A Recursive zk-based State Update System

Daniel Bloom, Sai Deng

Mozak

May 2024

1 Introduction

This paper introduces a ZKP (zero-knowledge proof) based state update system, where each block contains a SNARK proof aggregated from the user generated zkVM (zero knowledge virtual machine) proofs. It enables users to generate state update proofs in their local machines, contributing to a secure, decentralized verification process. Our main contribution in this paper, the recursive proofs system, addresses scalability by recursively verifying user proofs and aggregating them in a hierarchical tree structure up to a root proof, serving as a block proof. The proposed solution advances current blockchain paradigms by offering efficient recursive verification through ZKP, enhancing security and reducing computational load.

2 Background

2.1 Recursive Zero-Knowledge Proofs

A non-interactive zk system consists of a prover P and a verifier V. The prover's goal is to demonstrate that a computation C has been executed with some public input x and some secret input w, known as the witness. We follow the notations used in [4]. The prover generates a proof π using a publicly trusted arithmetic circuit implementing C: $P(x, w) \to \pi$. The verifier then checks the proof with $V(x, \pi) \to$ true/false, without needing to know w.

A recursive zk proof verifies other zk proofs within its circuit. P demonstrates that they have verified several inner proofs: $P(x_1, \pi_1, x_2, \pi_2, \ldots) \to \pi$. The recursive proving circuit enforces the constraints of verifiers for these inner proofs. When the outer proof π is verified, the inner proofs $\pi_1, \pi_2, \ldots$ are also verified: $V(x, \pi) \to \text{true} \Rightarrow \pi_1, \pi_2, \ldots$ are true.

Recursive proof composition was first introduced by [2] and later realized practically using cycles of elliptic curves by [1]. Subsequent research, such as Halo2 [3] and Nova [5], has further enhanced recursion speed and verification costs. Plonky2 [7] employs techniques from PLONK, FRI and Goldilocks fields, achieving the fastest recursion time so far.

[4] introduces a system that uses MapReduce to aggregate multiple different types of proofs into a single succinct root proof within a tree structure. [6] extends this tree structure into a vector commitment scheme, offering efficient, updatable batch proofs for blockchain applications.

2.2 zkVMs

zkVMs are a novel approach to achieving scalability and privacy in blockchain systems. Unlike traditional ZKPs that require custom circuits for each program, zkVMs act as virtual machines specifically designed for zk-proof computations. This allows developers to write programs in a familiar language and leverage the zkVM's built-in capabilities to generate proofs. zkVM designs can be categorized based on their underlying cryptographic techniques.

One of the primary methods utilized in zkVMs is zk-STARKs, the combination of AIR and FRI. This approach is exemplified by projects like Starkware, Risc0, Polygon Miden and SP1.

In parallel, folding-based methods have emerged as an alternative technique within zkVMs. These methods focus on the recursive composition of proofs, enabling the verification of complex computations through the aggregation of simpler proof instances.

Lasso and Jolt are innovative frameworks within the zkVM landscape. Lasso enables a new strategy of efficient lookups over very large tables. Jolt leverages Lasso to instantiate the "lookup singularity" by capturing the evaluation of each primitive CPU instruction within several large table lookups.

2.3 Trustless State Update System

In decentralized systems, trustless state update mechanisms are crucial for ensuring the integrity and security of the network without relying on a central authority. These systems use consensus algorithms to achieve agreement on the state of the blockchain among distributed participants. Bitcoin (BTC), Tendermint and Ethereum (ETH) are some examples.

Mina Protocol replaces the traditional blockchain with an easily verifiable proof. Rather than having each participant verify historical transactions independently, the network collaborates to generate zk proofs for transactions. This process allows end users to receive the state of the ledger accompanied by a zk-SNARK, which cryptographically guarantees its accuracy.

3 VM Proof Recursion

In this section, we introduce how zkVM proofs are generated locally on the user's machine and later used in our state update system.

In zk-STARKs-based zkVMs, multiple STARK tables are used, usually including tables for CPU, Memory, Program ROM, and more. These tables are connected by cross-table lookups, implemented by logUps. We express zkVM proving as: $P_{vm}(r, x) \to \pi$, where r is the program ROM that defines a unique program, and x are the program inputs. In RISC-V based zkVMs, the program to be executed is compiled into RISC-V machine code, and this compiled code is typically stored in an ELF file. This ELF file is then loaded into the zkVM for execution. r is a cryptographic commitment of the ELF file.

3.1 Fixed Recursion and Dynamic Recursion

Our recursion system is a hybrid proving system that includes both fixed recursion circuits and dynamic recursion circuits. The difference lies in that, in fixed recursion circuit, the verifier data (vd) is fixed in the circuit, meaning we can only verify the proofs generated from vd. In dynamic recursion circuits, verifier data is passed as inputs, supporting verification of proofs from different circuits. Mathematically, fixed recursion is $P(\pi) \to \pi'$, and dynamic recursion is $P(vd, \pi) \to \pi'$.

Polygon Zero uses fixed recursion for their zkEVM application. They predefined a degree range for each STARK table and used STARK to PLONK (more accurately speaking it is PLONKish + AIR) recursion to reduce its individual table proofs to a fixed type proof (has the same vd), then aggregate all the PLONK proofs. In Risc0 and SP1, they use zkVM to recursively verify their zkVM proofs, and thus they use dynamic recursion. The advantage of Polygon's method is its fast recursion speed, at least 2x faster than zkVM's self-recursion. The advantage of Risc0 and SP1's method is its flexibility; it can verify proofs produced by different programs, without worrying about the size of the program execution leading to different proof configurations.

In our system, we combine the benefits of both methods, providing a fast and flexible recursion framework to recursively verify zkVM proofs into a standard proof type and serve them in our later pipeline.

3.2 Batch STARKs

Similar to Plonky3, Batch STARKs are employed in the system. A Batch Merkle Tree is utilized to commit various components from different STARKs, specifically the execution trace polynomials, AIR constraints quotient polynomials, and cross-table lookup related polynomials. Values of polynomials from different tables with different degrees are merged into a single Batch Merkle Tree. Figure 1 illustrates how a Batch Merkle Tree is built from merging two Merkle Trees and used to compress Merkle proofs.

The Batch FRI protocol operates through random linear combinations [**?**]. Given a batch of L low-degree polynomials $q_0(X), \ldots, q_{L-1}(X)$, the verifier samples a random challenge λ. The prover then computes the linear combination

$$h(X) = \sum_{i=0}^{L-1} \lambda^i \cdot q_i(X),$$

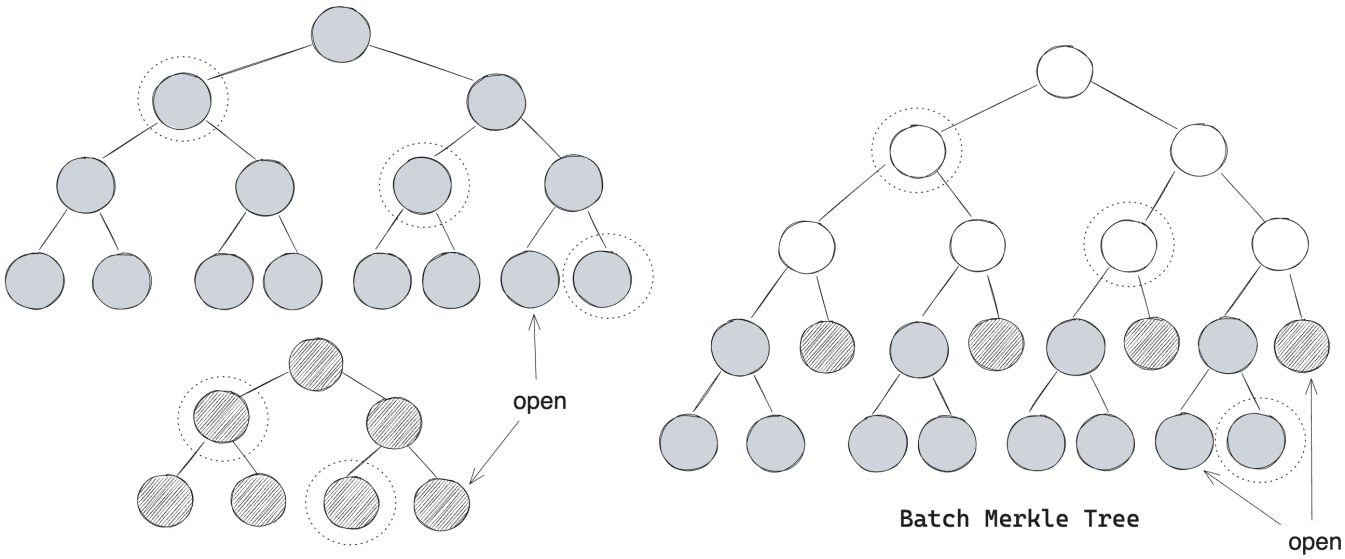

Figure 1: This figure illustrates the efficiency of using a Batch Merkle Tree to compress opening paths or Merkle proofs. On the left side, opening two values from two separate Merkle Trees requires five nodes for the Merkle proofs. On the right side, when the two opened values are on the same branch, only three nodes are needed. Thus, the Merkle proofs in shorter Merkle Trees are effectively cost-free.

and provides the oracle of $h(X)$ to the verifier. Subsequently, both the prover and verifier proceed with FRI for $h(X)$.

In the query phase of the FRI, each value is used to further verify the consistency between the oracle for $h(X)$ and those in the batch, $q_0(X), \ldots, q_{L-1}(X)$.

To accommodate polynomials of different degrees, similar to Plonky3, an enhancement has been made to the protocol. Consider a batch of D low-degree polynomials $h_0(X), \ldots, h_{D-1}(X)$ (these are the random linear combination polynomials) with varying degrees $2^{a_0}, \ldots, 2^{a_{D-1}}$.

During the folding process in FRI, we perform a random linear combination of $h_i(X)$ with the polynomial $f_i(X)$ at a specific FRI round i, provided the degree of $f_i(X)$ matches 2^{a_i}. Here $f(X) = f_E(X^2) + X \cdot f_O(X^2)$, where $f_E(X)$ and $f_O(X)$ are polynomials containing only even or odd terms of $f(X)$, respectively. We substitute $f(X)$ with $f_i'(X)$ and continue FRI as follows:

$$f_i'(X) = h_i(X) + \lambda_i \cdot f_i(X)$$

3.3 Program Hash and Final VM Proof

As discussed earlier, there are two primary types of recursion used in zk-STARK based systems nowadays:

- AIR Recursion: This involves writing the zkVM verifier in a language like Rust and running Rust verifiers within the zkVM to generate the recursive zkVM proof.
- PLONKish Recursion: This involves implementing the zkVM verifier in a zk circuit and generate the recursive proofs in PLONKish + FRI form.

The advantage of AIR recursion is the generation of a consistent recursion program hash, regardless of the program or its inputs, albeit at a slower speed. Conversely, PLONKish recursion offers faster performance but generates varying recursion circuits when zkVM execution table sizes differ due to the fixed number of gates associated with the circuits. This results in an undetermined recursion circuit hash.

Maintaining a consistent recursion circuit hash is crucial for proving systems. It ensures that only proofs generated from trusted circuits are included. While projects like Risc0, Valida, and SP1 opt for AIR recursion, Polygon Zero utilizes PLONKish recursion.

In the Plonky2 zkEVM, degree ranges for each STARK trace table are predefined. PLONKish recursion is used to reduce each table's proof to the proof in a fixed degree, and an additional PLONKish recursion circuit aggregates

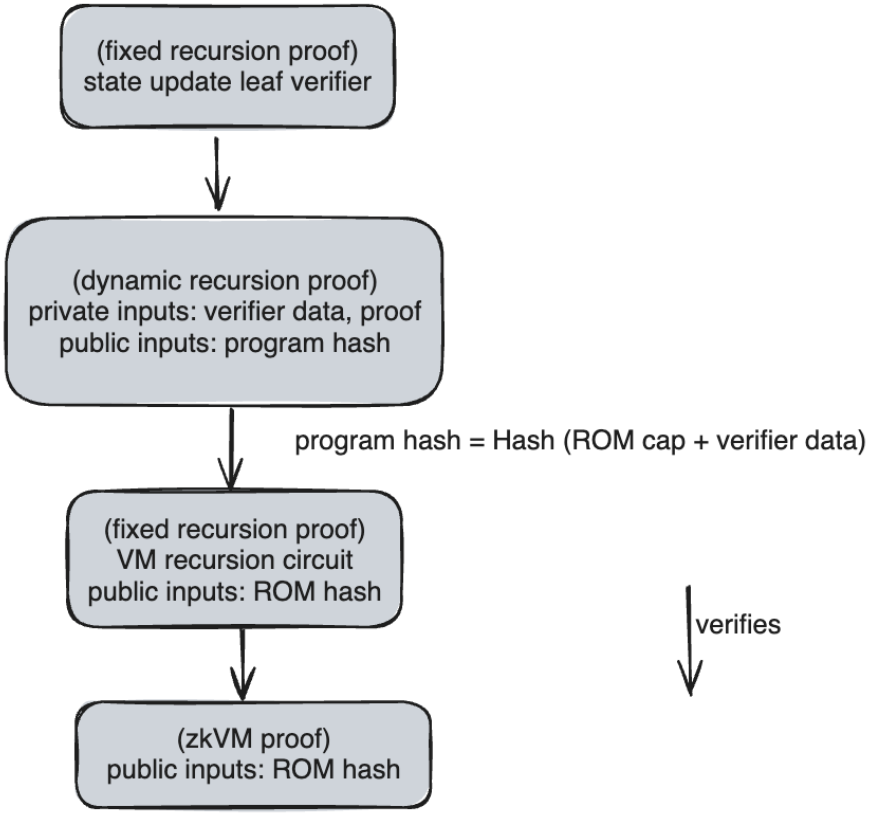

Figure 2: Recursion steps of a VM proof

them. While this approach yields consistent final recursion circuits, it sacrifices speed and deviates from the original intent of the method.

In our PLONKish based recursion system, the degrees of STARK tables are public settings of the recursion circuits. Given S STARK tables with degrees $2^{d_0}, 2^{d_1}, ...2^{d_{S-1}}$, the set $\{d_i\}, i = 0..S$ forms the public settings, with different sets resulting in different recursion circuit hashes. By testing various program inputs and their boundary values, we obtain multiple sets $\{d_{i,j}\}$, where $j = 0..T$. We then set $D_i = max_{j=0..T}\{d_{i,j}\}$ and use $\{D_i\}$ as the final public setting, padding each STARK table to degree $\{D_i\}$ during executions.

An alternative method involves adding trusted recursion circuits in public settings, such as using a Merkle tree to store all valid recursion circuit hashes. This method works better when the number of STARK tables used in zkVM is small.

With a trusted recursion circuit hash, we can now utilize Hash (Recursion Verifier Data — Program STARK Merkle Root) as our new ROM hash. This process is illustrated in Figure 2.

4 Recursion of State update proofs

In this section, we introduce the ZK-based distributed proving system to aggregate offline VM proofs and validate and combine all resulting state changes. Broadly this can be categorized into three steps:

1. Combining event attestations from zkVM proofs
2. Attesting to the relevant state data (updates, reads)
3. Proving the attested to data corresponds to the combined events

4.1 Optional Hashing

One of the tools these steps utilize is notion is similar to the *canonical hashing* of [6] which we call *optional hashing*. Optional hashing is a simple modification of any Merkle tree-capable hash function $h(L, R)$ defined as follows:

1. If both L and R are present $h'(L, R) = h(L, R)$
2. If only the left child L is present $h'(L, \emptyset) = L$
3. If only the right child R is present $h'(\emptyset, R) = R$
4. If neither child is present $h'(\emptyset, \emptyset) = \emptyset$

4.2 Proof of Events

Events are facilitate zkVM interactions with the on-chain data, and represent a constraint on the current and next block state. Events contain an address of the state node they apply to, and so consequently each corresponding constraint applies to only one node of state. Thus zkVM proofs typically attest to multiple events via a Merkle commitment. For reasons that will become clear, this tree must be sorted by the node address to which the event applies, and nodes are merged based on the Longest Common Prefix of the binary interpretation of these addresses. It is possible to enforce this sort-order constraint through additional proofs, but doing so at this stage is actually unnecessary, for reasons that will become clear.

One or more zkVM proofs can be arranged into a Merkle proof tree, the leaf nodes of which attest to the correctness of the zkVM proofs and their event commitments. Branch nodes combine two child proofs and attest to the merge of their events.

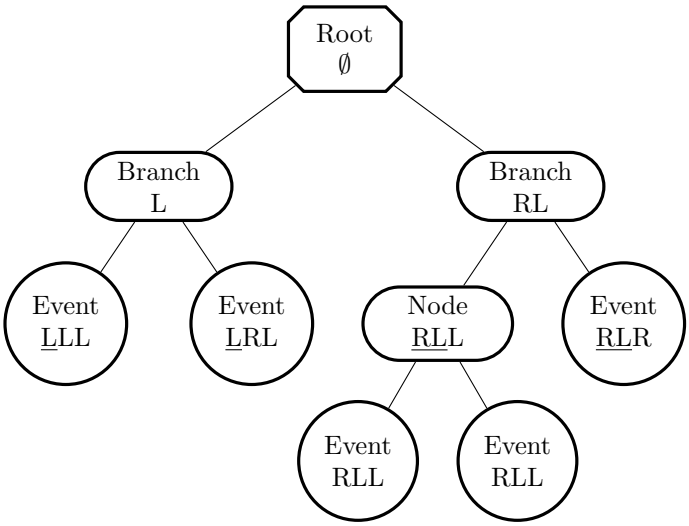

Figure 3: Event Merkle tree branching is based on LCP

4.2.1 Merge

Merging event trees (A and B) is done by reinterpreting them with optional hashing. A Merkle tree constructed with optional hashing will result in the same Merkle commitment, but enables the insertion of empty nodes.

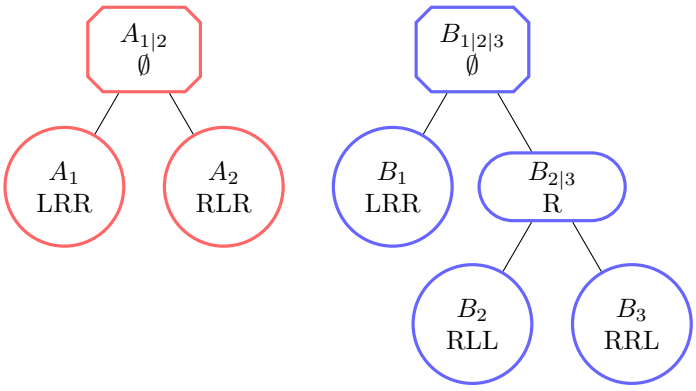

Figure 4: Two event trees A and B

By adding empty nodes to A for all B-only addresses, and vice-versa, we can create an A' and B' are isomorphic to each other.

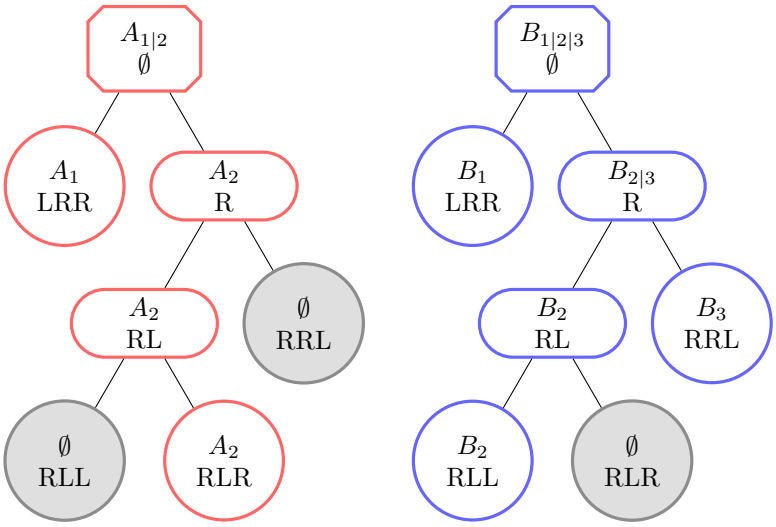

Figure 5: Two event trees A' and B' prepared to be merged

This isomorphic structure makes it easy to recursively prove the merge of the trees A and B results in tree C. If A and B are sorted, and the merge is performed correctly, C will be sorted, contain all the nodes from A and B. All that's needed are some simple circuits which will recursively attest to nodes of A', B', and C at each level.

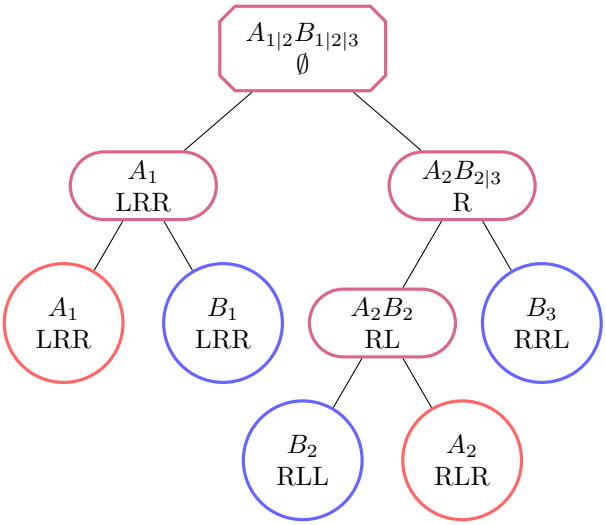

Figure 6: The merged tree C

Circuit M_0

Public input: (vk, V_A, V_B, V_C)
Computation: Check that $V_C = h'(V_A, V_B)$

Circuit M_i

Public input: (vk, V_L, V_R, V_M)
Witness: (π_L, L, V_{LA}, V_{LB}, π_R, R, V_{RA}, V_{RB})
Computation:

1. If $L = 0$ check Verify(vk_0, (vk, V_{LA}, V_{LB}, V_L), π_L)
2. If $R = 0$ check Verify(vk_0, (vk, V_{RA}, V_{RB}, V_R), π_R)
3. If $L \neq 0$ check Verify(vk, (vk, V_{LA}, V_{LB}, V_L), π_L)
4. If $R \neq 0$ check Verify(vk, (vk, V_{RA}, V_{RB}, V_R), π_R)
5. Check that $V_M = h'(V_L, V_R)$

4.3 Proof of State Update

State is committed to by means of a fixed-depth sparse Merkle tree. A state update consists of the previous and next Merkle commitments, as well as a summary commitment of all nodes for which any event exists (both reads and writes). This summary commitment is constructed using optional hashing and commits to the address, previous state, and next state of the selected nodes. Note how the summary tree is inherently sorted with merges occurring based on the Longest Common Prefix of the binary interpretation of the node addresses.

In order to ensure selected addresses are correct, we exploit the parent-child-sibling relationship of addresses, i.e. if the parent has an address of X, the left child must have an address of $2X$, and the right child must have an address of $2X + 1$. The root of the tree must then have an address of 0.

Circuit S_0

Public input: (A, S, H_P, H_N)
Computation:

1. Check that $S = \emptyset$ iff $A = \emptyset$
2. If $S = \emptyset$ check that $H_P = H_N$
3. If $S \neq \emptyset$ check that $S = H(A||H_P||H_N)$

Circuit S_i

Public input: (A, S, H_P, H_N)
Witness: (π_L, A_L, S_L, H_{LP}, H_{LN}, π_R, A_R, S_R H_{RP}, H_{RN})
Computation:

1. Check Verify(vk_{i-1}, A_L, S_L, H_{LP}, H_{LN}), π_L)
2. Check Verify(vk_{i-1}, A_R, S_R, H_{RP}, H_{RN}), π_R)
3. Check that $S = \emptyset$ iff $A = \emptyset$
4. If $A_L \neq \emptyset$ check that $A = 2 * A_L + 1$
5. If $A_R \neq \emptyset$ check that $A = 2 * A_R + 1$
6. Check that $S = h'(S_L, S_R)$
7. Check that $H_P = h'(H_L P, H_R P)$
8. Check that $H_N = h'(H_L N, H_R N)$

4.4 Proof Events and State Match

The final step in creating a block is to prove the event and state proofs correspond to each other. Note that the event proof is attesting to a sorted tree of events, and the state proof is attesting to a sorted tree of summaries (in addition to the new and old Merkle commitments). The event tree and the summary tree are paramorphic, i.e. they share the same branch structure and shape, but where the summary tree contains just a single leaf for each address, the event tree contains one or more events for each address.

This disparity in leaf multiplicity leaves us with two choices:

1. Combine the constraints of the events into a single super-constraint, which is validated against the summary.
2. Duplicate the summary leaf for each event and independently validate them, then combine these validations.

These approaches appear more different than they actually are in practice, as to prevent double-spends from duplicate write events, information about the number of writes an applied by an individual constraint must be passed upwards. As such, this paper will describe the former approach, while the latter is left as an exercise for the reader.

Circuit C_0

Public input: (vk, A, C, H_E)
Witness: (E)
Computation:

1. Check that $H_E = H(E)$
2. Check that $A = \text{GetAddress}(E)$
3. Check that $C = \text{BuildConstraint}(A, E)$

Circuit C_i

Public input: (vk, A, C, H_E)
Witness: (π_L, C_L, H_{LE}, π_R, C_R H_{RE})
Computation:

1. Check Verify(vk, (vk, A, C_L, H_{LE}), π_L)
2. Check Verify(vk, (vk, A, C_R, H_{RE}), π_R)
3. Check $C = \text{CombineConstraints}(C_L, C_R)$
4. Check $H_E = h(H_{LE}, H_{RE})$

After this small tree of recursive ZKPs has unified the structure of the summary and events trees, it becomes trivial to merklize with another recursive ZKP. This isomorphism is what necessitates the event sort order, and implicitly enforces it. The resulting root proof attest to the relationship between these trees. By combining all three proofs, along with a previous proof

References

[1] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. 79(4):1102–1160.

[2] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. https://eprint.iacr.org/2012/095.

[3] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. https://eprint.iacr.org/2019/1021.

[4] Sai Deng and Bo Du. zkTree: A zero-knowledge recursion tree with ZKP membership proofs.

[5] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, Lecture Notes in Computer Science, pages 359–388. Springer Nature Switzerland.

[6] Charalampos Papamanthou, Shravan Srinivasan, Nicolas Gailly, Ismael Hishon-Rezaizadeh, Andrus Salumets, and Stjepan Golemac. Reckle trees: Updatable merkle batch proofs with applications.

[7] Polygon. Plonky2: Fast recursive arguments with PLONK and FRI. https://github.com/mir-protocol/plonky2/blob/136cdd053f2175134cddc61abc587f1862e76921/plonky2/plonky2.pdf.