Eigen-zkVM: A Framework For More Efficient And Universal Incrementally Verifiable Computation

Eigen Labs

Abstract

Verifiable Computing has been proposed for many years, and now mass adoption is on the horizon with Eigen zkVM. There are strong use cases for verifiable computing, such as Rollup and zkML. The Rollups are a family of powerful scaling technologies that promise to significantly increase the capacity and privacy preservation of Smart Contract-based blockchains, including Ethereum, Cardano, etc. Securely outsourcing computation, like zkML, is another use case. Eigen zkVM is a universal verifiable computing framework, that allows developers to build their applications with high-level programming languages. It proposes a highly parallel pipeline architecture within a layered-proof system.

In this framework, We introduce a universal composable proof system, which is an extension of the existing R1CS (Rank-1 Constraint System) constraints. It has achieved interoperability and support for Plonk gates, R1CS, and AIR (Algebraic Intermediate Representation), and on this basis, it has expanded complex constraints such as Permutation check, Multiset equality, and Lookup table. In the arithmetic process, it has realized verifiable universal VM (Virtual Machine) computation, supporting instruction sets such as EVM and RISC-V, as well as general-purpose languages like Rust and Circom. In the proof system, by combining Stark recursion and aggregate proofs, it has achieved composable proofs for complex computations, and ultimately, using Snark (Succinct Non-interactive ARguments of Knowledge) technologies like Groth16 or Fflonk to compress the proof size from the 100KB level to the 100B level, achieving a three orders of magnitude reduction and significantly lowering the complexity of verification.

Finally, the strength of the universal composable proof system lies in its universal circuit layer, the fastest proof speed, and the lowest verification complexity, which has led to its practical application in Cardano, Ton, and Powdr Labs.

Keywords: Eigen zkVM, Universal Composable Proof System, Blockchain, Smart Contract

1 Introduction

One active research area in cryptography is improving the efficiency of Incrementally Verifiable Computation (IVC). IVC enables a party to prove the integrity of a program's execution by providing evidence that each step is accurate and properly builds on previous steps. Specifically, for any step N, a function F_N is applied to update the current state. This function takes as input the current state x_N and a proof π_N confirming the correct execution of all prior steps $1, 2, ..., N-1$. It outputs the updated state x_{N+1} along with a new proof π_{N+1} verifying the proper execution of this state transition.

IVC and its extension to Directed Acyclic Graphs (DAGs) and Proof-Carrying Data (PCD) [1, 2], have diverse applications across various domains. These include distributed computation [3, 4], blockchains [5, 6], verifiable delay functions [7], verifiable photo editing [8], and machine computations [9] using Succinct Non-Interactive Arguments of Knowledge (SNARKs). One notable use case is Decentralized Private Computation (DPC), which enables the delegation of program execution to untrusted parties. Furthermore, the construction of Verifiable Delay Functions (VDF) based on IVC is currently a leading candidate for implementation in Ethereum [10]. The most groundbreaking application of IVC and PCD is ZK-EVM, an initiative aimed at creating a proof system capable of validating the integrity of existing Ethereum blocks [11]. These applications demonstrate the versatility and importance of IVC and its generalizations, underscoring their potential to revolutionize various aspects of cryptography and distributed computing.

Early implementations of IVC [1, 2] depend on recursive forms of Succinct Non-Interactive Arguments of Knowledge (SNARKs) [12–15]. In this setup, at each incremental step i, the prover generates a SNARK that attests to the correct execution of that specific computational step, using the output from the previous step $i-1$ [3, 9]. This proof also confirms that a SNARK verifier—represented as a circuit—has accepted the SNARK generated during step $i-1$. One of the challenges in these implementations is the necessity to represent the SNARK verifier as a circuit, which could be computationally intensive. To address this, some works, like the one cited as [9], employ a 2-cycle of elliptic curves to reduce the verifier's size when it's encoded as a circuit. In this context, a 2-cycle of elliptic curves consists of a pair of elliptic curves (E_1, E_2) in which the scalar field of E_1 is equivalent to the base field of E_2, and vice versa.

Recent works [16–19] have shifted away from solely relying on SNARKs for IVC, leading to what are known as folding schemes. In a folding scheme, an untrusted prover and a verifier each have an NP instance of size N. The prover also holds witnesses for both instances. The protocol lets them combine these into a single N-sized NP instance, called the 'folded' instance. The folding scheme ensures this folded instance is only satisfiable if both original instances are valid. The scheme is considered nontrivial if it reduces the verifier's computational work and communication compared to not using the folding scheme. Essentially, folding simplifies verifying two structurally similar NP instances into checking one. Because of this simplification, folding schemes tend to be more efficient and easier to implement than SNARKs. Folding schemes move beyond SNARKs to provide more practical IVC.

Recent advancements in IVC include schemes like Nova [20], HyperNova [21], and ProtoStar [22], which leverage folding schemes instead of SNARKs. Despite this shift, the foundational work outlined in [9] remains the sole efficient approach for implementing these schemes using a cycle of elliptic curves. For instance, an implementation of Nova adapts the methodology from [9] to fit the context of folding-scheme-based recursive arguments. This adaptation still necessitates representing the verifier of a folding scheme as a circuit, which must be applied to both curves in the cycle.

2 Related works

2.1 Early IVC and PCD

Paul Valiant [1] pioneered a proof system for incrementally verifiable computation that pushes the limits of non-interactive proofs in efficiency. The prover's space is polynomial, and time is nearly linear, compared to classical methods, while the verifier operates in almost constant time and space. The system's highlight is its composability, allowing two proofs of length k to merge into one of the same length, efficiently upholding the combined assertions.

Chiesa and Tromer [2] extended incrementally verifiable computation to arbitrary graphs with PCD, focusing on output data properties to address faults and leakage. In PCD, the system's output requirements are predefined, and each data transmission is paired with a proof, verified by system components, ensuring data and its history meet the set properties.

Bitansky et al. [3] crafted the first succinct, publicly verifiable SNARK without the need for costly preprocessing, using only collision-resistant hashing. Their method also adapts to privately verifiable SNARKs with fully-homomorphic encryption. Central to their innovation is a recursive SNARK composition technique within the PCD framework, significantly advancing the efficiency of cryptographic proofs in distributed systems.

Spartan [23] constitutes a family of zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) crafted specifically for rank-1 constraint satisfiability (R1CS), an NP-complete language that extends the satisfiability concept of arithmetic circuits. It marks the introduction of the initial zkSNARKs that operate without a trusted setup (referred to as transparent zkSNARKs) for NP. Within this framework, the verification of proof incurs sub-linear costs while not mandating uniformity in the structure of the NP statement.

Halo Infinite [24] expands Halo to show that Proof-carrying data can be (heuristically) built from any homomorphic polynomial commitment scheme (PCS), even if the PCS evaluation proofs are neither succinct nor efficient. The Halo methodology extends to any PCS that has an even more general property, namely the ability to aggregate linear combinations of commitments into a new succinct commitment that can later be opened to this linear combination. It thus implies new constructions of SNARKs and PCD that were not previously described in the literature, and serve as a blueprint for future constructions as well.

2.2 The First Generation of IVC

Nova [20] presents an approach aimed at achieving IVC, wherein the prover engages in recursive proof to establish the precise execution of incremental computations in the format of $y = F^{(l)}(x)$. This approach actively avoids the use of succinct non-interactive arguments of knowledge (SNARKs) and arguments of knowledge in general. Instead, it introduces and employs folding schemes—a less intricate, simplified, and efficiently achievable primitive—to streamline the validation of two instances within a certain relationship to the verification of a single instance.

Drawing upon Nova as a foundation, SuperNova [25] is intricately crafted to produce concise proofs that validate the precise execution of programs on a stateful machine equipped with a specific instruction set (e.g., EVM, RISC-V). The cost associated with proving a program step is intrinsically linked to the size of the circuit embodying the instruction invoked during that particular step.

Mohnblatt introduced Sangria [26], a realization of IVC using a variant of the PLONK arithmetization. They expanded the relaxed PLONK arithmetization to accommodate custom gates of degree 2 and circuits with higher gate arity. Lastly, they delineated potential directions for future endeavors, including the incorporation of higher-degree gate folding, the facilitation of lookup gates, and the creation of an IOP for the relaxed PLONK arithmetization.

CycleFold [27] is a straightforward approach to instantiate recursive arguments based on folding schemes over a cycle of elliptic curves to achieve IVC. The recursive arguments based on folding schemes can be efficiently established without relying on a cycle of elliptic curves, except for a few scalar multiplications in their verifiers. To achieve this, CycleFold leverages the second curve in the elliptic curve cycle to primarily represent a single scalar multiplication (utilizing 1,000-1,500 multiplication gates). Subsequently, CycleFold folds invocations of this small circuit on the first curve within the cycle. This improvement is nearly an order of magnitude compared to the previous state-of-the-art in terms of circuit sizes on the second curve.

2.3 The Next Generation of IVC

Setty et al. [28] developed CCS, a flexible framework that generalizes R1CS to include various constraint systems like Plonkish and AIR without extra complexity. Their SuperSpartan SNARKs for CCS efficiently support high-degree constraints and avoid intensive cryptographic operations. Unlike HyperPlonk, SuperSpartan can handle uniform CCS instances more efficiently, without burdensome preprocessing for the verifier. It offers a significant advancement for AIR, providing a SNARK with a fast prover, minimal preprocessing, and compact proof size, with the added benefit of potential post-quantum security.

HyperNova [21] is a recursive argument designed to demonstrate incremental computations, where each step is defined using CCS (ePrint2023/552), a customizable constraint system that concurrently encompasses Plonkish, R1CS, and AIR, all without incurring additional overhead. It can minimize the prover's cost at every step, primarily relying on a single multi-scalar multiplication (MSM) with a size equivalent to the number of variables within the constraint system. This approach is particularly effective when leveraging an MSM-based commitment scheme.

Zheng et al. [29] developed KiloNova, a non-uniform PCD system that leverages generic folding schemes for enhanced zero-knowledge efficiency. It simplifies constraints for better performance, using optimizations like circuit aggregation to reduce complexity. KiloNova outperforms its predecessors, with prover costs mainly due to multi-scalar multiplication and recursive circuit efficiency achieved through logarithmic hashing and scalar multiplications.

ProtoStar [22] is a non-uniform IVC scheme designed for Plonk, offering support for high-degree gates and (vector) lookups. The core of the recursive circuit involves 3 group scalar multiplications and the hashing of $d*$ field elements, where $d*$ represents the degree of the highest gate. It operates without the need for a trusted setup or pairings, and the prover is not required to perform any FFT computations. Additionally, the computational load on the prover during each accumulation/IVC step remains logarithmic, aligning with the number of supported circuits, and remains unaffected by the table size within the lookup process.

Drawing inspiration from ProtoStar, ProtoGalaxy [30] is a folding scheme that enhances the recursive verifier's efficiency. In this scheme, the additional work required beyond the linear combination of witness commitments is minimized to include only a logarithmic count of field operations and a fixed number of hash operations. Furthermore, this folding scheme demonstrates strong performance when multiple instances are folded within a single step. In such cases, the marginal number of verifier field operations per instance remains constant, assuming gates with consistent degrees.

3 EigenVM

EigenVM is built on the RISC V ISA. RISC V is commonly used in embedded systems and low-power devices.

The Definition 1 gives a more formal definition of zkVM.

Definition 1. *A Turning complete zkVM can generate a proof for arbitrary computations and can be formalized to*

$$F(x, w) = y,$$

where the F is the instruction execution logic. The x comprises a program commitment, initial state commitment, and public memory. The w represents the program instructions, init state, and secret inputs. The y is the final state commitment.

3.1 EigenVM Execution Trace

EigenVM comprises several interconnected components, each fulfilling a specialized role. The key components include:

1. `Program Decoder`: Computes a commitment to the executing program and translates it into a sequence of operations for the VM to execute.
2. `Operand Stack`: Functions as a push-down stack, supplying operands for all VM-executed operations.
3. `Range Checker`: Provides essential 16-bit range checks for other components.

4. `Chiplets:` A collection of specialized circuits that expedite frequently-used complex calculations. The VM currently employs three types of chipsets:
   - `Hash Chiplet:` Computes Rescue Prime Optimized hashes for both sequential and Merkle tree hashing.
   - `Bitwise Chiplet:` Executes bitwise operations (e.g., AND, XOR) on 32-bit integers.
   - `Memory Chiplet:` Supports random-access memory within the VM.

These components are interconnected via buses implemented through multiset checks. Multiset checks are also employed internally to describe virtual tables within each component.

The execution trace of EigenVM features 66 main trace columns, 2 buses, and 6 virtual tables, as detailed in the accompanying diagram. While the decoder, stack, and range checker components have dedicated column sets, all chipsets share a common set of 18 columns. Chiplets are distinguished by a series of binary selector columns, which, when combined, uniquely identify each chipset. In addition to the aforementioned components, the execution trace incorporates 2 system columns:

- `clk:` Tracks the current VM cycle. The column values start at 0 and increment by 1 in each cycle.
- `fmp:` Contains the value of the free memory pointer, delineating the memory region available for procedure-local variables.

Constraints for the `fmp` column are elaborated in the system operations section. For the `clk` column, the constraints are straightforward and can be represented as:

$$clk' - (clk + 1) = 0(degree = 1) \tag{1}$$

This refinement aims to improve both the clarity and grammatical accuracy of the original description.

3.2 The Columns of the Execution Trace

The execution trace, often referred to simply as the *trace*, provides a comprehensive record of the machine's state at each clock cycle during computation. Typically, this trace is presented as a rectangular array. Each row corresponds to the machine's complete state at a specific instant, while each column represents the temporal evolution of a particular aspect of the computation (e.g., the value stored in a specific EigenVM register).

Categorization of Columns

The columns in the EigenVM execution trace can be classified into the following categories:

- `Control Columns - Public.` The data in these columns describe the EigenVM architecture and various control signals that define the stage of execution. This information helps to determine the constraints that are applicable at each computational stage.

- `Data Columns - Private.` These columns store data that represents the running state of the processor and memory. To efficiently verify the integrity of EigenVM memory operations, each register linked with these operations has two dedicated columns: the first shows the original execution order, while the second is sorted first by memory location and then by clock cycle.
- `Accumulator Columns - Private.` Accumulators are used to instantiate grand product constraints for PLONK-based permutation checks and PLOOKUP-based range checks. The Accumulator Columns contain the associated accumulator data. The content of these columns, along with the corresponding constraints, are formulated during the randomized preprocessing phase [31–33].

3.3 EigenVM Stack

EigenVM operates as a stack machine. The stack is a push-down stack with practically unlimited depth—in practice, the depth will not exceed 2^{32}. However, only the top 16 items are directly accessible to the VM. Each stack item is an element in a prime field with modulus $2^{64} - 2^{32} + 1$.

3.3.1 Stack Management Rules

To maintain a manageable constraint system for the stack, we enforce the following rules:

1. All operations executed by the VM can shift the stack by at most one item, meaning the stack size can either shrink or grow by one item or remain unchanged.
2. The stack depth must always be ≥ 16. Initially, the stack is populated with exactly 16 input values, which could all be zero.
3. At the end of program execution, the stack must contain exactly 16 items, which again can all be zero. These 16 items constitute the program's output.

To simplify stack depth management, we adopt an additional rule:

- When the stack depth is 16, removing additional items does not change its depth. Instead, zeroes are inserted at the deep end of the stack to maintain a depth of 16.

3.3.2 Stack Representation

The VM allocates 19 trace columns to represent the stack, as outlined below.

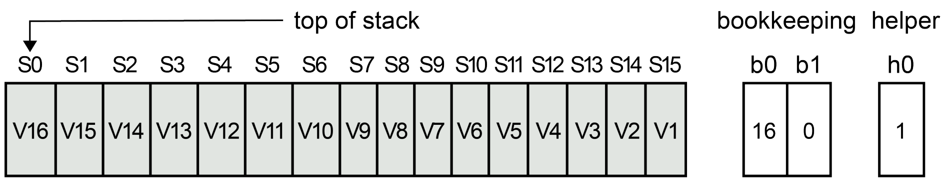
top of stack
bookkeeping
helper
S0 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15
V16 V15 V14 V13 V12 V11 V10 V9 V8 V7 V6 V5 V4 V3 V2 V1
b0 b1
16 0
h0
1

Fig. 1 EigenVM stack

3.3.3 Column Meanings

- $s_0, s_1, \ldots, s_{15}$ represent the top 16 slots of the stack.
- Column b_0 holds the current stack depth. For instance, if there are 16 items on the stack, then $b_0 = 16$.
- Column b_1 stores an address in a row in the `overflow table`. This table holds data that cannot fit into the top 16 stack slots. When $b_1 = 0$, it signifies that all stack data fit within these top slots.
- A helper column h_0 ensures that the stack depth never falls below 16. This column's values are set non-deterministically by the prover to $\frac{1}{b_0-16}$ when $b_0 \neq 16$ and to any other value otherwise.

3.4 EigenVM Memory

EigenVM supports linear, word-addressable read-write random-access memory (RAM). Each address holds four values, enabling batch reading and writing in groups of four. Each value exists as a field element in a 64-bit prime field with the modulus $2^{64} - 2^{32} + 1$. Any field element can serve as a memory address.

3.4.1 Design Idea

A straightforward and efficient alternative is a contiguous write-once memory. This requires only two trace columns, as depicted below:

- `addr Column`: Holds the memory address.
- `value Column`: Stores the field element representing the value at the given address.

Note that some rows in this table are duplicated because each memory access (read or write) necessitates a separate row. For instance, the value `b` was initially stored at address 11 and later read from the same address.

3.4.2 AIR Constraints

1. `Address Increment Constraint`: To maintain consistency in the `addr` column, values must either remain constant or increment by 11 from one row to the next. Mathematically, this constraint is expressed as:

$$(a' - a) \cdot (a' - a - 1) = 0, \tag{2}$$

   where a is the current row's `addr` value and a' is the next row's `addr` value.
2. `Value Persistence Constraint`: If the `addr` value remains the same, the corresponding `value` must also stay constant. This ensures that a value can only be set once at a given address. This constraint is:

$$(v' - v) \cdot (a' - a - 1) = 0, \tag{3}$$

   where v is the current row's `value` and v' is the next row's `value`.

Additional constraints for permutation checks are also necessary, but they are omitted here as they apply to all design alternatives discussed. The advantage of this alternative design lies in its efficiency: each memory access requires only two trace cells.

3.4.3 Read-Write Memory

We use a `clk` column to monitor the clock cycle of each memory access. Additionally, we use two columns, `old val` and `new val`, to differentiate between read and write operations. Specifically, if `old val` equals `new val`, a read operation occurred. Conversely, a difference between `old val` and `new val` signifies a write operation.

To support the above structure, the following AIR constraints are necessary:

1. `Contiguous Memory Address Constraint`:

$$(a' - a) \cdot (a' - a - 1) = 0 \tag{4}$$

2. `Memory Initialization Constraint`: When the memory address changes, `old val` should be set to 0.

$$(a' - a) \cdot \text{vold'} = 0 \tag{5}$$

3. `Value Consistency Constraint`: If the memory address remains the same, `new val` should equal the next row's `old val`.

$$(1 + a - a') \cdot (\text{vnew} - \text{vold'}) = 0 \tag{6}$$

4. `Clock Cycle Constraint`: For the same address, the values in the `clk` column should always be increasing. Introducing two additional columns, `d0` and `d1`, the constraint becomes:

$$(1 + a - a') \cdot ((i' - i - 1) - (2^{16} \cdot d1' + d0')) = 0 \tag{7}$$

   We also apply 16-bit range checks to `d0` and `d1`.

The overall cost of either reading or writing a single element now involves 6 trace cells and 2 16-bit range checks.

3.5 Continuation

If the majority of the leaves are empty, then they are called Sparse Merkle Trees. Using an SMT, it can be efficiently proven that specific data doesn't exist within a given Merkle Tree. In an SMT, the location of a piece of data (which leaf of the tree) and the data itself are bound to each other. This means that for a given piece of data, there is only one location within the tree at which that data could be placed. If that location is empty, the data is not present in the entire tree. To obtain this property, the contents of a leaf are hashed and a Merkle Tree is created in which the leaf's position corresponds to the hash. This requires a Merkle Tree of 256 levels and 2^{256} leaves. Generating such a large tree is efficient because the vast majority of the leaves are empty.

If most leaves are empty, they are called Sparse Merkle Trees (SMTs). Using an SMT, it can be efficiently proven specific data does not exist in a Merkle tree. In an SMT, the location of data (the leaf) and the data itself are bound together. For a given data item, there is only one location it could be placed. If that location is empty, the data is not in the tree. To achieve this, a leaf's contents are hashed and a 256-level Merkle tree is created, where the leaf's position matches the hash. This requires a tree with 2^{256} leaves. Generating such a large tree is efficient, as most leaves are empty.

4 The Interactive Oracle Proofs Protocol

In Probabilistically Checkable Proofs (PCPs), non-adaptivity is an inherent feature. A verifier, denoted as $\mathcal{V}$, has oracle access to a static proof string π. Given that π is static, $\mathcal{V}$ can query it multiple times, but each response π gives to a query q_i is solely dependent on q_i and not on any other queries q_j where $j \neq i$. Traditional PCP provers come with significant computational overhead, prompting the need for more efficient protocols. We introduce Interactive Oracle Proofs (IOPs), an extension of the PCP framework that also encompasses Interactive Proofs (IPs). In an IOP, the verifier is not obligated to read the entire message from the prover in each round. Instead, it has query-based access to the prover's message. This allows the verifier to focus on any specific symbol in the message at the expense of a single query, enabling sub-linear time operation relative to the total proof length—the cumulative length of all messages exchanged during the IOP.

Ben-Sasson, Chiesa, and Spooner [34] demonstrated that any IOP could be converted into a non-interactive argument using the random oracle model. This conversion uses Merkle hashing and the Fiat-Shamir transformation, paralleling the Kilian-Micali transformation that turns PCPs into succinct arguments. More precisely, instead of sending the entire message in each IOP round, the argument system prover sends a Merkle commitment of the IOP prover's message. The argument system verifier then mimics the IOP verifier to identify which elements of the message to query. The prover responds by revealing the pertinent symbols and providing the corresponding authentication paths within the Merkle tree. Finally, the Fiat-Shamir transformation is employed to make the interactive argument non-interactive.

Definition 2. *Given a function F, a Public-Coin Polynomial Interactive Oracle Proof (IOP) for F is an interactive protocol involving two parties: the prover $\mathcal{P}$ and the verifier $\mathcal{V}$. This protocol consists of k rounds of interaction. The prover $\mathcal{P}$ is provided with an input ω, while both $\mathcal{P}$ and $\mathcal{V}$ share a common input x. At the onset of the protocol, $\mathcal{P}$ submits a value y to $\mathcal{V}$ and asserts that there exists a ω such that $y = F(x, \omega)$. In the i-th round, $\mathcal{V}$ transmits a uniformly and independently random message α_i to $\mathcal{P}$.*

In response, $\mathcal{P}$ sends a message that takes one of two possible forms:

- *A string m_i that $\mathcal{V}$ reads in its entirety;*
- *An oracle for a polynomial f_i, which $\mathcal{V}$ can query via random access following the i-th round of interaction.*

Upon the protocol's conclusion, $\mathcal{V}$ outputs either `accept` or `reject`, signifying whether $\mathcal{V}$ validates $\mathcal{P}$'s initial claim.

The security concepts applicable to IOPs align with those of prior models, such as interactive proofs. In the subsequent definitions, probabilities are assessed over $\mathcal{V}$'s internal randomness.

Definition 3. *An IOP possesses perfect completeness and a soundness error of at most ε if it satisfies the following two conditions:*

- *`Perfect Completeness:` For any given x and any prover P who submits a value y such that $y = F(x, \omega)$ at the beginning of the protocol, the following must be true:*

$$\Pr[\mathcal{V}(x, \mathcal{P}(x, \omega)) = accept] = 1 \tag{8}$$

  $\mathcal{V}(x, \mathcal{P}(x, \omega))$ represents $\mathcal{V}$'s output after interacting with the prover on input x.

If the next condition holds as well, we say that the polynomial IOP has knowledge soundness error at most ε_{ks}.

- *`Soundness:` For any x and any prover $\mathcal{P}^*$ who submits a value y at the outset of the protocol, the following relationship must hold:*
  If $\Pr[\mathcal{V}(x, \mathcal{P}^*(x, \omega)) = accept] = \varepsilon_s$, then y must satisfies $y = F(x, \omega)$.

Additionally, if the IOP meets the following condition, it is said to have a knowledge soundness error of at most ε_{ks}:

- `Knowledge Soundness:` There exists a probabilistic polynomial-time algorithm E, known as the `knowledge extractor`, such that for any x and any prover $\mathcal{P}^*$ who submits a value y at the beginning of the protocol, the following holds:
  If $\Pr[\mathcal{V}(x, \mathcal{P}^*(x, \omega)) = accept] = \varepsilon_{ks}$, then $E(x, \mathcal{P}^*(x, \omega)) = \omega$ and y must satisfy $y = F(x, \omega)$.

To summarize, 1) `Soundness` ensures that a malicious prover cannot successfully assert the `existence` of w with a probability greater than ε_{ks}; 2) `Knowledge Soundness`, on the other hand, guarantees that a malicious prover cannot claim `possession` of w satisfying $y = F(x, \omega)$ with a probability greater than ε_{ks}. A Polynomial IOP that satisfies knowledge soundness is termed a `Polynomial IOP of Knowledge`. Interestingly, knowledge soundness implies soundness, and the converse is also true specifically for polynomial IOPs (as demonstrated in Lemma 2.3 of [35]). This indicates that establishing the soundness of a polynomial IOP is sufficient to assure its knowledge soundness.

Intriguingly, while it's intuitive that knowledge soundness implies soundness, the converse also holds specifically for Polynomial IOPs. In other words, establishing the soundness of a Polynomial IOP is, in itself, sufficient to guarantee its knowledge soundness.

Definition 4. *A Scalable Transparent Polynomial IOP of Knowledge (STIK) is a specialized form of Polynomial IOP with additional properties:*

- *`Transparent.` STIKs eliminate the need for a trusted setup before the protocol's execution, removing a single point of failure that could be exploited by malicious actors to forge false proofs.*

- `Doubly Scalable`. *The verifier operates in $O(log(n))$ time, while the prover operates in $O(nlog(n))$ time, where n represents the size of the computation F.*

When STIKs are adapted for real-world applications, they produce protocols that assume the prover to be computationally bounded. Such protocols are referred to as `argument systems`, as opposed to `proof systems`, and ensure soundness only against adversaries that operate in polynomial time.

Definition 5. *A Scalable Transparent Argument of Knowledge (STARK) is an implementation of an STIK that utilizes a family of collision-resistant hash functions. Specifically, the polynomial oracles transmitted from the prover to the verifier in the foundational Polynomial IOP are replaced by Merkle roots, which are calculated from polynomial evaluations over a set. When the verifier poses queries to a polynomial f at a variable v, the prover responds with $f(v)$ and its associated Merkle path.*

To eliminate the need for interaction between a specific prover and the verifier, making the protocols publicly verifiable, the Fiat-Shamir heuristic can be employed. This heuristic allows an STIK to be converted into a non-interactive argument of knowledge in the random oracle model. This is achieved by replacing the verifier's messages with queries to a random oracle, using the prover's prior messages as input. In practice, the random oracle is emulated using a cryptographic hash function. A non-interactive version of an STIK is termed a `non-interactive STARK`, although both types are commonly referred to simply as `STARKs`.

4.1 Execution Trace

Fibonacci's sequence

A prover needs to prove that the 1000th Fibonacci number Z starts from a public value X and a secret value Y, such that $F(X,Y) = Z$. This process is equivalent to proving

$$F_0 := X, F_1 := Y \tag{9}$$
$$F_i := F_{i-2} + F_{i-1} \tag{10}$$
$$Z := F_{1000}. \tag{11}$$

The computation requires n steps and w registers. The trace T is a $n \times w$ table, where $n = 1000, w = 2$. For example, if $X = 3, Y = 4$, the Table 1 can be constructed as follows: The above algorithm can be encoded as transition constraints and boundary

Table 1 Fibonacci's sequence.

n	$T_{n,0}$	$T_{n,1}$
0	3	4
1	4	7
2	7	11
3	11	18
4	18	29
...	...	...
999	F_{999}	F_{1000}

constraints as follows:

$$\begin{cases} T_{i+1,0} = T_{i,1}, \\ T_{i+1,1} = T_{i,0} + T_{i,1}. \end{cases} \tag{12}$$

$$\begin{cases} T_{0,0} = X, \\ T_{999,1} = Z. \end{cases} \tag{13}$$

Therefore, the following inference can be made:
Inference 1. *The prover proving that he knows the secret value Y satisfying the relation $F(X, Y) = Z$ is equivalent to the prover proving that it knows the trace T satisfying the transition constraints transition_constraints and the boundary constraints boundary_constraints.*

Trace Polynomial

Let $n = 0, ..., 999$ be the x-coordinates of a polynomial, and let the traces stored in register $T_{n,0}$ and register $T_{n,1}$ be the polynomial values, then the value expression of the trace polynomial can be constructed as follows:

$$P_0(i) = T_{i,0}, i = 0, ..., 999. \tag{14}$$

$$P_1(i) = T_{i,1}, i = 0, ..., 999. \tag{15}$$

Therefore, coefficient polynomials $P_0(x), P_1(x)$ of degree 999 can be constructed to express the traces.

The value expression of the trace polynomial can be equivalently transformed into the coefficient expression of the trace polynomial, so there is an equivalent transformation:

Fibonacci sequence ADD triple: $a_n = a_{n-1} + a_{n-2}$
Trace value expression: $T_{i+1,1} = T_{i,0} + T_{i,1}, i = 0, ..., 999$
Polynomial coefficient expression: $P_1(i + 1) = P_0(i) + P_1(i), i = 0, ..., 999$
Polynomial coefficient expression: $P_1(i + 1) - (P_0(i) + P_1(i)) = 0, i = 0, ..., 999$
Polynomial coefficient expression: $Q(x) := P_1(x + 1) - (P_0(x) + P_1(x)) = 0, x = 0, ..., 999$

Note that the degree of $Q(x)$ is 1000. We construct a public target polynomial:

$$R(x) = (x - 0)(x - 1)(x - 2)...(x - 999) \tag{16}$$

If $x = 0, ..., 999$, then the target polynomial $R(x) = 0$. There are other solutions x' to the equation $Q(x) = 0$. Thus, we can compute a quotient polynomial

$$C(x) = \frac{Q(x)}{R(x)}. \tag{17}$$

Note that the degree of the quotient polynomial is 1.

Similarly, we can construct a multiplication tuple.

Fibonacci's sequence MUL triple: $a_n = a_{n-1} * a_{n-2}$
Trace value expression: $T_{i+1,1} = T_{i,0} \cdot T_{i,1}, i = 0, ..., 999$
Polynomial coefficient expression: $P_1(i+1) = P_0(i) \cdot P_1(i), i = 0, ..., 999$
Polynomial coefficient expression: $P_1(i+1) - (P_0(i) \cdot P_1(i)) = 0, i = 0, ..., 999$
Polynomial coefficient expression: $Q(x) := P_1(x+1) - (P_0(x) \cdot P_1(x)) = 0, x = 0, ..., 999$

Note that the degree of $Q(x)$ is 2000. The public target polynomial is still:

$$R(x) = (x-0)(x-1)(x-2)...(x-999) \tag{18}$$

If $x = 0, ..., 999$, then the target polynomial $R(x) = 0$. There are other solutions x' to the equation $Q(x) = 0$. Thus, we can compute a quotient polynomial

$$C(x) = \frac{Q(x)}{R(x)}. \tag{19}$$

Note that the degree of the quotient polynomial is 1000.

Similarly, we can construct a multiplication quadruple.

Fibonacci's sequence MUL quadruple: $a_n = a_{n-1} * a_{n-2} * a_{n-3}$
Trace value expression: $T_{i+1,1} = T_{i,0} \cdot T_{i,1} \cdot T_{i,2}, i = 0, ..., 999$
Polynomial coefficient expression: $P_1(i+1) = P_0(i) \cdot P_1(i) \cdot P_2(i), i = 0, ..., 999$
Polynomial coefficient expression: $P_1(i+1) - (P_0(i) \cdot P_1(i) \cdot P_2(i)) = 0, i = 0, ..., 999$
Polynomial coefficient expression: $Q(x) := P_1(x+1) - (P_0(x) \cdot P_1(x) \cdot P_2(x)) = 0, x = 0, ..., 999$

Note that the degree of $Q(x)$ is 3000. The public target polynomial is still:

$$R(x) = (x-0)(x-1)(x-2)...(x-999) \tag{20}$$

If $x = 0, ..., 999$, then the target polynomial $R(x) = 0$. There are other solutions x' to the equation $Q(x) = 0$. Thus, we can compute a quotient polynomial

$$C(x) = \frac{Q(x)}{R(x)}. \tag{21}$$

Note that the degree of the quotient polynomial is 2000.

Therefore, the following inference can be made:

Inference 2. *Algorithms often involve multiple multiplications, so the degree of the quotient polynomial $C(x)$ is usually higher than the degree of the trace polynomial $T(x)$.*

Now we back to Fibonacci's sequence ADD triple, the transition constraints *transition_constraints* can be equivalently transformed into the following equations

$$T_{i+1,1} = T_{i,0} + T_{i,1} \Leftrightarrow C_0 = \frac{P_1(x+1) - (P_0(x) + P_1(x))}{\prod_{[0,\ldots,998]}^{i}(x-i)} \tag{22}$$

$$T_{i+1,0} = T_{i,1} \Leftrightarrow C_1 = \frac{P_0(x+1) - P_1(x)}{\prod_{[0,\ldots,998]}^{i}(x-i)} \tag{23}$$

$$\tag{24}$$

And the boundary constraints *boundary_constraints* can be equivalently transformed into the following

$$T_{0,0} = X \Leftrightarrow C_2(x) = \frac{P_0(x) - X}{x - 0} \tag{25}$$

$$T_{999,1} = Z \Leftrightarrow C_3(x) = \frac{P_1(x) - Z}{x - 999} \tag{26}$$

$$\tag{27}$$

Therefore, the following inference can be made:
Inference 3. *The prover proving that he knows the trace T satisfying the transition constraints transition_constraints and the boundary constraints boundary_constraints is equivalent to the prover proving that he knows the polynomials $P_0(x), P_1(x)$ such that $C_0(x), C_1(x), C_2(x), C_3(x)$ are degree D polynomials.*

On the contrary, if the verifier accepts that $C_0(x), C_1(x), C_2(x), C_3(x)$ are degree D polynomials, then the transition constraints and boundary constraints hold. Therefore, the verifier accepts the prover's proof that he knows the secret value Y satisfying the relation $F(X, Y) = Z$.

4.1.1 Stark AIR

An Arithmetic Interaction Row Proof (AIR P) over a field $\mathbb{F}$ has length n and width w. P is defined by a set of constraint polynomials f_i of predefined degree d in $2w$ variables. An execution trace T for P consists of n vectors of length w over $\mathbb{F}$, representing `rows of width` w. T is valid if substituting the $2w$ values from any two consecutive rows into any f_i evaluates to zero. A STARK enables proving knowledge of a valid T consistent with verifier-defined boundary constraints: for example, demanding the first value of the first row of T be zero (the same as public inputs in a SNARK). For the Fibonacci sequence, $w = 2$; the boundary conditions require the first row of T to contain two ones. And we use the constraint polynomials

$$f_1(X_1, X_2, Y_1, Y_2) = Y_1 - X_2 - X_1 \tag{28}$$

$$f_2(X_1, X_2, Y_1, Y_2) = Y_2 - Y_1 - X_2 \tag{29}$$

$$\tag{30}$$

A valid trace of length $n = 4$ would look like in Table 2. That is, a valid trace must

Table 2 Fibonacci's sequence.

1	1
2	3
5	8
13	21

contain consecutive elements of the Fibonacci sequence. So, for example, adding a boundary condition on the second value of the fourth row being 21 would validate this is indeed the correct 8-th Fibonacci element.

PAIRs - AIRs with preprocessed columns

In a Preprocessed AIR, we have an additional parameter t, and t preprocessed/predefined columns $c_1, \ldots, c_t \in \mathbb{F}_n$. An execution trace now consists of the $\{c_i\}$ in addition to the w. columns supplied by the prover. (We refer to the columns supplied by the prover as the witness part of the execution trace.)

For example, when $t = 1, w = 2, n = 4$ an execution trace could look like in Table 3. The constraint polynomials f_i will have $2(t + w)$ variables - i.e. the predefined

Table 3 Fibonacci's sequence.

$c_{1,1}$	a_1	b_1
$c_{1,2}$	a_2	b_2
$c_{1,3}$	a_3	b_3
$c_{1,4}$	a_4	b_4

values $c_{i,j}$ participate in the constraints.

A natural example is an AIR where for some rows we would like to perform an addition of the row values (and, say, obtain the addition result in the first column of the row below); and for the other rows we wish to perform multiplication.

For this purpose, we define the PAIR P as follows: Set $t = 1$, and define the column c_1 to be one on the rows where we want to add and zero when we want to multiply. The single constraint polynomial of P is

$$C_1 \cdot (Y_1 - (X_1 + X_2)) + (1 - C_1) \cdot (Y_1 - X_1 \cdot X_2). \tag{31}$$

The variable C_1 is assigned from the predefined column c_1. It is clear that an addition or multiplication relation is enforced according to the value of c_1.

For example, in a program where we wish to perform two additions and then a multiplication the execution trace could look like in Table 4. Because the predefined columns can be used in this way to select the operation, they are often referred to as `selectors`.

Table 4 Fibonacci's sequence.

1	1	1
1	2	5
0	7	3
0	21	0

Alternating between gates

The above example hints and suggests the typical way people design a PAIR: We predefine several sets of constraints, thinking of each one as a `gate`. Then, when designing the final program, we assign one of these gates to each row. As in the above example, selectors will be used to `compile` our program into a PAIR. It is worth noting that in addition to using the selectors to switch between gates, many times a gate itself will use selectors to enable more flexibility. A typical example is a gate for elliptic curve addition by a predefined point - the predefined point will be encoded in the selector values.

RAPs - PAIRs with interjected verifier randomness

Our final model introduces rounds of interaction where the verifier sends random field elements, and the prover can add more columns in response. In such a model, a RAP is a Randomized AIR with Preprocessing. Consider a width-2 AIR where we aim to verify that the columns sent by the prover are permutations of each other. Let the column values be $a_1, \dots, a_n$ and $b_1, \dots, b_n$.

By the Schwartz-Zippel Lemma, to verify that the columns are permutations of each other, it suffices to check that for a uniformly chosen $\gamma \in \mathbb{F}$,

$$\prod_{i\in[n]} (a_i + \gamma) = \prod_{i\in[n]} (b_i + \gamma). \tag{32}$$

With high probability over γ, the factors of the RHS are all non-zero. In this case, this check is equivalent to

$$1 = \prod_{i\in[n]} \frac{a_i + \gamma}{b_i + \gamma}. \tag{33}$$

A RAP of length $n + 1$ and total width, three can efficiently check this as follows:

1. The prover first sends the columns $(a_1, \dots, a_n, 0)$ and $(b_1, \dots, b_n, 0)$.
2. The verifier sends a random $\gamma \in \mathbb{F}$.
3. The prover sends a third column $(1, z_1, \dots, z_n)$ where z_i is defined as

$$z_i = \prod_{1\leq j\leq i} \frac{a_j + \gamma}{b_j + \gamma}. \tag{34}$$

If z is defined in this way, verifying that $z_n = 1$ serves as a permutation check. This can be added as a boundary constraint.

Moreover, the protocol must enforce that z was computed correctly. For this:

1. Add a boundary constraint ensuring the third column starts with one.
2. Add the constraint $Y_3(X_2 + \gamma) - X_3(X_1 + \gamma) = 0$.

Applying this constraint to row i verifies that

$$z_{i+1} \cdot (b_i + \gamma) = z_i \cdot (a_i + \gamma), \tag{35}$$

or, assuming $b_i + \gamma \neq 0$,

$$z_{i+1} = \frac{z_i \cdot (a_i + \gamma)}{b_i + \gamma}, \tag{36}$$

which inductively ensures that each z_i is computed correctly.

4.1.2 Customizable Constraint Systems

The Rank-1 Constraint Satisfaction (R1CS) is an NP-complete problem that finds its roots in Quadratic Arithmetic Programs (QAPs) [36]. In an R1CS instance, we consider a set of m constraints, and we define a vector z over the field $\mathbb{F}$ to be a solution to the instance if it satisfies all m constraints. The term `rank-1` signifies that each constraint is of a specialized form. To elaborate, each constraint states that the product of two particular linear combinations of the elements of z is equal to a third linear combination of those same elements. In practice, the last entries of z are usually constrained to equal some public input (an input that is known to both the prover and the verifier) followed by a constant one. Typically, an R1CS instance is divided into two main parts: a fixed `structure` detailing the constraints, and an `instance` containing just the public input. This division allows for preprocessing that is independent of the instance, thereby enabling the verifier's runtime to be succinct relative to the size of the structure.

Definition 6. *An R1CS structure S consists of size bounds $m, n, N, \ell \in \mathbb{N}$ where $n > \ell$, and three matrices $A, B, C \in \mathbb{F}^{m \times n}$ with at most $N = \Omega(\max(m, n))$ non-zero entries in total. An R1CS instance consists of public input $x \in \mathbb{F}^{\ell}$. An R1CS witness consists of a vector $w \in \mathbb{F}^{n-\ell-1}$. An R1CS structure-instance tuple (S, x) is satisfied by an R1CS witness w if*

$$(A \cdot z) \circ (B \cdot z) - C \cdot z = 0, \tag{37}$$

where $z = (w, 1, x) \in \mathbb{F}^n$, $\cdot$ is a matrix-vector multiplication operation, $\circ$ is the Hadamard (i.e., entry-wise) product between vectors, and $\mathbf{0}$ is a m-sized vector with entries equal to the additive identity in $\mathbb{F}$.

We now introduce a generalization of R1CS, which we refer to as CCS.

Definition 7. *(CCS). A CCS structure S consists of:*

- *size bounds $m, n, N, \ell, t, q, d \in \mathbb{N}$ where $n > \ell$;*
- *a sequence of matrices $\mathbf{M}_0, \ldots, \mathbf{M}_{t-1} \in \mathbb{F}^{m \times n}$ with at most $N = \Omega(\max(m, n))$ non-zero entries in total;*
- *a sequence of q multisets $[S_0, \ldots, S_{q-1}]$, where an element in each multiset is from the domain $\{0, \ldots, t-1\}$ and the cardinality of each multiset is at most d.*
- *a sequence of q constants $[c_0, \ldots, c_{q-1}]$, where each constant is from $\mathbb{F}$.*

A CCS instance consists of public input $x \in \mathbb{F}^{\ell}$. A CCS witness consists of a vector $w \in \mathbb{F}^{n-\ell-1}$. A CCS structure-instance tuple (S, x) is satisfied by a CCS witness w if

$$\sum_{i=0}^{q-1} c_i \cdot \bigcirc_{j \in S_i} \mathbf{M}_j \cdot z = \mathbf{0}, \tag{38}$$

where

$$z = (w, 1, x) \in \mathbb{F}^n, \tag{39}$$

$\mathbf{M}_j \cdot z$ denotes matrix-vector multiplication, $\circ$ denotes the Hadamard product between vectors, and $\mathbf{0}$ is a m-sized vector with entries equal to the the additive identity in $\mathbb{F}$.

Representing R1CS with CCS

Recall that an NP checker is an algorithm that takes an NP instance and an NP witness as input and verifies whether the witness satisfies the instance. The runtime of such a checker serves as a metric for evaluating the efficiency of our transformations. This runtime also indirectly indicates the size overheads; for example, in the case of a CCS witness transformed from an R1CS witness.

Lemma 1. *(R1CS to CCS transformation). Given an R1CS structure $S_{R1CS} = (m, n, N, \ell, A, B, C)$ there exists a CCS structure S_{CCS} such that for any instance $I_{R1CS} = x$ and witness $w_{R1CS} = w$, the tuple (S_{CCS}, x) is satisfied by w if and only if (S_{R1CS}, x) is satisfied by w. The transformation from S_{R1CS} to S_{CCS} runs in constant time. The natural NP checker's time to verify that the tuple $((S_{CCS}, x), w)$ satisfies Equation (2) is identical to the natural NP checker's time to verify that the tuple $((S_{R1CS}, x), w)$ satisfies Equation (1).*

Representing Plonkish with CCS

We propose a transformation from the Plonkish framework to CCS, allowing for greater interoperability and flexibility in the computational landscape.

Definition 8. *(Plonkish). A Plonkish structure S is comprised of:*

- *size bounds $m, n, \ell, t, q, d, e \in \mathbb{N}$;*
- *a multivariate polynomial g in t variables, where g is expressed as a sum of q monomials and each monomial has a total degree at most d;*
- *a vector of constants called selectors $s \in \mathbb{F}^e$; and*
- *a set of m constraints. Each constraint i is specified via a vector T_i of length t, with entries in the range $\{0, \dots, n+e-1\}$. T_i is interpreted as specifying t entries of a purported satisfying assignment z to feed into g.*

A Plonkish instance consists of public input $x \in \mathbb{F}^{\ell}$. A Plonkish witness consists of a vector $w \in \mathbb{F}^{n-\ell}$. A Plonkish structure-instance tuple (S, x) is satisfied by a Plonkish witness w if:

$$\forall i \in \{0, \dots, m-1\},\ g(z[T_i[1]], \dots, z[T_i[t]]) = 0, \tag{40}$$

where $z = (w, x, s) \in \mathbb{F}^{n+e}$.

Lemma 2. *(Plonkish to CCS transformation). Given a Plonkish structure $S_{Plonkish} = (m, n, \ell, t, q, d, e, g, T, s)$ there exists a CCS structure S_{CCS} such that the following* *holds. For any instance $I_{Plonkish} = x$ and witness vector $w_{Plonkish} = w$, the tuple (S_{CCS}, x) is satisfied by w if and only if $(S_{Plonkish}, x)$ is satisfied by w. The natural NP checker's time to verify the tuple $((S_{CCS}, x), w)$ is identical to the NP checker's time to verify the tuple $((S_{Plonkish}, x), w)$ that evaluates g monomial-by-monomial when checking that Equation (4) holds.*

Representing AIR with CCS.

We provide a transformation from AIR to CCS.
Definition 9. *(AIR). An AIR structure S consists of:*

- *size bounds $m, t, q, d \in \mathbb{N}$, where t is even;*
- *a multivariate polynomial g in t variables, where g is expressed as a sum of q monomials and each monomial has total degree at most d.*

An AIR instance consists of public input and output $x \in \mathbb{F}^t$. An AIR witness consists of a vector $w \in \mathbb{F}^{(m-1)\cdot t/2}$.

Let $z = (x[..t/2], w, x[t/2+1..]) \in (\mathbb{F}^{t/2})^{m+1}$ and $x[..t/2]$, $x[t/2+1..] \in \mathbb{F}^{t/2}$ refer to the first half and second half of x. Let us index the $m+1$ `rows` of z as $0, 1, \ldots, m$. An AIR structure-instance tuple (S, x) is satisfied by an AIR witness w if:

$$\forall i \in \{1, \ldots, m\},\ g(z[(i-1)\cdot t/2+1], \ldots, z[i\cdot t/2], z[i\cdot t/2+1], \ldots, z[(i+1)\cdot t/2]) = 0. \tag{41}$$

Lemma 3. *(AIR to CCS transformation). Given an AIR structure $S_{AIR} = (m, t, q, d, g)$, instance $I_{AIR} = x$, and witness $w_{AIR} = w$, there exists a CCS structure S_{CCS} such that the tuple (S_{CCS}, x) is satisfied by w if and only if (S_{AIR}, x) is satisfied by w. The natural NP checker's time to verify the tuple $((S_{CCS}, x), w)$ is identical to the runtime of the NP checker that evaluates g monomial-by-monomial when checking that $((S_{AIR}, x), w)$ satisfies Equation 41.*

Proof. Let $w_{CCS} = w_{AIR}$ and $I_{CCS} = I_{AIR}$. Let $S_{CCS} = (m, n, N, \ell, t, q, d, [\mathbf{M}_0, \ldots, \mathbf{M}_{t-1}], [S_0, \ldots, S_{q-1}], [c_0, \ldots, c_{q-1}])$, where m, t, q, d are from S_{AIR}. We now specify the remaining entries in S_{CCS}.
Deriving ℓ and n. Let $\ell \leftarrow t/2$ and $n \leftarrow m \cdot t/2$.
Deriving $\mathbf{M}_0, \ldots, \mathbf{M}_{t-1}$, and N. Recall that g in S_{AIR} is a multivariate polynomial in t variables. Unless set to a specific value below, any entry in $\mathbf{M}_0, \ldots, \mathbf{M}_{t-1} \in \mathbb{F}^{m\times n}$ equals 0, the additive identity of $\mathbb{F}$.

There is a CCS row for each of the m constraints in S_{AIR}, so if we index the CCS rows by $\{0, \ldots, m-1\}$, it suffices to specify how the ith row of these matrices is set, for $i = 0, \ldots, m-1$. We consider three cases. For all $j \in \{0, 1, \ldots, t-1\}$, let $k_j = i\cdot t/2 + j$.

- If $i = 0$ and $j < t/2$, we set $\mathbf{M}_j[i][j + |w_{AIR}|] = 1$.
- If $i = m-1$ and $j \geq t/2$, we set $\mathbf{M}_j[i][j + |w_{AIR}| + t/2] = 1$.
- Otherwise, we set $\mathbf{M}_j[i][k_j] = 1$.

We set $S_{CCS}.N$ to be the total number of non-zero entries in $\mathbf{M}_0, \ldots, \mathbf{M}_{t-1}$.
Deriving $S_0, \ldots, S_{q-1}$, and $c_0, \ldots, c_{q-1}$. Recall that g is a multivariate polynomial in t variables with q monomials where the degree of each monomial is at most d. For $i \in \{0, 1, \ldots, q-1\}$, set c_i to the coefficient of the ith monomial of g. For $i \in$

$\{0, 1, \ldots, q-1\}$, if the ith monomial contains a variable j, where $j \in \{0, 1, \ldots, t-1\}$, add j to multiset S_i with multiplicity equal to the degree of the variable.

By inspection, the tuple (S_{CCS}, I_{CCS}) is satisfied by w_{CCS} if and only if (S_{AIR}, I_{AIR}) is satisfied by w_{AIR}. Finally, the asserted bounds on the NP checker's time are immediate from the construction. □

4.2 FRI: Fast Reed-Solomon Interactive Oracle Proof of Proximity

4.2.1 Merkle Tree

In a Merkle tree, the green nodes represent the values of the polynomial committed to by the prover (such as the polynomial values $a = f(0), b = f(1), c = f(2), d = f(3)$. Each committed hash value forms the leaf nodes of the Merkle tree (nodes 4, 5, 6, and 7 in the diagram). All other non-leaf nodes are the hash of the concatenation of their corresponding child nodes (for example, $\mathsf{hash}(\#2) = \mathsf{hash}\left(\mathsf{hash}(\#4), \mathsf{hash}(\#5)\right)$. The root node is referred to as the Merkle root.

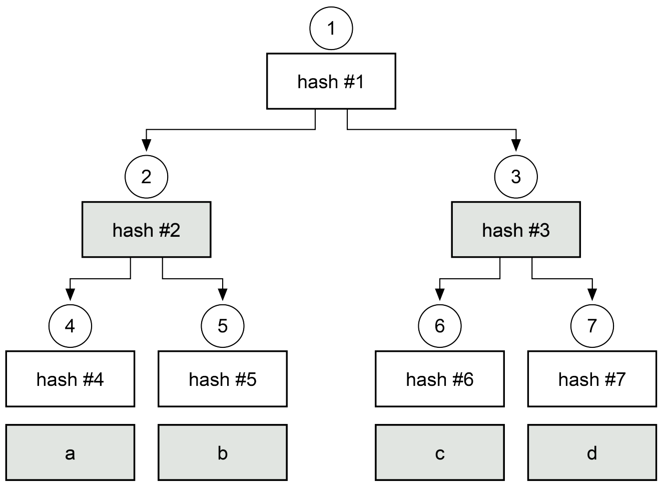

Fig. 2 Merkle Tree.

As illustrated in 2, a distinctive feature of the Merkle tree is that any change in a single leaf node will result in a change in the Merkle root. Conversely, if it can be proven that the root hash remains constant, the entire tree must also be constant. This means that the leaf node values have not been modified.

Polynomial Commitments Using Merkle Trees

- `Commit`: Generate a Merkle root based on the values of the polynomial (like the four nodes shown in the diagram).

- `Decommit:` Calculate a random point (such as node c) based on $root(f)$. The data transmitted is $(root(f), c, path(c))$ where, $path(c)$ is the hash of nodes 7 and 2, serving as the verification path for node c.
- `Verify:` Compute a new Merkle $root'$, using $(c, path(c))$. Perform a consistency check by comparing $root'$ and the received Merkle root $root$. If $root' = root$, accept; otherwise, reject.

4.2.2 Low-degree Detection

Direct Tests

For a constant polynomial $f(x) = c$, at a fixed point z_1 and a random point w calculated based on Fiat-Shamir, the verifier checks if

$$f(z_1) = f(w). \tag{42}$$

If it is true, the verifier accepts that the degree of the polynomial $f(x)$ is less than 1. For a linear polynomial $f(x) = bx + c$, at fixed points $(z_1, f(z_1))$ and $(z_2, f(z_2))$ and a random point w calculated based on Fiat-Shamir, the verifier checks if the third point $(w, f(w))$ lies on the same line as $(z_1, f(z_1))$ and $(z_2, f(z_2))$.

If this is the case, the verifier accepts that the degree of the polynomial $f(x)$ is less than 2. Extension: A polynomial $f(x)$ of degree d or lower requires d fixed points and one random point for verification. Given three points, a quadratic curve is determined, and a random point is selected for verification.

Linear Combination Test

Suppose there are two polynomials $f(x)$ and $g(x)$ of degree at most d. Using the direct testing method above requires $2(d+1)$ points for execution testing. To optimize this testing method, use the Fiat-Shamir heuristic to compute a random number α, and combine polynomials $f(x)$ and $g(x)$ as:

$$h(x) := f(x) + \alpha \cdot g(x). \tag{43}$$

This formula is called a linear combination. The prover constructs a Merkle tree based on the values of $h(x)$ and sends the Merkle root to the verifier. Then the verifier can verify the Merkle tree and only needs $d+1$ points to verify that the degree of $h(x)$ is less than d. Note that the degree of $h(x)$ is $\max\{\deg(f), \deg(g)\}$.

Polynomial Folding Lemma

Given that polynomial $f(x)$ have degree d, where $d = 2^n$. Let $g(x^2)$ be the even terms of $f(x)$, and $h(x^2)$ be the odd terms of $f(x)$. Then

$$f(x) = g(x^2) + x \cdot h(x^2). \tag{44}$$

The above formula is called the folding formula. Let $y = x^2$, then for y, the degrees of polynomials $g(y)$ and $h(y)$ are $d/2$.

For example: Given a polynomial $f(x)$ of degree $d = 2^3$

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7 + a_8x^8. \tag{45}$$

The even and odd terms of $f(x)$ are $f(x) = g(x^2) + x \cdot h(x^2)$

$$\begin{aligned} g(x^2) &= a_0 + a_2x^2 + a_4x^4 + a_6x^6 + a_8x^8 \\ xh(x^2) &= a_1x + a_3x^3 + a_5x^5 + a_7x^7 + 0x^9 \end{aligned} \tag{46}$$

Let $y = x^2$, then

$$\begin{aligned} g(y) &= a_0 + a_2y + a_4y^2 + a_6y^3 + a_8y^4 \\ h(y) &= a_1 + a_3y + a_5y^2 + a_7y^3 + 0y^4 \end{aligned} \tag{47}$$

Rewriting as

$$\begin{aligned} g(x) &= a_0 + a_2x + a_4x^2 + a_6x^3 + a_8x^4 \\ h(x) &= a_1 + a_3x + a_5x^2 + a_7x^3 + 0x^4 \end{aligned} \tag{48}$$

Therefore, the degrees of $g(x)$ and $h(x)$ are both $d/2$. The original polynomial of degree d is equivalent to the sum of odd and even terms, with degree decreased to $d/2$. Folding half each time, that is, 2, can generalize the folding 2^n, thus improving the folding efficiency.

4.2.3 FRI

FRI is used to verify polynomial degree is less than d. By combining the above direct test, linear combination test, and polynomial Folding Lemma, it is possible to test the degree of a polynomial $f_0(x)$ of degree $d = 2^n$ in only $O(\log d)$ rounds.

`Prover:` In step 1,
(1) Commit to the degree d polynomial $f_1(x)$ by computing its Merkle root $root(f_1)$.
(2) Compute the hash of the Merkle root to get a random number z:

$$z := \mathsf{hash}\left(root(f_1)\right) \tag{49}$$

(3) Computing polynomial values $f_1(z), f_1(-z)$.
`Send:` $\{root(f_1), f_1(z), f_1(-z), path(f_1(z)), path(f_1(-z))\}$.
`Verifier:`
(1) Use the Merkle root $root(f_1)$ and path $path(f_1(z))$ to verify the correctness of $f_1(z)$:

$$root(f_1) = \mathsf{Merkle}\left(f_1(z), path(f_1(z))\right) \tag{50}$$

(2) Similarly, verify $f_1(-z)$ using its path. If both checks pass, accept. Otherwise, reject.

`Prover:` In step 2,
(1) According to the folding formula, the degree d polynomial $f_1(x)$ can be expressed as:

$$f_1(x) = g_1(x^2) + x \cdot h_1(x^2) \tag{51}$$

Let this be Formula 1.
(2) Compute a hash based on the random number z to obtain a new random number α_2:

$$\alpha_2 := \mathsf{hash}(z) = \mathsf{hash}\left(\mathsf{hash}\left(root(f_1)\right)\right) \tag{52}$$

(3) Construct a linear combination polynomial:

$$f_2(x) = g_1(x) + \alpha_2 \cdot h_1(x) \tag{53}$$

(4) Commit to the polynomial $f_2(x)$ by computing its Merkle root $root(f_2)$. Here $f_2(x)$ has degree $d/2$.
(5) Computing polynomial values $f_2(-z^2)$.
`Send:` $\{root(f_2), f_2(-z^2), path(f_2(z^2)), path(f_2(-z^2))\}$.
Note that there is no need to send the value $f_2(z^2)$ since it can be computed directly.
`Verifier:`
(1) Hash the Merkle root $root(f_1)$ to obtain a random number z:

$$z := \mathsf{hash}\left(root(f_1)\right). \tag{54}$$

(2) Using the data $f_1(z), f_1(-z)$ from Step 1, the random number z, and the folding formula (Formula 1), obtain:

$$\begin{aligned} f_1(z) &= g_1(z^2) + z \cdot h_1(z^2) \\ f_1(-z) &= g_1(z^2) - z \cdot h_1(z^2) \end{aligned} \tag{55}$$

This allows solving equations to obtain $g_1(z^2), h_1(z^2)$.
(8) Hash z and $root(f_2)$ to obtain another random number α_2:

$$\alpha_2 := \mathsf{hash}(z, root(f_2)) \tag{56}$$

(9) Using $g_1(z^2), h_1(z^2)$, the random α_2, and the linear combination formula, compute:

$$f_2(z^2) := g_1(z^2) + \alpha_2 \cdot h_1(z^2) \tag{57}$$

to obtain $f_2(z^2)$.
(10) Use the received $root(f_2), path(f_2(z^2))$ to verify $f_2(z^2)$:

$$root(f_2) == \mathsf{Merkle}\left(f_2(z^2), path(f_2(z^2))\right). \tag{58}$$

(11) Similarly verify $f_2(-z^2)$. Accept if both checks pass, otherwise reject.

`Prover:` In step 3,
(1) According to the polynomial folding formula, the degree $d/2$ polynomial $f_2(x)$ can be expressed as:

$$f_2(x) = g_2(x^2) + x \cdot h_2(x^2) \tag{59}$$

Let this be Formula 2.
(2) Hash z and $root(f_2)$ to obtain a random number α_3:

$$\alpha_3 := \mathsf{hash}(z, root(f_2)) \tag{60}$$

(3) Construct a linear combination polynomial:

$$f_3(x) = g_2(x) + \alpha_3 \cdot h_2(x) \tag{61}$$

(4) Commit to $f_3(x)$ by computing its Merkle root $root(f_3)$. Here $f_3(x)$ has degree $d/2^2$.
(5) Computing polynomial values $f_3(-z^4)$.
`Send:` $\{root(f_3), f_3(-z^4), path(f_3(z^4)), path(f_3(-z^4))\}$.
Note that there is no need to send $f_3(z^4)$ since it can be directly computed.
`Verifier:`
(1) Using the verified value $f_2(z^2)$ from Step 1, z^2, and the folding formula (Formula 2), obtain:

$$\begin{aligned} f_2(z^2) &= g_2(z^4) + z^2 \cdot h_2(z^4) \\ f_2(-z^2) &= g_2(-z^4) - z^2 \cdot h_2(-z^4) \end{aligned} \tag{62}$$

This allows solving equations to obtain $g_2(z^4), h_2(z^4)$. (2) Hash z and $root(f_2)$ to obtain the random number α_3:

$$\alpha_3 := \mathsf{hash}(z, root(f_2)). \tag{63}$$

(3) Using $g_2(z^4), h_2(z^4), \alpha_3$, and the linear combination formula, compute:

$$f_3(z^4) := g_2(z^4) + \alpha_3 \cdot h_2(z^4) \tag{64}$$

to obtain$f_3(z^4)$.
(4) Use $root(f_3), path(f_3(z^4))$ to verify $f_3(z^4)$:

$$root(f_3) == \mathsf{Merkle}\left(f_3(z^4), path(f_3(z^4))\right). \tag{65}$$

(5) Similarly verify $f_3(-z^4)$. Accept if both checks pass, otherwise reject.

`Prover:` In step $\log(d)$,
`Send:` $\{root(f_{\log(d)}), f_{\log(d)}(-z^n), path(f_{\log(d)}(z^n)), path(f_{\log(d)}(-z^n))\}$

`Verifier:`
(1) Compute the polynomial value $f_{\log(d)}(z^n)$.
(2) Based on the polynomial folding and linear combination formulas, obtain a polynomial:

$$f_{\log(d)}(x) = g_{\log(d)}(x) + \alpha_{\log(d)} \cdot h_{\log(d)}(x) \tag{66}$$

This polynomial has degree $d/2^{\log(d)} = 1$, so it is a constant polynomial.
(3) The verifier uses direct testing to test the degree of the constant polynomial.

Note that:
(1) For the value range in the above protocol, it is necessary that for each value z in the range L, $-z$ is also in the range L.
(2) The commitment to the polynomial $f_1(x)$ is not over the range L, but rather over the range $L^2 = \{x^2 : x \in L\}$.

4.2.4 Probability Analysis

For the polynomial values $f_0(z), f_0(-z)$, assume the probability that $f_0(z) = f_0(-z) = 0$ is ε, where $0 < \varepsilon < 1$. Then the probability of other cases is $1-\varepsilon$, where $0 < 1-\varepsilon < 1$.

$$f(x) = g(x^2) + x \cdot h(x^2) \tag{67}$$

If the $1-\varepsilon$ probability case occurs, i.e. $f_0(z) = f_0(-z) \neq 0$, then there exists a unique α_1 that makes the linear combination zero:

$$f_1(z^2) = g_0(z^2) + \alpha_1 \cdot h_0(z^2) = 0 \tag{68}$$

However, α_1 is a random number computed via Fiat-Shamir, so the probability of manipulating α_1 to a desired value is equal to the difficulty of POW. Therefore, the probability that $f_1(z^2) \neq 0$ approaches 1. If the prover cheats on $f_1(z^2)$, the Merkle consistency check will fail. Thus, if the $1-\varepsilon$ probability case occurs, the verifier will reject it.

Conversely, the probability the prover successfully cheats in a round is ε. Over $\log(d)$ rounds, the probability the prover succeeds in every round is $\varepsilon^{\log(d)}$, which decays exponentially. Therefore, the probability the prover can successfully cheat is negligible. In summary, soundness holds except with negligible probability ε.

4.2.5 Batch FRI

In the Batched FRI Protocol, the prover aims to demonstrate the closeness of a set of functions $f_0, f_1, ..., f_N : H \leftarrow F$ to low-degree polynomials all at once. While it's possible to run individual FRI protocols for each function f_i in parallel, a more efficient approach is suggested in Section 8.2 of [37]. Specifically, the prover directly applies the FRI protocol to a random linear combination of each function f_i. After committing to the functions and receiving a uniformly sampled value ε from the verifier, the prover calculates a function

$$f(X) := f_0(X) + \sum_{i=1}^{N} \varepsilon^i f_i(X), \tag{69}$$

and applies the FRI protocol to it.

In [37], the function f is computed as $f_0(X) + \sum_{i=1}^{N} \varepsilon^i f_i(X)$, where a distinct random value $\varepsilon_i \in K$ is used for each function f_i rather than relying on powers of a single value ε. Although this alternative is secure, the soundness bound increases linearly with the number of functions N. Therefore, it's reasonable to assume that N is sublinear in K to maintain protocol security.

In the batched version, the verifier needs to confirm the correct relationship between functions $f_0, f_1, ..., f_N$ and the initial FRI polynomial $p_0 = f$. The verifier uses evaluations of p_0 received from the prover during each FRI query phase. To facilitate the verification

$$\begin{aligned} p_0(r) &:= f_0(r) + \sum_{i=1}^{N} \varepsilon^i f_i(r) \\ p_0(-r) &:= f_0(-r) + \sum_{i=1}^{N} \varepsilon^i f_i(-r), \end{aligned} \tag{70}$$

the prover sends evaluations of functions $f_0, f_1, ..., f_N$ at both r and $-r$, allowing the verifier to perform a local consistency check between $f_0, f_1, ..., f_N$ and p_0. These evaluations are accompanied by their respective Merkle tree paths.

Similar to the non-batched FRI protocol, both the query phase and the batched consistency check are repeated multiple times to ensure soundness. The soundness error is described in a theorem that informally adapts Theorems 7.2 and 8.3 from [37].

4.2.6 FRI as PCS

A polynomial commitment scheme is a specialized type of commitment scheme where the subject of the commitment is a low-degree polynomial. This allows the prover to commit to a specific polynomial p, constrained by a predetermined degree limit, without revealing all its evaluations at once. The prover can later disclose the value of $p(r)$ at a point r chosen by the verifier. During the initial commitment phase, although the prover does not share all evaluations of p with the verifier, the commitment still effectively locks the prover into that particular polynomial p. Consequently, when the verifier later requests the value of $p(r)$ at any point r of their choosing, the prover is obligated to provide the correct $p(r)$ based on the polynomial p fixed at the time of the original commitment. Importantly, the prover cannot adaptively select the polynomial p based on the query point r without violating the computational assumptions that underpin the security of the commitment scheme.

One might consider using a Merkle tree to construct a polynomial commitment scheme by having the prover P commit to a polynomial p. In this setup, P would generate a Merkle commitment to a string that includes all evaluations of p, namely $p(l_1), \dots, p(l_N)$, where $l_1, \dots, l_N$ represent all possible input values for the polynomial. When the verifier V requests $p(l_i)$, the prover can supply $p(l_i)$ along with its associated authentication information, which consists of $O(logN)$ hash values.

However, this methodology falls short of creating a true polynomial commitment scheme. While the Merkle tree does indeed bind the prover PP to a specific string of evaluations, it doesn't ensure that this string corresponds to evaluations of a genuine multilinear polynomial. In other words, when V asks for $p(r)$ for some input r, P is compelled by the Merkle tree's binding property to provide the r-th entry from the committed string, along with corresponding authentication data. Nevertheless, V has no means to verify that this string represents evaluations of an actual multilinear polynomial. The string could, in principle, consist of evaluations from an entirely arbitrary function.

To resolve the shortcomings of using Merkle trees alone, we integrate them with a low-degree test. This test guarantees that the prover is committed not just to a generic, possibly unstructured string, but specifically to a string that contains evaluations of a low-degree polynomial. More precisely, the test ensures that the committed string is `approximate` to the evaluation table of a low-degree polynomial. Although this produces an object that is not a full-fledged polynomial commitment scheme, it adds a layer of assurance.

The low-degree test provides this guarantee while examining only a small subset of the string's entries—often a quantity logarithmic in the string's length. This minimizes the amount of authentication information that the prover needs to transmit, thereby reducing communication overhead. Specifically, the low-degree test keeps this overhead lower than what would be required if the prover were to send a complete description of the polynomial to the verifier.

4.3 Arguments

In this section, we introduce `arguments` that extend the vanilla STARK protocol. Here, an argument refers to a relation between polynomials that cannot be directly expressed as an identity. We often call these arguments non-identity constraints. The three arguments presented are multiset equality, connection, and inclusion.

The protocols instantiating these arguments follow a common approach - computing a grand product polynomial over the vectors involved in the argument. Specifically, a polynomial is cumulatively computed by taking the quotient of a function of the first vector and a function of the second vector. The prover then provides identities to assure the verifier that the grand product was computed correctly and satisfies the protocol's intended relation. To ensure soundness, the computation incorporates random values sampled by the verifier.

4.3.1 Schwartz-Zippel Lemma

Let P be a n-variable polynomial $P = F(x_1, ..., x_n)$ of degree d over a finite field F. Let S be a subset of the finite field F. If $r_1, ..., r_n$ are chosen randomly from S, then the probability that the polynomial evaluates to zero is negligible, i.e.:

$$\Pr[P(r_1, ..., r_n) = 0] \leq \frac{d}{|S|} \tag{71}$$

In the univariate case, a degree d polynomial has at most d roots. According to the Schwartz-Zippel lemma, a nonzero polynomial will be evaluated to zero with negligible probability if the evaluation points are randomly chosen from a sufficiently large domain. This lemma is useful for analyzing the soundness of proofs involving polynomial evaluations.

4.3.2 Sum Check Protocol

A Polynomial-valued expression of $P(x)$ is $(x_i, y_i), i = 1, ..., n$, where x_i is x-coordinate and y_i is y-coordinate. For example, let x-coordinate be $(0, 1, 2, 3)$ and y-coordinate be $(-2, 1, 0, 1)$. Note that $y_1 = y_3 = 1$. Using the Lagrange interpolating polynomial, it can calculate the polynomial coefficient expression of $X(x)$ and $Y(x)$ respectively,

$$
\begin{aligned}
X(x) &= x, && (72)\\
Y(x) &= x^3 - 5x^2 + 7x - 2. && (73)\\
& && (74)
\end{aligned}
$$

Definition 10. *The recursive and initial term of the accumulator is as follows:*

$$
\begin{aligned}
P(n+1) &= P(n) \cdot (u + X(n) + v \cdot Y(n)) && (75)\\
P(0) &= 1 && (76)\\
n &= 0, 1, 2, ... && (77)
\end{aligned}
$$

Where u and v are two random numbers.

Therefore, based on the above recursion formula, the general expression of the accumulator can be calculated as follows:

$$
\begin{aligned}
P(n) &= P(n-1) \cdot (u + X(n-1) + v \cdot Y(n-1))\\
&= P(n-2) \cdot (u + X(n-2) + v \cdot Y(n-2)) \cdot (u + X(n-1) + v \cdot Y(n-1))\\
&= ...\\
&= P(0) \prod_{i=0}^{n-1} (u + X(i) + v \cdot Y(i))\\
&= \prod_{i=0}^{n-1} (u + X(i) + v \cdot Y(i))
\end{aligned} \tag{78}
$$

For the following mapping

$$
\begin{pmatrix} X(i) \\ \downarrow \\ Y(i) \end{pmatrix} = \begin{pmatrix} x_0, x_1, x_2, x_3 \\ \downarrow \\ y_0, y_1, y_2, y_3 \end{pmatrix} = \begin{pmatrix} 0, 1, 2, 3 \\ \downarrow \\ -2, 1, 0, 1 \end{pmatrix}, \tag{79}
$$

the corresponding value of the accumulator is

$$
P(4) = (u + x_0 + v \cdot y_0) \cdot (u + x_1 + v \cdot y_1) \cdot (u + x_2 + v \cdot y_2) \cdot (u + x_3 + v \cdot y_3) \tag{80}
$$

Perform transformation (commutative law of multiplication) on $P(4)$, we obtain $\tilde{P}(4)$

$$\tilde{P}(4) = (u + x_0 + v \cdot y_0) \cdot (u + x_3 + v \cdot y_3) \cdot (u + x_2 + v \cdot y_2) \cdot (u + x_1 + v \cdot y_1)\,. \quad (81)$$

Since $y_1 = y_3$, permute y_1 and y_3, we obtain $P(4)'$

$$P(4)' = (u + x_0 + v \cdot y_0) \cdot (u + x_3 + v \cdot y_1) \cdot (u + x_2 + v \cdot y_2) \cdot (u + x_1 + v \cdot y_3)\,. \quad (82)$$

It is obvious that

$$P(4) = \tilde{P}(4) = P(4)'. \quad (83)$$

The corresponding map of $P(4)$ is

$$\begin{pmatrix} X(i) \\ \downarrow \\ Y(i) \end{pmatrix} = \begin{pmatrix} x_0, x_1, x_2, x_3 \\ \downarrow \\ y_0, y_1, y_2, y_3 \end{pmatrix} = \begin{pmatrix} 0, 1, 2, 3 \\ \downarrow \\ -2, 1, 0, 1 \end{pmatrix}, \quad (84)$$

while the corresponding map of $P(4)'$ is

$$\begin{pmatrix} X(i) \\ \downarrow \\ Y(i) \end{pmatrix} = \begin{pmatrix} x_0, x_3, x_2, x_1 \\ \downarrow \\ y_0, y_1, y_2, y_3 \end{pmatrix} = \begin{pmatrix} 0, 3, 2, 1 \\ \downarrow \\ -2, 1, 0, 1 \end{pmatrix}. \quad (85)$$

Therefore, if $y_1 = y_3$, by changing the x-coordinate, the value of the accumulator remains unchanged $P(4) = P(4)'$. Therefore, the following inference can be drawn:
Inference 4. *By changing the x-coordinates $x_i \rightleftarrows x_j$, if the corresponding y-coordinates are equal $y_i = y_j$, then the value of the accumulator remains unchanged $P(n) = P(n)'$, where $n = \max\{i, j\} + 1$.*
Theorem 1. *Conversely, by changing the x-coordinates $x_i \rightleftarrows x_j$, if the values of the accumulator are equal $P(n) = P(n)'$, then the two y-coordinates are equal $y_i = y_j$, ensuring the line constraints are met.*

Proof. In the above example, by changing the x-coordinates $x_1 \rightleftarrows x_3$, and $P(4) = P(4)'$, then

$$\begin{gathered} (u + x_0 + v \cdot y_0) \cdot (u + x_1 + v \cdot y_1) \cdot (u + x_2 + v \cdot y_2) \cdot (u + x_3 + v \cdot y_3) \\ = \\ (u + x_0 + v \cdot y_0) \cdot (u + x_3 + v \cdot y_1) \cdot (u + x_2 + v \cdot y_2) \cdot (u + x_1 + v \cdot y_3) \end{gathered} \quad (86)$$

Since u and v are random numbers, according to the Schwartz-Zippel Lemma, except for negligible probabilities, the corresponding items in the above equation are equal. There are only the following two situations:
Case 1:

$$\begin{aligned} u + x_1 + v \cdot y_1 &= u + x_3 + v \cdot y_1 \\ u + x_3 + v \cdot y_3 &= u + x_1 + v \cdot y_3 \end{aligned} \quad (87)$$

Case 2:

$$\begin{aligned} u + x_1 + v \cdot y_1 &= u + x_1 + v \cdot y_3 \\ u + x_3 + v \cdot y_3 &= u + x_3 + v \cdot y_1 \end{aligned} \tag{88}$$

In case 1, when the formulation is simplified, the equality does not hold. In case 2, when the formulation is simplified, the equality holds, which yields $y_1 = y_3$. This completes the proof. $\square$

4.3.3 Permutation Check

Let $\{L_i\}_{i \in [n]}$ be the Lagrange basis on group $\mathbb{G}$.

$$\begin{aligned} L_i(j \cdot G) = 1, i = j \\ L_i(j \cdot G) = 0, i \neq j \end{aligned} \tag{89}$$

For $f, g \in \mathbb{F}_{<n}[X]$, the permutation map is $\sigma : [n] \to [n]$, $i \cdot G \in \mathbb{G}, i = 1, ..., n$. The permutation is

$$g = \sigma(f). \tag{90}$$

`Public preprocessing polynomials:` On the polynomial domain $\mathbb{F}_{<n}[X]$, polynomials $S_{ID}(g^i) = i, S_\sigma(g^i) = \sigma(i), i \in [n]$ and $g \in \mathbb{F}_{<n}[X]$.
`Prover's secret polynomial:` $f \in \mathbb{F}_{<n}[X]$.
`Verifier:` Chooses two random numbers (β, γ) which he sends to the Prover.
`Prover:` Computes:

$$\begin{aligned} f'(i \cdot G) = f(i \cdot G) + \beta \cdot i + \gamma \\ g'(i \cdot G) = g(i \cdot G) + \beta \cdot \sigma(i) + \gamma \end{aligned} \tag{91}$$

Then computes:

$$Z(i \cdot G) = \prod_{1 \leq j \leq i} \frac{f'(i \cdot G)}{g'(i \cdot G)}, i = 2, ..., n, \tag{92}$$

where $Z(G) = 1$. He sends the coordinatewise product value Z.
`Verifier:` for all $a \in \mathbb{G}$, compute

$$\begin{aligned} L_1(a)\,(Z(a) - 1) = 0 \\ Z(a)f'(a) = g'(a)Z(a \cdot G) \end{aligned} \tag{93}$$

If $a = 1 \cdot G$, then $L_1(1 \cdot G) = 1$, so

$$L_1(G)(Z(G) - 1) = 0 \Rightarrow Z(G) = 1 \tag{94}$$

If $a = i \cdot G, i = 2, ..., n$, then $L_1(i \cdot G) = 0$. So $L_1(a)(Z(a) - 1) = 0$ holds.

The order of $\mathbb{G}$ is n, so $G = (n+1) \cdot G$, thus

$$\begin{aligned} 1 &= Z(G) \\ &= Z((n+1) \cdot G) \\ &= Z(n \cdot G) \frac{f'(n \cdot G)}{g'(n \cdot G)} \\ &= Z((n-1) \cdot G) \frac{f'((n-1) \cdot G)}{g'((n-1) \cdot G)} \frac{f'(n \cdot G)}{g'(n \cdot G)} \\ &= ... \\ &= Z(G) \prod_{i=1}^{n} \frac{f'(i \cdot G)}{g'(i \cdot G)} \\ &= \prod_{i=1}^{n} \frac{f'(i \cdot G)}{g'(i \cdot G)} \\ &= \prod_{i=1}^{i} \frac{f(i \cdot G) + \beta \cdot i + \gamma}{g(i \cdot G) + \beta \cdot \sigma(i) + \gamma} \end{aligned} \tag{95}$$

So the coordinatewise product values computed by each are equal:

$$\prod_{i=1}^{i} f(i \cdot G) + \beta \cdot i + \gamma = \prod_{i=1}^{i} g(i \cdot G) + \beta \cdot \sigma(i) + \gamma \tag{96}$$

According to Schwartz-Zippel Lemma and Sum Check Protocol, corresponding terms are equal except with negligible probability. Therefore, the function value $g(i \cdot G)$ at index i equals the function value $f(\sigma(i) \cdot G)$ at index $\sigma(i)$, i.e.,

$$g(i \cdot G) = f(\sigma(i) \cdot G) \tag{97}$$

So the permutation relation $g = \sigma(f)$ is satisfied, achieving the permutation constraint.

4.3.4 Multiset Equality

Given the coefficients $a_1, ..., a_n$ of a polynomial f and the coefficients $b_1, ..., b_n$ of a polynomial g over a finite field $\mathbb{F}$, and a subset $H = x_1, ..., x_n \subset \mathbb{F}$.

(1) If the elements in the two sets $a_i, i = 1, ..., n$ and $b_i, i = 1, ..., n$ correspond equally $a_i = b_i, i \in [n]$, then the product values must be equal

$$\prod_{i \in [n]} a_i = \prod_{i \in [n]} b_i. \tag{98}$$

(2) Conversely, if the accumulators are equal $\prod_{i\in[n]} a_i = \prod_{i\in[n]} b_i$, then by the Schwartz-Zippel Lemma, we can construct a polynomial

$$P(X_1, ..., X_n) := \prod_{i\in[n]} (X_i - a_i) = 0. \tag{99}$$

Randomly choosing $(b_1, ..., b_n)$ from the subset H makes the polynomial $P(X_1, ..., X_n)$ equal to zero with negligible probability.

Therefore, if the polynomial $P(X_1, ..., X_n)$ equals zero, then $(b_1, ..., b_n)$ is a solution, so the elements in the two sets correspond equally $a_i = b_i, i \in [n]$. Thus, it can be deduced that polynomial f equals polynomial g.

By adding a random number, the product equality theorem can be reduced to multiset equality detection. Given two vectors $\vec{a} = (a_1, ..., a_n), \vec{b} = (b_1, ..., b_n)$, detect whether the two vectors contain the same elements, even in different degree counting repeats.

Reducing multiset equality to product consistency

The verifier V chooses a random $\gamma \in F$, and for the random shifted vectors $a' \triangleq a + \gamma, b' \triangleq b + \gamma$ (γ is added to all coordinates), it detects as follows

$$\prod_{i\in[n]} (a_i + \gamma) = \prod_{i\in[n]} (b_i + \gamma). \tag{100}$$

For γ in the polynomial, the Schwarz-Zippel Lemma shows that unless set $a' = b'$, the above equation holds with negligible probability.

5 Lookup Tables

zk-SNARKs have undergone significant refinements, primarily aimed at enhancing their succinctness and reducing both prover and verifier computation times. However, the majority of SNARKs are still optimized for arithmetic operations that can be readily transformed into polynomials, often described as `SNARK-friendly`. In contrast, operations considered `SNARK-unfriendly` have been largely overlooked. To address this gap, Lookup protocols were introduced, providing a means to handle some of these SNARK-unfriendly operations effectively.

5.1 Univariate Lookup

Fix integers n, d and vectors $f \in \mathbb{F}^n$, $t \in \mathbb{F}^d$. The notation $f \subset t$ means $\{f_i\}_{i\in[n]} \subset \{t_i\}_{i\in[d]}$. Let $\mathbb{H} = \{G, 2 \cdot G, \ldots, (n+1) \cdot G\}$ be a multiplicative subgroup of order $n+1$ in $\mathbb{F}$. For a polynomial $f \in \mathbb{F}[X]$ and $i \in [n+1]$ denote as $f_i := f(i \cdot G)$. For a vector $f \in \mathbb{F}^n$, it also denotes by f the polynomial in $\mathbb{F}_{<n}[X]$ with $f(i \cdot G) = f_i$. If $f \subset t$, then f is sorted by t when values appear in the same degree in f as they do in t. Formally, for any $i < i' \in [n]$ such that $f_i \neq f_{i'}$, if $j, j' \in [d]$ are such that $t_j = f_i, t_{j'} = f_{i'}$ then $j < j'$.

Now, given $t \in \mathbb{F}^d$, $f \in \mathbb{F}^n$, $s \in \mathbb{F}^{n+d}$, define bi-variate polynomials Ψ and Ω as

$$\Psi(\beta,\gamma) := (1+\beta)^{n+d-1} \cdot \prod_{i\in[n]} (\gamma + f_i) \prod_{i\in[d-1]} (\gamma + (t_i + \beta \cdot t_{i+1})/1+\beta) \tag{101}$$

$$\Omega(\beta,\gamma) := (1+\beta)^{n+d-1} \cdot \prod_{i\in[n+d-1]} (\gamma + (s_i + \beta \cdot s_{i+1})1+\beta) \tag{102}$$

$$\tag{103}$$

Theorem 2. *$\Psi(\beta,\gamma) \equiv \Omega(\beta,\gamma)$ if and only if $f \subset t$, $s = (f,t)$ sorted by t.*

Proof. (1) Suppose that $f \subset t$ and $s = (f,t)$ sorted by t. Then for each $j \in [d-1]$, there is a distinct index $i \in [n+d-1]$ such that

$$(t_j, t_{j+1}) = (s_i, s_{i+1}). \tag{104}$$

Therefore, the corresponding factors in Ψ and Ω are equal. That is,

$$(\gamma + (t_i + \beta \cdot t_{j+1})/(1+\beta)) = (\gamma + (s_i + \beta \cdot s_{j+1})/(1+\beta)). \tag{105}$$

Let $P' \subset [n+d-1]$ be these $d-1$ indices i, and the complementary set is $P = [n+d-1]\backslash P'$. The n indices $i \in P$ are such that $s_i = s_{i+1}$, and $\{f_s\}_{g^i \in P}$ equals $\{f_j\}_{j\in[n]}$ as multisets. For each $i \in P$, the factor of Ω, will be $\gamma + (s_i + \beta \cdot s_{i+1})/(1+\beta) = \gamma + s_i$ which equals the factor $\gamma + f_{j(i)}$ in F. Therefore, $\Psi(\beta,\gamma) \equiv \Omega(\beta,\gamma)$.

(2) For the other direction given $\Psi \equiv \Omega$. Since $\mathbb{F}(\beta)[\gamma]$ is a unique factorization domain, we know that the linear factors of Ψ and Ω, as written above must be equal. Thus, for each $i \in [d-1]$, Ω must have a factor equal to $\gamma + (t_i + \beta \cdot t_{i+1})/(1+\beta)$. In other words, for some $j \in [n+d-1]$,

$$\gamma + (t_i + \beta \cdot t_{i+1})/(1+\beta) = \gamma + (s_i + \beta \cdot s_{i+1})/(1+\beta). \tag{106}$$

According to Schwartz-Zippel Lemma, we have $t_i = s_j, t_{i+1} = s_{j+1}$.

Call $P' \subset [n+d-1]$ the set of these $d-1$ indices j. For any index $j \in [n+d-1]\backslash P'$, there must be a factor coming from f in Ψ that equals the corresponding factor in Ω. More precisely, for such j there exists $i \in [n]$ such that

$$\gamma + f_i = \gamma + (s_j + \beta \cdot s_{j+1})/(1+\beta). \tag{107}$$

According to Schwartz-Zippel Lemma, we have $f_i = s_j = s_{j+1}$. Therefore, $f \subset t$ and $s = (f,t)$ sorted by t.

5.1.1 Lookup Table

1. $s = (f,t)$ sorted by t; $h_1(i \cdot G) = s_i$ for $i \in [n+1]$; and $h_2(i \cdot G) = s_{n+i}$ for each $i \in [n+1]$.
2. P computes h_1, h_2 and commits them.

3. V chooses random $\beta, \gamma \in \gneqq$ and sends to P.
4. P computes a polynomial $Z \in \mathbb{F}_{<n+1}[X]$ that aggregates $\Psi(\beta,\gamma)/\Omega(\beta,\gamma)$. Let

$$Z(G) = 1 \tag{108}$$

$$Z(i \cdot G) = \frac{(1+\beta)^{n+d-1} \cdot \prod_{i\in[n]} (\gamma + f_i) \prod_{i\in[d-1]} (\gamma + (t_i + \beta \cdot t_{i+1})/1+\beta)}{(1+\beta)^{n+d-1} \cdot \prod_{i\in[n+d-1]} (\gamma + (s_i + \beta \cdot s_{i+1})1+\beta)}, i = 2, ..., n \tag{109}$$

$$Z((n+1)\cdot G) = 1 \tag{110}$$

$$\tag{111}$$

5. P sends Z to V who checks that:

$$L_1(x)(Z(x)-1) = 0, \tag{112}$$

$$((x-(n+1)\cdot G)Z(x)(1+\beta)) \cdot (\gamma + f(x)) \cdot (\gamma(1+\beta) + (t(x) + \beta \cdot t(x \cdot G)) \tag{113}$$

$$= (x-(n+1)\cdot G)Z(x\cdot G)\,(\gamma(1+\beta)+h_1(x)+\beta\cdot h_1(x\cdot G))\,(\gamma(1+\beta)+h_2(x)+\beta\cdot h_2(x\cdot G)), \tag{114}$$

$$L_{n+1}(x)(h_1(x) - h_2(x\cdot G)) = 0, \tag{115}$$

$$L_{n+1}(x)(Z(x)-1) = 0. \tag{116}$$

$$\tag{117}$$

Theorem 3. *If $\{f(g^i)\}_{i\in[n]} \not\subset \{t(g^i)\}_{i\in[n+1]}$, then in the above protocol, for any adversary $\mathcal{A}$ running the protocol in the role of the prover $\mathcal{P}$, the probability that the verifier $\mathcal{V}$ accepts is negligible. Furthermore, the proof length in the above protocol is $5n+4$.*

Proof. The adversary $\mathcal{A}$ acting as the prover $\mathcal{P}$ constructs three polynomials $h_1, h_2, Z \in \mathbb{F}_{<n+1}[X]$ and sends the proof to the verifier $\mathcal{V}$, where these three polynomials have length $3n+3$. The verifier $\mathcal{V}$ performs consistency checks of the second equation which has the highest degree, containing the product of these three polynomials h_1, h_2, Z and the linear polynomial $X-(n+1)\cdot G$. Among them, f has n elements, t has $n+1$ elements, Z has $n+1$ elements, the vanish polynomial H has $n+1$ elements, so the degree of the quotient polynomial is $2n+1$. Therefore, the proof length is $(3n+3)+(2n+1) = 5n+4$.

The first equation of the verification ensures $Z(g) = 1$, the fourth equation ensures $Z(g^{n+1}) = 1$; the third equation shows that $h_1(g^{n+1}) = h_2(g)$, so h_1, h_2 consistently describe a single vector $s \in \mathbb{F}^{2n+1}$.

Based on the necessary and sufficient condition, for any arbitrarily chosen s, if the set $\{t(g^i)\}_{i\in[n+1]}$ does not contain the set $\{f(g^i)\}_{i\in[n]}$, then the polynomials $\Psi(X,Y)$

and $\Omega(X, Y)$ are different. Based on the Schwartz-Zippel Lemma, except for a negligible probability, randomly choosing β, γ will make $\Psi(\beta, \gamma) \neq \Omega(\beta, \gamma)$, so $Z(g^{n+1}) \neq 1$. Therefore, the probability that the verifier $\mathcal{V}$ accepts is negligible.

On the contrary, if the verifier $\mathcal{V}$ accepts, then according to the 4th checking equation, it is ensured that $Z(g^{n+1}) = 1$. Combining the 1st and 2nd equations, the following equation holds

$$\begin{aligned}
&Z(g^{n+1}) \\
&= Z(g^n) \frac{(1+\beta) \cdot (\gamma + f(g^n))(\gamma(1+\beta) + t(g^n) + \beta t(g^{n+1}))}{(\gamma(1+\beta) + h_1(g^n) + \beta h_1(g^{n+1}))(\gamma(1+\beta) + h_2(g^n) + \beta h_2(g^{n+1}))} \\
&= Z(g^{n-1}) \frac{(1+\beta)^2 \cdot \prod_{j=n-1,n} (\gamma + f(g^j)) \prod_{j=n-1,n} (\gamma(1+\beta) + t(g^j) + \beta t(g^{j+1}))}{\prod_{j=n-1,n} (\gamma(1+\beta) + h_1(g^n) + \beta h_1(g^{n+1}))(\gamma(1+\beta) + h_2(g^n) + \beta h_2(g^{n+1}))} \\
&= \ldots \\
&= Z(g) \frac{(1+\beta)^n \cdot \prod_{j<n+1} (\gamma + f_j) \cdot \prod_{1<j<n+1} (\gamma(1+\beta) + t_j + \beta t_{j+1})}{\prod_{1<j<n+1} (\gamma(1+\beta) + s_j + \beta s_{j+1})(\gamma(1+\beta) + s_{n+j} + \beta s_{n+j+1})} \\
&= 1
\end{aligned} \tag{118}$$

Therefore, $\Psi \equiv \Omega$. It can be deduced that $f \subset t$. Therefore, the witness of the prover $\mathcal{P}$ is valid under the input-output constraints in the table, and the computation of the prover $\mathcal{P}$ is correct. □

5.2 Cached quotients Lookups

5.2.1 Logarithmic Derivative.

The Cached Quotient (CQ) protocol for lookup arguments utilizes the logarithmic derivative trick to establish the equality between two polynomials $p(x) = \prod_{a \in A}(x+a)$ and $q(x) = \prod_{b \in B}(x+b)$. According to the logarithmic derivative trick, $p(x)$ and $q(x)$ are equal if and only if their corresponding rational functions are equal:

$$\frac{p'(x)}{p(x)} = \sum_{a \in A} \frac{1}{x+a}, \tag{119}$$

$$\frac{q'(x)}{q(x)} = \sum_{b \in B} \frac{1}{x+b}. \tag{120}$$

The forward direction, $p(x) = q(x) \Rightarrow \frac{p'(x)}{p(x)} = \frac{q'(x)}{q(x)}$, is trivial. For the converse, assume that $\frac{p'(x)}{p(x)} = \frac{q'(x)}{q(x)}$. Then:

$$\left(\frac{p(x)}{q(x)}\right)' = \frac{p'(x)q(x) - q'(x)p(x)}{p^2(x)} = 0 \tag{121}$$

This implies that $\frac{p(x)}{q(x)} = c$ for some constant c. Given that both $p(x)$ and $q(x)$ have leading coefficients of 1, it follows that $c = 1$, and thus $p(x) = q(x)$.

To verify that a set of lookups f is contained within a table t, the following equality should hold:

$$\sum_{i=1}^{N} \frac{m_i}{x + t_i} = \sum_{j=1}^{m} \frac{1}{x + f_j} \tag{122}$$

Here, m_i represents the multiplicity of the value t_i in the lookup f_j. Note that each t_i is unique, f_j's can be repeated, and m_i's can be zero for many t_i's. The crux of the CQ protocol is to test this rational function identity at some random point $x = \beta$.

5.2.2 Reformulating Into Sumcheck.

The CQ protocol tests the identity by defining polynomials $A(x)$ and $B(x)$ such that their evaluations at points are given by:

$$A_i = \frac{m_i}{x + t_i}, \quad i = 1, \ldots, N, \tag{123}$$

$$B_j = \frac{1}{x + f_j}, \quad j = 1, \ldots, N. \tag{124}$$

Here, $A_i = A(g^i)$ where g is a generator of a multiplicative subgroup $V \subseteq F$ of order N, $V = \{g, g^2, \ldots, g^N = 1\}$. Similarly, $B_j = B(\omega^j)$ where ω is a generator of a multiplicative subgroup $H \subseteq F$ of order m, $H = \{\omega, \omega^2, \ldots, \omega^m = 1\}$.

To satisfy the required relationship 122 at the random point β, A_i and B_j must satisfy $\sum_i A_i = \sum_j B_j$. Since these are evaluations of polynomials defined over multiplicative subgroups, they obey:

$$\sum_{i=1}^{N} A_i = N \cdot A(0), \tag{125}$$

$$\sum_{j=1}^{m} B_j = m \cdot B(0). \tag{126}$$

The prover then must prove

$$N \cdot A(0) = m \cdot B(0), \tag{127}$$

which constitutes a univariate sumcheck problem.

5.2.3 Quotient Polynomials.

Consider the polynomials:

$$p(x) = A(x)(T(x) + \beta) - m(x), \tag{128}$$

$$q(x) = B(x)(F(x) + \beta) - 1, \tag{129}$$

where $T(x)$, $m(x)$, and $F(x)$ are the polynomials whose evaluations are t_i, m_i, and f_j respectively. Clearly, $p(x)$ must evaluate to zero on V and $q(x)$ must evaluate to zero on H. Hence, we can define quotient polynomials $Q_A(x)$ and $Q_B(x)$ as:

$$Q_A(x) = \frac{A(x)(T(x)+\beta) - m(x)}{Z_V(x)} \tag{130}$$

$$Q_B(x) = \frac{B(x)(F(x)+\beta) - 1}{Z_H(x)}, \tag{131}$$

where $Z_V(x)$ and $Z_H(x)$ are the vanishing polynomials on V and H respectively. The prover must now prove two things:

- That they know the polynomials $A(x)$, $B(x)$, $Q_A(x)$, $Q_B(x)$, $F(x)$, and m.
- That the polynomials satisfy the required relationships 127, 130, and 131.

This is achieved using KZG commitments, which can subsequently be verified using pairings. The details of these verifications are elaborated in the following subsections.

5.2.4 Proving Knowledge of the Polynomial $A(x)$ and its Sum.

To prove knowledge of a polynomial $\varphi(x)$ using a KZG commitment $[\varphi(x)]_1$, the prover follows these steps:

1. Evaluates $\varphi(x)$ at a specific point z.
2. Defines a new polynomial P_φ as follows:

$$P_\varphi(x) = \frac{\varphi(x) - \varphi(z)}{x - z} \tag{132}$$

3. Sends a commitment $[P_\varphi]_1$.

The verifier can then check the commitment using the pairing equation:

$$\hat{e}([\varphi(x)]_1 - [\varphi(z)]_1 + z \cdot [P_\varphi]_1, [1]_2) = \hat{e}([P_\varphi]_1, [x]_2) \tag{133}$$

To prove knowledge of another polynomial $A(x)$, the prover:

1. Evaluates $A(x)$ at $x = 0$.
2. Defines and commits to a polynomial $P_A(x)$:

$$P_A(x) = \frac{A(x) - A(0)}{x} \tag{134}$$

3. The commitment $[P_A]_1$ can then be verified as:

$$\hat{e}([A(x)]_1 - [A(0)]_1, [1]_2) = \hat{e}([P_A(x)]_1, [x]_2) \tag{135}$$

5.2.5 Low Degree Testing of $B(x)$.

To prove that $B(x)$ is of degree $< m$, the prover uses $P_B(x)$:

$$P_B(x) = \frac{B(x) - B(0)}{x} \tag{136}$$

Then, the prover commits to $[P_B(x) \cdot x^{N-m+1}]_1$ and verifies it as:

$$\hat{e}([P_B(x)]_1, [x^{N-m+1}]_2) = \hat{e}([P_B(x) \cdot x^{N-m+1}]_1, [x]_2) \tag{137}$$

Notice that any terms of $B(x)$ with degree $\geq m$ would appear as low-degree terms in the commitment $[P_B(x) \cdot x^{N-m+1}]_1$, since the SRS does not support commitments of terms with degree $\geq N$.

5.2.6 Proving the Relation 130.

To verify the relation 130, we utilize the following equation:

$$\hat{e}([A(x)]_1, [T(x)]_2) = \hat{e}([Q_A(x)]_1, [Z_V(x)]_2) \cdot e([m(x)]_1 - \beta[A(x)]_1, [1]_2). \tag{138}$$

Here, $[T]_2, [Z_V]_2, [1]_2, [x]_2$ are independent of the lookups and can therefore be precomputed.

The prover needs to compute several elements: $[A(x)]_1$, $[Q_A(x)]_1$, $[m(x)]_1$, $A(0)$, $h_A(x) - xA(0)i_1$. Almost all of these computations are $O(m)$ since they are only necessary for the lookups. If t_i is not in the lookups, then m_i and A_i are zero. The challenge arises in computing $[Q_A(x)]_1$, which is $O(N)$. To mitigate this, we precompute cached quotients $Q_i(x)$ as follows:

$$Q_i(x) = \frac{L_i(x)(T(x) - t_i)}{Z_V(x)} = \frac{T(x) - t_i}{k^i(x - g^i)} \tag{139}$$

Here, $L_i(x)$ are the Lagrange polynomials defined as:

$$L_i(x) = \frac{Z_V(x)}{k^i(x - g^i)} \tag{140}$$

$$k^i = Z_V'(g^i) = (x^N - 1)'\Big|_{x=g^i} = N \cdot g_i^{N-1} = \frac{N}{g^i} \tag{141}$$

The quotients $Q_i(x)$ are computed in $O(N \log N)$ time using the Number Theoretic Transform (NTT). With these precomputed quotients, $[Q_A(x)]_1$ can be computed in $O(m)$ time as:

$$[Q_A(x)]_1 = \sum_{A_i \neq 0} A_i \cdot [Q_i(x)]_1 \tag{142}$$

Here, A_i are the evaluations of the polynomial $A(x)$ as defined in Eq. 123.

5.2.7 Proving Relations (127, 131).

To prove the last set of relations, we introduce a new random point $x = \gamma$. The goal is to verify the knowledge of $F(\gamma)$ and $P_B(\gamma)$ through commitment to corresponding polynomials. We commit the polynomials as follows:

$$P_F(x) = \frac{F(x) - F(\gamma)}{x - \gamma}, \tag{143}$$

$$P_{B,\gamma}(x) = \frac{P_B(x) - P_B(\gamma)}{x - \gamma}. \tag{144}$$

Verification is carried out using the following pairings:

$$\hat{e}([F(x)]_1 - [F(\gamma)]_1 + \gamma[P_F(x)]_1, [1]_2) = \hat{e}([P_F(x)]_1, [x]_2), \tag{145}$$

$$\hat{e}([P_B(x)]_1 - [P_B(\gamma)]_1 + \gamma[P_{B,\gamma}(x)]_1, [1]_2) = \hat{e}([P_{B,\gamma}(x)]_1, [x]_2). \tag{146}$$

These equations confirm that $F(x)$ and $B(x)$ evaluate to $F(\gamma)$ and $B(\gamma)$ respectively.

To prove the targeted relations, we define:

$$Q_{b,\gamma} = \frac{(B(\gamma) - B(0) + A(0) \cdot \frac{N}{m})(F(\gamma) + \beta) - 1}{Z_H(\gamma)}, \tag{147}$$

$$P_{Q_B}(x) = \frac{Q_B(x) - Q_{b,\gamma}}{x - \gamma}. \tag{148}$$

We then commit $[Q_{b,\gamma}]_1$ and $[P_{Q_B}(x)]_1$, which can be verified by:

$$\hat{e}([Q_B(x)]_1 - [Q_{b,\gamma}]_1 + \gamma[P_{Q_B}(x)]_1, [1]_2) = \hat{e}([P_{Q_B}(x)]_1, [x]_2). \tag{149}$$

This construction provides evidence for the following:

$$Q_B(\gamma) = \frac{B(\gamma)(F(\gamma) + \beta) - 1}{Z_H(\gamma)}, \tag{150}$$

$$B(0) \cdot m = A(0) \cdot N. \tag{151}$$

Thus, the proofs for relations (127, 131) are finalized.

5.3 LogUP

5.3.1 The Lagrange kernel of the boolean hypercube

Let F denote a finite field, and F^* its multiplicative group. We regard the boolean hypercube $H = \{\pm 1\}^n$ as a multiplicative subgroup of $(F^*)^n$. For a multivariate function $f(X_1, \ldots, X_n)$, we will often use the vector notation $\vec{X} = (X_1, \ldots, X_n)$ for its arguments, writing $f(\vec{X}) := f(X_1, \ldots, X_n)$.

The *Lagrange kernel* of H is the multilinear polynomial

$$L_H(\vec{X},\vec{Y}) = \frac{1}{2^n}\cdot\prod_{j=1}^{n}(1+X_j\cdot Y_j). \tag{152}$$

It is noteworthy that $L_H(\vec{X},\vec{Y})$ is symmetric in $\vec{X}$ and $\vec{Y}$, i.e. $L_H(\vec{X},\vec{Y}) = L_H(\vec{Y},\vec{X})$, and that (152) is evaluated within only $\mathcal{O}\log|H|$ field operations.

Whenever $\vec{y}\in H$ we have that $L_H(\vec{X},\vec{y})$ is the Lagrange polynomial on H, which is the unique multilinear polynomial which satisfies $L_H(\vec{x},\vec{y}) = 1$ at $\vec{x}=\vec{y}$, and zero elsewhere on H. In particular, for a function $f: H\to F$ the inner product evaluation formula

$$\langle f, L_H(\,.\,,\vec{y})\rangle_H := \sum_{\vec{x}\in H} f(\vec{x})\cdot L_H(\vec{x},\vec{y}) = f(\vec{y}).$$

is valid for every $\vec{y}\in H$. Then, we have:
Let $p(\vec{X})$ be the unique multilinear extension of $f: H\to F$. Then for every $\vec{y}\in F^n$,

$$\langle f, L_H(\,.\,,\vec{y})\rangle_H = \sum_{x\in H} f(\vec{x})\cdot L_H(\vec{x},\vec{y}) = p(\vec{y}). \tag{153}$$

Note that for any $\vec{y}\in F^n$, the domain evaluation of $L_H(\vec{X},\vec{y})$ over H can be computed in $\mathcal{O}|H|$ field operations, by recursively computing the domain evaluation of the partial products $p_k(X_1,\ldots,X_k,y_1,\ldots,y_k) = \frac{1}{2^n}\cdot\prod_{j=1}^{k}(1+X_j\cdot y_j)$ over $H_k = \{\pm 1\}^k$ from the domain evaluation of p_{k-1}, where one starts with $f_0 = \frac{1}{2^n}$ over the single-point domain H_0. Each recursion step costs $|H_{k-1}|$ field multiplications, denoted by M, and the same number of additions, denoted by A, yielding overall

$$\sum_{k=1}^{n}|H_{k-1}|\cdot(\mathsf{M}+\mathsf{A}) < |H|\cdot(\mathsf{M}+\mathsf{A}). \tag{154}$$

5.3.2 The formal derivate

Given a univariate polynomial $p(X)=\sum_{k=0}^{d} c_k\cdot X^k$ over a general (possibly infinite) field F, its *derivative* is defined as

$$p'(X) := \sum_{k=1}^{d} k\cdot c_k\cdot X^{k-1}. \tag{155}$$

As in calculus, the derivative is linear, i.e. for every two polynomials $p_1(X), p_1(X)\in F[X]$, and coefficients $\lambda_1,\lambda_2\in F$,

$$(\lambda_1\cdot p_1(X)+\lambda_2\cdot p_1(X))' = \lambda_1\cdot p_1'(X)+\lambda_2\cdot p_2'(X)$$

and we have the product rule

$$(p_1(X)\cdot p_2(X))' = p_1'(X)\cdot p_2(X) + p_1(X)\cdot p_2'(X).$$

For a function $\frac{p(X)}{q(X)}$ from the rational function field $F(X)$, the derivative is defined as the rational function

$$\left(\frac{p(X)}{q(X)}\right)' := \frac{p'(X)\cdot q(X) - p(X)\cdot q'(X)}{q(X)^2}. \tag{156}$$

By the product rule for polynomials, the definition does not depend on the representation of $\frac{p(X)}{q(X)}$. Both linearity and the product rule extend to rational functions.

For any polynomial $p(X) \in F[X]$, if $p'(X) = 0$ then $p(X) = g(X^p)$ where p is the characteristic of the field F. In particular, if $\deg p(X) < p$, then the polynomial must be constant. As the analogous fact for fractions is not as commonly known, we give proof of the following inference.

Inference 5. *Let F be a field of characteristic $p \neq 0$, and $\frac{p(X)}{q(X)}$ a rational function over F with both $\deg p(X) < p$ and $\deg q(X) < p$. If the formal derivative $\left(\frac{p(X)}{q(X)}\right)' = 0$, then $\frac{p(X)}{q(X)} = c$ for some constant $c \in F$.*

5.3.3 The logarithmic derivative

The *logarithmic derivate* of a polynomial $p(X)$ over a (general) field F is the rational function $p'(X)/p(X)$.

Note that the logarithmic derivative of the product $p_1(X)\cdot p_2(X)$ of two polynomials $p_1(X), p_2(X)$ equals the sum of their logarithmic derivatives since by the product rule we have

$$\frac{(p_1(X)\cdot p_2(X))'}{p_1(X)\cdot p_2(X)} = \frac{p_1'(X)\cdot p_2(X) + p_1(X)\cdot p_2'(X)}{p_1(X)\cdot p_2(X)} = \frac{p_1'(X)}{p_1(X)} + \frac{p_2'(X)}{p_2(X)}.$$

In particular the logarithmic derivative of a product $p(X) = \prod_{i=1}^n (X + z_i)$, with each $z_i \in F$, is equal to the sum

$$\frac{p'(X)}{p(X)} = \sum_{i=1}^n \frac{1}{X + z_i}. \tag{157}$$

The following inference is a simple consequence of inference 5 and essentially states that under quite mild conditions on the field F, if two normalized polynomials have the same logarithmic derivative then they are equal. We state this fact for our use case of product representations.

Inference 6. *Let $(a_i)_{i=1}^n$ and $(b_i)_{i=1}^n$ be sequences over a field F with characteristic $p > n$. Then $\prod_{i=1}^n (X + a_i) = \prod_{i=1}^n (X + b_i)$ in $F[X]$ if and only if*

$$\sum_{i=1}^n \frac{1}{X + a_i} = \sum_{i=1}^n \frac{1}{X + b_i}$$

in the rational function field $F(X)$.

Given a product $p(X) = \prod_{i=1}^N (X + a_i)$ we can gather the poles of its logarithmic derivative obtaining the fractional decomposition

$$\frac{p'(X)}{p(X)} = \sum_{a \in F} \frac{m(a)}{X + a},$$

where $m(a) \in \{1, \ldots, N\}$ is the multiplicity of the value a in $(a_i)_{i=1}^N$. Fractional decompositions are unique, as shown by the following deduction.

Inference 7. *Let F be an arbitrary field and $m_1, m_2 : F \to F$ any functions. Then $\sum_{z \in F} \frac{m_1(z)}{X - z} = \sum_{z \in F} \frac{m_2(z)}{X - z}$ in the rational function field $F(X)$, if and only if $m_1(z) = m_2(z)$ for every $z \in F$.*

This leads to the following algebraic criterion for set membership, which is the key tool for our lookup arguments.

Inference 8 (Set inclusion)**.** *Let F be a field of characteristic $p > N$, and suppose that $(a_i)_{i=1}^N$, $(b_i)_{i=1}^N$ are arbitrary sequences of field elements. Then, $\{a_i\} \subseteq \{b_i\}$ as sets (with multiples of values removed), if and only if there exists a sequence $(m_i)_{i=1}^N$ of field elements from $F_q \subseteq F$ such that*

$$\sum_{i=1}^N \frac{1}{X + a_i} = \sum_{i=1}^N \frac{m_i}{X + b_i} \tag{158}$$

in the function field $F(X)$. Moreover, we have equality of the sets $\{a_i\} = \{b_i\}$, if and only if $m_i \neq 0$, for every $i = 1, \ldots, N$.

5.3.4 Lookups based on the logarithmic derivative

Assume that F is a finite field, and that $f_1, \ldots, f_M$ and $t : H \to F$ are functions over the Boolean hypercube $H = \{\pm 1\}^n$. By deduction 8, it holds that $\bigcup_{i=1}^M \{f_i(\vec{x})\}_{\vec{x} \in H} \subseteq \{t(\vec{x})\}_{\vec{x} \in H}$ as sets, if and only if there exists a function $m : H \to F$ such that

$$\sum_{\vec{x} \in H} \sum_{i=1}^M \frac{1}{X + f_i(\vec{x})} = \sum_{\vec{x} \in H} \frac{m(\vec{x})}{X + t(\vec{x})}, \tag{159}$$

assuming that the characteristic of F is larger than M times the size of the hypercube. If t is injective (which is typically the case for lookup tables) then m is the multiplicity function, counting the number of occurrences for each value $t(\vec{x})$ in $f_1, \ldots, f_M$ altogether, i.e. $m(\vec{x}) = m_f(t(\vec{x})) = \sum_{i=1}^M |\{\vec{y} \in H : f_i(\vec{y}) = t(\vec{x})\}|$. If t is not one-to-one, we set m as the *normalized* multiplicity function

$$m(\vec{x}) = \frac{m_f(t(\vec{x}))}{m_t(t(\vec{x}))} = \frac{\sum_{i=1}^M |\{\vec{y} \in H : f_i(\vec{y}) = t(\vec{x})\}|}{|\{\vec{y} \in H : t(\vec{y}) = t(\vec{x})\}|}. \tag{160}$$

The plot for proving that $\bigcup_{i=1}^M \{f_i(\vec{x})\}_{\vec{x} \in H} \subseteq \{t(\vec{x})\}_{\vec{x} \in H}$ is as follows. Given a random challenge $x \in F$, the prover shows that the rational identity (159) holds at $X = x$, whenever evaluation is possible. However, to make (159) applicable to the sumcheck argument, the prover needs to provide multilinear helper functions for the rational expressions. We use a single multilinear function for the entire fractional expression in (159), which is subject to a domain identity over H which has $\mathcal{O}M$ variables and absolute degree $\mathcal{O}M$. This will lead to a protocol with a $\mathcal{O}M^2$ prover.

We provide a single helper function

$$h(\vec{x}) = \sum_{i=1}^{M} \frac{1}{x + f_i(\vec{x})} - \frac{m(\vec{x})}{x + t(\vec{x})}, \tag{161}$$

subject $\sum_{\vec{x}\in H} h(\vec{x}) = 0$. Correctness of h is ensured by the domain identity

$$\big(h(\vec{x}) \cdot (x + t(\vec{x})) + m(\vec{x})\big) \cdot \prod_{i=1}^{M} (x + f_i(\vec{x})) = (x + t(\vec{x})) \cdot \sum_{i=1}^{M} \prod_{j\neq i} (x + f_j(\vec{x})) \tag{162}$$

over H, and we apply the Lagrange kernel $L_H(\,.\,, \vec{z})$ at a randomly chosen $\vec{z} \in F^n$ to reduce the domain identity to another sumcheck over H. Both sumchecks, the one for h and the one for the domain identity, are then combined into a single one, using another randomness $\lambda \in F$.

Multi-column lookup over $H = \{\pm 1\}^n$
Let $M \geq 1$ be an integer, and F a finite field with characteristic $p > M \cdot 2^n$. Given any functions $f_1, \ldots, f_M, t : H \to F$ on the boolean hypercube $H = \{\pm 1\}^n$, the Lagrange IOP for that $\bigcup_{i=1}^{M} \{f_i(\vec{x}) : \vec{x} \in H\} \subseteq \{t(\vec{x}) : \vec{x} \in H\}$ as sets is as follows.

1. The prover determines the (normalized) multiplicity function $m : H \to F$ as defined in (160), and sends the oracle for m to the verifier. The verifier answers with a random sample $x \in F/\{-t(\vec{x}) : \vec{x} \in H\}$.
2. Given the challenge x from the verifier, the prover computes the randomized functions $\varphi_i(\vec{x}) = x + f_i(\vec{x})$, $i = 1, \ldots, M$, and $\tau(\vec{x}) = x + t(\vec{x})$. It determines the values for
$$h(\vec{x}) = \sum_{i=1}^{M} \frac{1}{\varphi_i(\vec{x})} - \frac{m(\vec{x})}{\tau(\vec{x})}, \tag{163}$$
over H, and sends the oracle for h to the verifier.
3. The verifier responds with a random vector $\vec{z} \in F^n$ and a batching randomness $\lambda \in F$. Now, both the prover and verifier engage in the sumcheck protocol for
$$\sum_{\vec{x}\in H} Q(L_H(\vec{x}, \vec{z}), h(\vec{x}), m(\vec{x}), \varphi_1(\vec{x}), \ldots, \varphi_M(\vec{x}), \tau(\vec{x})) = 0,$$

where

$$Q(L,h,m,\varphi_1,\ldots,\varphi_M,\tau) = L\cdot\left((h\cdot\tau+m)\cdot\prod_{i=1}^{M}\varphi_i - \tau\cdot\sum_{i=1}^{M}\prod_{j\neq i}\varphi_j\right)+\lambda\cdot h. \tag{164}$$

The sumcheck protocol outputs the expected value v for the multivariate polynomial

$$Q(L_H(\vec{X},\vec{z}),h(\vec{X}),m(\vec{X}),\varphi_1(\vec{X}),\ldots,\varphi_M(\vec{X}),\tau(\vec{X})) \tag{165}$$

at $\vec{X}=\vec{r}$ sampled by the verifier in the course of the protocol.

4. The verifier queries $[f_1],\ldots,[f_M],[t],[m],[h]$ for their inner product with $L_H(\,.\,,\vec{r})$, and uses the answers to check whether (165) equals the expected value v at $\vec{X}=\vec{r}$. (The value $L_H(\vec{r},\vec{z})$ is computed by the verifier.)

6 Parallel Optimization

Eigen zkVM supports infinite recursion, and proof composition, using Randomized AIR with preprocess for arithmetization to achieve high reliability. EigenVM supports universal on-chain verification curves, including BN254 and BLS12381, and allows developers to roll up their off-chain computation to different L1 blockchain protocols.

In the STARK verification process, an entity known as the verifier relies on proof, public inputs, and additional verifier-specific parameters to verify the information. The concept of `composing proofs` involves integrating multiple proving systems to produce a unified proof. This is typically done to enhance efficiency in a certain component of the cryptographic system. In our scenario, we initially employed a STARK as the first proving system. The crux of our compositional approach is to delegate the STARK proof verification task to a specialized verification circuit, denoted as C. If the prover can supply a valid proof, $\pi_{CIRCUIT}$, confirming the accurate execution of this verification circuit, then that alone suffices for verifying the original STARK proof, π_{STARK}. As illustrated in Figure 3, the verifier only needs to validate $\pi_{CIRCUIT}$, not π_{STARK}. One key advantage of this compositional strategy is that $\pi_{CIRCUIT}$ is both smaller and faster to verify compared to π_{STARK}.

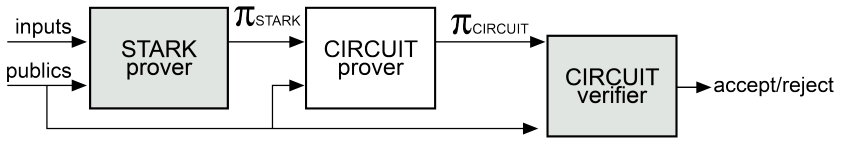

Fig. 3 Proof composition

6.1 Recursive Proof

In our architecture, we exploit the efficiency of verifiers over provers to construct a recursive cascade of STARK verifiers, as illustrated in Figure 4. Specifically, we employ intermediate circuits to define these STARK verifiers. Circuits are optimal for computations with limited branching, making them ideal for verifier computations.

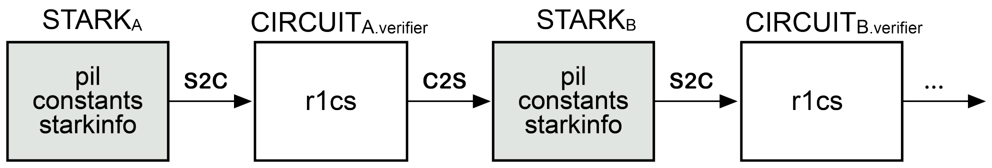

Fig. 4 Recursive Proof

To begin, consider the parameters describing the first STARK (STARKA)—comprising pil, constants, and starkinfo. During an initial setup phase, STARKA is translated into its corresponding verifier circuit, represented by Rank-1 Constraint System (R1CS) constraints. This conversion can be pre-processed before proof computation. We utilize Circom as the intermediate language for circuit description.

Following this, the R1CS-defined circuit is converted into a new STARK definition (STARKB) with new pil, constants, and stark info—a process we term as C2S (CIRCUIT-to-STARK). Also conducted in the setup phase, STARKB essentially uses a PlonKish arithmetization with custom gates based on STARKA's verification circuit. Note that this recursive step can be repeated as needed: each iteration compresses the proof, making it quicker to verify at the expense of increased prover complexity. During the setup phase, various artifacts are generated to facilitate STARK prover creation.

The first proof is initiated by supplying appropriate inputs and public values to STARKA's prover. The resultant proof is then passed to the next STARK prover, along with its public inputs. This process is recursively executed, as shown in Figure 5, to create a chain of STARK provers.

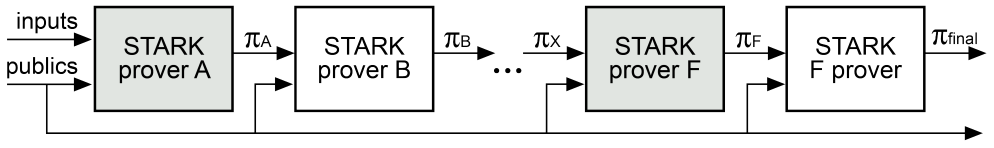

Fig. 5 Recursive provers

6.2 Aggregation Proof

Our architecture also supports proof aggregation, a specific form of proof composition. Multiple valid proofs can be consolidated into a single aggregated proof, which alone needs to be validated. Aggregators are integrated within intermediate circuits, with Figure 6 providing an example featuring binary aggregators.

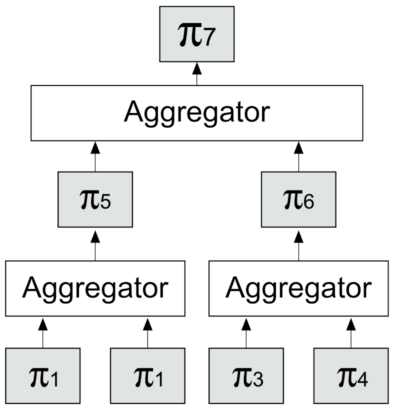

Fig. 6 Aggregation Proof

6.3 Proving Pipeline

Eigen zkVM Design Philosophy

The faster the proof system, the better, and the lower the gas, the better. The proof system is divided into two phases: the recursive phase and the on-chain phase.
The goals for each phase are:

- `On-chain phase:` The smaller the proof size, the better. Thus, it has a lower storage gas; The lower the verification complexity, the better. Thus, it has a lower verification gas.
- `Recursive phase:` No Setup or universal Setup. Thus, it has a good recursion. The faster the proof speed, the better, without needing to consider verification complexity and proof size.

Table 5 Proof systems.

commitment	Proof time	Proof size	Verify time	Tx Upper
KZG	$O(d)$	$O(1)$	$O(1)$	Small
FRI	$O(d * \log_2(d))$	$O(\log_2(d))$	$O(\log_2(d))$	Big
IPA	$O(d)$	$O(\log(d))$	$O(\log(d))$	Big

The on-chain phase only cares about the proof size and verify time. According to Table 5, the KZG commitments are optimal, IPA is second best, and FRI is the worst. Therefore, in the on-chain phase, the Plonk system based on KZG commitments should be used to achieve minimum storage gas and verification gas.

The recursive phase only cares about proving speed and recursive properties. Verification using KZG commitments requires bilinear mapping, and the verification of bilinear mapping needs to be constantly deferred during the recursion. Therefore, developing KZG in the recursive phase is relatively complex and is not recommended. Verification of FRI and IPA is simple, and the corresponding circuit development is relatively simple. For a proof time, under the same parameters, the proof speed of KZG and IPA is the fastest, while the FRI proof speed is the slowest. However, KZG and IPA can only run under a secure cryptosystem. In other words, it can only run on a group with a 256-bit scalar field. However, FRI can run on a 110-bit scalar field, with the best parallelization effect. Therefore, the best choice for the recursive phase is a proof system based on FRI, as shown in Figure 7.

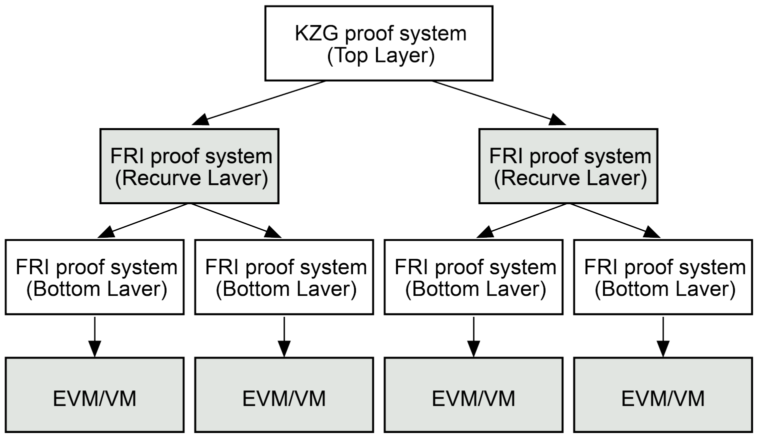
KZG proof system (Top Layer)
FRI proof system (Recurve Laver)
FRI proof system (Recurve Laver)
FRI proof system (Bottom Laver)
FRI proof system (Bottom Laver)
FRI proof system (Bottom Laver)
FRI proof system (Bottom Laver)
EVM/VM
EVM/VM
EVM/VM
EVM/VM

Fig. 7 Eigen zkVM super proof system

7 Core Features and Applications

Eigen zkVM is a virtual machine designed specifically for building secure, efficient, and scalable smart contracts and digital assets. Its core advantage lies in its ability to validate and execute contracts without revealing any sensitive information, thereby ensuring the privacy and security of transactions.

7.1 Multi-Language and Multi-Architecture Support

Eigen zkVM supports RISC V naturally, and upon this, Eigen zkVM also supports Rust, EVM, and Wasm.

7.1.1 Rust zkVM

One of the unique advantages of Eigen zkVM is its support for the Rust programming language. Rust is widely praised for its excellent memory safety and concurrency features. Through Rust, developers can write highly optimized, low-latency smart contracts without worrying about common security vulnerabilities, such as buffer overflows or data races. This not only improves the efficiency of contract execution but also significantly reduces security risks.

7.1.2 EVM

Another notable feature of Eigen zkVM is its compatibility with the Ethereum Virtual Machine (EVM). This means that existing Ethereum smart contracts and DApps (decentralized applications) can be seamlessly migrated to the Eigen zkVM platform. This compatibility provides developers with a smooth transition path, allowing them to take advantage of Eigen zkVM's advanced features, such as zero-knowledge proofs, without the need for extensive code refactoring.

7.1.3 High Flexibility and Portability

Due to its support for multiple programming languages and hardware architectures, Eigen zkVM offers extremely high flexibility and portability. Developers can choose the most suitable language and platform for development based on specific application scenarios and requirements. This flexibility allows Eigen zkVM to adapt to various application scenarios and business needs, thereby having a broader application prospect.

In summary, Eigen zkVM, with its support for multiple programming languages and hardware architectures, as well as its built-in zero-knowledge proof features, provides a powerful and flexible platform for building secure, efficient, and scalable smart contracts and digital assets. Whether it's in high-performance computing, big data analytics, or low-power environments like IoT and mobile devices, Eigen zkVM can deliver outstanding performance and security guarantees. This makes it an ideal choice for the future development of digital assets and smart contracts.

7.2 Fully on-chain Architecture Support

Eigen zkVM takes its cryptographic capabilities to the next level by offering support for a diverse set of elliptic curves, including sec256k1, ed25519, bn256, and BLS12381. This extensive curve support is a cornerstone for implementing robust zero-knowledge proofs across a variety of blockchain applications, thereby fortifying Eigen zkVM's position as a versatile and secure platform for decentralized computing.

7.2.1 Comprehensive Elliptic Curves Support

Eigen zkVM stands out for its robust support for multiple cryptographic curves, including secp256k1, ed25519, bn256, and BLS12381. This multi-curve support offers a range of advantages and opens up diverse application scenarios, making zkVM a highly versatile platform for secure and private transactions.

7.2.2 Proofs for the Full on-Chain

The diverse set of supported curves enables Eigen zkVM to facilitate zero-knowledge proofs across an entire blockchain ecosystem. This is invaluable for creating seamless and secure cross-chain interactions. For instance, a user could prove asset ownership on one blockchain while executing a confidential transaction on another, all without revealing any sensitive information.

7.2.3 Cryptographic Flexibility

The extensive curve support provides developers with unparalleled flexibility in choosing the most suitable cryptographic primitives for their applications. This is a significant advantage for building customized solutions that meet diverse security, performance, and functionality requirements. Conclusion

In summary, Eigen zkVM's comprehensive support for a variety of elliptic curves like sec256k1, ed25519, bn256, and BLS12381 amplifies its capabilities in delivering secure, efficient, and versatile zero-knowledge proofs for a wide array of blockchain applications. This positions Eigen zkVM as an exceptionally robust and flexible platform for developers aiming to construct decentralized applications with strong cryptographic features.

7.3 Applications

1. **Verifiable Computation** Eigen zkVM allows users to prove the correctness of a computation's result without revealing the original data. Eigen zkVM allows developers to convert the high-level language computation to be verifiable. This significantly decreases the pressure and burden of developers without much cryptography knowledge to launch their ZK-powered applications.
2. **Universal Rollup Service** Verifiable Computation is very computation-intensive and may bring a big resource cost to any ZK-powered service provider, and Leveraging Eigen zkVM's universal constraint system and composable proof system, the Universal Rollup Service can use parallel computing and CPU-friendly hardware acceleration to decrease the proving cost.

References

[1] Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5, pp. 1–18 (2008). Springer

[2] Chiesa, A., Tromer, E.: Proof-carrying data and hearsay arguments from signature cards. In: ICS, vol. 10, pp. 310–331 (2010)

[3] Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for snarks and proof-carrying data. In: Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing, pp. 111–120 (2013)

[4] Chiesa, A., Tromer, E., Virza, M.: Cluster computing in zero knowledge. In: Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II 34, pp. 371–403 (2015). Springer

[5] Bonneau, J., Meckler, I., Rao, V., Shapiro, E.: Coda: Decentralized cryptocurrency at scale. Cryptology ePrint Archive (2020)

[6] Kattis, A., Bonneau, J.: Proof of necessary work: Succinct state verification with fairness guarantees. Cryptology ePrint Archive (2020)

[7] Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Annual International Cryptology Conference, pp. 757–788 (2018). Springer

[8] Naveh, A., Tromer, E.: Photoproof: Cryptographic image authentication for any set of permissible transformations. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 255–271 (2016). IEEE

[9] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II 34, pp. 276–294 (2014). Springer

[10] Khovratovich, D., Maller, M., Tiwari, P.R.: Minroot: Candidate sequential function for ethereum vdf. Cryptology ePrint Archive (2022)

[11] Buterin, V.: The different types of zk evm. https://vitalik.ca/general/2022/08/04/zkevm.html. Accessed: 2023-04-27 (2022)

[12] Gentry, C., Wichs, D.: Separating succinct non-interactive arguments from all falsifiable assumptions. In: Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing, pp. 99–108 (2011)

[13] Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp. 326–349 (2012)

[14] Masip-Ardevol, H., Guzmán-Albiol, M., Baylina-Melé, J., Muñoz-Tapia, J.L.: estark: Extending starks with arguments. Cryptology ePrint Archive (2023)

[15] Thaler, J., *et al.*: Proofs, arguments, and zero-knowledge. Foundations and Trends® in Privacy and Security **4**(2–4), 117–660 (2022)

[16] Bowe, S., Grigg, J., Hopwood, D.: Recursive proof composition without a trusted setup. Cryptology ePrint Archive (2019)

[17] Bünz, B., Chiesa, A., Mishra, P., Spooner, N.: Proof-carrying data from accumulation schemes. Cryptology ePrint Archive (2020)

[18] Boneh, D., Drake, J., Fisch, B., Gabizon, A.: Halo infinite: Recursive zk-snarks from any additive polynomial commitment scheme. Cryptology ePrint Archive (2020)

[19] Bünz, B., Chiesa, A., Lin, W., Mishra, P., Spooner, N.: Proof-carrying data without succinct arguments. In: Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part I 41, pp. 681–710 (2021). Springer

[20] Kothapalli, A., Setty, S., Tzialla, I.: Nova: Recursive zero-knowledge arguments from folding schemes. In: Annual International Cryptology Conference, pp. 359–388 (2022). Springer

[21] Kothapalli, A., Setty, S.: Hypernova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive (2023)

[22] Bünz, B., Chen, B.: Protostar: Generic efficient accumulation/folding for special sound protocols. Cryptology ePrint Archive (2023)

[23] Setty, S.: Spartan: Efficient and general-purpose zksnarks without trusted setup. In: Annual International Cryptology Conference, pp. 704–737 (2020). Springer

[24] Boneh, D., Drake, J., Fisch, B., Gabizon, A.: Halo infinite: Proof-carrying data from additive polynomial commitments. In: Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part I 41, pp. 649–680 (2021). Springer

[25] Kothapalli, A., Setty, S.: Supernova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive (2022)

[26] Mohnblatt, N.: Sangria: a folding scheme for PLONK (2023)

[27] Kothapalli, A., Setty, S.: Cyclefold: Folding-scheme-based recursive arguments over a cycle of elliptic curves. Cryptology ePrint Archive (2023)

[28] Setty, S., Thaler, J., Wahby, R.: Customizable constraint systems for succinct arguments. Cryptology ePrint Archive (2023)

[29] Zheng, T., Gao, S., Guo, Y., Xiao, B.: Kilonova: Non-uniform pcd with zero-knowledge property from generic folding schemes. Cryptology ePrint Archive (2023)

[30] Eagen, L., Gabizon, A.: Protogalaxy: Efficient protostar-style folding of multiple instances. Cryptology ePrint Archive (2023)

[31] Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019)

[32] Gabizon, A., Williamson, Z.J.: plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive (2020)

[33] Aztec: From airs to to raps - how plonk-style arithmetization works. https://hackmd.io/@aztec-network/plonk-arithmetiizationair (2023)

[34] Ben-Sasson, E., Chiesa, A., Gabizon, A., Riabzev, M., Spooner, N.: Interactive oracle proofs with constant rate and query complexity. Cryptology ePrint Archive (2016)

[35] Chen, B., Bünz, B., Boneh, D., Zhang, Z.: Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 499–530 (2023). Springer

[36] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcps. In: Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32, pp. 626–645 (2013). Springer

[37] Ben-Sasson, E., Carmon, D., Ishai, Y., Kopparty, S., Saraf, S.: Proximity gaps for reed–solomon codes. In: 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS), pp. 900–909 (2020). IEEE