Polynomial IOPs for Memory Consistency Checks in Zero-Knowledge Virtual Machines

Yuncong Zhang[1][0000−0001−8826−5844], Shi-Feng Sun[1][0000−0003−3742−5275]⋆, Ren Zhang[2][0000−0003−2063−1769], and Dawu Gu[1][0000−0002−0504−9538]⋆

[1] Shanghai Jiao Tong University, China
{shjdzhangyuncong,shifeng.sun,dwgu}@sjtu.edu.cn
[2] Cryptape Co. Ltd. and Nervos, China
ren@nervos.org

Abstract. Zero-Knowledge Virtual Machines (ZKVMs) have gained traction in recent years due to their potential applications in a variety of areas, particularly blockchain ecosystems. Despite tremendous progress on ZKVMs in the industry, no formal definitions or security proofs have been established in the literature. Due to this lack of formalization, existing protocols exhibit significant discrepancies in terms of problem definitions and performance metrics, making it difficult to analyze and compare these advancements, or to trust the security of the increasingly complex ZKVM implementations.

In this work, we focus on random-access memory, an influential and expensive component of ZKVMs. Specifically, we investigate the state-of-the-art protocols for validating the correct functioning of memory, which we refer to as the *memory consistency checks*. Isolating these checks from the rest of the system allows us to formalize their definition and security notion. Furthermore, we summarize the state-of-the-art constructions using the Polynomial IOP model and formally prove their security.

Observing that the bottleneck of existing designs lies in sorting the entire memory trace, we break away from this paradigm and propose a novel memory consistency check, dubbed Permem. Permem bypasses this bottleneck by introducing a technique called the address cycle method, which requires fewer building blocks and—after instantiating the building blocks with state-of-the-art constructions—fewer online polynomial oracles and evaluation queries. In addition, we propose gcq, a new construction for the lookup argument—a key building block of the memory consistency check, which costs fewer online polynomial oracles than the state-of-the-art construction cq.

Keywords: Proof System, SNARK, ZKVM, Random Access Memory

1 Introduction

Zero-Knowledge Virtual Machine (ZKVM) [zkS22, TV22, Pol22, Scr22, Mid22, Ris22, GPR21] is a type of program execution system that can produce a proof

⋆ Corresponding author.

of the validity of the execution without revealing any secret inputs. These proofs can be verified quickly without re-executing the program. ZKVMs are considered more user-friendly than traditional circuit-based SNARKs [Gro16, CHM+20, GWC19] for programmers, because ZKVMs support instruction-based programs that can be easily constructed from high-level languages. Some ZKVMs [zkS22, Pol22, Scr22], often referred to as zkEVMs, are designed to be compatible with the Ethereum Virtual Machine (EVM), and have the potential to improve the scalability and privacy of Ethereum, a decentralized platform with the second-largest market value as of 2023, the time of this writing. Other ZKVMs support various types of machine architectures, such as RISC-V [Ris22] for wider applications, or SNARK-friendly machines [TV22, Scr22, GPR21] for increased efficiency.

Constructing a ZKVM involves designing protocols for checking the consistent functioning of all its components, including the instruction fetcher, register file, arithmetic logic unit, and memory. The most technically challenging protocol among them is the *memory consistency check (MCC)*, whose complexity roots in the *history-dependent* nature of memory: the output of memory access depends on the entire history of its inputs. This characteristic causes the MCC to be more resource-intensive than other protocols. Consequently, many ZKVM projects such as Scroll [Scr22] and Triton VM [TV22] devoted continuing efforts to optimizing the MCC.

However, there is an absence of literature discussing recent advances in MCC, as the constructions have mostly been developed in a haphazard manner and tightly connected to their engineering projects. This leads to a lack of agreement on the formal definition of the security goals, the context of protocol design, and the performance metrics, rendering it challenging to analyze and compare different constructions. Furthermore, it is uncertain whether they contain vulnerabilities due to their lack of formal security analysis. Although there is a family of related works investigating RAM-based SNARKs [BCGT13, BCG+13, BCTV13, BBC+17, BBHR18, BCG+18], recent implementations adopt a richer and more advanced family of new techniques not covered in this literature. To address the above issues, *it is crucial to formalize the problem of MCC and to conduct a systematic examination of existing solutions*, which can help deepen our understanding of the problem, eliminate potential security risks, and identify and address the performance bottleneck.

1.1 Our Contributions

This work offers a formal analysis of the MCCs employed in popular ZKVMs in the industry and improves their performance via a new design method and a new building block. Specifically, we provide a formal definition of MCC and its security, formulated within the Polynomial IOP (PIOP) model [BFS20], which is a widely used SNARK construction model in the literature [GWC19, CHM+20, CBBZ22,SZ22,ZSZ+22]. We also extract and formalize the underlying techniques of existing MCCs in PIOPs and prove their security. Inspired by our formalization, we propose (1) a more efficient construction method called the *address*

Table 1. Comparing the MCCs, our Permem achieves the largest memory size with fewer building blocks and online polynomials outside of the building blocks. The protocols are sorted by the address space, from the most limited contiguous memory to the largest full space read-write memory. "Sort." stands for the sorting paradigm, and "AC." stands for the address cycle method, which is extracted from Arya [BCG+18] and formalized in this paper. The constant $c \geq 1$ is a user-selected integer, but $c = 1$ usually suffices as $\mathbb{F}$'s size is usually 256-bit or larger. N is the number of execution steps, which is usually orders of magnitude less than 2^{32}. **Perm.** is the number of permutation arguments. **Lookup** is the number of lookup arguments (one "Double" lookup argument achieves the same result as two "Single" lookup arguments with smaller amortized cost). **Poly.** is the number of polynomial oracles sent to the verifier online *excluding those in the building blocks*. **Queries** is the number of evaluation queries issued by the verifier *excluding those in the building blocks*.

Protocol	Method	Address Space	Writable	Build blocks		Poly.	Queries
				Perm.	Lookup		
Cairo [GPR21]	Sort.	Contiguous	×	1	0	4	0
AryaMem (optimized based on [BCG+18])	AC.	$[1..N]$	✓	2	1 Single	4	0
Miden [Mid22] etc.	Sort.	$32k$-bit	✓	1	$2k$ Double	$7 + 2k$	0
Triton [TV22]	Sort.	$\mathbb{F}^c$	✓	1	1 Single	$10 + c$	2
Permem	AC.	$\mathbb{F}^c$	✓	1	1 Single	$6 + c$	2

cycle method, which instantiates into a novel MCC named Permem, and (2) a new lookup argument called gcq. Our main contributions are as follows.

- We formally define the notion *memory consistency check* and its security (Section 3), and formalize the state-of-the-art constructions in PIOPs with security proofs under our definition (Section 4). Specifically, observing that all these constructions follow a common pattern, which we refer to as the *sorting paradigm*, we identify the key subprotocol (*sorting check*) that differentiates these constructions. We summarize all the sorting checks into three PIOPs, each for a different memory model, respectively: (a) contiguous read-only memory (Section 4.1), used by Cairo [GPR21]; (b) memory with 32- or 256-bit addresses (Section 4.2), used by Miden [Mid22], RiscZero [Ris22], and all zkEVMs; and (c) memory with the *full address space*, i.e., $\mathbb{F}^c$ for $c \geq 1$ (Section 4.3), used by Triton VM [TV22], which supports memory spaces larger than 32- or even 256-bit address with less cost.
- Next, observing the bottleneck of the sorting paradigm, we introduce a more efficient method for constructing MCCs, named *address cycle method* (Section 5.1). We extract the address-shifting permutation of Arya [BCG+18], a zero-knowledge proof for TinyRAM [BCG+13], and develop it into a method for constructing MCCs. This general method reduces the MCC construction into designing a *distinctness check*, which is to prove that all entries in a vector are distinct. This reduction not only simplifies the design workflow but also improves the performance of MCC. Using our method, we propose a new MCC, called Permem (Section 5.1); it supports the *full address space*

Table 2. Comparing the MCCs with different instantiations of the lookup argument, our new lookup argument gcq reduces the number of polynomials and queries by ≈ 3 compared to the state-of-the-art construction cq (the rows without the gcq mark). Here double-gcq is the batched version of gcq with smaller amortized costs. The protocols are sorted by the address space (column **A. Space**), from the most limited contiguous memory to the largest full space read-write memory. The star "*" means the double-gcq is alternatively constructed where the grand-sum vector $\widetilde{\mathbf{u}}$ is split (see Section 6 for details). N is the number of executed steps of the machine. **Deg.** is the maximal degree of the polynomial oracles sent from the prover. **Poly.** is the number of polynomial oracles sent to the verifier online. **Queries** is the number of evaluation queries. **Dist.** is the number of distinct evaluation points.

Protocol	A. Space	Deg.	Poly.	Queries	Dist.
Cairo [GPR21]	Contiguous	N	5	8	2
AryaMem (optimized based on [BCG+18])	$[1..N]$	2N	13	20	2
AryaMem (gcq)	$[1..N]$	2N	10	18	2
Miden etc. [Mid22, Scr22, Pol22, Ris22, zkS22]	$32k$-bit	2N	$14+4k$	$20+5k$	2
Miden etc. (double-gcq)	$32k$-bit	5N	$12+3k$	$19+4k$	2
Miden etc. (double-gcq*)	$32k$-bit	3N	$12+4k$	$19+5k$	2
Triton [TV22]	$\mathbb{F}$	2N	18	26	3
Triton (gcq)	$\mathbb{F}$	2N	15	24	3
Permem	$\mathbb{F}$	2N	15	23	3
Permem (gcq)	$\mathbb{F}$	2N	12	21	3

as Triton VM does, by extracting the core of the *contiguity check* of Triton VM and formalizing it into a distinctness check, which may also be useful in constructing PIOPs other than MCC. As shown in Table 1 and 2, Permem costs fewer building blocks, thus fewer online polynomial oracles and evaluation queries compared to Triton VM and $32k$-bit ZKVMs. Note that, as it is hard to compare Permem directly with Arya, which is constructed in the ILC model [BCG+17], we adapt the memory component of Arya in the PIOP model, named AryaMem, and include it in our tables along with the recent ZKVMs.

– Finally, we propose a novel lookup argument, which is an essential building block for most MCCs and is widely used in SNARKs [PFM+22, ABST22, CBBZ22]. We name it gcq for *grand-sum version of* cq [EFG22]. The key idea behind gcq is to replace the univariate sumcheck of cq with the grand-sum check; this technique is simple but effective, because the grand-sum check fits perfectly in the context of cq, especially when cq is used in MCC. Table 2 and 3 show that when the lookup argument is instantiated with gcq, the MCCs use fewer online polynomial oracles and evaluation queries (by 2 to 10), and has smaller proof sizes (by 2 to 10 group and field elements), compared to using the state-of-the-art construction cq[3].

[3] Strictly speaking, the corresponding PIOP protocol behind cq is without the KZG-specific optimizations.

Table 3. Comparing the MCCs with the PIOP instantiated with KZG [KZG10], our new lookup argument gcq reduces the proof sizes by $2 \sim 10$ group and field elements, and the prover costs by $2 \sim 10$ FFTs/MSMs, compared to the state-of-the-art construction cq (the rows without the gcq mark). Here double-gcq is the batched version of gcq with smaller amortized costs. The protocols are sorted by the address space, from the most limited contiguous memory to the largest full space read-write memory. The star "*" means the double-gcq is alternatively constructed where the grand-sum vector $\tilde{\mathbf{u}}$ is split (see Section 6 for details). N is the number of executed steps of the machine. The prover is dominated by FFT ($O(N \log N)$), MSM ($O(N \cdot \lambda / \log N)$), and MPE (multi-point evaluation, $O(S \log^2 S)$), where the unit is field operations, and S is the number of addresses touched by the program in the execution. In practice, S is usually at least an order of magnitude smaller than N. For all protocols, the cost of the verifier is dominated by one pairing, which is omitted from the table.

Protocol	**Address Space**	**SRS**		**Proof**		**Prover**		
		$\mathbb{G}_1$	$\mathbb{G}_2$	$\mathbb{G}_1$	$\mathbb{F}$	FFT	MSM	MPE
Cairo [GPR21]	Contiguous	N	2	7	8	5	7	0
AryaMem (optimized based on [BCG+18])	$[1..N]$	2N	2	15	20	13	15	0
AryaMem (gcq)	$[1..N]$	2N	2	12	18	10	12	0
Miden [Mid22] etc.	$32k$-bit	2N	2	$16+4k$	$20+5k$	$14+4k$	$16+4k$	0
Miden etc. (double-gcq)	$32k$-bit	5N	2	$14+3k$	$19+4k$	$12+3k$	$14+3k$	0
Miden etc. (double-gcq*)	$32k$-bit	3N	2	$14+4k$	$19+5k$	$12+4k$	$14+4k$	0
Triton [TV22]	$\mathbb{F}$	2N	2	21	26	18	21	1
Triton (gcq)	$\mathbb{F}$	2N	2	18	24	15	18	1
Permem	$\mathbb{F}$	2N	2	18	23	15	18	1
Permem (gcq)	$\mathbb{F}$	2N	2	15	21	12	15	1

1.2 Technical Overview

To better understand the protocols presented in this work, we provide an overview of the underlying intuitions. We start by introducing the necessary background concepts.

PIOP. Almost all SNARKs, including ZKVMs, follow the PIOP pipeline, which designs a PIOP and then compiles it into a non-interactive scheme via cryptographic tools [BFS20]. A PIOP is an interactive protocol between two parties, the prover and the verifier. The prover is able to send polynomials, e.g., $f(X) \in \mathbb{F}[X]$, which may be much larger than the verifier's storage. The verifier, however, only has oracle access to $f(X)$, meaning it is able to query for $y = f(z)$ for any given $z \in \mathbb{F}$. This oracle access allows the verifier to check if the polynomials satisfy certain relations, using the Swartz-Zippel Lemma. For example, by checking $f(z) + g(z) = h(z)$ for uniformly random z, the verifier ensures $f(X) + g(X) = h(X)$.

PIOPs can also be used to verify relations between vectors besides polynomials, by exploiting the natural transformations between polynomials and vectors. One popular transformation is the polynomial interpolation over a specific domain $\mathbb{D}$ of size $N = 2^\mu$. With this correspondence, the verifier can verify a

vector equation, e.g., $\mathbf{a} + \mathbf{b} \circ \mathbf{c} = \mathbf{0}$, where "$\circ$" is the entrywise product between vectors. This vector equation is equivalent to the polynomial equation $f_{\mathbf{a}}(X) + f_{\mathbf{b}}(X) \cdot f_{\mathbf{c}}(X) = q(X) \cdot Z(X)$ for some quotient polynomial $q(X)$, where $Z(X) := \prod_{x \in \mathbb{D}}(X - x)$ is the vanishing polynomial over $\mathbb{D}$. Apart from the above vector equations, the verifiers can also check more complex relations such as:

- *The permutation relation* [GWC19], which states that two vectors are permutations of each other. For example, $\mathbf{a} = (1, 2, 2, 3)^\mathsf{T}$ and $\mathbf{b} = (2, 3, 1, 2)^\mathsf{T}$, denoted by $\mathbf{a} \sim \mathbf{b}$ for convenience.
- *The lookup relation* [GW20], which states that all the elements in one vector are contained in the other vector. For example, $\mathbf{a} = (1, 2, 2, 3)^\mathsf{T}$ and $\mathbf{b} = (1, 2, 3, 4)^\mathsf{T}$, denoted by $\mathbf{a} \subset \mathbf{b}$ for convenience.

The protocols for checking these relations are referred to as *permutation arguments* [GWC19] and *lookup arguments* [GW20], respectively. These arguments can be extended to apply to tuples of vectors, also referred to as *tables*. For example, $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \sim (\mathbf{a}', \mathbf{b}', \mathbf{c}') \in \mathbb{F}^{N \times 3}$ means these two tables have the same multiset of rows in potentially different orders, i.e., the multisets of tuples $\{(\mathbf{a}_{[i]}, \mathbf{b}_{[i]}, \mathbf{c}_{[i]})\}_{i=1}^{N}$ and $\{(\mathbf{a}'_{[i]}, \mathbf{b}'_{[i]}, \mathbf{c}'_{[i]})\}_{i=1}^{N}$ are equal to each other.

A PIOP can be compiled into a SNARK by standard techniques [BFS20], i.e., instantiating the polynomial oracles with cryptographic constructions such as polynomial commitment scheme (PCS) [KZG10]. The performance of the resulting SNARK is determined by that of the PIOP in various aspects.

Next, we present a high-level overview of ZKVMs and our systemization over the current state of MCCs.

Workflow of ZKVM. On input $\mathbf{x}$, a machine $\mathbf{M}$ executes for T steps and produces an output $\mathbf{y} := \mathbf{M}(\mathbf{x})$. For simplicity, throughout this work, we assume $T = N$, the size of interpolation domain $\mathbb{D}$[4]. Assume the machine has m field elements as the internal state. The *execution trace* of the machine is a table $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)}) \in \mathbb{F}^{T \times m}$ where the t-th row represents the state values at step t. Given the pair $\mathbf{x}$ and $\mathbf{y}$, proving that $\mathbf{y} = \mathbf{M}(\mathbf{x})$ is equivalent to proving the existence of an execution trace that is consistent with $\mathbf{x}$, $\mathbf{y}$ and the architecture of the machine. Since we are in the PIOP model, the prover, after executing the program, may directly send the execution trace to the verifier, without exhausting the verifier's storage and computational resources.

After sending the execution trace, the prover tries to convince the verifier that these vectors are consistent with the machine. In practice, the machine is broken down into smaller components, such as instruction fetching, decoding, arithmetic logic unit, and memory access. Each component's consistency is formalized using building blocks including vector equations, permutation relation, and lookup relation. We focus on the memory component, whose checking protocol is the most challenging to design for reasons that will be explained later.

[4] For example, the machine can be designed such that executing the last instruction (e.g., a STOP instruction) does not change the state of the machine, so that this instruction can be repeated as many times as needed until T reaches N.

Memory consistency check. Although the memory is typically modeled as a dictionary that maps from the address space to the value space, we describe its functionality via an alternative approach that matches our definition of memory consistency. This model involves three variables $\mathbf{op}_{[t]}, \mathbf{addr}_{[t]}, \mathbf{val}_{[t]}$ representing the *operator*, the *address*, and the *value*, respectively. For step t from 1 to N, the machine computes two state variables $\mathbf{op}_{[t]}$ and $\mathbf{addr}_{[t]}$ from the current instruction or other internal states of the machine. The variable $\mathbf{op}_{[t]}$ is either Read or Write, which are constant field elements specified by the machine. The machine then computes another variable $\mathbf{val}_{[t]}$ as follows:

- If $\mathbf{op}_{[t]} = \mathsf{Read}$, find the maximal $t' < t$ such that $\mathbf{addr}_{[t']} = \mathbf{addr}_{[t]}$, then set $\mathbf{val}_{[t]} = \mathbf{val}_{[t']}$. If no satisfying t' is found, set $\mathbf{val}_{[t]}$ arbitrarily.
- If $\mathbf{op}_{[t]} = \mathsf{Write}$, compute $\mathbf{val}_{[t]}$ from the other internal states of the machine by a specified procedure. However, to decouple the memory from this potentially complex procedure, we consider $\mathbf{val}_{[t]}$ to be an arbitrary value set by the machine executor.

Given the traces of these variables, namely the vectors $\mathbf{op}, \mathbf{addr}, \mathbf{val}$ of size N, the memory consistency check should ensure that they represent a consistent execution trace of the memory, where the consistency can be informally defined as follows: for every t, $\mathbf{val}_{[t]}$ is honestly computed from $\mathbf{op}_{[t]}$, $\mathbf{addr}_{[1]}, \cdots, \mathbf{addr}_{[t]}$, $\mathbf{val}_{[1]}, \cdots, \mathbf{val}_{[t-1]}$ by the above procedure.

Among all the components of the machine, memory is the only one that is *history-dependent*[5]: the next state depends on the entire history of the machine states, instead of only on the previous state. This characteristic complicates the consistency checking protocol for the following reason. Without history dependency, the consistency of the entire trace can be decomposed into a sequence of *local relations* between adjacent rows in the trace. These local relations can be captured by one or more low-degree multivariate polynomials that verify the transition between adjacent states. The number of variables in these polynomials has only the size of two states. This allows efficient verification using the vector equation checks. However, if the current state of the machine depends on the entire history, capturing the relations between dependent states would require multivariate polynomials with $O(N)$ variables, which renders the vector equation check infeasible.

The sorting paradigm. The reason for memory consistency being history-dependent is that different memory addresses are accessed in an interleaved manner. This observation inspires the idea of sorting the memory execution trace by address. After sorting the table $(\mathbf{op}, \mathbf{addr}, \mathbf{val})$ into $(\widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}})$ using the column $\widetilde{\mathbf{addr}}$ as the key, accesses to identical addresses are grouped together, and as a result, $\widetilde{\mathbf{val}}_{[t]}$ depends only on $\widetilde{\mathbf{op}}_{[t]}, \widetilde{\mathbf{addr}}_{[t]}, \widetilde{\mathbf{addr}}_{[t-1]}$ and $\widetilde{\mathbf{val}}_{[t-1]}$.

[5] Except for some special-purpose components designed particularly for ZKVMs, e.g., the hash table in Triton VM and some builtins in Cairo, that are not in a traditional CPU architecture. The stack in stack-based architectures like EVM can be considered as a simpler version of random-access memory, whose consistency checks are similar to those for memories.

The above idea is formalized as the *sorting paradigm*, which captures the MCCs in all ZKVMs as of 2023, the time of this writing. This paradigm is described by the following procedure:

1. The prover sorts $\mathbf{op}, \mathbf{addr}, \mathbf{val}$ by the entries in $\mathbf{addr}$ and obtains $\widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$. These sorted vectors are sent to the verifier.
2. The verifier confirms that $\mathbf{op}, \mathbf{addr}, \mathbf{val}$ and $\widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$ are permutations of each other.
3. The verifier ensures the expected local property of the sorted memory trace using one or more vector equations.
4. The final step varies in different ZKVMs, but all involve proving that $\widetilde{\mathbf{op}}$, $\widetilde{\mathbf{addr}}$, $\widetilde{\mathbf{val}}$ is the sorting of $\mathbf{op}, \mathbf{addr}, \mathbf{val}$ by $\mathbf{addr}$.

In current ZKVMs, step 4—the sorting check—is accomplished using one of the following three protocols, each for a different memory model:

1. *Contiguous read-only memory.* This model requires that the values in the vector $\mathbf{addr}$ span a contiguous region in $\mathbb{F}$ and that $\mathbf{op}_{[t]}$ is always Read. With these requirements, $\widetilde{\mathbf{addr}}_{[t-1]} \leq \widetilde{\mathbf{addr}}_{[t]}$ is equivalent to $\widetilde{\mathbf{addr}}_{[t]} - \widetilde{\mathbf{addr}}_{[t-1]} \in \{0, 1\}$, which is captured by the vector equation $(\widetilde{\mathbf{addr}}_{[t]} - \widetilde{\mathbf{addr}}_{[t-1]}) \cdot (\widetilde{\mathbf{addr}}_{[t]} - \widetilde{\mathbf{addr}}_{[t-1]} - 1) = 0$.
2. *Read-write memory with 32k-bit addresses.* For this memory model, the constraint $\widetilde{\mathbf{addr}}_{[t-1]} \leq \widetilde{\mathbf{addr}}_{[t]}$ is checked by a $32k$-bit range check over the vector $\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}$, where $\widetilde{\mathbf{addr}}^{\leftarrow 1}$ is the cyclic left-shifting of $\widetilde{\mathbf{addr}}$ by one position.
3. *Read-write memory with full address space: $\mathbb{F}^c$ for $c \geq 1$.* Since the case $c > 1$ can be reduced to $c = 1$ by random linear combination, we proceed assuming that $c = 1$. The statement that $\widetilde{\mathbf{addr}}$ is sorted is proved by the *contiguity check*, designed by Triton VM [TV22], explained as follows. Given the vector $\widetilde{\mathbf{addr}}$, initialize the polynomial $f(X) = X - \widetilde{\mathbf{addr}}_{[1]}$, then for each t from 2 to N, if $\widetilde{\mathbf{addr}}_{[t]} \neq \widetilde{\mathbf{addr}}_{[t-1]}$, multiply $f(X)$ by $X - \widetilde{\mathbf{addr}}_{[t]}$, otherwise do nothing. Obviously, the vector $\widetilde{\mathbf{addr}}$ is *contiguous* (which means repeated elements fall in contiguous regions; this is equivalent to $\widetilde{\mathbf{addr}}$ being sorted by some custom order over $\mathbb{F}$) if and only if no monomial $X - \widetilde{\mathbf{addr}}_{[t]}$ is multiplied to $f(X)$ more than once, if and only if $f(X)$ has no multiple roots. The prover sends the polynomial $f(X)$ to the verifier, who checks that $f(X)$ is correctly computed and that $\gcd(f(X), Df(X)) = 1$, where $Df(X)$ is the formal derivative of $f(X)$.

For read-write memories, there is an additional issue: as $\mathbf{addr}$ may contain duplicate elements, multiple permutations exist for sorting the memory execution trace. However, the sorting technique works only if the permutation is the unique one that preserves the order of rows with identical addresses as in the original table. This unique permutation is referred to as the *canonical sorting*. To ensure that the sorting is canonical, the following modifications should be applied to

the second and third protocols: the memory execution trace is sorted together with the incrementing vector $\mathbf{incs} = (1, 2, \cdots, N)$. The verifier then ensures that if $\widetilde{\mathbf{addr}}_{[t]} = \widetilde{\mathbf{addr}}_{[t-1]}$, the difference $\widetilde{\mathbf{incs}}_{[t]} - \widetilde{\mathbf{incs}}_{[t-1]}$ must be in the range $1, 2, \cdots, N$, by a lookup relation.

Our improvement: address cycle method. Note that in the sorting paradigm, the prover sends at least four vectors to the verifier, each for a column in the sorted memory trace. This is somewhat wasteful because, compared to the unsorted memory trace, the additional information conveyed in these four vectors is no more than a single permutation. We propose an alternative way that saves these costs. Instead of reordering the memory execution trace, we *redefine* the meaning of *adjacency* such that identical addresses become adjacent under this new definition. This insight is extracted from Arya [BCG+18], a zero-knowledge protocol for TinyRAM [BCG+13]. We name this technique the *address cycle method*.

This method involves defining a permutation σ over the index set $\{1, \cdots, N\}$. The permutation σ maps each index t to the previous time when $\mathbf{addr}_{[t]}$ was accessed, i.e. $\sigma(t) = \max\{j < t | \mathbf{addr}_{[j]} = \mathbf{addr}_{[t]}\}$, if such maximal value is well-defined. If otherwise, this maximal value does not exist, i.e., $\mathbf{addr}_{[t]}$ is accessed for the first time, $\sigma(t)$ maps it to the last time the same address was accessed, i.e., in this case, $\sigma(t) = \max\{j \leq N | \mathbf{addr}_{[j]} = \mathbf{addr}_{[t]}\}$. This way, for each distinct address, all the positions where it appears are linked into a cycle by σ. Obviously, $\mathbf{addr}$ is invariant under the permutation σ.

Now we observe the behavior of $\mathbf{val}$ as it is permuted by σ. By definition, if the memory trace is consistent, then $\mathbf{val}$ is *almost* invariant under the permutation. Specifically, $\mathbf{val}_{[\sigma(t)]} = \mathbf{val}_{[t]}$ for every t except for those with $\mathbf{op}_{[t]} = \mathsf{Write}$ or $\mathbf{addr}_{[t]}$ is accessed the first time.

It turns out that the aforementioned behaviors of $\mathbf{addr}$ and $\mathbf{val}$ when permuted by σ suffice to guarantee memory consistency, as shown in Theorem 7. To summarize, the MCC can be accomplished by proving the existence of a permutation σ with the following properties:

1. There exists a vector $\mathbf{first} \in \{0, 1\}^N$ such that $t > \sigma(t)$ for every t except for those t where $\mathbf{first}_{[t]} = 1$.
2. $\mathbf{addr}$ is invariant under σ and $\mathbf{val}$ is almost invariant: $\mathbf{val}_{[\sigma(t)]} = \mathbf{val}_{[t]}$ for every t except for those with $\mathbf{op}_{[t]} = \mathsf{Write}$ or $\mathbf{first}_{[t]} = 1$.
3. For every address a, all the positions where a appears in $\mathbf{addr}$ fall in the same cycle of σ.

The first two properties are simple to check, as they are captured by vector equations, permutation relations, and lookup relations. See Section 5.1 for details. Checking the last property is the most challenging part of this method. We proved in Lemma 5 (Section 5.1) that this property can be ensured by showing that the elements $\{\mathbf{addr}_{[t]} | 1 \leq t \leq N, \mathbf{first}_{[t]} = 1\}$ are distinct. Therefore, our address cycle method reduces the MCC problem to the *distinctness check* problem, which is the key component of different constructions. Here we present two constructions for this component.

1. *Permem.* We note that the contiguity check of Triton VM can be generalized into a distinctness check that does not pose any restriction on the address space. This distinctness check protocol produces a new MCC with full address space, i.e., $\mathcal{A} = \mathbb{F}^c$ for any $c \geq 1$. We name this new MCC Permem.
2. *AryaMem.* For a clear comparison between Permem and Arya, which is originally described in the ILC model [BCG+17], we adapt the memory component of Arya into a PIOP, called AryaMem, which is optimized with the standard PIOP techniques. We remark that the memory component of the original Arya does not strictly follow the pattern of our address cycle method, whereas AryaMem is adapted to follow this method strictly. In particular, in Arya, the last property of σ is not verified via distinctness check, but instead by a protocol called blookup, which is constructed with two lookup arguments. We replace this blookup protocol with a distinctness check, which is much simpler thanks to Arya's limited memory address space (the set $\{1, 2, \cdots, M\}$ for $M \approx N$). Specifically, to prove that **addr** satisfies the distinctness condition, it suffices to show that there exists a vector that is both a permutation of $(1, 2, \cdots, M)$ and is identical to **addr** in places where $\mathbf{first}_{[t]} = 1$. This can be implemented using a single permutation argument.

Lookup argument. The lookup argument is an influential building block in most MCCs. We construct a new lookup argument, named gcq for *grand-sum version of* cq [EFG22], based on the *logarithmic derivative* technique. The insight of logarithmic derivative is that every element in $A = \{a_1, \cdots, a_n\}$ appears in $B = \{b_1, \cdots, b_n\}$ if and only if $\sum \left(\frac{1}{X - a_i} - \frac{m_i}{X - b_i} \right) = 0$, where $m_i \geq 0$ is the number of times b_i appears in A. Proving that this sum is zero is the core of the logarithmic derivative technique. In cq, this is accomplished by the popular univariate sumcheck protocol [BCR+19], which is also extensively used in building general-purpose SNARKs [BCR+19, CHM+20, COS20, RZ21].

However, we notice that a simpler technique, called *grand-sum check*, has multiple benefits which are, somewhat surprisingly, undervalued in the literature[6]. Compared to the univariate sumcheck, the grand-sum check has the following three advantages: (1) it works in both monomial basis and Lagrange basis; (2) it does not require an individual degree bound of the PIOP; and (3) most importantly, the grand-sum check contributes (almost) no additional cost at all, in terms of the number of online polynomials and evaluation queries, which is explained as follows. Note that: (1) for any vector $\mathbf{u}$, the vector $\mathbf{u}^{\leftarrow 1} - \mathbf{u} + s \cdot \mathbf{e}_1$ is guaranteed to have sum s, where $\mathbf{u}^{\leftarrow 1}$ is $\mathbf{u}$ circularly left-shifted by one position; and (2) for any vector $\mathbf{v}$, we can always find $\tilde{v}$ such that $\mathbf{v} = \widetilde{\mathbf{v}}^{\leftarrow 1} - \widetilde{\mathbf{v}} + s \cdot \mathbf{e}_1$, e.g., $\widetilde{\mathbf{v}} := (v_1, v_1 + v_2, \cdots, \sum v_i)$, the grand-sum vector of $\mathbf{v}$. Therefore, the prover could have directly sent $\widetilde{\mathbf{v}}$, without sending $\mathbf{v}$ in the first place, and the verifier simulates the polynomial oracle for $\mathbf{v}$ using that of $\widetilde{\mathbf{v}}$ wherever $\mathbf{v}$ appears.

[6] It is indeed used in some works, but very rarely, e.g., in Flookup [GK22]. It is used only in a small component of Flookup, where univariate sumcheck is unusable.

Although the grand-sum check has some disadvantages compared to the univariate check, which may partially explain the rare usage of grand-sum check, these disadvantages are avoided in the context of MCCs:

- Grand-sum check involves shifting the vector $\tilde{\mathbf{v}}$ by one position, which requires simulating the polynomial oracle $f_{\tilde{\mathbf{v}}}(\omega X)$, causing one additional distinct evaluation point ωz in the PIOP. However, this is not a problem in MCC, as ωz is already required by the permutation check.
- It is unclear how to exploit the KZG-specific optimizations in grand-sum check, which is interesting for future research. In particular, cq exploits these techniques to allow the prover cost to depend only on the size N of the execution trace and independent of the lookup table size. However, in Permem, the lookup table also has size N, rendering the table-size-independence unnecessary.
- The grand-sum check does not look intuitive when the polynomial oracle $f_{\mathbf{v}}(X)$ has a degree greater than N, in which case the simulation additionally involves the quotient polynomial $q(X)$. However, the grand-sum check still saves one polynomial oracle compared to the univariate sumcheck in this scenario. Moreover, in cq or gcq, the target polynomials of the sumcheck have degrees bounded by the domain size, so the quotient polynomials are unnecessary.

For the above reasons, grand-sum check fits perfectly in MCC, especially in our Permem. Table 2 shows that the MCCs use three fewer online polynomials and two fewer evaluation queries by replacing cq (that uses univariate sum-check) with gcq (that uses grand-sum check).

Zero knowledge. We will not address the zero-knowledge aspect in this work for the following two reasons. First, despite the "ZK" in the name, ZKVMs are more valued for their succinctness than their zero-knowledge property. This preference is evident from the fact that currently ZKVMs are mainly used in zkRollups [zkS22, Azt22, Pol22, Loo22], which prioritize scalability over privacy. Second, zero-knowledge can be achieved as an added property in SNARKs using standard techniques such as adding a masking polynomial $\delta(X) \cdot Z(X)$ to the interpolated polynomials, where $\delta(X)$ is a uniformly random small polynomial. It is unnecessary to repeat these standard techniques, so we omit them for clarity and simplicity.

1.3 Related Works

Although there is a lack of literature discussing the recent developments in ZKVMs [zkS22, TV22, Pol22, Scr22, Mid22, Ris22, GPR21], these ZKVMs are the result of more than ten years of progress in the field of verifiable computations (VC) [GGP10]. VC constructions can be categorized based on their model of computation, primarily the circuit model and the RAM model. Circuit-based VCs, particularly SNARKs, have gained greater attention and undergone active research since 2018 [PHGR13, Gro16, CHM+20, GWC19, Set20, XZZ+19,

BDFG20,BBB+18,Eag22,SL20,BFS20,ZSZ+22,SZ22,ZXZS20,BFH+20,COS20, BCR+19,WTS+18]. Nonetheless, they have a significant drawback as circuits are inconvenient to program for, especially when branching and loops are involved.

Although RAM-based VCs potentially support more intuitive programming interfaces like high-level programming languages, they are more inefficient than the circuit-based ones, with MCC being a major bottleneck. The Merkle-tree-based memory check [BFR+13] is barely practical, and outperformed by the sorting technique, which is initially based on routing networks [BCGT13, BCG+13, BCTV13,BBC+17,BBHR18] and later adopts the more efficient permutation argument developed in circuit-based SNARKs [GWC19], as in [ZGK+18,BCG+18]. Many works only support memory space as small as $\{1, \cdots, M\}$ for $M = O(T)$, and those supporting 32-bit memory addresses tend to be quite slow.

The recent rapid development of ZKVMs has been largely aided by the introduction of lookup arguments [GW20,ZBK+22,PK22,GK22,ZGK+22,EFG22, Hab22, CBBZ22, SLST23], which have significantly boosted the efficiency of 32- and 64-bit MCCs. However, 256-bit memory checks remain very expensive. Triton VM [TV22] mitigates this issue by its MCC with full address space, which is sufficiently large to cover the functionality of 256-bit memory. Our new protocol, Permem, further reduces the number of online polynomials of Triton VM.

Recent lookup arguments based on logarithmic derivatives [Hab22, EFG22, SLST23] are a promising new approach, offering both high performance and appealing properties such as homomorphic additions. Our new lookup argument gcq improves the state-of-the-art construction cq, costing fewer online polynomials and evaluation queries.

2 Preliminaries

Let λ be the security parameter. For $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, 2, \cdots, n\}$. For $i \leq j$, $[i..j]$ denotes $\{i, \cdots, j\}$. Throughout the paper, we use a unique finite field $\mathbb{F} = \mathbb{F}_p$ where p is a prime of $O(\lambda)$ bits. When the context is clear, we use integers and field elements interchangeably, so the sets $[n], [i..j]$ may also represent the corresponding $\mathbb{F}$ elements after reducing modulo p. For algorithm A, $\mathsf{A} \to c$ means the algorithm outputs c.

An *indexed relation* $\mathcal{R}$ is a set of triples $(\mathtt{i}, \mathtt{x}, \mathtt{w})$, where $\mathtt{i}$ is called the index, $\mathtt{x}$ is the instance, and $\mathtt{w}$ is the witness. The language induced from $\mathcal{R}$ is $\mathcal{L}(\mathcal{R}) := \{(\mathtt{i}, \mathtt{x}) : \exists\ \mathtt{w}, s.t.\ (\mathtt{i}, \mathtt{x}, \mathtt{w}) \in \mathcal{R}\}$.

2.1 Vectors and Polynomials

A vector of length N over $\mathbb{F}$ is denoted by $\mathbf{v} \in \mathbb{F}^N$. The length of $\mathbf{v}$ is $|\mathbf{v}|$. The i-th entry of the vector $\mathbf{v}$ is denoted by $\mathbf{v}_{[i]}$. The subvector of $\mathbf{v}$ from index i to j is denoted by $\mathbf{v}_{[i..j]}$. Let $\mathsf{Elems}(\mathbf{v})$ be the set of *distinct* elements in $\mathbf{v}$, and $\mathsf{MultiElems}(\mathbf{v})$ be the *multiset* of the elements in $\mathbf{v}$. We write $a \in \mathbf{v}$ if $a \in \mathsf{Elems}(\mathbf{v})$ and $\mathbf{u} \subset \mathbf{v}$ if $\mathsf{Elems}(\mathbf{u}) \subseteq \mathsf{Elems}(\mathbf{v})$. We say $\mathbf{u}$ and $\mathbf{v}$ are permutations of each other if $\mathsf{MultiElems}(\mathbf{u}) = \mathsf{MultiElems}(\mathbf{v})$. For permutation σ over $[N]$ and $\mathbf{v} \in \mathbb{F}^N$,

define $\sigma(\mathbf{v}) := (\mathbf{v}_{[\sigma(t)]})_{t=1}^{N}$. For vectors $\mathbf{u}, \mathbf{v}$ with $|\mathbf{u}| = |\mathbf{v}|$, $\mathbf{u} \circ \mathbf{v}$ is their *Hadamard product* (entry-wise product). $\mathbf{u} \| \mathbf{v}$ is the concatenation of two vectors. $\mathbf{v}^{\leftarrow k} := \mathbf{v}_{[k+1..N]} \| \mathbf{v}_{[1..k]}$ is the circular shift of $\mathbf{v}$ by k positions to the left or $-k$ positions to the right if $k < 0$. Let $\mathbf{v}_1, \cdots, \mathbf{v}_c \in \mathbb{F}^N$, then the tuple $(\mathbf{v}_1, \cdots, \mathbf{v}_c)$ is a *table* with N rows and c columns. The notations for tables, including $\sigma(\mathbf{v}_1, \cdots, \mathbf{v}_c)$, $(a_1, \cdots, a_c) \in (\mathbf{v}_1, \cdots, \mathbf{v}_c)$ and $(\mathbf{u}_1, \cdots, \mathbf{u}_c) \subset (\mathbf{v}_1, \cdots, \mathbf{v}_c)$, are defined similarly as for vectors. Particularly, let $\mathsf{Rows}(\mathbf{v}_1, \cdots, \mathbf{v}_c)$ denote the set of *distinct* tuples $\{(a_1, \cdots, a_c) \in (\mathbf{v}_1, \cdots, \mathbf{v}_c)\}$, and $\mathsf{MultiRows}(\mathbf{v}_1, \cdots, \mathbf{v}_c)$ be the *multiset* of the tuples $\{(a_1, \cdots, a_c) \in (\mathbf{v}_1, \cdots, \mathbf{v}_c)\}$.

For any constant $C \in \mathbb{F}$, let $\mathbf{C}^N$ be a shorthand of the size-N vector consisting of only C. In particular, $\mathbf{0}^N, \mathbf{1}^N$ are the vectors consisting of N zeros or ones. Let $\mathbf{e}_i^N := \mathbf{0}^{i-1} \| 1 \| \mathbf{0}^{N-i}$ be the i-th unit vector. The superscript may be omitted if the length is clear from the context.

Let $f(X) \in \mathbb{F}[X]$ denote a polynomial over $\mathbb{F}$. We call a subset $\mathbb{D} \subset \mathbb{F}$ a *domain*. Given a domain $\mathbb{D}$ of size N where the elements are ordered by $a_1, \cdots, a_N$, let $f(\mathbb{D})$ be the vector $(f(a_1), \cdots, f(a_N))$. Given a vector $\mathbf{v}$ of size $|\mathbb{D}|$, we can find at least one polynomial $f_{\mathbf{v}}(X)$ such that $f_{\mathbf{v}}(\mathbb{D}) = \mathbf{v}$, and call it an interpolation of $\mathbf{v}$ over $\mathbb{D}$. We usually take $\mathbb{D} = \{1, \omega, \cdots, \omega^{N-1}\}$ where ω is the N-th root of unity. In this setting, the polynomial interpolation of $\mathbf{e}_i$ is $f_{\mathbf{e}_i}(X) = \frac{\omega^{i-1} \cdot (X^N - 1)}{N \cdot (X - \omega^{i-1})}$ and can be evaluated by $O(\log(N))$ field operations. The identity polynomial $f_{\mathbf{id}}(X) := X$ corresponds to the vector $\mathbf{id} := (1, \omega, \cdots, \omega^{N-1})$.

2.2 Interactive Proof System

An interactive proof system [GMR85] is a protocol between two parties, the prover P and the verifier V. The prover tries to convince the verifier of a statement $(\mathtt{i}, \mathtt{x}) \in \mathcal{L}$. In this work, we consider arguments of knowledge with preprocessing. That is, before the protocol starts, the index $\mathtt{i}$ is preprocessed offline by the indexer I, which produces helpful information for both the prover and the verifier, such that the verifier does not need to learn the entire index, but only the preprocessed information.

Definition 1 (Preprocessing Proof System). *A preprocessing proof system for indexed relation $\mathcal{R}$ is a triple of PPT algorithms $(\mathsf{I}, \mathsf{P}, \mathsf{V})$. For any triple $(\mathtt{i}, \mathtt{x}, \mathtt{w})$, the indexer I takes as input $\mathtt{i}$, and outputs $\mathtt{i}_P$ and $\mathtt{i}_V$. The prover P takes as input $\mathtt{i}_P, \mathtt{x}, \mathtt{w}$, and the verifier V takes as input $\mathtt{i}_V, \mathtt{x}$, and they interact with each other. At the end of the interaction, the verifier outputs $b \in \{0, 1\}$, indicating if it accepts $(b = 1)$ or rejects $(b = 0)$. Denote this procedure by $\langle \mathsf{I}(\mathtt{i}), \mathsf{P}(\mathtt{x}, \mathtt{w}), \mathsf{V}(\mathtt{x}) \rangle \to b$.*

The protocols should satisfy the following properties:

- *Completeness. For any $(\mathtt{i}, \mathtt{x}, \mathtt{w}) \in \mathcal{R}$,*

$$\Pr[b = 0 | \langle \mathsf{I}(\mathtt{i}), \mathsf{P}(\mathtt{x}, \mathtt{w}), \mathsf{V}(\mathtt{x}) \rangle \to b] \leq e_c$$

  where e_c is a negligible value called the completeness error. If e_c is zero, then we say this protocol has perfect completeness.

– *Soundness. For any* $(\mathtt{i}, \mathtt{x}) \notin \mathcal{L}(\mathcal{R})$ *and unbounded algorithm* P^**,*

$$\Pr[b = 1 | \langle \mathsf{I}(\mathtt{i}), \mathsf{P}^*, \mathsf{V}(\mathtt{x}) \rangle \to b] \leq e_s$$

where e_s *is a negligible value called the soundness error. If* e_s *is zero, then we say this protocol has perfect soundness.*

Moreover, a proof system may also enjoy other properties:

– *public coin*, if all the verifier messages are fresh random coins;
– *statistical honest-verifier zero-knowledge*, if there exists a simulator S such that for any $(\mathtt{i}, \mathtt{x}, \mathtt{w}) \in \mathcal{R}$ and any unbounded distinguisher D

$$|\Pr[\mathsf{D}(\mathsf{View}(\mathtt{i}, \mathtt{x}, \mathtt{w}))] - \Pr[\mathsf{D}(\mathsf{S}(\mathtt{i}, \mathtt{x}))]| = \mathsf{negl}$$

where $\mathsf{View}(\mathtt{i}, \mathtt{x}, \mathtt{w})$ is the view of the verifier during the execution.
– *succinctness*, if the verification time and/or the online communication cost is sublinear with respect to the size of the witness;
– *proof (resp. argument* [7]*) of knowledge*, if for any $\mathtt{i}$ and (resp. PPT) prover P^*, there exists a PPT extractor E, which has access to the same input and random tape of P^*, such that for any efficient adversary A

$$\Pr[b = 1 \wedge (\mathtt{i}, \mathtt{x}, \mathtt{w}) \notin \mathcal{R} | \mathsf{A} \to \mathtt{x}, \langle \mathsf{I}(\mathtt{i}), \mathsf{P}^*, \mathsf{V}(\mathtt{x}) \rangle \to b, \mathsf{E}^{\mathsf{P}^*}(\mathtt{i}) \to \mathtt{w}] \leq e_s.$$

A public coin argument of knowledge can be transformed into a non-interactive argument of knowledge via the Fiat-Shamir heuristic [FS86]. If the protocol is also succinct (and zero-knowledge), then the resulting non-interactive scheme is called a SNARK (or zkSNARK).

2.3 Polynomial IOP

A Polynomial Interactive Oracle Proof (PIOP) [BFS20] is a type of interactive proof system where the prover's messages sent to the verifier are restricted to be polynomial oracles or field elements. PIOPs can be converted into conventional interactive proofs through cryptographic compilers [BFS20] based on polynomial comitments [KZG10, BFS20].

Definition 2 (Polynomial IOP). *Given a finite field* $\mathbb{F}$*, a preprocessing PIOP of degree bound* D *for indexed relation* $\mathcal{R}$ *is a triple of PPT algorithms* $(\mathsf{I}, \mathsf{P}, \mathsf{V})$ *such that:*

– $(\mathsf{I}, \mathsf{P}, \mathsf{V})$ *is a public coin preprocessing interactive proof system for* $\mathcal{R}$ *with completeness error* e_c *and soundness error* e_s*;*
– I, P *sends polynomials* $f_i(X) \in \mathbb{F}[X]$ *of degree at most* D *to* V*;*
– V *sends challenges* $\alpha_k \in \mathbb{F}$ *to* P*;*

[7] If soundness holds only against a polynomial-bounded prover, then we say this protocol is an *argument*.

- V *is an oracle machine with access to a list of oracles, which contains one oracle for each polynomial received from* I *and* P*;*
- *on receiving a query* $z \in \mathbb{F}$*, the oracle for* $f_i(X)$ *responds with* $f_i(z)$*.*

Having oracle access to $f(X)$ gives the verifier the ability to evaluate $f(X)$ at arbitrary point z without learning the content of $f(X)$ itself. Moreover, given oracle access to $f(X)$ and $g(X)$, the verifier also gains the ability to evaluate other polynomials, e.g., $a \cdot f(X) + b \cdot g(X)$ and $f(c \cdot X)$. We say the verifier *simulates the oracle access* to these polynomials. The verifier may also simulate the oracle access to polynomials that admit fast evaluation. For example, the constant polynomial $f(X) = C$, the identity polynomial $f_{\mathsf{id}}(X) = X$, and the polynomial $\frac{\omega^{i-1} \cdot (X^N - 1)}{N \cdot (X - \omega^{i-1})}$, i.e., the polynomial obtained from interpolating $\mathbf{e}_i$ over $\mathbb{D}$.

We adopt the following notations for describing a PIOP:

- "I sends $f(X)$" means the indexer sends $f(X)$ to the prover and the oracle access of $f(X)$ to the verifier, and "P sends $f(X)$" means the prover sends the oracle access of $f(X)$ to the verifier.
- "V samples $\alpha \xleftarrow{\$} \mathbb{F}$" implies that V sends a uniformly random α to P.
- "V checks $f(z) \cdot g(z) = h(z)$" (or similar equations) means the verifier queries the oracles for $f(X), g(X), h(X)$ at point z, receives y_f, y_g, y_h respectively, and checks if $y_f \cdot y_g = y_h$.

2.4 PIOP for Vector Languages

Exploiting the polynomial interpolation, we may describe a PIOP as if the parties are communicating with vectors instead of polynomials. We adopt the following change of notations for ease of description:

- We say "I sends $\mathbf{v}$" or "P sends $\mathbf{v}$" in place of "I sends $f_{\mathbf{v}}(X)$" or "P sends $f_{\mathbf{v}}(X)$", where $f_{\mathbf{v}}(X)$ is an interpolation of $\mathbf{v}$ over $\mathbb{D}$.
- Vector expressions stand for polynomials: $\mathbf{u} \circ \mathbf{v}$ for $f_{\mathbf{u}}(X) \cdot f_{\mathbf{v}}(X)$, $a \cdot \mathbf{u} + b \cdot \mathbf{v}$ for $a \cdot f_{\mathbf{u}}(X) + b \cdot f_{\mathbf{v}}(X)$, and $\mathbf{v}^{\leftarrow k}$ for $f_{\mathbf{v}}(\omega^k X)$.
- We say "V checks $\mathbf{u} + \mathbf{v} \circ \mathbf{w}^{\leftarrow k} = \mathbf{0}$" (or other vector equations) when the verifier samples $z \in \mathbb{F}$ uniformly, the prover sends $q(X) = \frac{f_{\mathbf{u}}(X) + f_{\mathbf{v}}(X) \cdot f_{\mathbf{w}}(\omega^k X)}{Z(X)}$, and the verifier checks $f_{\mathbf{u}}(z) + f_{\mathbf{v}}(z) \cdot f_{\mathbf{w}}(\omega^k z) = q(z) \cdot Z(z)$, where $Z(X) = X^N - 1$ is the vanishing polynomial over $\mathbb{D}$. When a protocol contains more than one such checks, say $F_i = \mathbf{0}$ for i from 1 to m, where F_i is a vector expression, the verifier samples $\beta \in \mathbb{F}$ and checks $\sum_{i=1}^{m} \beta^{i-1} F_i = \mathbf{0}$ instead. By Schwartz-Zippel Lemma, this check incurs a soundness error of $(d + m - 1)/|\mathbb{F}|$, where d is the degree of the polynomial divided by $Z(X)$.

Although a PIOP may involve polynomial oracles in its execution, the parties of a PIOP cannot take polynomial oracles as inputs, because the relation $\mathcal{R}$ is not well-defined when oracles are involved. However, in constructing a PIOP, we frequently encounter situations where it would be convenient to design a

building-block *subprotocol* for proving statements that involve polynomial oracles, e.g., "given the oracle access to $f(X)$ that was previously sent from the prover, $f(X)$ satisfies certain property". In fact, all the PIOPs presented in this work are such subprotocols, including the MCC, which works as a building block of the entire ZKVM protocol. To formally define such subprotocols in the PIOP model, we introduce the notion *vector languages*.

Definition 3 (Vector Language). *Let m, N be positive integers. A vector language $\mathcal{R}$ of width m and length N is a set of vector tuples, where each tuple contains m vectors and each vector has length N.*

Definition 4 (PIOP for Vector Language). *Let $\mathcal{R}$ be a vector language of width m and length N. We say a PIOP $\Pi = (\mathsf{I}, \mathsf{P}, \mathsf{V})$ is a PIOP for $\mathcal{R}$ if I takes inputs the description of $\mathcal{R}$, and for any vectors $\mathbf{v}_1, \cdots, \mathbf{v}_m$, P takes inputs $\mathbf{v}_1, \cdots, \mathbf{v}_m$, and V has oracle access to $f_{\mathbf{v}_1}(X), \cdots, f_{\mathbf{v}_m}(X)$ at the start. The PIOP is complete if for any tuple $(\mathbf{v}_1, \cdots, \mathbf{v}_m) \in \mathcal{R}$, V accepts except with probability at most e_c. The PIOP is sound if for any $(\mathbf{v}_1, \cdots, \mathbf{v}_m) \notin \mathcal{R}$, V accepts with probability no more than e_s.*

2.5 Building Blocks

MCCs have two key building blocks: the permutation argument and the lookup argument. Given $\mathbf{u}_1, \cdots, \mathbf{u}_m$ and $\mathbf{v}_1, \cdots, \mathbf{v}_m$, the *permutation argument* [GWC19] (also referred to as the *multi-set check* [CBBZ22]) allows the verifier to check that the tables $(\mathbf{u}_1, \cdots, \mathbf{u}_m)$ and $(\mathbf{v}_1, \cdots, \mathbf{v}_m)$ have the same *multi-set* of rows, potentially in different orders. Formally, a permutation argument is a PIOP, referred to as Perm, for the vector language

$$\mathcal{R}_{\mathsf{Perm}} := \left\{ (\{\mathbf{u}_i \in \mathbb{F}^N\}_{i=1}^m, \{\mathbf{v}_i \in \mathbb{F}^N\}_{i=1}^m) \;\middle|\; \begin{array}{l} \mathsf{MultiRows}(\mathbf{u}_1, \cdots, \mathbf{u}_m) = \\ \mathsf{MultiRows}(\mathbf{v}_1, \cdots, \mathbf{v}_m) \end{array} \right\}$$

with completeness error $e_{c,\mathsf{Perm}}$ and soundness error $e_{s,\mathsf{Perm}}$. PLONK [GWC19] provides an example construction of the permutation argument. The idea of the PLONK construction is to prove that for given random values $\alpha_0, \cdots, \alpha_m$, the two grand products $\prod_i(\alpha_0+\alpha_1\cdot\mathbf{u}_{1,[i]}+\cdots+\alpha_m\cdot\mathbf{u}_{m,[i]})$ and $\prod_i(\alpha_0+\alpha_1\cdot\mathbf{v}_{1,[i]}+\cdots+\alpha_m\cdot\mathbf{v}_{m,[i]})$ are equal. We denote the vectors satisfying the permutation relation by $(\mathbf{u}_1, \cdots, \mathbf{u}_m) \sim (\mathbf{v}_1, \cdots, \mathbf{v}_m)$, and we say "$\mathsf{V}$ checks $(\mathbf{u}_1, \cdots, \mathbf{u}_m) \sim (\mathbf{v}_1, \cdots, \mathbf{v}_m)$" when the parties run the Perm protocol with inputs $\mathbf{u}_1, \cdots, \mathbf{u}_m, \mathbf{v}_1, \cdots, \mathbf{v}_m$.

The *lookup argument* [GW20] allows the verifier to confirm that the set of rows of the table $(\mathbf{u}_1, \cdots, \mathbf{u}_m)$ is contained in the set of rows of the table $(\mathbf{v}_1, \cdots, \mathbf{v}_m)$. In ZKVM design, it is often necessary to prove this relationship for only a selected subset of the rows in $(\mathbf{u}_1, \cdots, \mathbf{u}_m)$. To deal with this, we modify the traditional lookup relation by introducing a selector vector $\mathbf{b}$ with values of either 0 or 1. This selector vector is used to specify the positions of the rows to be selected. Formally, for any table $(\mathbf{u}_1, \cdots, \mathbf{u}_m)$ and vector $\mathbf{b} \in \{0,1\}^N$, let $\mathsf{Rows}_{\mathbf{b}}(\mathbf{u}_1, \cdots, \mathbf{u}_m)$ denote the set of tuples $\{(\mathbf{u}_{1,[j]}, \cdots, \mathbf{u}_{m,[j]}) | j \in [N], \mathbf{b}_{[j]} = 1\}$.

Then a lookup argument is a PIOP for the vector language

$$\mathcal{R}_{\mathsf{Lookup}} := \left\{ \begin{pmatrix} \{\mathbf{u}_i \in \mathbb{F}^N\}_{i=1}^m, \{\mathbf{v}_i \in \mathbb{F}^N\}_{i=1}^m, \\ \mathbf{b} \in \{0,1\}^N \end{pmatrix} \middle| \begin{array}{l} \mathsf{Rows}_{\mathbf{b}}(\mathbf{u}_1, \cdots, \mathbf{u}_m) \subseteq \\ \mathsf{Rows}(\mathbf{v}_1, \cdots, \mathbf{v}_m) \end{array} \right\}$$

with completeness error $e_{c,\mathsf{Lookup}}$ and soundness error $e_{s,\mathsf{Lookup}}$. We denote the vectors satisfying the lookup relation by $(\mathbf{u}_1, \cdots, \mathbf{u}_m) \subset_{\mathbf{b}} (\mathbf{v}_1, \cdots, \mathbf{v}_m)$, and we say "$\mathsf{V}$ checks $(\mathbf{u}_1, \cdots, \mathbf{u}_m) \subset_{\mathbf{b}} (\mathbf{v}_1, \cdots, \mathbf{v}_m)$" when the parties run the Lookup protocol with inputs $\mathbf{u}_1, \cdots, \mathbf{u}_m, \mathbf{v}_1, \cdots, \mathbf{v}_m, \mathbf{b}$. We omit $\mathbf{b}$ when $\mathbf{b} = \mathbf{1}$.

Although existing lookup argument constructions [GW20, ZBK+22, PK22, GK22, ZGK+22, EFG22, Hab22, CBBZ22, SLST23] do not involve the selector vector $\mathbf{b}$, they can be adapted to take $\mathbf{b}$ into consideration. We will present our construction in Section 5.2.

A widely used application of the lookup argument is the range check, particularly 32-bit range checks in ZKVMs. Formally, a 32-bit range check is a PIOP, referred to as $\mathsf{Range32}$, for the vector language

$$\mathcal{R}_{\mathsf{R32}} = \left\{ (\mathbf{v} \in \mathbb{F}^N, \mathbf{b} \in \{0,1\}^N) | \forall t \in [N], \mathbf{b}_{[t]} = 0 \vee \mathbf{v}_{[t]} \in [0..2^{32} - 1] \right\}$$

with completeness error $e_{c,\mathsf{Range32}}$ and soundness error $e_{s,\mathsf{Range32}}$.

3 The Memory Consistency Check Problem

We start from defining the problem of memory consistency check (MCC). We call a table $(\mathbf{op}, \mathbf{addr}, \mathbf{val}) \in \mathbb{F}^{N \times 3}$ a consistent memory trace if the value $\mathbf{val}_{[t]}$ for each row with $\mathbf{op}_{[t]} = \mathsf{Read}$ is equal to the value associated with the address $\mathbf{addr}_{[t]}$ the last time it was accessed. This concept is formalized in Definition 5, where we set $\mathsf{Read} = 0$ and $\mathsf{Write} = 1$ for simplicity and without loss of generality.

Definition 5 (Memory Consistency Check). *Let N be an integer and $\mathcal{A} \subset \mathbb{F}$. A memory consistency check for memory address space $\mathcal{A}$ is a PIOP $\Pi = (\mathsf{I}, \mathsf{P}, \mathsf{V})$ for the following vector language:*

$$\mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}} := \left\{ \begin{array}{l} \mathbf{op} \in \{0,1\}^N, \\ \mathbf{addr} \in \mathcal{A}^N, \\ \mathbf{val} \in \mathbb{F}^N \end{array} \middle| \begin{array}{ll} \forall t \in [N], \textit{either} & \mathbf{op}_{[t]} = 1 \textit{ or} \\ & \mathsf{prev}(t; \mathbf{addr}) = \perp \textit{ or} \\ & \mathbf{val}_{[t]} = \mathbf{val}_{[\mathsf{prev}(t;\mathbf{addr})]} \end{array} \right\} \textit{ where}$$

$$\mathsf{prev}(t; \mathbf{addr}) := \begin{cases} \max J := \left\{ t' \middle| t' < t \wedge \mathbf{addr}_{[t']} = \mathbf{addr}_{[t]} \right\}, & \textit{if } J \neq \emptyset \\ \perp, & \textit{otherwise} \end{cases}.$$

We may write $\mathsf{prev}(t)$ instead of $\mathsf{prev}(t; \mathbf{addr})$ for simplicity when the choice of $\mathbf{addr}$ is unambiguous. Intuitively, $\mathsf{prev}(t)$ maps t to the previous time t' when the same address appeared. The memory consistency requires that $\mathbf{val}_{[t]}$ is equal to $\mathbf{val}_{[t']}$, unless the current instruction is writing ($\mathbf{op}_{[t]} = 1$) or this address was never accessed before ($t' = \perp$).

Next, we will explain the mainstream approach for constructing MCCs, which is referred to as the *sorting paradigm* and is used in all of the ZKVM projects discussed in this work.

4 The Sorting Paradigm

The main challenge in MCCs is handling the *history-dependency* of the memory: the value retrieved from a memory operation is dependent on previous operations that may be far ahead. A natural solution to this challenge is to sort the table $(\mathbf{op}, \mathbf{addr}, \mathbf{val})$ to group related operations together. To avoid affecting the consistency of this trace, the sorting should satisfy the following criterion: it should *never* swap the order between two rows with the same address. The sortings that follow this criterion are formalized by the following definitions.

Definition 6 (Sorting). *Let* $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)}) \in \mathbb{F}^{N \times m}$ *and* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}) \in \mathbb{F}^{N \times m}$ *be two tables with* m *columns and* N *rows. Let* $k_1, k_2, \cdots, k_\ell$ *be* ℓ *distinct integers in* $[m]$*, and* $\preceq_1, \cdots, \preceq_\ell$ *be* ℓ *total orders respectively over* $\mathsf{Elems}(\mathbf{v}^{(k_1)}), \cdots, \mathsf{Elems}(\mathbf{v}^{(k_\ell)})$*. We say* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)})$ *is a sorting of* $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$ *by* $\mathbf{v}^{(k_1)}_{[t]}, \cdots, \mathbf{v}^{k_\ell}_{[t]}$ *if:*

- $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)}) \sim (\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)})$*, i.e., there exists a permutation* σ *over* $[N]$ *such that* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}) = \sigma(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$*; and*
- *for every* $t \in [N-1]$*,* $(\tilde{\mathbf{v}}^{(k_1)}_{[t]}, \cdots, \tilde{\mathbf{v}}^{(k_\ell)}_{[t]}) \preceq (\tilde{\mathbf{v}}^{(k_1)}_{[t+1]}, \cdots, \tilde{\mathbf{v}}^{(k_\ell)}_{[t+1]})$*, where* $\preceq$ *is defined lexigraphically from* $\preceq_1, \cdots, \preceq_\ell$*.*

We say $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)})$ *is sorted by keys* $\mathbf{v}^{(k_1)}, \cdots, \mathbf{v}^{(k_\ell)}$ *with total orders* $\preceq_1, \cdots, \preceq_\ell$*, and* $\mathbf{v}^{(k_i)}$ *is the* i*-th key of this sorting.*

Definition 7 (Canonical Sorting). *Given the two tables* $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)}) \in \mathbb{F}^{N \times m}$ *and* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}) \in \mathbb{F}^{N \times m}$ *satisfying Definition 6, we say* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)})$ *is* the *canonical sorting of* $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$ *if there exists a permutation* σ *such that* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}) = \sigma(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$ *and for every* $t \in [N-1]$*, if* $(\tilde{\mathbf{v}}^{(k_1)}_{[t]}, \cdots, \tilde{\mathbf{v}}^{(k_\ell)}_{[t]}) = (\tilde{\mathbf{v}}^{(k_1)}_{[t+1]}, \cdots, \tilde{\mathbf{v}}^{(k_\ell)}_{[t+1]})$ *then* $\sigma^{-1}(t) < \sigma^{-1}(t+1)$*.*

For convenience, we denote the set of vector tuples satisfying the above definition by the vector relation $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)}, \tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}) \in \mathcal{R}^m_{\mathsf{CN}}(k_1, \cdots, k_\ell, \preceq_1, \cdots, \preceq_\ell)$*.*

Let $\mathcal{R}^m_{\mathsf{CN}}(k_1, \cdots, k_\ell) := \bigcup_{\preceq_1, \cdots, \preceq_\ell} \mathcal{R}^m_{\mathsf{CN}}(k_1, \cdots, k_\ell, \preceq_1, \cdots, \preceq_\ell)$ *be the vector relation of all such vector tuples for arbitrary total order. We may write* $\mathcal{R}^m_{\mathsf{CN}}(\preceq_{\mathbf{v}^{k_1}}, \cdots, \preceq_{\mathbf{v}^{k_\ell}})$ *and* $\mathcal{R}^m_{\mathsf{CN}}(\mathbf{v}^{k_1}, \cdots, \mathbf{v}^{k_\ell})$*, respectively, for ease of description.*

With the above definition of sorted tables, we now describe how to sort the memory trace. Let $\preceq_{\mathbf{addr}}$ be any total order over the address space $\mathcal{A}$ and $\mathcal{R}^{\mathcal{A}}_{\mathsf{CN}}(\preceq_{\mathbf{addr}})$ denote the vector language that consists of all the vector tuples ($\mathbf{op}$, $\mathbf{addr}$, $\mathbf{val}$, $\widetilde{\mathbf{op}}$, $\widetilde{\mathbf{addr}}$, $\widetilde{\mathbf{val}}$) such that $(\widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}})$ is the canonical sorting of $(\mathbf{op}, \mathbf{addr}, \mathbf{val})$ by the key $\mathbf{addr}$ with total order $\preceq_{\mathbf{addr}}$. Let $\mathcal{R}^{\mathcal{A}}_{\mathsf{CN}}(\mathbf{addr})$ be the union of all these vector languages, i.e., $\mathcal{R}^{\mathcal{A}}_{\mathsf{CN}}(\mathbf{addr}) := \bigcup_{\preceq_{\mathbf{addr}}} \mathcal{R}^{\mathcal{A}}_{\mathsf{CN}}(\preceq_{\mathbf{addr}})$. The following theorem is the central idea behind the sorting technique for MCCs.

Theorem 1. *Given any* $\mathcal{A} \subset \mathbb{F}$ *and tuple* $((\mathbf{op}, \mathbf{addr}, \mathbf{val}), (\widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}})) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{CN}}(\mathbf{addr})$*,* $(\mathbf{op}, \mathbf{addr}, \mathbf{val}) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}}$ *if and only if* $(\widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}}$*.*

To prove Theorem 1, we first introduce the following lemma, which essentially claims that the permutation for the canonical sorting can be decomposed into a sequence of transpositions such that each transposition swaps two rows with different keys.

Lemma 1. *Given two tables* $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)}) \in \mathbb{F}^{N\times m}$ *and* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}) \in \mathbb{F}^{N\times m}$ *satisfying the definition of Definition 6 with total orders* $\preceq_1, \cdots, \preceq_\ell$*. Then* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)})$ *is the canonical sorting of* $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$ *by keys* $\mathbf{v}^{(k_1)}, \cdots, \mathbf{v}^{(k_\ell)}$ *if and only if there exists a sequence of transpositions* $\sigma_1, \cdots, \sigma_s$*, where* σ_i *swaps* t_i *and* t_i+1*, such that* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}) = \sigma_s(\sigma_{s-1}(\cdots\sigma_1(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})\cdots))$ *and that for each* $i \in [s]$*,*

$$\left(\mathbf{v}^{(k_1)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i+1)\cdots))]}, \cdots, \mathbf{v}^{(k_\ell)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i+1)\cdots))]}\right) \prec \left(\mathbf{v}^{(k_1)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}^{-1}(t_i)\cdots))]}, \cdots, \mathbf{v}^{(k_\ell}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i)\cdots))]}\right).$$

Proof. Sufficiency. Let $\sigma(\cdot) := \sigma_s(\sigma_{s-1}(\cdots\sigma_1(\cdot)\cdots))$. Then we have $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}) = \sigma(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$. It remains to show that for every $t \in [N-1]$, if $(\tilde{\mathbf{v}}^{(k_1)}_{[t]}, \cdots, \tilde{\mathbf{v}}^{(k_\ell)}_{[t]}) = (\tilde{\mathbf{v}}^{(k_1)}_{[t+1]}, \cdots, \tilde{\mathbf{v}}^{(k_\ell)}_{[t+1]})$ then $\sigma(t) < \sigma(t+1)$. Suppose t^* satisfies $(\tilde{\mathbf{v}}^{(k_1)}_{[t^*]}, \cdots, \tilde{\mathbf{v}}^{(k_\ell)}_{[t^*]}) = (\tilde{\mathbf{v}}^{(k_1)}_{[t^*+1]}, \cdots, \tilde{\mathbf{v}}^{(k_\ell)}_{[t^*+1]})$. Since for each $i \in [s]$, $(\mathbf{v}^{(k_1)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i+1)\cdots))]}, \cdots, \mathbf{v}^{(k_\ell)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i+1)\cdots))]})$ is different from $(\mathbf{v}^{(k_1)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i)\cdots))]}, \cdots, \mathbf{v}^{(k_\ell)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i)\cdots))]})$, we have for each $i \in [s]$, $\{t_i, t_i+1\} \neq \{\sigma_i(\cdots\sigma_s(t^*)\cdots), \sigma_i(\cdots\sigma_s(t^*+1)\cdots)\}$. In particular, $\{t_s, t_s+1\} \neq \{\sigma_s(t^*), \sigma_s(t^*+1)\}$, i.e., $t_s \neq t^*$, which implies that either $\sigma_s(t^*) = t^*-1, \sigma_s(t^*+1) = t^*+1$ (in case $t_s = t^*-1$), or $\sigma_s(t^*) = t^*, \sigma_s(t^*+1) = t^*+2$ (in case $t_s = t^*+1$), or $\sigma_s(t^*) = t^*, \sigma_s(t^*+1) = t^*+1$ (otherwise). In all cases, $\sigma_s(t^*) < \sigma_s(t^*+1)$. Similarly, we rule out the possibility that $\sigma_s(t^*) = t_{s-1}, \sigma_s(t^*+1) = t_{s-1}+1$ (which would imply $\{\sigma_{s-1}(\sigma_s(t^*)), \sigma_{s-1}(\sigma_s(t^*+1))\} = \{t_{s-1}, t_{s-1}+1\}$), and by similar case studies we can conclude that $\sigma_{s-1}(\sigma_s(t^*)) < \sigma_{s-1}(\sigma_s(t^*+1))$. Continuing this process, we finally get $\sigma_1(\cdots\sigma_{s-1}(\sigma_s(t^*))\cdots) < \sigma_1(\cdots\sigma_{s-1}(\sigma_s(t^*+1))\cdots)$, which is exactly $\sigma^{-1}(t^*) < \sigma(t^*+1)$ (note that transpositions are inverses of themselves).

Necessity. Suppose $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)})$ is the canonical sorting of $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$ by keys $\mathbf{v}^{(k_1)}, \cdots, \mathbf{v}^{(k_\ell)}$, we construct the sequence $\sigma_1, \cdots, \sigma_s$ as follows. Starting from the table $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$, and initialize $i = 1$, then repeat the following steps until we cannot proceed any further: find the first t such that $(\mathbf{v}^{(k_1)}_{[t]}, \cdots, \mathbf{v}^{(k_\ell)}_{[t+1]}) \prec (\mathbf{v}^{(k_1)}_{[t]}, \cdots, \mathbf{v}^{(k_\ell)}_{[t]})$, set $t_i = t$, swap the rows t and $t+1$, i.e., apply the swap σ_i to the current table, and increment i.

Since $\preceq$ is a total order over the tuples $(\mathbf{v}^{(k_1)}_{[t]}, \cdots, \mathbf{v}^{(k_\ell)}_{[t]})$, the above procedure will terminate. Obviously, by construction, we have for each $i \in [s]$,

$$\left(\mathbf{v}^{(k_1)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i+1)\cdots))]}, \cdots, \mathbf{v}^{(k_\ell)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i+1)\cdots))]}\right) \prec \left(\mathbf{v}^{(k_1)}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}^{-1}(t_i)\cdots))]}, \cdots, \mathbf{v}^{(k_\ell}_{[\sigma_1(\sigma_2(\cdots\sigma_{i-1}(t_i)\cdots))]}\right).$$

Moreover, for each $t \in [N-1]$, we have $(\mathbf{v}^{(k_1)}_{[\sigma_1(\sigma_2(\cdots\sigma_s(t)\cdots))]}, \cdots, \mathbf{v}^{(k_\ell)}_{[\sigma_1(\sigma_2(\cdots\sigma_s(t)\cdots))]}) \preceq (\mathbf{v}^{(k_1)}_{[\sigma_1(\sigma_2(\cdots\sigma_s(t+1)\cdots))]}, \cdots, \mathbf{v}^{(k_\ell)}_{[\sigma_1(\sigma_2(\cdots\sigma_s(t+1)\cdots))]})$, since otherwise the procedure would not have stopped. Let $\sigma = \sigma_s(\cdots\sigma_1(\cdot))$. By the same argument as in the previous part, we conclude that $\sigma^{-1}(\sigma(t)) < \sigma^{-1}(\sigma(t)+1)$ for every $\sigma(t) \in [N-1]$ where $(\mathbf{v}^{(k_1)}_{[\sigma(t)]}, \cdots, \mathbf{v}^{(k_1)}_{[\sigma(t)]}) = (\mathbf{v}^{(k_1)}_{[\sigma(t)+1]}, \cdots, \mathbf{v}^{(k_1)}_{[\sigma(t)+1]})$, i.e., $\sigma(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$ is the canonical sorting of $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$. □

Proof (of Theorem 1). Let $\preceq$ be any total order over $\mathsf{Elems}(\mathbf{addr})$. By Lemma 1, we can find a sequence of transpositions $\sigma_1, \cdots, \sigma_s$ with the desired properties. It suffices to show that the memory consistency is preserved by each transposition, i.e., for each $i \in [s]$, $\sigma_i(\cdots\sigma_1(\mathbf{op}, \mathbf{addr}, \mathbf{val})\cdots) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}}$ if and only if $\sigma_{i-1}(\cdots\sigma_1(\mathbf{op}, \mathbf{addr}, \mathbf{val})\cdots) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}}$. The proof of Lemma 1 demonstrates that we can find the sequence $\sigma_1, \cdots, \sigma_s$ by repeatedly swapping adjacent rows of $(\mathbf{op}, \mathbf{addr}, \mathbf{val})$ where $\mathbf{addr}_{[t+1]} \prec \mathbf{addr}_{[t]}$ until $\mathbf{addr}$ is completely sorted. We only need to show that each operation in this procedure preserves consistency. Obviously, the properties $\mathbf{op} \in \{0,1\}^N$ and $\mathbf{addr} \in \mathcal{A}^N$ are not affected by swapping, so we only need to show that the property $\forall t \in [N], \mathbf{op}_{[t]} = 1 \vee \mathsf{prev}(t; \mathbf{addr}) \vee \mathbf{val}_{[t]} = \mathbf{val}_{[\mathsf{prev}(t;\mathbf{addr})]}$ is not affected. Assume the operation swaps the rows t^* and t^*+1. Obviously, the property still holds for $t \in [t^*-2]$. Since $\mathbf{addr}_{[t^*]} \neq \mathbf{addr}_{[t^*+1]}$, after the swapping, we simultaneously swap $\mathbf{op}_{[t^*]}$ with $\mathbf{op}_{[t^*+1]}$, $\mathsf{prev}(t^*; \mathbf{addr})$ with $\mathsf{prev}(t^*+1; \mathbf{addr})$, and $\mathbf{val}_{[t^*]}$ with $\mathbf{val}_{[t^*+1]}$. Therefore, the consistencies of these two positions are still preserved. Finally, for $t > t^*+1$, if $\mathsf{prev}(t; \mathbf{addr})$ is t^* (or t^*+1), then after the swap $\mathsf{prev}(t; \mathbf{addr})$ becomes t^*+1 (or t^*). However, since $\mathbf{val}_{[t^*]}$ and $\mathbf{val}_{[t^*+1]}$ are also swapped, the value $\mathbf{val}_{[\mathsf{prev}(t;\mathbf{addr})]}$ stays unchanged. Otherwise, $\mathsf{prev}(t; \mathbf{addr}) \notin \{t^*, t^*+1\}$, then $\mathsf{prev}(t; \mathbf{addr})$ is not affected and $\mathbf{val}_{[\mathsf{prev}(t;\mathbf{addr})]}$ also stays unchanged. This finishes the proof. □

Based on Theorem 1, Algorithm 1 presents the common workflow of all the MCCs using the sorting technique, where the $\mathsf{CSort}_{\mathcal{A}}$ protocol is decided by the concrete constructions. We call this workflow the *sorting paradigm*. In Algorithm 1, we use the following trick for proving a statement of the form "if $r = 0 \wedge s = t$ then $u = v$". We note that this statement is equivalent to "$\exists a, b$ such that $a \cdot r + b \cdot (s - t) = u - v$". We use this trick to prove the statement that whenever $\mathbf{op}$ is 0 for reading and the sorted address matches with the previous one, then the value should also match.

Theorem 2. *If $\mathsf{CSort}_{\mathcal{A}}$ is a PIOP for the vector language $\mathcal{R}^{\mathcal{A}}_{\mathsf{CN}}(\preceq_{\mathbf{addr}})$ with completeness error e_c and soundness error e_s, where $\preceq_{\mathbf{addr}}$ is any total order over $\mathcal{A}$, then the $\mathsf{CSortMCC}_{\mathcal{A}}$ protocol in Algorithm 1 is an MCC for $\mathcal{A}$ with completeness error e_c and soundness error $e_s + (2N+1)/|\mathbb{F}|$.*

Proof. Consider the following sequence of statements:

$$(\mathbf{op}, \mathbf{addr}, \mathbf{val}) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}} \tag{1}$$

$$\mathbf{addr} \in \mathcal{A}^N \tag{2}$$

Algorithm 1 Sorting Paradigm

procedure $\mathsf{CSortMCC}_{\mathcal{A}}(\mathbf{op}, \mathbf{addr}, \mathbf{val})$

P sends $\widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$, the canonical sorting of $(\mathbf{op}, \mathbf{addr}, \mathbf{val})$;

P sends $\mathbf{a}, \mathbf{b}$ such that $\forall t \in [N-1]$, $\mathbf{a}_{[t]} \cdot \widetilde{\mathbf{op}}_{[t+1]} + \mathbf{b}_{[t]} \cdot (\widetilde{\mathbf{addr}}_{[t+1]} - \widetilde{\mathbf{addr}}_{[t]}) = \widetilde{\mathbf{val}}_{[t+1]} - \widetilde{\mathbf{val}}_{[t]}$;

V checks $\mathbf{op} \circ \mathbf{op} = \mathbf{op}$;

V checks $(\mathbf{id} - \mathbf{1}) \circ (\mathbf{a} \circ \widetilde{\mathbf{op}}^{\leftarrow 1} + \mathbf{b} \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) - (\widetilde{\mathbf{val}}^{\leftarrow 1} - \widetilde{\mathbf{val}})) = \mathbf{0}$ where $\mathbf{id} = \{1, \omega, \cdots, \omega^{N-1}\}$ is the evaluation of $f(X) := X$ over $\mathbb{D}$;

P and V run the $\mathsf{CSort}_{\mathcal{A}}$ protocol with inputs $\mathbf{op}$, $\mathbf{addr}$, $\mathbf{val}$, $\widetilde{\mathbf{op}}$, $\widetilde{\mathbf{addr}}$, $\widetilde{\mathbf{val}}$.

$$(\mathbf{op}, \mathbf{addr}, \mathbf{val}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{CN}}(\preceq_{\mathbf{addr}}) \tag{3}$$

$$(\tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}} \tag{4}$$

$$\mathbf{op} \in \{0,1\}^N \tag{5}$$

$$\mathsf{V} \text{ accepts } \mathbf{op} \circ \mathbf{op} = \mathbf{op} \tag{6}$$

$$\forall t \in [N], \mathsf{prev}(t; \tilde{\mathbf{addr}}) = \begin{cases} \perp, & \text{if } t = 1 \vee \mathbf{addr}_{[t-1]} \neq \mathbf{addr}_{[t]} \\ t-1, & \text{otherwise} \end{cases} \tag{7}$$

$$\forall t \in [N-1], \tilde{\mathbf{op}}_{[t+1]} = 1 \vee \tilde{\mathbf{addr}}_{[t+1]} \neq \tilde{\mathbf{addr}}_{[t]} \vee \tilde{\mathbf{val}}_{[t+1]} = \tilde{\mathbf{val}}_{[t]} \tag{8}$$

$$\forall t \in [N-1], \mathbf{a}_{[t]} \cdot \tilde{\mathbf{op}}_{[t+1]} + \mathbf{b}_{[t]} \cdot (\tilde{\mathbf{addr}}_{[t+1]} - \tilde{\mathbf{addr}}_{[t]}) = \tilde{\mathbf{val}}_{[t+1]} - \tilde{\mathbf{val}}_{[t]} \tag{9}$$

$$\mathsf{V} \text{ accepts } (\mathbf{id} - \mathbf{1}) \circ (\mathbf{a} \circ \tilde{\mathbf{op}}^{\leftarrow 1} + \mathbf{b} \circ (\tilde{\mathbf{addr}}^{\leftarrow 1} - \tilde{\mathbf{addr}}) - (\tilde{\mathbf{val}}^{\leftarrow 1} - \tilde{\mathbf{val}})) = \mathbf{0} \tag{10}$$

$$\mathsf{V} \text{ accepts in } \mathsf{CanonicalSort}_{\mathcal{A}} \tag{11}$$

Completeness follows from the inductions

- (1) (3) $\Rightarrow$ (4) (7) $\Rightarrow$ (8) $\Rightarrow$ (9) $\Rightarrow$ (10), where where (1) (3) $\Rightarrow$ (4) follows from Theorem 1;
- (5) $\Rightarrow$ (6);
- (1) $\Rightarrow$ (2);
- (2) (3) $\Rightarrow$ (11) fails with probability e_c.

The completeness error is therefore e_c.

Soundness follows from the inductions

- (6) $\Rightarrow$ (5) and (10) $\Rightarrow$ (9) fail with probability $(2N+1)/|\mathbb{F}|$;
- (11) $\Rightarrow$ (3) (2) fails with probability e_s;
- (9) $\Rightarrow$ (8);
- (3) $\Rightarrow$ (7);
- (8) (7) (5) (2) $\Rightarrow$ (4);
- (3) (4) $\Rightarrow$ (1).

The soundness error is given by the union bound. □

In the following subsections, we will introduce three different constructions of the CSort protocol, each for a different memory model.

4.1 Contiguous Read-Only Memory

We start from the simplest case—the *contiguous read-only* memory setting, which is adopted by Cairo [GPR21]. In this setting, there is no writing operation, which means $\mathbf{op}$ is restricted to be the zero vector, hence "read-only". Moreover, all accessed memory addresses form a contiguous region, which means $\mathsf{Elems}(\mathbf{addr}) = \{s, s+1, \cdots, s+S-1\}$ for some $s \in \mathbb{F}$ and $S \in \mathbb{N}$. Formally, all valid execution traces for contiguous read-only memory constitute the vector language

$$\mathcal{R}_{\mathsf{CROM}} := \left\{ (\mathbf{addr}, \mathbf{val}) \,\middle|\, (\mathbf{0}, \mathbf{addr}, \mathbf{val}) \in \mathcal{R}_{\mathsf{Mem}}^{\mathbb{F}}, \mathbf{addr} \in \mathcal{R}_{\mathsf{CONT}} \right\}$$

where $\mathcal{R}_{\mathsf{CONT}} = \left\{ \mathbf{addr} \,\middle|\, \exists s \in \mathbb{F}, S \in [N], \mathsf{Elems}(\mathbf{addr}) = \{s+i\}_{i=0}^{S-1} \right\}$.

Contiguous read-only memories are more restricted and have fewer capabilities compared to read-write memories, thus the programming process is more challenging for programmers. On the positive side, the contiguous read-only memory model enables a much simpler protocol for checking memory consistency. Being read-only, the vectors $\mathbf{op}$, $\widetilde{\mathbf{op}}$ and $\mathbf{a}$ are eliminated from Algorithm 1, and every sorting of $(\mathbf{0}, \mathbf{addr}, \mathbf{val})$ is the canonical sorting. By contiguity, $\widetilde{\mathbf{addr}}$ satisfies that adjacent addresses differ by at most one. These observations lead to Algorithm 2. In this protocol, the vector equation checked by the verifier ensures that $\widetilde{\mathbf{addr}}_{[t]} - \widetilde{\mathbf{addr}}_{[t+1]}$ is either 0 or 1, except for the edge case $t = N$, which is eliminated by multiplying the vector $\mathbf{id} - \omega^{N-1} \cdot \mathbf{1} = (1 - \omega^{N-1}, \omega - \omega^{N-1}, \cdots, \omega^{N-1} - \omega^{N-1})$, which is zero only at $t = N$.

Algorithm 2 Canonical Sort for Contiguous Read-Only Memory

procedure $\mathsf{CROMSort}(\mathbf{addr}, \mathbf{val}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}})$
  V checks $(\mathbf{addr}, \mathbf{val}) \sim (\widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}})$;
  V checks $(\mathbf{id} - \omega^{N-1} \cdot \mathbf{1}) \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}} - \mathbf{1}) = \mathbf{0}$.

Theorem 3. *Assuming the input vectors satisfy* $\forall t \in [N-1], \widetilde{\mathbf{addr}}_{[t]} \neq \widetilde{\mathbf{addr}}_{[t+1]} \vee \widetilde{\mathbf{val}}_{[t]} = \widetilde{\mathbf{val}}_{[t+1]}$*, then the* $\mathsf{CROMSort}$ *protocol in Algorithm 2 is a PIOP for the vector language*

$$\mathcal{R}_{\mathsf{CN}}^{\mathsf{CROM}} := \left\{ (\mathbf{addr}, \mathbf{val}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}) \,\middle|\, \begin{array}{l} \mathbf{addr} \in \mathcal{R}_{\mathsf{CONT}} \\ (\mathbf{0}, \mathbf{addr}, \mathbf{val}, \mathbf{0}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}) \in \mathcal{R}_{\mathsf{CN}}^{\mathcal{A}}(\preceq_{\mathbf{addr}}) \end{array} \right\}$$

with completeness error $e_{c,\mathsf{Perm}}$ *and soundness error* $e_{s,\mathsf{Perm}} + 2N/|\mathbb{F}|$*, where* $\preceq_{\mathbf{addr}}$ *is the total order over* $\mathsf{Elems}(\mathbf{addr}) = \{s+i\}_{i=1}^{S}$ *such that* $s+i \preceq_{\mathbf{addr}} s+j \Leftrightarrow i \leq j$ *for every pair of* $(i, j) \in [0..S-1]^2$.

Proof. Consider the following statements:

$$\forall t \in [N-1], \tilde{\mathbf{addr}}_{[t]} \neq \tilde{\mathbf{addr}}_{[t+1]} \vee \tilde{\mathbf{val}}_{[t]} = \tilde{\mathbf{val}}_{[t+1]} \tag{12}$$

$$(\mathbf{addr}, \mathbf{val}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \in \mathcal{R}_{\mathsf{CN}}^{\mathsf{CROM}} \tag{13}$$

$$(\mathbf{0}, \mathbf{addr}, \mathbf{val}, \mathbf{0}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \in \mathcal{R}_{\mathsf{CN}}^{\mathcal{A}}(\preceq_{\mathbf{addr}}) \tag{14}$$

$$\mathsf{Elems}(\mathbf{addr}) = \{s+i\}_{i=0}^{S-1} \tag{15}$$

$$(\mathbf{addr}, \mathbf{val}) \sim (\tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \tag{16}$$

$$\mathsf{V} \text{ accepts } (\mathbf{addr}, \mathbf{val}) \sim (\tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \tag{17}$$

$$\forall t \in [N-1], \tilde{\mathbf{addr}}_{[t+1]} - \tilde{\mathbf{addr}}_{[t]} \in \{0,1\} \tag{18}$$

$$\mathsf{V} \text{ accepts } (\mathbf{id} - \omega^{N-1} \cdot \mathbf{1}) \circ (\mathbf{addr}^{\leftarrow 1} - \mathbf{addr}) \circ (\mathbf{addr}^{\leftarrow 1} - \mathbf{addr} - \mathbf{1}) = \mathbf{0} \tag{19}$$

Completeness follows from the induction sequences

- (13) ⇒ (14) (15);
- (14) ⇒ (16) ⇒ (17) where (16) ⇒ (17) fails with probability $e_{c,\mathsf{Perm}}$;
- (14) (15) ⇒ (18) ⇒ (19).

The completeness error is then $e_{c,\mathsf{Perm}}$.

Soundness follows from the induction sequences

- (19) ⇒ (18) ⇒ (15) where (19) ⇒ (18) fails with probability at most $2N/|\mathbb{F}|$;
- (18) (12) (16) ⇒ (14);
- (17) ⇒ (16) where (17) ⇒ (16) fails with probability at most $e_{s,\mathsf{Perm}}$;
- (14) (15) ⇒ (13).

The soundness error is then given by union bound. □

Next, we handle the most popular case where the memory is writable and the memory address space is the set of 32-bit integers.

4.2 Read-Write Memory with 32-bit Addresses

We explain how to construct CSort when $\mathcal{A} = [0..2^{32}-1]$. This 32-bit address space is adopted by Miden [Mid22] and 32-bit RiscZero [Ris22]. The techniques can also be extended to 64- and 256-bit address spaces; the later is used in zkEVMs, e.g., Scroll [Scr22], Polygon Hermez [Pol22], zkSync [zkS22].

In the 32-bit randomly accessible memory, the differences between adjacent entries in the sorted addresses $\widetilde{\mathbf{addr}}$ are no longer restricted to $\{0,1\}$, but fall in a larger range, namely $[0..2^{32}-1]$. Therefore, the boolean check in Algorithm 2 is replaced with the 32-bit range check. Moreover, without the read-only setting, the canonicity of the sorting is no longer automatically guaranteed. Instead, the prover sorts the memory trace together with the vector $\mathbf{incs} = (1, 2, \cdots, N)$ and the verifier ensures the sorted vector $\widetilde{\mathbf{incs}}$ satisfies certain properties, as detailed in Algorithm 3. In this protocol, the vectors **inv** and **diff** are used to indicate the positions where $\widetilde{\mathbf{addr}}_{[t+1]}$ differs from $\widetilde{\mathbf{addr}}_{[t]}$, i.e. $\widetilde{\mathbf{addr}}_{[t+1]} - \widetilde{\mathbf{addr}}_{[t]}$ is invertible. The second Range32 in Algorithm 3 uses the tricks from Hermez and Miden, which applies the 32-bit range check simultaneously to two vectors, exploiting the fact that we only need to ensure $\widetilde{\mathbf{incs}}_{[t+1]} - \widetilde{\mathbf{incs}}_{[t]} \in [0, 2^{32})$ for $\mathbf{diff}_{[t]} = 0$ and $\widetilde{\mathbf{addr}}_{[t+1]} - \widetilde{\mathbf{addr}}_{[t]} \in [0, 2^{32})$ for $\mathbf{diff}_{[t]} = 1$. The masking vector $\mathbf{1} - \mathbf{e}_N$ excludes the case $t = N$ from the range check.

Algorithm 3 Canonical Sort for 32-bit Read-Write Memory

procedure RW32Sort($\mathbf{op}, \mathbf{addr}, \mathbf{val}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}}$)

I sends $\mathbf{incs} = (1, 2, \cdots, N)$;

P sends $\widetilde{\mathbf{incs}}$ satisfying $(\mathbf{incs}, \mathbf{op}, \mathbf{addr}, \mathbf{val}) \sim (\widetilde{\mathbf{incs}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}})$;

P sends $\mathbf{inv}$ such that for every $t \in [N]$, $\mathbf{inv}_{[t]} = 0$ if $\widetilde{\mathbf{addr}}_{[t+1]} = \widetilde{\mathbf{addr}}_{[t]}$, and $\mathbf{inv}_{[t]} = (\widetilde{\mathbf{addr}}_{[t+1]} - \widetilde{\mathbf{addr}}_{[t]})^{-1}$ elsewhere, where $\widetilde{\mathbf{addr}}_{[N+1]}$ is treated as $\widetilde{\mathbf{addr}}_{[1]}$;

P sends $\mathbf{diff}$ such that for every $t \in [N]$, $\mathbf{diff}_{[t]} = 0$ if $\widetilde{\mathbf{addr}}_{[t+1]} = \widetilde{\mathbf{addr}}_{[t]}$, and $\mathbf{diff}_{[t]} = 1$ elsewhere, where $\widetilde{\mathbf{addr}}_{[N+1]}$ is treated as $\widetilde{\mathbf{addr}}_{[1]}$;

V checks $(\mathbf{incs}, \mathbf{op}, \mathbf{addr}, \mathbf{val}) \sim (\widetilde{\mathbf{incs}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}})$;

V checks $(\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) \circ \mathbf{inv} = \mathbf{diff}$ and $\mathbf{diff} \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}) = \widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}}$;

P and V run the Range32 protocol with inputs $\mathbf{addr}, \mathbf{1}$;

P and V run the Range32 protocol with inputs $(\mathbf{1} - \mathbf{diff}) \circ (\widetilde{\mathbf{incs}}^{\leftarrow 1} - \widetilde{\mathbf{incs}}) + \mathbf{diff} \circ (\widetilde{\mathbf{addr}}^{\leftarrow 1} - \widetilde{\mathbf{addr}})$ and $\mathbf{1} - \mathbf{e}_N$.

Theorem 4. *The* RW32Sort *protocol in Algorithm 3 is a PIOP for the vector language* $\mathcal{R}_{\mathsf{CN}}^{[0..2^{32}-1]}(\preceq_{\mathbf{addr}})$ *with completeness error* $e_{c,\mathsf{Perm}} + 2 \cdot e_{c,\mathsf{Range32}}$ *and soundness error* $(2N+1)/|\mathbb{F}| + e_{s,\mathsf{Perm}} + 2 \cdot e_{s,\mathsf{Range32}}$*, where* $\preceq_{\mathbf{addr}}$ *is the natural order over integers.*

We use the following lemma for an equivalent condition for canonical sorting.

Lemma 2. *Given two tables* $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)}) \in \mathbb{F}^{N \times m}$ *and* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}) \in \mathbb{F}^{N \times m}$*.* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)})$ *is the canonical sorting of* $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)})$ *by keys* $\mathbf{v}^{(k_1)}, \cdots, \mathbf{v}^{(k_\ell)}$ *with total orders* $\preceq_1, \cdots, \preceq_\ell$ *if and only if there exists* $\tilde{\mathbf{incs}} \in \mathbb{F}^N$ *such that* $(\tilde{\mathbf{v}}^{(1)}, \cdots, \tilde{\mathbf{v}}^{(m)}, \tilde{\mathbf{incs}})$ *is a sorting of* $(\mathbf{v}^{(1)}, \cdots, \mathbf{v}^{(m)}, \mathbf{incs})$ *by keys* $\mathbf{v}^{(k_1)}, \cdots, \mathbf{v}^{(k_\ell)}, \mathbf{incs}$ *with total orders* $\preceq_1, \cdots, \preceq_\ell, \preceq_{\mathbf{incs}}$*, where* $\mathbf{incs} = (1, 2, \cdots, N)$ *and* $\preceq_{\mathbf{incs}}$ *is the natural order over* $1, 2, \cdots, N$*.*

Proof. Sufficiency is obvious. Necessity follows from the observation that the vector $\tilde{\mathbf{incs}} = (\sigma(1), \cdots, \sigma(N))$ satisfies the requirement. □

Proof (Of Theorem 4). Consider the following statements

$$(\mathbf{op}, \mathbf{addr}, \mathbf{val}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \in \mathcal{R}_{\mathsf{CN}}^{[0..2^{32}-1]} \tag{20}$$

$$(\mathbf{op}, \mathbf{addr}, \mathbf{val}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \in \mathcal{R}_{\mathsf{CN}}^{[0..2^{32}-1]}(\preceq_{\mathbf{addr}}) \tag{21}$$

$$\mathbf{addr} \in [0..2^{32}-1]^N \tag{22}$$

$$\mathsf{V} \text{ accepts in Range32 with inputs } \mathbf{addr}, \mathbf{1} \tag{23}$$

$$(\tilde{\mathbf{incs}}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \text{ is a sorting of } (\mathbf{incs}, \mathbf{op}, \mathbf{addr}, \mathbf{val}) \text{ by } \mathbf{addr}, \mathbf{incs} \text{ with } \preceq_{\mathbf{addr}} \tag{24}$$

$$\tilde{\mathbf{addr}} \text{ is sorted by } \preceq_{\mathbf{addr}} \tag{25}$$

$$\forall t \in [N-1], \tilde{\mathbf{addr}}_{[t+1]} - \tilde{\mathbf{addr}}_{[t]} \in [0..2^{32}-1] \tag{26}$$

$$\forall t \in [N-1], \tilde{\mathbf{incs}}_{[t+1]} - \tilde{\mathbf{incs}}_{[t]} \in [1..2^{32}] \vee \tilde{\mathbf{addr}}_{[t+1]} \neq \tilde{\mathbf{addr}}_{[t]} \tag{27}$$

$$(\mathbf{incs}, \mathbf{op}, \mathbf{addr}, \mathbf{val}) \sim (\tilde{\mathbf{incs}}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \tag{28}$$

$$\mathsf{V} \text{ accepts } (\mathbf{incs}, \mathbf{op}, \mathbf{addr}, \mathbf{val}) \sim (\tilde{\mathbf{incs}}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \tag{29}$$

$$\forall t \in [N], \mathbf{inv}_{[t]} = \begin{cases} 0, & \text{if } \tilde{\mathbf{addr}}_{[t+1]} = \tilde{\mathbf{addr}}_{[t]} \\ (\tilde{\mathbf{addr}}_{[t+1]} - \tilde{\mathbf{addr}}_{[t]})^{-1}, & \text{otherwise} \end{cases} \tag{30}$$

$$\forall t \in [N], \mathbf{diff}_{[t]} = \begin{cases} 0, \text{ if } \tilde{\mathbf{addr}}_{[t+1]} = \tilde{\mathbf{addr}}_{[t]} \\ 1, \text{ otherwise} \end{cases} \tag{31}$$

$$\mathsf{V} \text{ accepts in } \mathsf{Range32} \text{ with inputs} \\ (\mathbf{1} - \mathbf{diff}) \circ (\tilde{\mathbf{incs}}^{\leftarrow 1} - \tilde{\mathbf{incs}}) + \mathbf{diff} \circ (\tilde{\mathbf{addr}}^{\leftarrow 1} - \tilde{\mathbf{addr}}) \\ \text{and } \mathbf{1} - \mathbf{e}_N \tag{32}$$

$$(\tilde{\mathbf{addr}}^{\leftarrow 1} - \tilde{\mathbf{addr}}) \circ \mathbf{inv} = \mathbf{diff} \tag{33}$$

$$\mathbf{diff} \circ (\tilde{\mathbf{addr}}^{\leftarrow 1} - \tilde{\mathbf{addr}}) = \tilde{\mathbf{addr}}^{\leftarrow 1} - \tilde{\mathbf{addr}} \tag{34}$$

$$\mathsf{V} \text{ accepts } (\tilde{\mathbf{addr}}^{\leftarrow 1} - \tilde{\mathbf{addr}}) \circ \mathbf{inv} = \mathbf{diff} \tag{35}$$

$$\mathsf{V} \text{ accepts } \mathbf{diff} \circ (\tilde{\mathbf{addr}}^{\leftarrow 1} - \tilde{\mathbf{addr}}) = \tilde{\mathbf{addr}}^{\leftarrow 1} - \tilde{\mathbf{addr}} \tag{36}$$

Completeness follows from the sequence of inductions

- (20) ⇒ (21) (22);
- (22) ⇒ (23) fails with probability $e_{c,\mathsf{Range32}}$;
- (21) ⇒ (24) ⇒ (25) (28), where (21) ⇒ (24) follows from Lemma 2;
- (25) ⇒ (26);
- (28) ⇒ (29) fails with probability $e_{c,\mathsf{Perm}}$;
- (30) (31) ⇒ (33) (34);
- (31) (27) (26) ⇒ (32) fails with probability $e_{c,\mathsf{Range32}}$;
- (33) (34) ⇒ (35) (36).

The completeness error is the sum of these probabilities by union bound.

Soundness follows from the sequence of inductions

- (35) (36) ⇒ (33) (34) fails with probability $(2N+1)/|\mathbb{F}|$;
- (33) (34) ⇒ (31);
- (31) (32) ⇒ (27) (26) fails with probability $e_{s,\mathsf{Range32}}$;
- (23) ⇒ (22) fails with probability $e_{s,\mathsf{Range32}}$;
- (22) (26) ⇒ (25);
- (27) (25) ⇒ (24);
- (29) ⇒ (28) fails with probability $e_{s,\mathsf{Perm}}$;
- (28) (24) ⇒ (21) by Lemma 2;
- (21) (22) ⇒ (20).

The soundness error is the sum of these probabilities by the union bound. □

The above protocol can be extended to $32k$-bit address space for arbitrary k, as long as $|\mathbb{F}| > 2^{32k+1}$. In practice, we are interested in cases where $k = 1, 2, 4$ and 8. However, the cost also grows linearly with k, particularly the number of online polynomial oracles and queries. In comparison, an MCC protocol with the full address space, i.e., $\mathcal{A} = \mathbb{F}$, provides a sufficiently large space with a smaller cost. We introduce the related techniques next.

4.3 Read-Write Memory with the Full Address Space

We present the canonical sorting extracted from Triton VM [TV22], which, at the time of writing, is the *only* ZKVM that adopts the setting with both full address space (i.e., $\mathcal{A} = \mathbb{F}$) and a read-write memory. This address space covers the functionalities of both 64-bit and 256-bit memories[8]. In cases where $|\mathbb{F}| < 2^{256}$, one can use two or more field elements as a memory address, making $\mathcal{A} = \mathbb{F}^c$ for $c > 1$, and linearly combine the c address traces with random challenges supplied by the verifier. Each extra address trace enlarges the memory space by a factor of $|\mathbb{F}|$ with the cost of only one online polynomial oracle. For simplicity, we assume $c = 1$ in the rest of this section.

When $\mathcal{A} = \mathbb{F}$, the greatest challenge is to define a total order over $\mathsf{Elems}(\mathbf{addr})$ that is efficiently verifiable, since $\mathsf{Elems}(\mathbf{addr})$ is neither contiguous nor restricted to a small subset. Triton VM overcomes this issue by designing a new technique for showing that $\widetilde{\mathbf{addr}}$ is sorted by any total order, which we summarize in Algorithm 4. Algorithm 4 is slightly different from that of Triton VM, dropping the engineering-related details. Moreover, we extract the core of the Triton VM memory protocol, the *contiguity check*, and generalize and reformulate it as the *distinctness check*, which will also be used in our construction in the next section.

The contiguity check of Triton VM relies on the following lemma, which presents an equivalent condition for a vector being sorted.

Lemma 3. *Given any vector $\mathbf{v} \in \mathbb{F}^N$, the following two statements are equivalent:*

1. *There exists a total order "$\preceq$" over $\mathbb{F}$ such that $\mathbf{v}$ is sorted by "$\preceq$".*
2. *There exists a vector $\mathbf{b} \in \{0,1\}^N$ such that*
   - *$\mathbf{b}_{[1]} = 1$, and for every $t \in [N-1]$, either $\mathbf{v}_{[t]} = \mathbf{v}_{[t+1]}$ or $\mathbf{b}_{[t+1]} = 1$;*
   - *the elements $\{\mathbf{v}_{[t]} | t \in [N], \mathbf{b}_{[t]} = 1\}$ are distinct.*

Proof. Sufficiency. If there exists such a vector $\mathbf{b}$ satisfying both conditions, we claim that for every pair of $i < j$, if $\mathbf{v}_{[i]} = \mathbf{v}_{[j]}$ then for every k from i to j, $\mathbf{v}_{[i]} = \mathbf{v}_{[k]}$. Assume for contradiction that there exist $i < k < j$ such that $\mathbf{v}_{[i]} = \mathbf{v}_{[j]} \neq \mathbf{v}_{[k]}$, then by the first condition, there exists $i' \leq i$ such that $\mathbf{v}_{[i']} = \mathbf{v}_{[i]}$ and $\mathbf{b}_{[i']} = 1$, and exists $k < j' \leq j$ such that $\mathbf{v}_{[j']} = \mathbf{v}_{[j]}$ and $\mathbf{b}_{[j']} = 1$. Then $\mathbf{v}_{[i']} = \mathbf{v}_{[j']}$ but $i' \neq j'$, contradicting the second condition.

[8] Unless for extremely special cases where the program relies on the memory check to decide whether to abort or not.

With the above claim, we define the total order "$\preceq$" over the elements in $\mathbf{v}$ as follows. For every pair of $a, b \in \mathbf{v}$, $a \preceq b$ if and only if there exist $i \leq j$ such that $\mathbf{v}_{[i]} = a$ and $\mathbf{v}_{[j]} = b$. This total order is well-defined, since if both $a \preceq b$ and $b \preceq a$, then there exist $i \leq k \leq j$ such that $\mathbf{v}_{[i]} = \mathbf{v}_{[j]} = a$ and $\mathbf{v}_{[k]} = b$. By the above claim, $a = b$.

Necessity. If $\mathbf{v}$ is sorted, we construct $\mathbf{b}$ as follows. Let $\mathbf{b}_{[1]} = 1$, and for every $t \in [N-1]$, let $\mathbf{b}_{[t+1]} = 1$ if $\mathbf{v}_{[t+1]} \neq \mathbf{v}_{[t]}$, otherwise $\mathbf{b}_{[t+1]} = 0$. Obviously, $\mathbf{b}$ satisfies the first condition. If the second condition is not satisfied, i.e., there are $\mathbf{v}_{[i]} = \mathbf{v}_{[j]}$ for $i < j$ and $\mathbf{b}_{[i]} = \mathbf{b}_{[j]} = 1$, then by the first condition $\mathbf{v}_{[j]} \neq \mathbf{v}_{[j-1]}$, and $\mathbf{v}_{[i]} \prec \mathbf{v}_{[j-1]} \prec \mathbf{v}_{[j]} = \mathbf{v}_{[i]}$, contradicting the definition of the total order. □

Lemma 3 reduces the problem of proving that $\mathbf{v}$ is sorted to proving that the elements in certain positions of $\mathbf{v}$ are distinct, where the positions are specified by $\mathbf{b}$. This problem is then handled by the following lemma.

Lemma 4 (Proposition 6.6 of [Mig92]). *Let $\mathbb{F}$ be a finite field and $f(X) = \sum_{i=0}^{d} f_i X^i \in \mathbb{F}[X]$. Then $f(X)$ is squarefree if and only if $\gcd(f(X), Df(X)) = 1$, where $Df(X)$ is the shorthand for formal derivative of $f(X)$, i.e., $Df(X) := \frac{d(f(X))}{dX} = \sum_{i=0}^{d-1}(i+1)f_{i+1}X^i$.*

Lemma 4 inspires the protocol $\mathsf{DstFull}$ in Algorithm 4, which proves that given $\mathbf{b} \in \{0,1\}^N$, the elements $\{\mathbf{v}_{[i]} | i \in [N], \mathbf{b}_{[i]} = 1\}$ are distinct. This protocol exploits the formulae of logarithmic derivative $Df(X) = f(X) \cdot D(\log(f(X)))$ and $D(\log \prod_i (X - v_i)) = \sum_i \frac{1}{X - v_i}$ from calculus. The vectors $\mathbf{f}$ and $\mathbf{u}$ in this protocol are constructed so that their last entries are exactly $f(\beta)$ and $g(\beta)/f(\beta)$, respectively. Based on the $\mathsf{DstFull}$ protocol, we construct the $\mathsf{FullSort}$ protocol, as presented in Algorithm 4.

Theorem 5. *The $\mathsf{DstFull}$ protocol in Algorithm 4 is a PIOP for the vector language*

$$\mathcal{R}_{\mathsf{Distinct}} := \left\{ (\mathbf{v} \in \mathbb{F}^N, \mathbf{b} \in \{0,1\}^N) \;\middle|\; \forall (i,j) \in [N]^2, (\mathbf{b}_{[i]}, \mathbf{b}_{[j]}) \neq (1,1) \vee \mathbf{v}_{[i]} \neq \mathbf{v}_{[j]} \right\}$$

with completeness error $e_{c,\mathsf{DstFull}} = N/|\mathbb{F}|$ and soundness error $e_{s,\mathsf{DstFull}} = (4N + D + 4)/|\mathbb{F}|$ where D is the degree bound of PIOP.

Proof. Consider the following statements

$$(\mathbf{v}, \mathbf{b}) \in \mathcal{R}_{\mathsf{Distinct}} \tag{37}$$

$$\mathbf{b} \in \{0,1\}^N \tag{38}$$

$$\mathsf{V} \text{ accepts } \mathbf{b} \circ \mathbf{b} = \mathbf{b} \tag{39}$$

$$\gcd(f(X), Df(X)) = 1 \tag{40}$$

$$s(X)f(X) + t(X)Df(X) = 1 \tag{41}$$

$$s(\beta)f(\beta) + t(\beta)Df(\beta) = 1 \tag{42}$$

$$\forall i \in [N], \mathbf{f}_{[i]} = \begin{cases} (\beta - \mathbf{v}_{[1]}) \cdot \mathbf{b}_{[1]} + 1 - \mathbf{b}_{[1]}, & i = 1 \\ \mathbf{f}_{[i-1]} \cdot \left((\beta - \mathbf{v}_{[i]}) \cdot \mathbf{b}_{[i]} + 1 - \mathbf{b}_{[i]}\right), & i > 1 \end{cases} \tag{43}$$

$$\forall i \in [N], \mathbf{u}_{[i]} = \begin{cases} \frac{\mathbf{b}_{[1]}}{\beta - \mathbf{v}_{[1]}}, & i = 1 \\ \mathbf{u}_{[i-1]} + \frac{\mathbf{b}_{[i]}}{\beta - \mathbf{v}_{[i]}}, & i > 1 \end{cases} \tag{44}$$

$$\mathsf{V} \text{ accepts } \mathbf{e}_1 \circ ((\boldsymbol{\beta} - \mathbf{v}) \circ \mathbf{b} + \mathbf{1} - \mathbf{b} - \mathbf{f}) = \mathbf{0} \tag{45}$$

$$\mathsf{V} \text{ accepts } (\mathbf{id} - \mathbf{1}) \circ \left(\mathbf{f}^{\leftarrow -1} \circ ((\boldsymbol{\beta} - \mathbf{v}) \circ \mathbf{b} + \mathbf{1} - \mathbf{b}) - \mathbf{f}\right) = \mathbf{0} \tag{46}$$

$$\mathsf{V} \text{ accepts } (\mathbf{id} - \mathbf{1}) \circ ((\mathbf{u} - \mathbf{u}^{\leftarrow -1}) \circ (\boldsymbol{\beta} - \mathbf{v}) - \mathbf{b}) = \mathbf{0} \tag{47}$$

$$\mathbf{f}_{[N]} = f(\beta) \tag{48}$$

$$\mathbf{u}_{[N]} = Df(\beta)/f(\beta) \tag{49}$$

$$\mathsf{V} \text{ accepts } (\mathbf{f} \circ (s(\beta) \cdot \mathbf{1} + t(\beta) \cdot \mathbf{u}) - \mathbf{1}) \circ \mathbf{e}_N = \mathbf{0} \tag{50}$$

Completeness follows from the sequence of inductions

- (37) ⇒ (40) (38);
- (38) ⇒ (39);
- (40) ⇒ (41) ⇒ (42);
- (43) (44) (38) ⇒ (45) (46) (47);
- (43) (44) (38) ⇒ (48) (49) where (44) ⇒ (49) follows from the identity $Df(X)/f(X) = D(\log f(X)) = \sum_{i=1}^{n} \frac{\mathbf{b}_{[i]}}{X - \mathbf{v}_{[i]}}$, and (44) is valid if and only if $f(\beta) \neq 0$ which fails with probability bounded by $N/|\mathbb{F}|$;
- (48) (49) (42) ⇒ (50).

The completeness error follows from the union bound.

Soundness follows from the sequence of inductions

- (39) ⇒ (38), this and the following two implications together fail with probability bounded by $(3N + 4)/|\mathbb{F}|$;
- (38) (45) (46) (47) ⇒ (38) (43) (44) ⇒ (48) (49);
- (50) (49) (48) ⇒ (42);
- (42) ⇒ (41) ⇒ (40) where (42) ⇒ (41) fails with probability bounded by $(N + D)/|\mathbb{F}|$;
- (40) (38) ⇒ (37).

The soundness error follows from the union bound. □

Theorem 6. *The* FullSort *protocol in Algorithm 4 is a PIOP for the relation* $\mathcal{R}^{\mathbb{F}}_{\mathsf{CN}}(\mathbf{addr})$ *with completeness error* $e_{c,\mathsf{Perm}} + e_{c,\mathsf{Lookup}} + e_{c,\mathsf{DstFull}}$ *and soundness error* $(2N + 1)/|\mathbb{F}| + e_{s,\mathsf{Perm}} + e_{s,\mathsf{Lookup}} + e_{s,\mathsf{DstFull}}$.

Algorithm 4 Canonical Sort for Full Address Space

procedure $\mathsf{DstFull}(\mathbf{v}, \mathbf{b})$

P locally computes $f(X) = \prod_{i\in[N],\mathbf{b}_{[i]}=1}(X - \mathbf{v}_{[i]})$, $g(X) = Df(X)$;

P sends $s(X), t(X)$ such that $s(X)f(X) + t(X)g(X) = 1$ computed as follows:

- Let $\{r_1, \cdots, r_k\} = \{\mathbf{v}_{[i]} | i \in [N], \mathbf{b}_{[i]} = 1\}$;
- Compute $t_i = 1/g(r_i)$ for $i \in [k]$ by multi-point evaluation and batched inversion;
- Interpolate $t(X)$ such that $t(r_i) = t_i$, and compute $s(X) = \frac{1-t(X)g(X)}{f(X)}$;

V samples $\beta \xleftarrow{\$} \mathbb{F}$ and queries for $s(\beta), t(\beta)$;

P sends $\mathbf{f}, \mathbf{u}$ where $\mathbf{f}_{[i]} = \begin{cases} (\beta - \mathbf{v}_{[1]}) \cdot \mathbf{b}_{[1]} + 1 - \mathbf{b}_{[1]}, & i = 1 \\ \mathbf{f}_{[i-1]} \cdot \left((\beta - \mathbf{v}_{[i]}) \cdot \mathbf{b}_{[i]} + 1 - \mathbf{b}_{[i]}\right), & i > 1 \end{cases}$ and $\mathbf{u}_{[i]} = \begin{cases} \frac{\mathbf{b}_{[1]}}{\beta - \mathbf{v}_{[1]}}, \text{ for } i = 1 \\ \mathbf{u}_{[i-1]} + \frac{\mathbf{b}_{[i]}}{\beta - \mathbf{v}_{[i]}}, \text{ for } i > 1 \end{cases}$;

V checks $\mathbf{b} \circ \mathbf{b} = \mathbf{b}$ and $(\mathbf{f} \circ (s(\beta) \cdot \mathbf{1} + t(\beta) \cdot \mathbf{u}) - \mathbf{1}) \circ \mathbf{e}_N = \mathbf{0}$;

V checks $\mathbf{e}_1 \circ ((\boldsymbol{\beta} - \mathbf{v}) \circ \mathbf{b} + \mathbf{1} - \mathbf{b} - \mathbf{f}) = \mathbf{0}$;

V checks $(\mathbf{id} - \mathbf{1}) \circ \left(\mathbf{f}^{\leftarrow -1} \circ ((\boldsymbol{\beta} - \mathbf{v}) \circ \mathbf{b} + \mathbf{1} - \mathbf{b}) - \mathbf{f}\right) = \mathbf{0}$;

V checks $(\mathbf{id} - \mathbf{1}) \circ ((\mathbf{u} - \mathbf{u}^{\leftarrow -1}) \circ (\boldsymbol{\beta} - \mathbf{v}) - \mathbf{b}) = \mathbf{0}$.

procedure $\mathsf{FullSort}(\mathbf{op}, \mathbf{addr}, \mathbf{val}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}})$

I sends $\mathbf{incs} = (1, 2, \cdots, N)$;

P sends $\widetilde{\mathbf{incs}}$ as in Algorithm 3 and $\mathbf{b}$ computed from $\widetilde{\mathbf{addr}}$ as in Lemma 3;

V checks $\mathbf{b} \circ \mathbf{e}_1 = \mathbf{e}_1$ and $(\widetilde{\mathbf{addr}} - \widetilde{\mathbf{addr}}^{\leftarrow -1}) \circ (\mathbf{1} - \mathbf{b}) = \mathbf{0}$;

V checks $(\mathbf{incs}, \mathbf{op}, \mathbf{addr}, \mathbf{val}) \sim (\widetilde{\mathbf{incs}}, \widetilde{\mathbf{op}}, \widetilde{\mathbf{addr}}, \widetilde{\mathbf{val}})$ and $\widetilde{\mathbf{incs}} - \widetilde{\mathbf{incs}}^{\leftarrow -1} \subset_{\mathbf{1-b}} \mathbf{incs}$;

P and V run the $\mathsf{DstFull}$ protocol with inputs $\widetilde{\mathbf{addr}}, \mathbf{b}$.

Proof. Consider the following statements

$$(\mathbf{op}, \mathbf{addr}, \mathbf{val}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \in \mathcal{R}^{\mathbb{F}}_{\mathsf{CN}}(\mathbf{addr}) \tag{51}$$

$\exists$total order $\preceq_{\mathbf{addr}}$ over $\mathsf{Elems}(\mathbf{addr})$ s.t.

$$(\tilde{\mathbf{incs}}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \text{ is a sorting of } (\mathbf{incs}, \mathbf{op}, \mathbf{addr}, \mathbf{val}) \text{ with } \preceq_{\mathbf{incs}}, \preceq_{\mathbf{addr}} \tag{52}$$

$$\exists\text{total order } \preceq_{\mathbf{addr}} \text{ over } \mathsf{Elems}(\mathbf{addr}) \text{ s.t. } \tilde{\mathbf{addr}} \text{ is sorted by } \preceq_{\mathbf{addr}} \tag{53}$$

$$\forall t \in [N], \mathbf{b}_{[t]} = \begin{cases} 1, \text{ if } t = 1 \vee \tilde{\mathbf{addr}}_{[t-1]} \neq \tilde{\mathbf{addr}}_{[t]} \\ 0, \text{ otherwise} \end{cases} \tag{54}$$

$$\mathsf{V} \text{ accepts } \mathbf{b} \circ \mathbf{e}_1 = \mathbf{e}_1 \tag{55}$$

$$\mathsf{V} \text{ accepts } (\tilde{\mathbf{addr}} - \tilde{\mathbf{addr}}^{\leftarrow -1}) \circ (\mathbf{1} - \mathbf{b}) = \mathbf{0} \tag{56}$$

$$(\tilde{\mathbf{addr}}, \mathbf{b}) \in \mathcal{R}_{\mathsf{Distinct}} \tag{57}$$

$$\mathsf{V} \text{ accepts in } \mathsf{DstFull} \text{ with input } \tilde{\mathbf{addr}}, \mathbf{b} \tag{58}$$

$$(\mathbf{incs}, \mathbf{op}, \mathbf{addr}, \mathbf{val}) \sim (\tilde{\mathbf{incs}}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \tag{59}$$

$$\mathsf{V} \text{ accepts } (\mathbf{incs}, \mathbf{op}, \mathbf{addr}, \mathbf{val}) \sim (\tilde{\mathbf{incs}}, \tilde{\mathbf{op}}, \tilde{\mathbf{addr}}, \tilde{\mathbf{val}}) \tag{60}$$

$$\forall t \in [2..N], \tilde{\mathbf{incs}}_{[t]} - \tilde{\mathbf{incs}}_{[t-1]} \in [N] \vee \tilde{\mathbf{addr}}_{[t]} \neq \tilde{\mathbf{addr}}_{[t-1]} \tag{61}$$

$$\mathsf{V} \text{ accepts } \tilde{\mathbf{incs}} - \tilde{\mathbf{incs}}^{\leftarrow -1} \subset_{\mathbf{1}-\mathbf{b}} \mathbf{incs} \tag{62}$$

Completeness follows from the sequence of inductions

- (51) ⇒ (52);
- (52) ⇒ (53) (61) (59);
- (53) (54) ⇒ (57) where (53) (54) ⇒ (57) follows from Lemma 3;
- (54) ⇒ (55) (56);
- (57) ⇒ (58) fails with probability $e_{c,\mathsf{DstFull}}$;
- (59) ⇒ (60) fails with probability $e_{c,\mathsf{Perm}}$;
- (54) (61) ⇒ (62) fails with probability $e_{c,\mathsf{Lookup}}$.

The completeness error is the sum of these probabilities by union bound.

Soundness follows from the sequence of inductions

- (58) ⇒ (57);
- (56) (55) (57) ⇒ (54) follows from the observation that if $\mathbf{b}_{[t]} = 1$ for some t such that $\tilde{\mathbf{addr}}_{[t-1]} = \tilde{\mathbf{addr}}_{[t]}$, we can find some $t' < t$ with $\tilde{\mathbf{addr}}_{[t']} \neq \tilde{\mathbf{addr}}_{[t]}$ but $\mathbf{b}_{[t']} = 1$, which contradicts $(\tilde{\mathbf{addr}}, \mathbf{b}) \in \mathcal{R}_{\mathsf{Distinct}}$. The two vector equation checks together fail with probability $(2N+1)/|\mathbb{F}|$;
- (57) (54) ⇒ (53) fails with probability $e_{s,\mathsf{DstFull}}$;
- (54) (62) ⇒ (61) fails with probability $e_{s,\mathsf{Lookup}}$;
- (60) ⇒ (59) fails with probability $e_{s,\mathsf{Perm}}$;
- (61) (53) (59) ⇒ (52) ⇒ (51).

The soundness error follows from the union bound. □

5 Permem: New Construction with the Full Address Space

Inspired by our systemization of existing protocols, we propose several ways to optimizing the performance of previous works. First, observing the bottleneck of the sorting paradigm, in which the sorted vectors inevitably cost four online polynomials, we propose an alternative method for addressing the history-dependency of memory, called *address cycle method*, that avoids these costs. Then, using this method, we propose Permem, a more efficient MCC that supports the full memory address space. Finally, observing the significant impact of lookup arguments on the efficiency of MCCs, which is also an independently important target of research [GW20,ZBK+22,PK22], we propose a more efficient lookup argument gcq.

5.1 Address Cycle Method and Permem

As an alternative to the sorting paradigm, we propose a new method for addressing the history-dependency issue of MCC, which we call the address cycle method. This method is extracted from Arya [BCG+18]. Using this method, we construct a new MCC protocol, named Permem.

The insight behind the address cycle method is the following observation on the definition of $\mathcal{R}_{\mathsf{Mem}}$. The consistency of the memory trace $\mathbf{op}, \mathbf{addr}, \mathbf{val}$ is equivalent to a series of equality checks over elements in $\mathbf{val}$, where these equality checks are determined by $\mathbf{op}$ and $\mathbf{addr}$. The equality checks can be accomplished using the technique of PLONK [GWC19] that checks if $\mathbf{val}$ remains invariant under some permutation σ. However, unlike in PLONK, the permutation σ here is kept secret from the verifier. As a result, the prover must demonstrate, in the online phase, that the committed permutation is consistent with $\mathbf{op}$ and $\mathbf{addr}$.

Multiple permutations exist that capture the equality checks induced by $\mathcal{R}_{\mathsf{Mem}}$. We choose the following one that is conceptually simpler.

Definition 8 (Previous Access Permutation). *The* previous access permutation *of* $\mathbf{addr} \in \mathbb{F}^N$ *is the permutation over* $[N]$ *defined by*

$$\sigma_{\mathbf{addr}}(t) = \begin{cases} \mathsf{prev}(t; \mathbf{addr}), & \text{if } \mathsf{prev}(t; \mathbf{addr}) \neq \perp \\ \max\{j \in [N] | \mathbf{addr}_{[j]} = \mathbf{addr}_{[t]}\}, & \text{otherwise} \end{cases},$$

where $\mathsf{prev}(t; \mathbf{addr})$ *is defined as in Definition 5.*

Given this permutation, $(\mathbf{op}, \mathbf{addr}, \mathbf{val}) \in \mathcal{R}_{\mathsf{Mem}}$ is equivalent to the following statement: the vector $\sigma_{\mathbf{addr}}(\mathbf{val}) - \mathbf{val}$ should be zero except at positions where $\mathsf{prev}(t; \mathbf{addr}) \neq \perp$ or $\mathbf{op}_{[t]} = 1$, as demonstrated in the following theorem.

Theorem 7. *Given* $\mathbf{op} \in \{0,1\}^N$, $\mathbf{addr} \in \mathcal{A}^N, \mathbf{val} \in \mathbb{F}^N$. *Let* $\mathbf{first} \in \{0,1\}^N$ *be defined as* $\mathbf{first}_{[t]} = 1$ *if and only if* $\mathbf{addr}_{[t]} \notin \mathbf{addr}_{[1..t-1]}$. *Then* $(\mathbf{op}, \mathbf{addr}, \mathbf{val}) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}}$ *if and only if* $(\sigma_{\mathbf{addr}}(\mathbf{val}) - \mathbf{val}) \circ (\mathbf{1} - \mathbf{first}) \circ (\mathbf{1} - \mathbf{op}) = \mathbf{0}$, *where* $\sigma_{\mathbf{addr}}$ *is the previous access permutation of* $\mathbf{addr}$.

Proof. Note that $\mathbf{first}_{[t]} = 1$ is equivalent to $\mathsf{prev}(t; \mathbf{addr}) = \perp$. Then according to definitions of $\mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}}$ and $\sigma_{\mathbf{addr}}$, $(\mathbf{op}, \mathbf{addr}, \mathbf{val}) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}}$ if and only if for every $t \in [N]$, $\mathbf{first}_{[t]} = 1$ or $\mathbf{op}_{[t]} = 1$ or $\mathbf{val}_{[t]} = \mathbf{val}_{[\sigma_{\mathbf{addr}}(t)]}$, which is equivalent to the claimed vector equation. □

The following lemma shows the properties of $\sigma_{\mathbf{addr}}$ with which the prover can prove to the verifier that a committed permutation is indeed $\sigma_{\mathbf{addr}}$. Theorem 7 and Lemma 5 together leads to our new MCC, which we call the address cycle method, in Algorithm 5, where the $\mathsf{Distinct}$ protocol is yet to instantiate. Note that the notation "$\subset_{\mathbf{1}-\mathbf{first}}$" flips "$\subset_{\mathbf{first}}$" in the sense that the lookup check is only applied to the subset of positions where $\mathbf{first}_{[i]} = 0$ instead of $\mathbf{first}_{[i]} = 1$.

Lemma 5. *Given* $\mathbf{addr} \in \mathbb{F}^N$, *the previous access permutation* $\sigma_{\mathbf{addr}}$ *of* $\mathbf{addr}$ *is the unique permutation over* $[N]$ *that satisfies the following properties: (a)* $\sigma_{\mathbf{addr}}(\mathbf{addr}) = \mathbf{addr}$; *and (b) for any pair* $t \neq t'$ *such that* $\sigma_{\mathbf{addr}}(t) \geq t$ *and* $\sigma_{\mathbf{addr}}(t') \geq t'$, $\mathbf{addr}_{[t]} \neq \mathbf{addr}_{[t']}$.

Proof. If $\sigma_{\mathbf{addr}}$ is the previous access permutation, then the first property is satisfied by definition. For any $t < t'$ such that $\mathbf{addr}_{[t]} = \mathbf{addr}_{[t']}$, $\mathsf{prev}(t'; \mathbf{addr})$ must be smaller than t', which leads to the second property.

Next, we show the uniqueness. Suppose σ satisfies the two properties. For any $a \in \mathsf{Elems}(\mathbf{addr})$, let $\mathcal{I}_a = \{t | \mathbf{addr}_{[t]} = a\}$, then by the first property, $\mathcal{I}_a$ is closed under σ. Suppose $\mathcal{I}_a = \{t_1, t_2, \cdots, t_k\}$ such that $t_1 < t_2 < \cdots < t_k$. By the second property, there is at most one $t \in \mathcal{I}_a$ such that $\sigma(t) \geq t$. Obviously, this t must be t_1. Since $\sigma(t_2) < t_2$, $\sigma(t_2)$ can only be t_1. Continuing this process, we conclude that there is a unique choice of $\sigma(t)$ for every $t \in \mathcal{I}_a$. Since this argument holds for every $a \in \mathsf{Elems}(\mathbf{addr})$, we conclude that σ is unique. □

Theorem 8. *Let $\mathcal{A} \subset \mathbb{F}$ be the set of addresses. Assume that $\mathsf{Distinct}_{\mathcal{A}}$ is a PIOP for the vector language*

$$\mathcal{R}_{\mathsf{Distinct},\mathcal{A}} := \left\{ (\mathbf{v} \in \mathbb{F}^N, \mathbf{b} \in \{0,1\}^N) \;\middle|\; \begin{array}{l} \forall (i,j) \in [N]^2, \text{ if } \mathbf{b}_{[i]} = \mathbf{b}_{[j]} = 1 \\ \text{then } (\mathbf{v}_{[i]}, \mathbf{v}_{[j]}) \in \mathcal{A}^2 \wedge \mathbf{v}_{[i]} \neq \mathbf{v}_{[j]} \end{array} \right\}$$

with completeness error $e_{c,\mathsf{Distinct}}$ and soundness error $e_{s,\mathsf{Distinct}}$, then the $\mathsf{ACMCC}_{\mathcal{A}}$ protocol in Algorithm 5 is a PIOP for $\mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}}$ with completeness error $e_{c,\mathsf{Perm}} + e_{c,\mathsf{Lookup}} + e_{c,\mathsf{Distinct}}$ and soundness error $(3N+1)/|\mathbb{F}| + e_{s,\mathsf{Perm}} + e_{s,\mathsf{Lookup}} + e_{s,\mathsf{Distinct}}$.

Proof. Consider the following statements

$$(\mathbf{op}, \mathbf{addr}, \mathbf{val}) \in \mathcal{R}^{\mathcal{A}}_{\mathsf{Mem}} \tag{63}$$

$$\mathbf{op} \in \{0,1\}^N \tag{64}$$

$$\mathsf{V} \text{ accepts } \mathbf{op} \circ \mathbf{op} = \mathbf{op} \tag{65}$$

$$\sigma_{\mathbf{addr}} \text{ is previous access permutation of } \mathbf{addr} \tag{66}$$

$$\sigma_{\mathsf{addr}}(\mathbf{addr}) = \mathbf{addr} \tag{67}$$

$$\sigma_{\mathsf{addr}}(\mathbf{val}) = \mathbf{sval} \tag{68}$$

$$\forall t \in [N], \mathsf{prev}(t; \mathbf{addr}) = \perp \vee \sigma_{\mathsf{addr}}(t) < t \tag{69}$$

$$(\boldsymbol{\sigma}_{\mathsf{addr}}, \mathbf{addr}, \mathbf{sval}) \sim (\mathbf{incs}, \mathbf{addr}, \mathbf{val}) \tag{70}$$

$$\mathsf{V} \text{ accepts } (\boldsymbol{\sigma}_{\mathsf{addr}}, \mathbf{addr}, \mathbf{sval}) \sim (\mathbf{incs}, \mathbf{addr}, \mathbf{val}) \tag{71}$$

$$\mathbf{first}_{[t]} = \begin{cases} 1, \mathbf{addr}_{[t]} \notin \mathbf{addr}_{[1..t-1]} \\ 0, \text{otherwise} \end{cases} \tag{72}$$

$$(\mathbf{1} - \mathbf{op}) \circ (\mathbf{1} - \mathbf{first}) \circ (\mathbf{sval} - \mathbf{val}) = \mathbf{0} \tag{73}$$

$$\mathsf{V} \text{ accepts } (\mathbf{1} - \mathbf{op}) \circ (\mathbf{1} - \mathbf{first}) \circ (\mathbf{sval} - \mathbf{val}) = \mathbf{0} \tag{74}$$

$$\mathbf{incs} - \boldsymbol{\sigma}_{\mathsf{addr}} \subset_{\mathbf{1}-\mathbf{first}} \mathbf{incs} \tag{75}$$

$$\mathsf{V} \text{ accepts } \mathbf{incs} - \boldsymbol{\sigma}_{\mathsf{addr}} \subset_{\mathbf{1}-\mathbf{first}} \mathbf{incs} \tag{76}$$

$$(\mathbf{addr}, \mathbf{first}) \in \mathcal{R}_{\mathsf{Distinct},\mathcal{A}} \tag{77}$$

$$\mathsf{V} \text{ accepts in } \mathsf{Distinct}_{\mathcal{A}} \text{ with inputs } \mathbf{addr}, \mathbf{first} \tag{78}$$

Completeness follows from the sequence of inductions

– (63) ⇒ (64) ⇒ (65);

- (66) ⇒ (67) (68) (69);
- (67) (68) ⇒ (70) ⇒ (71) where (70) ⇒ (71) fails with probability $e_{c,\mathsf{Perm}}$;
- (63) (72) ⇒ (73) ⇒ (74);
- (69) (72) ⇒ (75) (77);
- (75) ⇒ (76) fails with probability $e_{c,\mathsf{Lookup}}$;
- (77) ⇒ (78) fails with probability $e_{c,\mathsf{Distinct}}$.

The completeness error is the sum of these probabilities by the union bound.

Soundness follows from the sequence of inductions

- (71) ⇒ (70) ⇒ (67) (68) where (71) ⇒ (70) fails with probability $e_{s,\mathsf{Perm}}$;
- (78) ⇒ (77) fails with probability $e_{s,\mathsf{Distinct}}$;
- (76) ⇒ (75) fails with probability $e_{s,\mathsf{Lookup}}$;
- (67) (75) (77) ⇒ (66);
- (74) ⇒ (73), which, together with the next induction, fails with probability $(3N+1)/|\mathbb{F}|$;
- (65) ⇒ (64);
- (66) (68) (73) (64) ⇒ (63).

The soundness error follows from the union bound. □

Algorithm 5 Address Cycle Method

procedure $\mathsf{ACMCC}_{\mathcal{A}}(\mathbf{op}, \mathbf{addr}, \mathbf{val})$
- I sends $\mathbf{incs} = (1, 2, \cdots, N)$;
- P computes σ_{addr} as in Definition 8;
- P sends $\boldsymbol{\sigma}_{\mathsf{addr}} := (\sigma_{\mathsf{addr}}(1), \cdots, \sigma_{\mathsf{addr}}(N))$;
- P sends $\mathbf{sval} := \sigma_{\mathbf{addr}}(\mathbf{val})$ (which is $(\mathbf{val}_{[\sigma_{\mathbf{addr}}(1)]}, \cdots, \mathbf{val}_{[\sigma_{\mathbf{addr}}(N)]})$)
- P sends $\mathbf{first}$ where $\mathbf{first}_{[t]} = 1$ if $\mathbf{addr}_{[t]} \notin \mathbf{addr}_{[1..t-1]}$, otherwise $\mathbf{first}_{[t]} = 0$;
- V checks $(\boldsymbol{\sigma}_{\mathsf{addr}}, \mathbf{addr}, \mathbf{sval}) \sim (\mathbf{incs}, \mathbf{addr}, \mathbf{val})$ and $\mathbf{incs} - \boldsymbol{\sigma}_{\mathsf{addr}} \subset_{\mathbf{1}-\mathbf{first}} \mathbf{incs}$;
- V checks $\mathbf{op} \circ \mathbf{op} = \mathbf{op}$ and $(\mathbf{1} - \mathbf{first}) \circ (\mathbf{1} - \mathbf{op}) \circ (\mathbf{sval} - \mathbf{val}) = \mathbf{0}$;
- P and V run the $\mathsf{Distinct}_{\mathcal{A}}$ protocol (Theorem 8), with inputs $\mathbf{addr}$ and $\mathbf{first}$.

We then provide two instantiations of the address cycle method.

Full memory address space. Note that $\mathcal{R}_{\mathsf{Distinct},\mathcal{A}}$ in Theorem 8 becomes the same as $\mathcal{R}_{\mathsf{Distinct}}$ in Theorem 5 when $\mathcal{A} = \mathbb{F}$. Therefore, the DstFull protocol in Algorithm 4 satisfies the requirement of $\mathsf{Distinct}_{\mathbb{F}}$. Placing this protocol directly into the address cycle method produces a new MCC which we name Permem.

Linear-size memory address space. When the memory address space $\mathcal{A}$ is the set $[M]$ for memory size $M \approx N$, as in Arya, there exists a simpler and more efficient construction of $\mathsf{Distinct}_{\mathcal{A}}$, denoted by DistLinear. Combining this distinctness check with the address cycle method gives us AryaMem, which roughly follows the same pattern as the memory component of Arya but adopts many PIOP-specific techniques. We summarize this distinctness check as the DistLinear protocol in Algorithm 6.

Algorithm 6 Distinctness Check of Arya

procedure $\mathsf{DistLinear}(\mathbf{v}, \mathbf{b})$

I sends $\mathbf{incs} = (1, 2, \cdots, N)$;

P sends $\mathbf{comp}$ such that $\mathbf{comp}_{[t]} = 0$ for every $t \in \mathcal{T} := \{t \in [N] : \mathbf{b}_{[t]} = 1\}$, and for those $t \notin \mathcal{T}$, $\mathbf{comp}_{[t]}$ are distinct from each other and from all entries in $\mathbf{v}$;

V checks $\mathbf{b} \circ \mathbf{b} = \mathbf{b}$ and $\mathbf{v} \circ \mathbf{b} + \mathbf{comp} \circ (\mathbf{1} - \mathbf{b}) \sim \mathbf{incs}$.

Theorem 9. *The* $\mathsf{DistLinear}$ *protocol in Algorithm 6 is a PIOP for the vector relation* $\mathcal{R}_{\mathsf{Distinct},[N]}$ *(defined in Theorem 8) with completeness error* $e_{c,\mathsf{Perm}}$ *and soundness error* $2N/|\mathbb{F}| + e_{s,\mathsf{Perm}}$.

Proof. Consider the following statements

$$(\mathbf{v}, \mathbf{b}) \in \mathcal{R}^{[N]}_{\mathsf{Distinct}} \tag{79}$$

$$\mathbf{b} \in \{0, 1\}^N \tag{80}$$

Verifier accepts (at random z) the polynomial identity corresponding to

$$\mathbf{b} \circ \mathbf{b} = \mathbf{b} \tag{81}$$

$$\{\mathbf{v}_{[t]} : \mathbf{b}_{[t]} = 1\} \subset [M] \tag{82}$$

$$\{\mathbf{v}_{[t]} : \mathbf{b}_{[t]} = 1\} \text{ are distinct} \tag{83}$$

$$\exists \tilde{\mathbf{v}} \in \mathbb{F}^N, \mathbf{v} \sim \mathbf{incs} \wedge \mathbf{v}_{[t]} = \tilde{\mathbf{v}}_{[t]} \forall t \text{ where } \mathbf{b}_{[t]} = 1 \tag{84}$$

$$\mathbf{v} + \mathbf{diff} \circ (\mathbf{1} - \mathbf{b}) \sim \mathbf{incs} \tag{85}$$

$$\text{Verifier accepts } \mathbf{v} + \mathbf{diff} \circ (\mathbf{1} - \mathbf{b}) \sim \mathbf{incs} \tag{86}$$

Completeness follows from the sequence of inductions

- (79) ⇒ (80)(82)(83);
- (80) ⇒ (81);
- (82)(83) ⇒ (84) ⇒ (85) ⇒ (86) where the last step fails with probability at most $e_{c,\mathsf{Perm}}$.

Therefore, the completeness error is $e_{c,\mathsf{Perm}}$.

Soundness follows from the sequence of inductions

- (86) ⇒ (85) ⇒ (84) ⇒ (82)(83) where the first step fails with probability at most $e_{s,\mathsf{Perm}}$;
- (81) ⇒ (80) which fails with probability at most $2N/|\mathbb{F}|$;
- (80)(82)(83) ⇒ (79).

The completeness error is given by the union bound. □

5.2 Grand-Sum-based Lookup Argument

Observing that the lookup argument presents an influential factor in the efficiency of most MCCs, we provide gcq, in Algorithm 7, a novel lookup argument with improved performance. Our protocol takes inspiration from cq [EFG22] and differs from cq by replacing the univariate sumcheck with the grand-sum check, resulting in a smaller number of online polynomials. Our protocol assumes that the invoker will ensure that the input vector $\mathbf{b}$ is a boolean vector, as is the case in the MCCs. Our protocol exploits the following technique from the grand-sum check: a vector $\mathbf{a}$ satisfies $\sum_{i=1}^{N} \mathbf{a}_{[i]} = 0$ if and only if there exists vector $\mathbf{c}$ such that $\mathbf{a} = \mathbf{c}^{\to 1} - \mathbf{c}$. Therefore, we can simply eliminate $\mathbf{a}$ from the protocol, and simulate it using $\mathbf{c}^{\to 1} - \mathbf{c}$ wherever $\mathbf{a}$ is needed.

Theorem 10. *Assume that* $\mathbf{b} \in \{0,1\}^N$ *is guaranteed. The* single-gcq *protocol in Algorithm 7 is a PIOP for* $\mathcal{R}_{\mathsf{Lookup}}^{(m=1)}$ *(defined in Section 2.5) with completeness error* $2N/|\mathbb{F}|$ *and soundness error* $7N/|\mathbb{F}|$*. The* gcq *protocol is a PIOP for* $\mathcal{R}_{\mathsf{Lookup}}$ *with completeness error* $2N/|\mathbb{F}|$ *and soundness error* $(m+8N)/|\mathbb{F}|$*.*

Proof. Consider the following statements

$$(\mathbf{u}, \mathbf{v}, \mathbf{b}) \in \mathcal{R}_{\mathsf{Lookup}}^{(m=1)} \tag{87}$$

$$\forall i \in [N], m_i = |\{j \in [N] | \mathbf{u}_{[j]} = \mathbf{v}_{[i]}, \mathbf{b}_{[j]} = 1\}| \tag{88}$$

$$\prod_{j=1}^{N} (\mathbf{u}_{[j]} + X)^{\mathbf{b}_{[j]}} = \prod_{j=1}^{N} (\mathbf{v}_{[j]} + X)^{\mathbf{m}_{[j]}} \tag{89}$$

$$\sum_{j=1}^{N} \frac{\mathbf{b}_{[j]}}{\mathbf{u}_{[j]} + X} = \sum_{j=1}^{N} \frac{\mathbf{m}_{[j]}}{\mathbf{v}_{[j]} + X} \tag{90}$$

$$\sum_{j=1}^{N} \frac{\mathbf{b}_{[j]} \cdot \prod_{k=1}^{N} (\mathbf{u}_{[k]} + X)(\mathbf{v}_{[k]} + X)}{\mathbf{u}_{[j]} + X} = \sum_{j=1}^{N} \frac{\mathbf{m}_{[j]} \cdot \prod_{k=1}^{N} (\mathbf{u}_{[k]} + X)(\mathbf{v}_{[k]} + X)}{\mathbf{v}_{[j]} + X} \tag{91}$$

$$\sum_{j=1}^{N} \frac{\mathbf{b}_{[j]} \cdot \prod_{k=1}^{N} (\mathbf{u}_{[k]} + \beta)(\mathbf{v}_{[k]} + \beta)}{\mathbf{u}_{[j]} + \beta} = \sum_{j=1}^{N} \frac{\mathbf{m}_{[j]} \cdot \prod_{k=1}^{N} (\mathbf{u}_{[k]} + \beta)(\mathbf{v}_{[k]} + \beta)}{\mathbf{v}_{[j]} + \beta} \tag{92}$$

$$\sum_{j=1}^{N} \frac{\mathbf{b}_{[j]}}{\mathbf{u}_{[j]} + \beta} = \sum_{j=1}^{N} \frac{\mathbf{m}_{[j]}}{\mathbf{v}_{[j]} + \beta} \tag{93}$$

$$\forall i \in [N], \mathbf{c}_{[i]} = \sum_{j=1}^{i} \left(\frac{\mathbf{b}_{[j]}}{\mathbf{u}_{[j]} + \beta} - \frac{\mathbf{m}_{[j]}}{\mathbf{v}_{[j]} + \beta} \right) \tag{94}$$

$$\mathbf{c}_{[N]} = 0 \tag{95}$$

$$\forall i \in [N], \mathbf{c}_{[i]} - \mathbf{c}_{[i-1]} = \frac{\mathbf{b}_{[i]}}{\mathbf{u}_{[i]} + \beta} - \frac{\mathbf{m}_{[i]}}{\mathbf{v}_{[i]} + \beta} \tag{96}$$

$$\forall i \in [N], (\mathbf{c}_{[i]} - \mathbf{c}_{[i-1]}) \cdot (\mathbf{u}_{[i]} + \beta) \cdot (\mathbf{v}_{[i]} + \beta) = \mathbf{b}_{[i]} \cdot (\mathbf{v}_{[i]} + \beta) - \mathbf{m}_{[i]} \cdot (\mathbf{u}_{[i]} - \beta) \quad (97)$$

Verifier accepts (at random z) the polynomial identity corresponding to

$$(\mathbf{u} + \boldsymbol{\beta}) \circ (\mathbf{v} + \boldsymbol{\beta}) \circ (\mathbf{c} - \mathbf{c}^{\leftarrow -1}) = \mathbf{b} \circ (\mathbf{v} + \boldsymbol{\beta}) - \mathbf{m} \circ (\mathbf{u} + \boldsymbol{\beta}) \quad (98)$$

Completeness of single-gcq follows from the sequences of inductions

- (87) ⇒ (89) ⇒ (90) ⇒ (93), where (90) ⇒ (93) fails when $-\beta \in \mathbf{u} \| \mathbf{v}$, which happens with probability at most $2N/|\mathbb{F}|$;
- (93) (94) ⇒ (95);
- (95) (94) ⇒ (96) ⇒ (97) ⇒ (98).

Therefore, the completeness error is $2N/|\mathbb{F}|$ by the union bound.

Soundness of single-gcq follows from the following sequences of inductions:

- (98) ⇒ (97) follows from Schwartz-Zippel Lemma and fails with probability at most $3N/|\mathbb{F}|$;
- (97) ⇒ (96) fails with probability at most $2N/|\mathbb{F}|$;
- (96) ⇒ (93), which follows from summing up both sides of (96) over $[N]$;
- (93) ⇒ (92) ⇒ (91) ⇒ (90), where (92) ⇒ (91) follows from Schwartz-Zippel Lemma and fails with probability at most $2N/|\mathbb{F}|$;
- (90) ⇒ (89) follows from the fact that the two sides of (90) are respectively the derivative of the logarithm of the two sides of (89), so (90) implies that the ratio between the two sides of (89) is a constant, which can only be one because both sides are monic;
- (89) ⇒ (87) by the assumption that $\mathbf{b} \in \{0,1\}^N$ is guaranteed.

The soundness error is $7N/|\mathbb{F}|$ by the union bound.

Completeness of gcq follows directly from that of single-gcq. Soundness of gcq follows from that of single-gcq, the fact that $\exists m_i$ such that the polynomial equation $\sum_{i=1}^{m} \sum_{j=1}^{N} (Y - \mathbf{u}_{i,[j]}) X^{i-1} = \sum_{i=1}^{m} \sum_{j=1}^{N} (Y - \mathbf{v}_{i,[j]})^{m_i} X^{i-1}$ holds if and only if $(\mathbf{u}_1, \cdots, \mathbf{u}_m, \mathbf{v}_1, \cdots, \mathbf{v}_m) \in \mathcal{R}_{\mathsf{Lookup}}$, and Schwartz-Zippel Lemma, which fails with probability $(m+N)/|\mathbb{F}|$, where $m+N$ is the total degree of the polynomials. □

Our protocol can be extended to achieve smaller amortized costs for k lookup arguments with the same superset $\mathbf{v}$. Specifically, the extended version requires $\lceil \frac{k+1}{2} \rceil$ online polynomial oracles, compared to $2k$ for naïvely invoking single-gcq by k times. The example for $k = 2$, called double-gcq, is presented in Algorithm 8.

Theorem 11. *The* double-gcq *protocol in Algorithm 8 is a PIOP for* $\mathcal{R}_{\mathsf{Lookup}}^{\mathsf{double}} = \left\{ (\mathbf{u}, \mathbf{u}', \mathbf{v}, \mathbf{b}, \mathbf{b}') \mid (\mathbf{u} \| \mathbf{u}') \subset_{\mathbf{b} \| \mathbf{b}'} \mathbf{v} \right\}$ *with completeness error* $3N/|\mathbb{F}|$ *and soundness error* $(10N + 2)/|\mathbb{F}|$.

Proof. Consider the following statements

$$(\mathbf{u}, \mathbf{u}', \mathbf{v}, \mathbf{b}, \mathbf{b}') \in \mathcal{R}_{\mathsf{Lookup}}^{(\mathsf{double})} \quad (99)$$

Algorithm 7 Lookup Argument

procedure $\mathsf{gcq}(\mathbf{u}_1, \cdots, \mathbf{u}_m, \mathbf{v}_1, \cdots, \mathbf{v}_m, \mathbf{b})$

V samples $\alpha \xleftarrow{\$} \mathbb{F}$;

P and V run the single-gcq protocol with inputs $\sum_{i=1}^{m} \alpha^{i-1}\mathbf{u}_i$, $\sum_{i=1}^{m} \alpha^{i-1}\mathbf{v}_i$, $\mathbf{b}$.

procedure $\mathsf{single\text{-}gcq}(\mathbf{u}, \mathbf{v}, \mathbf{b})$

P sends $\mathbf{m} = (m_i)_{i=1}^{N}$ where $m_i = |\{j \in [N] | \mathbf{u}_{[j]} = \mathbf{v}_{[i]}, \mathbf{b}_{[j]} = 1\}|$;

V samples $\beta \xleftarrow{\$} \mathbb{F}$;

P sends $\mathbf{c} := \left(\sum_{j=1}^{i} \left(\frac{\mathbf{b}_{[j]}}{\mathbf{u}_{[j]}+\beta} - \frac{\mathbf{m}_{[j]}}{\mathbf{v}_{[j]}+\beta}\right)\right)_{i=1}^{N}$;

V checks $(\mathbf{u}+\boldsymbol{\beta}) \circ (\mathbf{v}+\boldsymbol{\beta}) \circ (\mathbf{c} - \mathbf{c}^{\leftarrow -1}) = \mathbf{b} \circ (\mathbf{v}+\boldsymbol{\beta}) - \mathbf{m} \circ (\mathbf{u}+\boldsymbol{\beta})$.

Algorithm 8 Double Lookup Argument

procedure $\mathsf{double\text{-}gcq}(\mathbf{u}, \mathbf{u}', \mathbf{v}, \mathbf{b}, \mathbf{b}')$

P sends $\mathbf{m} = (m_i)_{i=1}^{N}$ where $m_i = |\{j\}_{\mathbf{u}_{[j]}=\mathbf{v}_{[i]}, \mathbf{b}_{[j]}=1}| + |\{j\}_{\mathbf{u}'_{[j]}=\mathbf{v}_{[i]}, \mathbf{b}'_{[j]}=1}|$;

V samples $\beta \xleftarrow{\$} \mathbb{F}$;

P sends $\widetilde{\mathbf{u}} := \left(\sum_{j=1}^{i} \frac{\mathbf{b}_{[j]}}{\mathbf{u}_{[j]}+\beta} + \frac{\mathbf{b}'_{[j]}}{\mathbf{u}'_{[j]}+\beta}\right)_{i=1}^{N}$ and $\widetilde{\mathbf{v}} := \left(\sum_{j=1}^{i} \frac{\mathbf{m}_{[j]}}{\mathbf{v}_{[j]}+\beta}\right)_{i=1}^{N}$;

V checks $\mathbf{e}_N \circ (\widetilde{\mathbf{u}} - \widetilde{\mathbf{v}}) = \mathbf{0}$;

V checks $(\mathbf{u}+\boldsymbol{\beta}) \circ (\mathbf{u}'+\boldsymbol{\beta}) \circ (\widetilde{\mathbf{u}} - (\mathbf{1}-\mathbf{e}_1) \circ \widetilde{\mathbf{u}}^{\leftarrow -1}) = \mathbf{b}' \circ \mathbf{u} + \mathbf{b} \circ \mathbf{u}' + \beta \cdot (\mathbf{b}+\mathbf{b}')$;

V checks $(\mathbf{v}+\boldsymbol{\beta}) \circ (\widetilde{\mathbf{v}} - (\mathbf{1}-\mathbf{e}_1) \circ \widetilde{\mathbf{v}}^{\leftarrow -1}) = \mathbf{m}$.

$\forall i \in [N]$,

$$m_i = |\{j \in [N] | \mathbf{u}_{[j]} = \mathbf{v}_{[i]}, \mathbf{b}_{[j]} = 1\}| + |\{j \in [N] | \mathbf{u}'_{[j]} = \mathbf{v}_{[i]}, \mathbf{b}'_{[j]} = 1\}| \quad (100)$$

$$\prod_{j=1}^{N} (\mathbf{u}_{[j]} + X)^{\mathbf{b}_{[j]}} \cdot (\mathbf{u}'_{[j]} + X)^{\mathbf{b}'_{[j]}} = \prod_{j=1}^{N} (\mathbf{v}_{[j]} + X)^{\mathbf{m}_{[j]}} \quad (101)$$

$$\sum_{j=1}^{N} \left(\frac{\mathbf{b}_{[j]}}{\mathbf{u}_{[j]} + X} + \frac{\mathbf{b}'_{[j]}}{\mathbf{u}'_{[j]} + X}\right) = \sum_{j=1}^{N} \frac{\mathbf{m}_{[j]}}{\mathbf{v}_{[j]} + X} \quad (102)$$

$$\sum_{j=1}^{N} \left(\frac{\mathbf{b}_{[j]}}{\mathbf{u}_{[j]} + \beta} + \frac{\mathbf{b}'_{[j]}}{\mathbf{u}'_{[j]} + \beta}\right) = \sum_{j=1}^{N} \frac{\mathbf{m}_{[j]}}{\mathbf{v}_{[j]} + X} \quad (103)$$

$$\forall i \in [N], \tilde{\mathbf{u}}_{[i]} = \sum_{j=1}^{i} \left(\frac{\mathbf{b}_{[j]}}{\mathbf{u}_{[j]} + \beta} + \frac{\mathbf{b}'_{[j]}}{\mathbf{u}'_{[j]} + \beta}\right) \quad (104)$$

$$\forall i \in [N], \tilde{\mathbf{v}}_{[i]} = \sum_{j=1}^{i} \frac{\mathbf{m}_{[j]}}{\mathbf{v}_{[j]} + \beta} \quad (105)$$

$$\tilde{\mathbf{u}}_{[N]} = \tilde{\mathbf{v}}_{[N]} \quad (106)$$

Verifier accepts (at random z) the polynomial identity corresponding to

$$\mathbf{e}_N \circ (\tilde{\mathbf{u}} - \tilde{\mathbf{v}}) = \mathbf{0} \quad (107)$$

$$\forall i \in [N], \tilde{\mathbf{u}}_{[i]} - (1 - \mathbf{e}_{1,[i]}) \cdot \mathbf{u}_{[i-1]} = \frac{\mathbf{b}_{[j]}}{\mathbf{u}_{[i]} + \beta} + \frac{\mathbf{b}'_{[j]}}{\mathbf{u}'_{[i]} + \beta} \tag{108}$$

$$\forall i \in [N], (\mathbf{u}_{[i]} + \beta) \cdot (\mathbf{u}'_{[i]} + \beta) \cdot (\tilde{\mathbf{u}}_{[i]} - (1 - \mathbf{e}_{1,[i]}) \cdot \tilde{\mathbf{u}}_{[i-1]}) = \mathbf{b}'_{[i]} \cdot (\mathbf{u}_{[i]} + \beta) + \mathbf{b}_{[i]} \cdot (\mathbf{u}'_{[i]} + \beta) \tag{109}$$

Verifier accepts (at random z) the polynomial identity corresponding to

$$(\mathbf{u} + \boldsymbol{\beta}) \circ (\mathbf{u}' + \boldsymbol{\beta}) \circ (\tilde{\mathbf{u}} - (\mathbf{1} - \mathbf{e}_1) \circ \tilde{\mathbf{u}}^{\leftarrow -1}) = \mathbf{b}' \circ \mathbf{u} + \mathbf{b} \circ \mathbf{u}' + \boldsymbol{\beta} \circ (\mathbf{b} + \mathbf{b}') \tag{110}$$

$$\forall i \in [N], \tilde{\mathbf{v}}_{[i]} - (1 - \mathbf{e}_{1,[i]}) \cdot \tilde{\mathbf{v}}_{[i-1]} = \frac{\mathbf{m}_{[i]}}{\mathbf{v}_{[i]} + \beta} \tag{111}$$

$$\forall i \in [N], (\mathbf{v}_{[i]} + \beta) \cdot (\tilde{\mathbf{v}}_{[i]} - (1 - \mathbf{e}_{1,[i]}) \cdot \tilde{\mathbf{v}}_{[i-1]}) = \mathbf{m}_{[i]} \tag{112}$$

Verifier accepts (at random z) the polynomial identity corresponding to

$$(\mathbf{v} + \boldsymbol{\beta}) \circ (\tilde{\mathbf{v}} - (\mathbf{1} - \mathbf{e}_1) \circ \mathbf{v}^{\leftarrow -1}) = \mathbf{m} \tag{113}$$

Completeness of double-gcq follows from the sequences of inductions

- (99) (100) ⇒ (101) ⇒ (102) ⇒ (103) where (102) ⇒ (103) fails when $-\beta \in \mathbf{u} \| \mathbf{u}' \| \mathbf{v}$, which happens with probability at most $3N/|\mathbb{F}|$;
- (103) (104) (105) ⇒ (106) ⇒ (107);
- (104) ⇒ (108) ⇒ (109) ⇒ (110);
- (105) ⇒ (111) ⇒ (112) ⇒ (113).

Therefore, the completeness error is $3N/|\mathbb{F}|$ by the union bound.

Soundness of double-gcq follows from the following sequences of inductions:

- (113) ⇒ (112), (110) ⇒ (109), and (107) ⇒ (106) follow from Schwartz-Zippel Lemma and together fails with probability at most $(4N + 2)/|\mathbb{F}|$;
- (112) ⇒ (111) fails with probability at most $N/|\mathbb{F}|$;
- (109) ⇒ (108) fails with probability at most $2N/|\mathbb{F}|$;
- (111) ⇒ (105) and (108) ⇒ (104);
- (105) (104) (106) ⇒ (103);
- (103) ⇒ (102) follows from Schwartz-Zippel Lemma and fails with probability at most $3N/|\mathbb{F}|$, similarly as the induction (93) ⇒ (92) ⇒ (91) ⇒ (90) in the proof of Theorem 10;
- (102) ⇒ (101) follows from the fact that the two sides of (102) are respectively the derivative of the logarithm of the two sides of (101), so (102) implies that the ratio between the two sides of (101) is a constant, which can only be one because both sides are monic;
- (101) ⇒ (99).

The soundness error is $(10N + 2)/|\mathbb{F}|$ by the union bound. □

We write "V checks $(\mathbf{u} \| \mathbf{u}') \subset_{\mathbf{b} \| \mathbf{b}'} \mathbf{v}$" when the parties run the double lookup argument. An immediate application is the Range32 protocol for the 32-bit range check, which can be directly extended to 64- or 256-bit ranges.

6 Efficiency Analysis

We evaluate and compare the performance of the different MCCs. We measure their performance based on two factors: the number of times building blocks Perm and Lookup are used, and the number of online polynomial oracles and evaluation queries outside of these building blocks. A summary of these costs is presented in Table 1.

For more concrete comparisons, we instantiate the permutation argument with the construction from PLONK [GWC19], and the lookup argument with two constructions respectively: the state-of-the-art construction cq [EFG22] and our gcq. See Appendix A of this paper for the PIOP version of cq after applying the standard optimization techniques and adding the masking vector $\mathbf{b}$. We also analyze the performance of the $32k$-bit memory with a different tradeoff between the maximal degree and the number of online polynomials. This alternative approach is characterized by not merging the grand-sum vectors in double-gcq. The concrete performance results are summarized in Table 2.

We present in Table 3 the estimated proof sizes and the costs of the prover and the verifier after instantiating the PIOP with the KZG polynomial commitment scheme. These numbers are the costs of the memory consistency checks when they are compiled into a SNARK as a standalone PIOP. They can only partially reflect the additional costs these protocols contribute to the entire ZKVM, as batching is extensively used in the compilation. Specifically, MCC can share the verifier-sampled evaluation point z with the other parts, and the proof for opening polynomial commitments can be batched with those for the rest of the PIOP, just like other building blocks like permutation/lookup arguments. Therefore, the additional cost brought by MCC in practice would be smaller than the numbers shown in Table 3.

Comparison among prior MCCs. As Table 2 shows, the simplest memory model, contiguous read-only memory, has the most efficient MCC protocols with either instantiation of the building blocks. For non-contiguous read-write memories, the Triton VM MCC requires roughly the same number of online polynomial oracles and evaluation queries as the check for 32-bit memory. However, for 256-bit memory, the Triton VM check costs approximately 20 to 30 fewer online polynomial oracles and evaluation queries. Although AryaMem has the fewest polynomials and queries, its memory space is also the most limited among the read-write memories.

Performance of Permem. Our new protocol costs three fewer online oracles and three fewer evaluation queries compared to Triton VM. Both Permem and Triton VM cost one more distinct evaluation point—the β in the DstFull protocol (Algorithm 4), compared to all other works. This additional evaluation point and the $O(S \cdot \log^2 S)$ complexity of multi-point evaluation seems inevitable for achieving the full address space.

Compared to all existing works with read-write memories, Permem is outperformed only by AryaMem, which is also based on the address cycle method. In detail, Permem costs three more polynomials and two more evaluation queries, in exchange for the larger memory address space.

Comparison between lookup arguments. When used in MCCs, our new lookup argument gcq reduces the number of online polynomial oracles by 2 to 10 and the number of online queries by 1 to 9, compared to the state-of-the-art construction cq. In particular, the performance of the MCCs with the full address space has overall improvements when instantiated with gcq.

7 Conclusion

In this work, we have analyzed the current methods for performing MCCs, a crucial and expensive component in ZKVMs, and formalized all of them as variants of the sorting paradigm. Our study provides a comprehensive overview of the various techniques used to build MCCs. Inspired by the techniques covered in this systemization, we suggest improvements to existing protocols in two aspects: a novel MCC protocol Permem that costs fewer building blocks, and a new lookup argument also with improved efficiency.

We hope that our work will inspire further research that explores new combinations of these techniques or improves existing components. In particular, for full address space, the DstFull protocol presents a bottleneck in terms of performance. It requires four online polynomials, one distinct evaluation point, and has a prover cost of $O(S \log^2 S)$. Improving its performance or eliminating the dependence on this protocol would be a valuable avenue for future work.

Acknowledgement

This work is partially supported by the National Key Research and Development Project (Grant No. 2020YFA0712300) and the National Natural Science Foundation of China (Grant No. 62272294). We thank Alan Szepieniec and the anonymous reviewers for their valuable comments.

References

ABST22. Miguel Ambrona, Marc Beunardeau, Anne-Laure Schmitt, and Raphaël R. Toledo. aPlonK : Aggregated PlonK from Multi-Polynomial Commitment Schemes. https://eprint.iacr.org/2022/1352, 2022.

Azt22. Team of Aztec. Aztec. https://zk.money/, 2022.

BBB+18. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. In *SP 2018, Proceedings*, pages 315–334. IEEE Computer Society, 2018.

BBC+17. Eli Ben-Sasson, Iddo Bentov, Alessandro Chiesa, Ariel Gabizon, Daniel Genkin, Matan Hamilis, Evgenya Pergament, Michael Riabzev, Mark Silberstein, Eran Tromer, and Madars Virza. Computational Integrity with a Public Random String from Quasi-Linear PCPs. In *EUROCRYPT 2017*, volume 10212, pages 551–579. 2017. http://link.springer.com/10.1007/978-3-319-56617-7_19.

BBHR18. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018. http://eprint.iacr.org/2018/046.

BCG+13. Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO 2013*, volume 8043, pages 90–108. Springer Berlin Heidelberg, 2013. http://link.springer.com/10.1007/978-3-642-40084-1_6.

BCG+17. Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune K. Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 336–365. Springer, 2017.

BCG+18. Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution. In *ASIACRYPT 2018*, Lecture Notes in Computer Science, pages 595–626, Cham, 2018.

BCGT13. Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: Extended abstract. In *ITCS '13*, page 401, Berkeley, California, USA, 2013. ACM Press. http://dl.acm.org/citation.cfm?doid=2422436.2422481.

BCR+19. Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent Succinct Arguments for R1CS. In *EUROCRYPT 2019*, volume 11476, pages 103–128. 2019. http://link.springer.com/10.1007/978-3-030-17653-2_4.

BCTV13. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. Technical Report 879, 2013. https://eprint.iacr.org/2013/879.

BDFG20. Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo Infinite: Recursive zk-SNARKs from any Additive Polynomial Commitment Scheme. Technical Report 1536, 2020. http://eprint.iacr.org/2020/1536.

BFH+20. Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In *CCS 2020*, pages 2025–2038, 2020.

BFR+13. Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 341–357, Farminton Pennsylvania, November 2013. ACM. https://dl.acm.org/doi/10.1145/2517349.2522733.

BFS20. Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK Compilers. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020*, pages 677–706, Cham, 2020.

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

CHM+20. Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020*, Lecture Notes in Computer Science, pages 738–768, Cham, 2020. Springer International Publishing.

COS20. Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and Transparent Recursive Proofs from Holography. In *EUROCRYPT 2020*, Lecture Notes in Computer Science, pages 769–793, Cham, 2020.

Eag22. Liam Eagen. Bulletproofs++. Technical Report 510, 2022. https://eprint.iacr.org/2022/510.

EFG22. Liam Eagen, Dario Fiore, and Ariel Gabizon. Cq: Cached quotients for fast lookups. https://eprint.iacr.org/2022/1763, 2022.

FS86. Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986. https://doi.org/10.1007/3-540-47721-7_12.

GGP10. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *CRYPTO 2010*, pages 465–482, Berlin, Heidelberg, 2010. Springer.

GK22. Ariel Gabizon and Dmitry Khovratovich. Flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size. https://eprint.iacr.org/2022/1447, 2022.

GMR85. S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing - STOC '85*, pages 291–304, Providence, Rhode Island, United States, 1985. ACM Press. http://portal.acm.org/citation.cfm?doid=22145.22178.

GPR21. Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a Turing-complete STARK-friendly CPU architecture. Technical Report 1063, 2021. http://eprint.iacr.org/2021/1063.

Gro16. Jens Groth. On the Size of Pairing-Based Non-interactive Arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016*, volume 9666, pages 305–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. http://link.springer.com/10.1007/978-3-662-49896-5_11.

GW20. Ariel Gabizon and Zachary J. Williamson. Plookup: A simplified polynomial protocol for lookup tables. Technical Report 315, 2020. http://eprint.iacr.org/2020/315.

GWC19. Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Technical Report 953, 2019. https://eprint.iacr.org/2019/953.

Hab22. Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. https://eprint.iacr.org/2022/1530, 2022.

KZG10. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT 2010*, volume 6477, pages 177–194. 2010. http://link.springer.com/10.1007/978-3-642-17373-8_11.

Loo22. Team of Loopring. Loopring - zkRollup Layer2 for Trading and Payment. https://loopring.org/#/, 2022.

Mid22. Team of Miden. Miden VM Documentation. https://maticnetwork.github.io/miden/, 2022.

Mig92. Maurice Mignotte. *Mathematics for Computer Algebra*. Springer, 1992.

PFM+22. Luke Pearson, Joshua Fitzgerald, Héctor Masip, Marta Bellés-Mu
textasciitilde noz, and Jose Luis Mu
textasciitilde noz-Tapia. PlonKup: Reconciling PlonK with plookup. Technical Report 086, 2022. https://eprint.iacr.org/2022/086.

PHGR13. B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *SP 2013*, pages 238–252, Berkeley, CA, May 2013. IEEE. http://ieeexplore.ieee.org/document/6547113/.

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

Pol22. Team of Polygon. Polygon Hermez. https://polygon.technology/solutions/polygon-hermez/, 2022.

Ris22. Team of RiscZero. RISC Zero : General-Purpose Verifiable Computing. https://risczero.com/, 2022.

RZ21. Carla Ràfols and Arantxa Zapico. An algebraic framework for universal and updatable SNARKs. In *Annual International Cryptology Conference*, pages 774–804. Springer, 2021.

Scr22. Team of Scroll. Scroll. https://scroll.io/, 2022.

Set20. Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Annual International Cryptology Conference*, pages 704–737. Springer, 2020.

SL20. Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zkSNARKs. Technical Report 1275, 2020. http://eprint.iacr.org/2020/1275.

SLST23. Alan Szepieniec, Alexander Lemmens, Jan Ferdinand Sauer, and Bobbin Threadbare. The Tip5 Hash Function for Recursive STARKs. https://eprint.iacr.org/2023/107, 2023.

SZ22. Alan Szepieniec and Yuncong Zhang. Polynomial IOPs for Linear Algebra Relations. In *PKC 2022*, volume 13177 of *Lecture Notes in Computer Science*, pages 523–552. Springer, 2022. https://doi.org/10.1007/978-3-030-97121-2_19.

TV22. Team of Triton VM. Triton VM. Triton VM, September 2022. https://github.com/TritonVM/triton-vm.

WTS+18. R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-Efficient zkSNARKs Without Trusted Setup. In *SP 2018*, pages 926–943, May 2018.

XZZ+19. Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *CRYPTO 2019*, pages 733–764, Cham, 2019.

ZBK+22. Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup Arguments in Sublinear Time. Technical Report 621, 2022. https://eprint.iacr.org/2022/621.

ZGK+18. Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *SP 2018*, pages 908–925. IEEE, 2018.

ZGK+22. Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. Baloo: Nearly Optimal Lookup Arguments. https://eprint.iacr.org/2022/1565, 2022.

zkS22. Team of zkSync. zkSync. https://zksync.io/, 2022.

ZSZ+22. Yuncong Zhang, Alan Szepeniec, Ren Zhang, Shi-Feng Sun, Geng Wang, and Dawu Gu. VOProof: Efficient zkSNARKs from Vector Oracle Compilers. In *CCS 2022*, CCS '22, pages 3195–3208, New York, NY, USA, November 2022. https://doi.org/10.1145/3548606.3559387.

ZXZS20. Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *SP 2020*, pages 859–876. IEEE, 2020.

A The PIOP version of cq

We present the PIOP underlying cq lookup argument [EFG22] in Algorithm 9.

Algorithm 9 PIOP underlying cq

procedure $\mathsf{Lookup}^{\mathsf{cq}}(\{\mathbf{u}_i\}_{i=1}^k, \mathbf{v}, \{\mathbf{b}_i\}_{i=1}^k)$

P sends $\mathbf{m} = (m_i)_{i=1}^N$ where $m_i = \sum_{\ell=1}^k |\{j \in [N] | \mathbf{u}_{\boldsymbol{\ell}[j]} = \mathbf{v}_{[i]}, \mathbf{b}_{\boldsymbol{\ell}[j]} = 1\}|$;

V samples $\beta \xleftarrow{\$} \mathbb{F}$;

for ℓ from 1 to k **do**

P sends $\tilde{\mathbf{u}}_\ell := \left(\frac{\mathbf{b}_{\boldsymbol{\ell}[i]}}{\mathbf{u}_{\boldsymbol{\ell}[i]}+\beta}\right)_{i=1}^N$;

P sends $\tilde{\mathbf{v}} := \left(\frac{\mathbf{m}_{[i]}}{\mathbf{v}_{[i]}+\beta}\right)_{i=1}^N$;

P sends $r(X) \in \mathbb{F}^{N-2}[X]$ such that $f_{\tilde{\mathbf{v}}}(X) - \sum_{\ell=1}^k f_{\tilde{\mathbf{u}}_\ell}(X) - r(X) \cdot X$ is divided by $X^N - 1$ and $\tilde{r}(X) = r(X) \cdot X^{D-N+2}$;

V checks $f_{\tilde{\mathbf{v}}}(X) - \sum_{\ell=1}^k f_{\tilde{\mathbf{u}}_\ell}(X) - r(X) \cdot X$ is divided by $X^N - 1$ by sending the quotient polynomial $q(X)$ and opening the polynomials at z;

for ℓ from 1 to k **do**

V checks $(\mathbf{u}_\ell + \boldsymbol{\beta}) \circ \tilde{\mathbf{u}} = \mathbf{b}_\ell$;

V checks $(\mathbf{v} + \boldsymbol{\beta}) \circ \tilde{\mathbf{v}} = \mathbf{m}$;

V checks $\tilde{r}(z) = r(z) \cdot z^{D-N+2}$, where z can reuse the one used before.