we are given were constructed in a similar way as in the classic offline memory-checking process [BEGKN91; CDvGS03; SAGL18]. More importantly, our algorithm makes specific use of the "initial reads" and "final writes" in the memory-checking process, which we explicitly define in this section.

Consider an untrusted server who performs read/write operations to a memory. The memory is represented as a T-sized vector of memory values, where the addresses are the indices $1, \ldots, T$. Suppose OM is the starting, old memory. The server locally intializes two lists, RL and WL, to empty lists. 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 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 $\mathsf{RL}$
3 : stores $(v, t^*)$ at the memory
4 : append $(a, v, t^*)$ to $\mathsf{WL}$
5 : $t^* \leftarrow t^* + 1$
```

When a write operation of value v' to address a occurs, the checker updates RL, WL in the same way except that the entry appended to WL will contain the new value v'.

Naturally, the entries in RL and WL would be sorted in increasing order of timestamp. Then we construct the "initial reads" R and "final writes" W as follows:

[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].

```
$R, W, A_R, A_W \leftarrow \{\}$
for $(a, v, t) \in \mathsf{RL}$ do
  if $a \notin A_R$ then do
    append $(a, v, t)$ to $R$
    $A_R \leftarrow A_R \cup \{a\}$
for $(a, v, t) \in \mathsf{WL}.rev$ do
  if $a \notin A_W$ then do
    append $(a, v, t)$ to $W$
    $A_W \leftarrow A_W \cup \{a\}$
```

Finally, we sort R, W by addresses in the same order[8], and return $\mathsf{Rd} := \mathsf{RL}||W$ and $\mathsf{Wr} := \mathsf{WL}||R$.

Lemma 2. *(Contrapositive of Lemma 1 from [BEGKN91]) If* Rd *and* Wr *are permutations of each other, then the read/write operations are consistent with each other. In other words, for every address, the value and timestamp read are consistent with the value and timestamp previously written.*

Remark 1. The protocol guarantees that $|\mathsf{RL}| = |\mathsf{WL}|$ and $\mathsf{RL}.\boldsymbol{a} = \mathsf{WL}.\boldsymbol{a}$ 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.

Let $\ell := |R| = |W|$ and $k := |\mathsf{RL}| = |\mathsf{WL}|$. Note that k is at most 2ℓ, therefore $k = O(\ell)$.

Remark 2. The memory accesses and the memory updates were performed correctly if and only if the following three properties are satisfied:

1. Rd and Wr are permutations of each other, as described in Lemma 2
2. All the initially read values $\mathbf{R}.\boldsymbol{v}$ are consistent with the old memory OM.
3. The new memory NM is updated only at the addresses written to and with the correct amount. In other words, the T-sized vector $\mathsf{NM} - \mathsf{OM}$ should be an ℓ sparse representation of the ℓ-sized vector $W - R$.

4 Special-Sound Subprotocols for Memory-Proving

We introduce the three LogUp-style 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 might be tuples of addresses, values, and/or timestamps.

[8] Sorting takes $O(\ell \log \ell)$ time, but this is entirely prior to and not a part of our memory-proving protocol.

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.

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 $\mathbf{h}_j = \frac{\boldsymbol{n}_j}{\boldsymbol{d}_j}$ $\forall j \in [|\mathbf{h}|]$, 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.

4.1 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 3. *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*

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

See Appendix A.1 for a proof of Lemma 3.

We can therefore describe a special-sound protocol Π_{perm} for $\mathcal{R}_{perm}$ by simply adding the check $\ell \overset{?}{=} T$ 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** $\mathsf{V}(\mathbf{t} \in \mathbb{F}^T)$
	$\xrightarrow{\quad\mathbf{w}\quad}$	
Compute $\mathbf{h} \in \mathbb{F}^\ell, \mathbf{g} \in \mathbb{F}^T$	$\xleftarrow{\quad x_1\quad}$	$x_1 \leftarrow\$\, \mathbb{F}$
$\mathbf{h}_j \leftarrow \frac{1}{x_1 + \mathbf{w}_j} \;\; \forall j \in [\ell]$		
$\mathbf{g}_i \leftarrow \frac{1}{x_1 + \mathbf{t}_i} \;\; \forall i \in [T]$	$\xrightarrow{\quad\mathbf{h}, \mathbf{g}\quad}$	$\ell \overset{?}{=} T$
		$\sum_{j=1}^{\ell} \mathbf{h}_j \overset{?}{=} \sum_{i=1}^{T} \mathbf{g}_i$
		$\mathbf{h}_j \cdot (x_1 + \mathbf{w}_j) \overset{?}{=} 1 \;\; \forall j \in [\ell]$
		$\mathbf{g}_i \cdot (x_1 + \mathbf{t}_i) \overset{?}{=} 1 \;\; \forall i \in [T]$

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 $j \in [\ell], i \in [T]$.

4.2 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 4 and 5 in the following are extensions on Lemma 4 and 5 from [Hab22], respectively. See Appendix A.2 for proofs of Lemma 4 and Lemma 5.

Lemma 4. *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 $z_1, z_2 \in \mathbb{F}^2$.

Lemma 5. *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 $(\boldsymbol{b}, \mathbf{w})$.

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

Special-sound protocol Π_{ivlk} for $\mathcal{R}_{ivlk}$

Prover $\mathsf{P}(\mathcal{C}_{ivlk}, \mathbf{w} \in \mathbb{F}^\ell, \boldsymbol{b} \in \mathbb{F}^\ell)$		**Verifier** $\mathsf{V}(\mathcal{C}_{ivlk})$
Compute $\boldsymbol{m} \in \mathbb{F}^T$ such that		
$\boldsymbol{m}_i = \sum_{j=1}^{\ell} \mathbb{1}(\mathbf{w}_j = \mathbf{t}_i) \ \ \forall i \in [T]$	$\xrightarrow{\mathbf{w}, \boldsymbol{b}, \boldsymbol{m}}$	
	$\xleftarrow{x_1, x_2}$	$x_1, x_2 \xleftarrow{\$} \mathbb{F}^2$
Compute $\mathbf{h} \in \mathbb{F}^\ell, \mathbf{g} \in \mathbb{F}^T$		
$\mathbf{h}_j \leftarrow \frac{1}{x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{w}_j} \ \ \forall j \in [\ell]$		
$\mathbf{g}_i \leftarrow \frac{\boldsymbol{m}_i}{x_1 + x_2 \cdot i + \mathbf{t}_i} \ \ \forall i \in [T]$	$\xrightarrow{\mathbf{h}, \mathbf{g}}$	$\sum_{j=1}^{\ell} \mathbf{h}_j \overset{?}{=} \sum_{i=1}^{T} \mathbf{g}_i$
		$\mathbf{h}_j \cdot (x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{w}_j) \overset{?}{=} 1 \ \ \forall j \in [\ell]$
		$\mathbf{g}_i \cdot (x_1 + x_2 \cdot i + \mathbf{t}_i) \overset{?}{=} \mathbf{m}_i \ \ \forall i \in [T]$

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

See Appendix A.2 for a proof of Lemma 6.

4.3 Mem-Update Relation

Definition 11 (Mem-Update Relation). *Given configuration $\mathcal{C}_{mu} := (T, \ell, \Delta)$ 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 $\mathcal{R}_{mu}$ 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 7. *Let $\mathbb{F}$ be a field of characteristic $p > \max\{\ell, T\}$. Given the sequences of field elements $\mathbf{w} \in \mathbb{F}^\ell, \boldsymbol{b} \in \mathbb{F}^\ell, \varDelta \in \mathbb{F}^T$, we have $(T, \ell, \varDelta, \mathbf{w}, \boldsymbol{b}) \in \mathcal{R}_{mu}$ if and only if the following equation holds in the function field $F(X, Y)$*

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

See Appendix A.3 for a proof of Lemma 7.

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

Special-sound protocol $\varPi_{\text{mu}}$ for $\mathcal{R}_{\text{mu}}$

Prover $\mathsf{P}(\mathcal{C}_{\text{mu}}, \mathbf{w} \in \mathbb{F}^\ell, \boldsymbol{b} \in \mathbb{F}^\ell)$		**Verifier** $\mathsf{V}(\mathcal{C}_{\text{mu}})$
	$\xrightarrow{\mathbf{w}, \boldsymbol{b}}$	
	$\xleftarrow{x_1, x_2}$	$x_1, x_2 \leftarrow\!\$\, \mathbb{F}^2$
Compute $\mathbf{h} \in \mathbb{F}^\ell, \mathbf{g} \in \mathbb{F}^T$		
$\mathbf{h}_j \leftarrow \dfrac{\mathbf{w}_j}{x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{w}_j} \quad \forall j \in [\ell]$		
$\mathbf{g}_i \leftarrow \dfrac{\varDelta_i}{x_1 + x_2 \cdot i + \varDelta_i} \quad \forall i \in [T]$	$\xrightarrow{\mathbf{h}, \mathbf{g}}$	$\sum_{j=1}^{\ell} \mathbf{h}_j \overset{?}{=} \sum_{i=1}^{T} \mathbf{g}_i$
		$\mathbf{h}_j \cdot (x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{w}_j) \overset{?}{=} \mathbf{w}_j \;\; \forall j \in [\ell]$
		$\mathbf{g}_i \cdot (x_1 + x_2 \cdot i + \varDelta_i) \overset{?}{=} \varDelta_i \;\; \forall i \in [T]$

Complexity. $\varPi_{\text{mu}}$ 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 8. *$\varPi_{mu}$ is $((\ell+T), 2(\ell+T))$-special-sound, assuming each entry $\mathbf{w}_j, \varDelta_i$ for all $j \in [\ell], i \in [T]$ is a single value.*

See Appendix A.3 for a proof of Lemma 8.

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 7. 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.

5 The LogUp-Powered Memory-Proving Algorithm

5.1 Using LogUp-style Relations for Memory-Proving

The full Read/Write Memory-Proving Algorithm $\varPi_{\mathsf{MP}}$ is given in Appendix B.

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 7. 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	$O(\ell)$	$2\ell\,\mathbb{F} + 2\ell\mathbb{T}$	3	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 + $O(\ell\log\ell)\mathbb{F}$	$(c+1)\log T\mathbb{G}$

Given the old memory $\mathsf{OM} = [\boldsymbol{v}_i]_{i=1}^{T}$; $\mathsf{Rd} = \mathsf{RL}||W = [(\boldsymbol{a}_i, \boldsymbol{v}_i, \boldsymbol{t}_i)]_{i=1}^{k}$ and $\mathsf{Wr} = \mathsf{WL}||R = [(\boldsymbol{a}_i, \boldsymbol{v}_i, \boldsymbol{t}_i)]_{i=1}^{k}$, which were constructed as described in Section 3. Let $\ell := |W| = |R|$.

In Π_{MP}, the prover takes as input $(\mathsf{OM}, \mathsf{Rd} = \mathsf{RL}||W, \mathsf{Wr} = \mathsf{WL}||R)$, and the verifier takes as input $(\mathsf{OM}^{\mathsf{V}}, \mathsf{RL}, \mathsf{WL})$, 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. $R.\boldsymbol{a} \overset{?}{=} W.\boldsymbol{a}$. The rest of the protocol is composed of the following three LogUp-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 independent of 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 $\mathcal{R}_{\mathrm{mu}}$.

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 independent of T: running Π_{perm} is linear in k, and Π_{ivlk} and Π_{mu} 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 $O(\ell)$ and independent of T.

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

Computing commitments in the accumulation scheme When we use the ProtoStar compiler to turn our memory-proving protocol Π_{MP} into an accumulation scheme, the resulting accumulation prover $\mathsf{P}_{\mathsf{acc}}$ will send the homomorphic commitments to the prover messages instead of the plain vectors. The homomorphic commitments to the $O(\ell)$-sized and ℓ-sparse vectors can all be computed in time independent of T since committing to 0 is free. Moreover, the commitment to NM can be computed in one step by adding the commitment to Δ and the commitment to OM.

Speeding up Memory-Proving with LogUp-GKR (described in Section 7). In the memory-proving protocol the prover's messages are either $O(\ell)$ sized or $O(\ell)$ sparse. However, a more fine-grained 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 7, 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 $\mathit{\Pi}_{\mathsf{MP}}$, some of the vectors of fractions sent by the prover are sparse (E.g. $\boldsymbol{g}^{\mathrm{ivlk}}, \boldsymbol{g}^{\mathrm{mu}}$). 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 independent of 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.

Extending to key-value store Our protocol can be extended to prove the correctness of key-value store, which is very similar to memory access but the storage does not have a fixed size T. We describe the details of this extension in Appendix B.1.

5.2 Accumulation prover runs in time independent of T

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

Underlying special-sound prover runs in $O(\ell)$ time As can be seen in Appendix C, all computations of the prover in $\mathit{\Pi}_{\mathsf{MP}}$ can be done in $O(\ell)$ time. Vectors $\boldsymbol{h}^{\mathrm{perm}}, \boldsymbol{g}^{\mathrm{perm}}, \boldsymbol{h}^{\mathrm{ivlk}}, \boldsymbol{h}^{\mathrm{mu}}$ all have $O(\ell)$ size, so they can clearly be computed in $O(\ell)$ time. Vectors $\boldsymbol{m}, \boldsymbol{\mathit{\Delta}}, \boldsymbol{g}^{\mathrm{ivlk}}, \boldsymbol{g}^{\mathrm{mu}}$ 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 $O(\ell)$ time for an honest prover, since only $\mathit{\Delta}$ is sparse and only ℓ locations in the memory table need to be changed.

Computing the cross error terms in $O(\ell)$ time 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 $O(\ell)$ time, as the vectors will be either $O(\ell)$-sized or ℓ-sparse. The detailed 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 time independent of T in the kth round of accumulation is given Appendix C.1.

Note that this helper algorithm is only required when LogUp-GKR (described in Section 7) 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.

Updating the accumulator in $\boldsymbol{O(\ell)}$ 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 $O(\ell)$ time because $\pi.\mathbf{g}$ is ℓ-sparse, the complexity for naively computing $\mathsf{acc}'.\mathsf{OM}$ is linear in T. We show a trick in Appendix C.2 that enables us to accumulate OM in time independent of T.

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 B), the prover first commits to R, W and $\boldsymbol{m}$. It also commits to $\varDelta$ 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 $\varDelta$ 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 5.2 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	$O(\ell)$	$5\ell\mathbb{T} + 6\ell\mathbb{F}$	3	2	$(6\ell, \mathbb{T})$-MSM $+(9\ell, \mathbb{F})$-MSM	$4\mathbb{G}$
Using GKR	$O(\ell \log \ell)$	$5\ell\mathbb{T} + O(T^{1/c})$	7	$(c+1) \cdot \log T$	$(6\ell, \mathbb{T})$-MSM	$(c+1) \log T\mathbb{G}$

6 Accumulation-Friendly GKR

Right now, th prover in our memory-proving IVC scheme needs to commit to 15 field elements per memory access, 9 of which are small memory entries, and 6 of which are large field elements, i.e. $\log |\mathbb{F}|$-bit, since they are the inverses of the memory entries. As an example, say the memory only contains 32-bit entries. Using homomorphic commitments require fields of size at least 2^{256}, leading to a factor $256/32 = 8$ blowup when computing commitments.

An intuitive solution is to employ the GKR protocol, since it has the advantage of only requiring committing to the inputs/outputs and not any intermediate values of the circuit wires. Unfortunately, naively using GKR in accumulation results in $\log^2 n$ rounds (assuming n is the number of inputs), which is expensive since the ProtoStar accumulation compiler pays linearly in the number of rounds. We design a version of the GKR protocol that is better suited for accumulation. It takes fewer rounds but retains the desired property of not requiring committing to any intermediate values. The core ingredient is a bivariate sumcheck protocol which only has two rounds.

6.1 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 Π_{eval} below.

Subprotocol Π_{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 $\mathsf{P}(f, \boldsymbol{a} = [\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k])$

$\boldsymbol{a}^i \leftarrow [\boldsymbol{a}_1^i, \ldots, \boldsymbol{a}_k^i] \quad \forall i \in \{2, 4, \ldots, m\}$

$A := (\boldsymbol{a}, \boldsymbol{a}^2, \boldsymbol{a}^4 \ldots, \boldsymbol{a}^m)$

$L_x^{\mathbb{H}}(u) := \frac{c_x(u^m - 1)}{u - x} \quad \forall x \in \mathbb{H}$

$\xrightarrow{A,\ L_x^{\mathbb{H}}(u)\ \forall x \in \mathbb{H}}$

Verifier $\mathsf{V}(f, \boldsymbol{a}, [f(\boldsymbol{x})]_{\boldsymbol{x} \in \mathbb{H}^k})$

$A(0) \stackrel{?}{=} \boldsymbol{a} \quad A(i) \stackrel{?}{=} A(i-1)^2 \quad \forall i \in \{1, \ldots, \log m - 1\}$

$\prod_{j=1}^{k} L_{\boldsymbol{x}_j}^{\mathbb{H}}(\boldsymbol{a}_j) \cdot \prod_{j=1}^{k} (\boldsymbol{a}_j - \boldsymbol{x}_j) \stackrel{?}{=} \prod_{j=1}^{k} c_{\boldsymbol{x}_j} \cdot (A(\log m, j) - 1) \quad \forall \boldsymbol{x} \in \mathbb{H}^k$

$f(\boldsymbol{a}) \leftarrow \sum_{\boldsymbol{x} \in \mathbb{H}^k} \Big(\prod_{j=1}^{k} L_{\boldsymbol{x}_j}^{\mathbb{H}}(\boldsymbol{a}_j) f(\boldsymbol{x}) \Big)$

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

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

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.

6.2 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 in sumcheck. 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** $\mathsf{V}(f, T)$
$f_1(X) \leftarrow \sum_{y \in \mathbb{G}_2} f(X, y)$	$\xrightarrow{f_1(\omega_i)\ \ \forall i \in [m]}$	
	$\xleftarrow{a}$	$a \leftarrow\!\!\$\ \mathbb{F} \setminus \mathbb{H}$
$f_2(Y) \leftarrow f(a, Y)$	$\xrightarrow{f_2(\omega_i)\ \ \forall i \in [m]}$	
	$\xleftarrow{b}$	$b \leftarrow\!\!\$\ \mathbb{F} \setminus \mathbb{H}$
$T^* \leftarrow f_2(b)$	$\xrightarrow{T^*}$	Use $\mathit{\Pi}_{eval}$ to evaluate $f_1(a), f_2(b)$ $\sum_{x \in \mathbb{G}_1} f_1(x) \overset{?}{=} T$ $\sum_{y \in \mathbb{G}_2} f_2(y) \overset{?}{=} f_1(a)$ $T^* \overset{?}{=} f_2(b)$ $T^* \overset{?}{=} f(a, b)$

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 (independent of the memory size T) when performing memory-proving using our LogUp-powered protocol, the actual number of P messages will be much smaller.

6.3 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

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

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:

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

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 $g(r)=0$.

7 LogUp GKR protocol using the batching subprotocol

We incorporate the subprotocols described in Section 6 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 $\mathbb{F}$. The full protocol is in Appendix D. We will use this protocol to do the verifier checks for the LogUp-style arguments in Π_{MP}.

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. Here, n is width of the GKR circuit, c is the number of variant in the sumcheck protocol, and k is the degree of the sumcheck polynomial.

Variant	P Time	$\lvert\mathsf{P}$ Msg$\rvert$	deg(V)	# P Msgs	$\mathsf{P}_{\mathsf{acc}}$ Time	$\mathsf{V}_{\mathsf{acc}}$ Time
bivariate $\sum$	$O(n \log n)$	$O(n^{1/2})$	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 $+O(n \log n)\mathbb{F}$	$(c+1) \cdot k$ $+2\mathbb{G}$

7.1 Other Applications of GKR in IVC

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 $s_{\lambda-1} \dots s_1 s_0$ where s_i is either 0 or 1 for every $i \in \{0, \dots, \lambda - 1\}$ 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 $\lambda/(k+1)$.

In Appendix D.1, we give a concrete formula for scalar multiplication $s \cdot G$ using GKR in the short Weierstass curves $Y^2 = X^3 + b$.

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. We would also like to thank Sebastian Angel for the discussion on Spice and key-value store.

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.

[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.

[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.

[BCTV14] 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.

[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.

[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.

[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.

[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.

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

[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`.

[Val08] 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`.

[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 Proofs for Lemmas

A.1 Proof for Lemma in Section 4.1

Lemma 3. *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.

A.2 Proofs for Lemmas in Section 4.2

Lemma 4. *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 $z_1 \in \mathbb{F}, z_2 \in \mathbb{F}$,

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

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 5. *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 4, 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 $(\boldsymbol{b}, \mathbf{w})$.

Lemma 6. *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

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

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 7 $(\mathcal{C}_{ivlk}; \mathbf{w}, \boldsymbol{b}) \in \mathcal{R}_{\mathrm{mu}}$. 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.

A.3 Proof for Lemma in Section 4.3

Lemma 7. *Let $\mathbb{F}$ be a field of characteristic $p > \max\{\ell, T\}$. Given the sequences of field elements $\mathbf{w} \in \mathbb{F}^{\ell}, \boldsymbol{b} \in \mathbb{F}^{\ell}, \varDelta \in \mathbb{F}^{T}$, we have $(T, \ell, \varDelta, \mathbf{w}, \boldsymbol{b}) \in \mathcal{R}_{mu}$ if and only if the following equation holds in the function field $F(X,Y)$*

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

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

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

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

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

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 4, 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 $\sum_{i=1}^{T} \frac{\varDelta_i}{X+Y\cdot i+\varDelta_i}$ decomposition with equivalent numerator and denominator. This implies that for non-zero $z_2 \cdot \mu_{\mathbf{w}}(z_1, z_2) = \varDelta_{z_1} = z_2$, so $\mu_w(z_1, z_2) = 1$. Thus, for all $j \in [\ell]$ such that $\mathbf{w}_j \neq 0$, $\varDelta_{\boldsymbol{b}_j} = \mathbf{w}_j$.

For every $\frac{\varDelta_i}{X+Y\cdot i+\varDelta_i}$, 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 $\varDelta_i = z_2 \cdot \mu_{\mathbf{w}}(z_1, z_2) = z_2$ for every $\varDelta_i$. Therefore if $\varDelta_i \neq 0$ then $\mu_{\mathbf{w}}(z_1, z_2) \neq 0$, and thus there exists $j \in [\ell]$ such that $\boldsymbol{b}_j = i$ and $\varDelta_i = \mathbf{w}_j$.

Lemma 8. *Π_{mu} is $((\ell{+}T), 2(\ell{+}T))$-special-sound, assuming each entry $\mathbf{w}_j, \varDelta_i$ 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 $x_1^{(p)} + x_2^{(q)} \cdot i + \varDelta_i \neq 0$ 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 $\mathbf{g}_i = \frac{\varDelta_i}{x_1^{(p)} + x_2^{(q)} \cdot i + \varDelta_i}$. Define the degree $\ell + T - 1$ polynomial

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

If $f(X,Y)$ is the zero polynomial then $\sum_{j=1}^{\ell} \frac{\mathbf{w}_j}{X + Y \cdot \boldsymbol{b}_j + \mathbf{w}_j} = \sum_{i=1}^{T} \frac{\varDelta_i}{X + Y \cdot i + \varDelta_i}$ and by Lemma 7 $(\mathcal{C}_{\text{mu}}; \mathbf{w}, \boldsymbol{b}) \in \mathcal{R}_{\text{mu}}$. 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.

B Full Memory-Proving Protocol

Special-sound Lookup-Powered Memory-Proving Protocol Π_{MP}

Prover $\mathsf{P}(\mathsf{OM}, \mathsf{Rd} = \mathsf{RL}||W, \mathsf{Wr} = \mathsf{WL}||R)$

Verifier $\mathsf{V}(\mathsf{OM}^{\mathsf{V}}, \mathsf{RL}, \mathsf{WL})$

Compute $\boldsymbol{b} \in \mathbb{F}^{\ell}, \mathbf{r} \in \mathbb{F}^{\ell}, \boldsymbol{m} \in \mathbb{F}^{T}$,
$\mathbf{w} \in \mathbb{F}^{\ell}, \varDelta \in \mathbb{F}^{T}$ such that:

$\boldsymbol{b} := R.\boldsymbol{a} \quad \mathbf{r} := R.\boldsymbol{v}$

$$\boldsymbol{m}_i := \sum_{j=1}^{\ell} \mathbb{1}(\mathbf{r}_j = \mathsf{OM}_i) \;\; \forall i \in [T]$$

$\mathbf{w} := W.\boldsymbol{v} - R.\boldsymbol{v}$

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

Efficiently computes $\mathsf{NM} \leftarrow \varDelta + \mathsf{OM}$

$$\xrightarrow{R, W, \boldsymbol{m}, \mathsf{NM}}$$

$R.\boldsymbol{a} \overset{?}{=} W.\boldsymbol{a} \quad \mathsf{Rd} := \mathsf{RL}||W \quad \mathsf{Wr} := \mathsf{WL}||R$

$$\xleftarrow{x_1, x_2, x_3}$$

$x_1, x_2, x_3 \leftarrow\!\!\$\; \mathbb{F}^3$

Compute, on the fly, the linear combinations
of the values in the tuples of Rd, Wr:

$\mathsf{Rd}'_j \leftarrow \mathsf{Rd}_j.a + x_2 \cdot \mathsf{Rd}_j.v + x_3 \cdot \mathsf{Rd}_j.t \;\; \forall j \in [k]$

$\mathsf{Wr}'_j \leftarrow \mathsf{Wr}_j.a + x_2 \cdot \mathsf{Wr}_j.v + x_3 \cdot \mathsf{Wr}_j.t \;\; \forall j \in [k]$

$$\mathbf{h}_j^{\mathrm{perm}} := \frac{1}{x_1 + \mathsf{Rd}'_j} \;\; \forall j \in [k]$$

$$\mathbf{g}_j^{\mathrm{perm}} = \frac{1}{x_1 + \mathsf{Wr}'_j} \;\; \forall j \in [k]$$

$$\mathbf{h}_j^{\mathrm{ivlk}} := \frac{1}{x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{r}_j} \;\; \forall j \in [\ell]$$

$$\mathbf{g}_i^{\mathrm{ivlk}} = \frac{\boldsymbol{m}_i}{x_1 + x_2 \cdot i + \mathsf{OM}_i} \;\; \forall i \in [T]$$

$$\mathbf{h}_j^{\mathrm{mu}} := \frac{\mathbf{w}_j}{x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{w}_j} \;\; \forall j \in [\ell]$$

$$\mathbf{g}_i^{\mathrm{mu}} = \frac{\varDelta_i}{x_1 + x_2 \cdot i + \varDelta_i} \;\; \forall i \in [T]$$

Verifier:

$\boldsymbol{b}^{\mathsf{V}} := R.\boldsymbol{a} \quad \mathbf{r}^{\mathsf{V}} := R.\boldsymbol{v}$

$\mathbf{w} := W.\boldsymbol{v} - R.\boldsymbol{v} \quad \varDelta := \mathsf{NM} - \mathsf{OM}^{\mathsf{V}}$

Similarly, compute the linear combinations
on the fly:

$\mathsf{Rd}'_j \leftarrow \mathsf{Rd}_j.a + x_2 \cdot \mathsf{Rd}_j.v + x_3 \cdot \mathsf{Rd}_j.t \;\; \forall j \in [k]$

$\mathsf{Wr}'_j \leftarrow \mathsf{Wr}_j.a + x_2 \cdot \mathsf{Wr}_j.v + x_3 \cdot \mathsf{Wr}_j.t \;\; \forall j \in [k]$

$$\xrightarrow{\mathbf{h}^{\mathrm{perm}}, \mathbf{g}^{\mathrm{perm}}, \mathbf{h}^{\mathrm{ivlk}}, \mathbf{g}^{\mathrm{ivlk}}, \mathbf{h}^{\mathrm{mu}}, \mathbf{g}^{\mathrm{mu}}}$$

$$\sum_{j=1}^{k} \mathbf{h}_j^{\mathrm{perm}} \overset{?}{=} \sum_{j=1}^{k} \mathbf{g}_j^{\mathrm{perm}}$$

$$\sum_{j=1}^{\ell} \mathbf{h}_j^{\mathrm{ivlk}} \overset{?}{=} \sum_{i=1}^{T} \mathbf{g}_i^{\mathrm{ivlk}}$$

$$\sum_{j=1}^{\ell} \mathbf{h}_j^{\mathrm{mu}} \overset{?}{=} \sum_{i=1}^{T} \mathbf{g}_i^{\mathrm{mu}}$$

$\mathbf{h}_j^{\mathrm{perm}} \cdot (x_1 + \mathsf{Rd}'_j) \overset{?}{=} 1 \;\; \forall j \in [k]$

$\mathbf{g}_j^{\mathrm{perm}} \cdot (x_1 + \mathsf{Wr}'_j) \overset{?}{=} 1 \;\; \forall j \in [k]$

$\mathbf{h}_j^{\mathrm{ivlk}} \cdot (x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{r}_j) \overset{?}{=} 1 \;\; \forall j \in [\ell]$

$\mathbf{g}_i^{\mathrm{ivlk}} \cdot (x_1 + x_2 \cdot i + \mathsf{OM}_i) \overset{?}{=} \boldsymbol{m}_i \;\; \forall i \in [T]$

$\mathbf{h}_j^{\mathrm{mu}} \cdot (x_1 + x_2 \cdot \boldsymbol{b}_j + \mathbf{w}_j) \overset{?}{=} \mathbf{w}_j \;\; \forall j \in [\ell]$

$\mathbf{g}_i^{\mathrm{mu}} \cdot (x_1 + x_2 \cdot i + \varDelta_i) \overset{?}{=} \varDelta_i \;\; \forall i \in [T]$

Updates $\mathsf{OM}^{\mathsf{V}} \leftarrow \mathsf{NM}$

B.1 Extending to key-value store

Key-value store bears a strong resemblance to memory access, with two notable differences: 1) Instead of reading/writing to addresses, the server gets/inserts values under specified keys. 2) Unlike memory, the storage does not have a fixed size T; rather, it is only as large as the number of distinct keys inserted. In other words, the storage is initially empty and only grows as values are inserted under new keys. Key-value store was the setting considered in Spice [SAGL18]. Although the absence of a fixed storage bound T seems problematic when applying our memory-proving protocol, our protocol can be extended to support a verifiable key-value store. We give a brief description below.

Set T to be a cryptographically large number (e.g. 2^{256}) and initialize a T-sized storage table to all zeroes (To support actually storing the value 0, one can either use a one-bit flag to indicate whether a 0 entry is empty, or store a special symbol for the value 0). Suppose we perform ℓ key-value get/insert operations. Whenever we need to insert a value under key k, we write that value to the kth entry in the storage table. Since T is huge, obviously $\ell << T$, and only a small subset of $\{1, \ldots, T\}$ would ever be used as keys.

In each round of accumulation, $\mathsf{P}_{\mathsf{acc}}$ needs to commit to the prover messages of the underlying interactive protocol, which includes some T-sized vectors. While committing to the T-sized vectors seems overwhelmingly expensive, we note that those vectors will be sparse with at most ℓ nonzero entries, so the commitment cost is $O(\ell)$. However, we do need random access to the commitment keys in order to avoid generating all T commitment keys. For instance, we can use a hash function that maps from $\{1, \ldots, T\}$ to a prime order group; then, we only need to use the hash function for the keys inserted into the storage. Since we only need to do $O(\ell)$ commitments, only $O(\ell)$ hashes will be needed in each round.

At the end of each IVC step, the prover needs to send all the decommitted, accumulated vectors to the decider for the final verification checks. Note that the T-sized vectors will again be sparse: suppose we do $O(\ell)$ get/insert operations per accumulation round and have a total of a accumulation rounds, then the total number of nonsparse entries in the T-sized vectors will be at most $O(a\ell) << T$. Thus, instead of sending the full vectors, we can send a table of the nonsparse vector values.

There are two limitations in extending our memory-proving protocol to verifiable key-value store. First, as previously noted, our scheme necessitates random access to commitment keys. Second, this scheme for verifiable key-value store may not be compatible with our optimization using the accumulation-friendly GKR, because the GKR circuit will be very large with $\log T$ depth. We leave these as open questions for future research.

C Subalgorithms for $O(\ell)$ accumulation prover time

C.1 Computing the cross error terms in $O(\ell)$ time

Let acc be the current accumulator, π the current proof, and acc' the updated accumulator. We need to compute the cross error terms for

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

We give a description of the idea below. See Figure 1 for the algorithm.

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 $\boldsymbol{i} = [1, 2, \ldots, T]$ 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,

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

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 $O(\ell)$; moreover, since the two resulting Hadamard products will also be sparse vectors, committing them only takes $O(\ell)$ 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,

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

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 independent of T since $\hat{\pi}.\mathit{\Delta}$ 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 independent of T, which means the prover will be able to compute and commit to $\mathbf{e}_1$ in time independent of T as well.

```
Given:
$\rho_{\mathsf{acc}}$ – random oracle for accumulation,
$\mathsf{ck}$ – 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:
    $\mathsf{a\hat{c}c}.\mathbf{g}$ – item accumulated up to $(k-2)$th round,
    $\hat{\pi}.(\mathbf{g}, \mathit{\Delta})$ – proof items from the $(k-1)$th round,
    $r_{(k-1)}$ – random accumulation challenge from the $(k-1)$th round,
    $\hat{E}_{a2} = \mathsf{Commit}(\mathsf{ck}, \mathsf{a\hat{c}c}.\mathbf{g} \circ \boldsymbol{i})$, $\hat{E}_{a3} = \mathsf{Commit}(\mathsf{ck}, \mathsf{a\hat{c}c}.\mathbf{g} \circ \hat{\pi}.\mathsf{OM})$ – 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:
        $E_{a2} \leftarrow \hat{E}_{a2} + r_{(k-1)} \cdot \mathsf{Commit}(\mathsf{ck}, \hat{\pi}.\mathbf{g} \circ \boldsymbol{i})$,
        $E_{a3} \leftarrow \hat{E}_{a3} + \mathsf{Commit}(\mathsf{ck}, \mathsf{a\hat{c}c}.\mathbf{g} \circ \hat{\pi}.\mathit{\Delta} + r_{(k-1)} \cdot \hat{\pi}.\mathbf{g} \circ \pi.\mathsf{OM})$,
        $E_a \leftarrow \pi.x_1 \cdot C_{\mathbf{g}} + \pi.x_2 \cdot E_{a2} + E_{a3}$,
        $\mathbf{e}_b \leftarrow \pi.\mathbf{g} \circ \mathsf{acc}.\mathsf{OM}'$,
        $\mathbf{e}_c \leftarrow \mathsf{acc}.\mu \cdot \pi.\boldsymbol{m}$,
        $E_1 \leftarrow E_a + \mathsf{Commit}(\mathsf{ck}, \mathbf{e}_b + \mathbf{e}_c) + C_{\boldsymbol{m}}$,
        Use $\rho_{\mathsf{acc}}$ 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,
    $\pi.(\mathbf{g}, \mathit{\Delta})$ – 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 $\mathbf{g}_i^{\mathrm{ivlk}} \cdot (x_1 + x_2 \cdot i + \mathsf{OM}_i) \overset{?}{=} \boldsymbol{m}_i \quad \forall i \in [m]$ in time independent of 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.

C.2 Updating the accumulator in $O(\ell)$ 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 $O(\ell)$ time because $\pi.\mathbf{g}$ is ℓ-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 time independent of T. 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 $\pi_{(j-1)}.\varDelta_i \neq 0$. Then, for $k > j > 0$ such that j is most recent round before round k such that $\pi_{(j-1)}.\varDelta_i \neq 0$, the accumulator of OM_i for every $i \in [T]$ in round k will be

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

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 $\pi_{(k-1)}.\varDelta$ [9], 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 $\pi_{(j)}.\varDelta_i \neq 0$ respectively for every $i \in [T]$. Then, $\mathsf{acc}_{(k)}.\mathsf{OM}$ can be expressed as,

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

In round k, for every $i \in [T]$, we evaluate $\mathsf{acc}_{(k)}$ and set $\mathbf{R}_i \leftarrow r^*$ if and only if $\pi_{(k-1)}.\varDelta_i = 0$. This way, in every round, only ℓ positions of the accumulated OM and ℓ positions of $\mathbf{R}$ need to be computed, achieving $O(\ell)$ 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 $\pi_{(k)}.\mathsf{OM}$.

[9] 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 C

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 $\pi_{(j)}.\mathit{\Delta}_i \neq 0$ for the corresponding $i \in [T]$,
$\mathsf{acc}_{(k-1)}.\mathsf{OM}$ – the current accumulated OM,
$\pi_{(k)}.(\mathsf{OM}, \mathit{\Delta})$ – memory and update vectors in the new proof.
Item stored from the previous round:
$\hat{\pi}_{(k-1)}.\mathit{\Delta}$ – 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. $\pi_{(k-1)}.\mathit{\Delta}_i \neq 0$ :
        $\mathsf{acc}_{(k)}.\mathsf{OM}_i \leftarrow \mathsf{acc}_{(k-1)}.\mathsf{OM}_i +$
                  $(r^* - \mathbf{R}_i) \cdot (\pi_{(k)}.\mathsf{OM}_i - \pi_{(k-1)}.\mathit{\Delta}_i) + r_{(k)} \cdot \pi_{(k-1)}.\mathit{\Delta}_i$
        $\mathbf{R}_i \leftarrow r^*$
      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:
$\pi_{(k)}.\mathit{\Delta}$ – update vector of this round.

Fig. 2. Algorithm for accumulating OM in the kth round of accumulation in time independent of 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.

D Full Adapted LogUp-GKR Protocol

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

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

  - 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 $(1, \omega \cdot y_1)$:

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

  - 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 $i = 1, \dots, \log m/2$: 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

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

  - P claims that $\sum_{v \in \mathbb{H}_i} f^{(i)}_{r_i}(v) = T_i$.
  - P and V apply the sum-check protocol to $f^{(i)}_{r_i}$, up until V's final check in that protocol, when V must evaluate $f^{(i)}_{r_i}$ at a randomly chosen point $y_{i+1} \in \mathbb{F}$.
  - P sends over $[T^{(i+1)}_j]_{j=1}^4$, 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 $[T^{(i+1)}_j]_{j=1}^4$ to perform the final check in the sumcheck protocol.
  - V chooses $r_{i+1} \leftarrow\$\ \mathbb{F}$ at random and sets $T_{i+1} \leftarrow \sum_{j=1}^4 r_{i+1}^{j-1} T^{(i+1)}_j$.

Phase 3:

- For $i = \log m/2 + 1, \dots, \log m - 1$: 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

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

  - P claims that $\sum_{u, v \in \mathbb{H}_{i'}, \mathbb{H}} f^{(i)}_{r_i}(u, v) = T_i$.

- 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

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

D.1 Concrete Formula for short Weierstass curves $Y^2 = X^3 + b$

Suppose we are given a scalar s in bit representation $s_{\lambda-1} \dots s_1 s_0$, 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 $s = s_{\lambda-1} \dots s_0$ 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

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

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 6.3). If we are doing multiple EC multiplications in parallel then these can be combined using Lagrange polynomials.