Recproofs: Vector Commitments with Updatable Batch Proofs and their Applications
(Draft)

Charalampos Papamanthou[*,2], Shravan Srinivasan[1], and Ismael Hishon-Rezaizadeh[1]

[1]*Lagrange Labs*
[2]*Yale University*

Abstract

We propose Recproofs, a Merkle-based vector commitment that computes a batch proof of a subset of k leaves in a Merkle tree of n leaves using recursive SNARKs. Our construction folds the computation of the subset hash inside the computation of Merkle verification via *canonical hashing*. Due to this folding, our batch proofs can be updated in logarithmic time, whenever a Merkle tree element (belonging in the batch or not) changes, by maintaining a data structure that stores previously computed recursive proofs. Our batch proofs are also computable in $O(\log n)$ parallel time. We also extend our framework to provide updatable proofs for MapReduce computations: A prover can commit to a memory M and produce a succinct proof for a MapReduce computation over a subset I of M. The proof can be efficiently updated whenever I or M changes. Updatable proofs find applications in the blockchain space when a proof needs to be computed and efficiently maintained over a moving stream of blocks (e.g., moving average). Our *preliminary* evaluation using Plonky2 shows that our approach has small memory footprint, significantly outperforms previous approaches in terms of updates (potentially up to $135\times$) and performs similarly with other approaches in terms of aggregation time.

1 Introduction

A vector commitment is a cryptographic primitive that enables a party to commit, via a small digest, to a vector M of n slots in a way that an individual proof π_i can be provided for proving the correctness of an arbitrary vector slot $\mathsf{M}[i]$. Vector commitments have found applications in the blockchain space, such as in Ethereum state compression (via Merkle Patricia tries [1]), stateless validation [4], zero-knowledge proofs [20] as well as in verifiable cross-chain computation [2]. A growing body of work on vector commitments (e.g., aSVC, AMT, pointproofs [8, 17, 18]) has focused on optimizing the size of individual proofs as well as the size of batch proofs (A batch proof is a proof that simultaneously proves a set of k vector slots without paying a proof cost proportional to k and can be typically computed by aggregating k individual proofs.) Unfortunately, these vector commitments are not *maintainable*: Whenever a vector slot changes, $\Omega(n)$ time is required to update all individual proofs, which can be a bottleneck for many applications. *Maintainable* vector commitments on the other hand [11, 12, 16, 19] maintain a data structure that can be used to update all proofs in logarithmic time—however, they suffer from increased batch proof sizes and aggregation and verification times. This work is focusing on designing maintainable vector commitments with small batch proof sizes (45.13 KiB, independent of the batch size and the size of the vector), fast updates to the batch proof (35.46 seconds for a tree height of 29) and verification (8.76 *ms*).

The Merkle-SNARK approach. The basis of this work is the celebrated Merkle tree data structure [12], which can serve as a vector commitment. Merkle trees are naturally maintainable, but have large individual proofs and more importantly, large batch proofs. They can be turned into vector commitments with small (batch) proofs via either one of the following two Merkle-SNARK approaches that have been proposed in the literature.

1. View the verification of the k Merkle tree proofs as a whole computation circuit C, and apply a SNARK system as a black box on C. This method was benchmarked in the recent Hyperproofs [16] work by Srinivasan et al. One of the drawbacks of this approach is that it is not naturally parallelizable—it can only be parallelized to the extent that the prover algorithm of the used SNARK can be parallelized;

2. Build a SNARK circuit for the verification of a single Merkle proof, but instead of verifying k SNARK proofs natively, build a SNARK that verifies two leaf SNARK proofs, leading to $k/2$ SNARK proofs. Use this method recursively to eventually end up with a single proof that verifies all k initial Merkle proofs. This technique was

[*]Contribution to this work was made in individual capacity and not as part of Yale University duties or responsibilities.

recently proposed by Deng and Du [7] and has the benefit of being naturally parallelizable: The SNARK proofs at the leaf level can be computed independently and in parallel, and the same holds for the proofs of the next level, leading to $\log n + \log k$ parallel complexity.

1.1 Our central contribution: Recproofs

We consider a dynamic setting for batch proofs: Assume a batch proof for a subset I of leaves has been computed, and consider a leaf $x \in I$ or a leaf $y \notin I$ changes its value. Clearly, the batch proof changes completely. The question we are asking is whether we can update the batch proof in time sublinear in k, i.e., without recomputing the proof from scratch. This could have applications in settings where a batch proof must be updated as new blocks are generated (e.g., while computing a moving average for pricing). It is worth noting that none of the directions above ((1) and (2)) can support updatability: Every single change requires recomputation of all the SNARK proofs from scratch.

We propose a new method to compute batch Merkle proofs that allows efficient updates. In particular, by maintaining a batch-specific data structure, the batch proof can be updated in $\log n$ time per update. To the best of our knowledge, this is the first construction for batch Merkle proofs that supports efficient updates.

Our approach uses recursive SNARKs, but in a fundamentally different way. Instead of computing individual SNARK proofs for every Merkle proof (as in (2)), we run the recursion directly on the Merkle paths that belong to elements in the batch. While we are traversing the paths, we not only verify that the elements belong to the Merkle tree but we also compute a "batch" hash for the elements in the batch and make this batch hash part of the public statement. We compute the batch hash via *canonical hashing*, a deterministic and secure way to represent any subset of k leaves succinctly (see Fig. 1). If the recursive proof verifies, that means that the batch hash corresponds to some valid elements of the tree, and can be recomputed using the actual claimed batch as an input.

This technique naturally supports updates in $O(\log n)$ time, just like Merkle trees support updates in $O(\log n)$ time: Given a batch proof, one can keep a data structure with all the recursive proofs corresponding to the interior nodes that were computed. When a leaf changes (either in the batch or not), the batch proof can be updated by traversing only the path that corresponds to the changed element, and by reusing recursive proofs for subtrees that were not affected by the change. Other notions of updatability could also be possible, e.g., when the indices participating in the batch change or when the Merkle tree itself grows or shrinks.

Finally, we mention that an added benefit of our approach is that it is naturally parallelizable with $O(\log n)$ parallel complexity, as opposed to $O(\log n + \log k)$ that direction (2) above has.

1.2 Second contribution: Dynamic verifiable MapReduce

We observe that our technique can be generalized on verifying MapReduce [6] computations on dynamic data: In particular, we propose a technique that allows a prover to commit to a memory M, and provide a proof of correctness for a *fixed* MapReduce computation on any subset of M. Moreover, this proof can be easily updated whenever the subset changes, without having to recompute it from scratch. While recursive SNARKs have been used before for MapReduce computations [5], we are the first to show how to verify MapReduce on arbitrary subsets of memory, as well as how to update those proofs.

1.3 Evaluation

We perform an initial evaluation of our approach. We use Plonky2 [15] to implement our approach. Our preliminary evaluation shows that our approach has a small memory footprint, significantly outperforms previous approaches in terms of updates (potentially up to $135\times$) and fast verification (8.76 *ms*) and performs similarly with other approaches in terms of aggregation time.

2 Preliminaries

Notation. Let λ be the security parameter and H denote a collision-resistant hash function. Let $[n] = [0,n) = \{0,1,\ldots,n-1\}$, and $r \in_R S$ denote picking an element from S uniformly at random. Bolded, lower-case symbols such as $\mathbf{a} = [a_0,\ldots,a_{n-1}]$ typically denote vector of binary strings, where $a_i \in \{0,1\}^{2\lambda}, \forall i \in [n]$. If a_i's are arbitrarily long, we use the H function to reduce it to a fixed size. $|\mathbf{a}|$ denotes the size of the vector $\mathbf{a}$.

Succinct Non-Interactive Argument of Knowledge [10]. Let $\mathcal{R}$ be an efficiently computable binary relation that consists of pairs of the form (x,w), where x is a statement and w is a witness.

Definition 2.1. *A SNARK is a triple of PPT algorithms* $\Pi = (\mathsf{Setup},\mathsf{Prove},\mathsf{Verify})$ *defined as follows:*

- $\mathsf{Setup}(1^\lambda,\mathcal{R}) \to (\mathsf{pk},\mathsf{vk})$*: takes a security parameter* λ *and the binary relation* $\mathcal{R}$ *and outputs a common reference string consisting of the prover key and the verifier key* $(\mathsf{pk},\mathsf{vk})$.
- $\mathsf{Prove}(\mathsf{pk},x,w) \to \pi$ *: on input* pk*, a statement* x *and the witness* w*, outputs a proof* π.
- $\mathsf{Verify}(\mathsf{vk},x,\pi) \to 1/0$*: on input* vk*, a statement* x*, and a proof* π*, it outputs either 1 indicating accepting the argument or 0 for rejecting it.*

It also satisfies the following properties:

- Completeness: For all $(x,w) \in \mathcal{R}$, the following holds:

$$\Pr\left(\mathsf{Verify}(\mathsf{vk},x,\pi)=1 \;\middle|\; \begin{array}{r}(\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda,\mathcal{R}) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk},x,w)\end{array}\right)=1$$

- Knowledge Soundness: For any PPT adversary $\mathcal{A}$, there exists a PPT extractor $\mathcal{X}_\mathcal{A}$ such that the following probability is negligible in λ:

$$\Pr\left(\begin{array}{l}\mathsf{Verify}(\mathsf{vk},x,\pi) \\ \wedge\, \mathcal{R}(x,w)=0\end{array}\middle|\; \begin{array}{r}(\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda,\mathcal{R}) \\ ((x,\pi);w) \leftarrow \mathcal{A} \,\|_{\mathcal{X}_\mathcal{A}}\, ((\mathsf{pk},\mathsf{vk}))\end{array}\right)$$

- Succinctness: For any x and w, the length of the proof π is given by $|\pi|$=poly$(\lambda)\cdot$polylog$(|x|+|w|)$.

Merkle trees. Let M be a memory of n slots. A Merkle tree [12] is an algorithm to compute a succinct, collision-resistant representation C of M (also called digest) so that one can provide a small proof for the correctness of any memory slot $\mathsf{M}[i]$, while at the same time being able to update C in logarithmic time whenever a slot changes. We assume the memory slot values and the output of the H function have size 2λ bits each. The Merkle tree on an n-sized memory M can be constructed as follows:

Without loss of generality, assume n is a power of two, and consider a full binary tree built on top of memory M. For every node v in the Merkle tree T, the Merkle hash of v, C_v, is computed as follows:

1. If v is a leaf node, then $\mathsf{C}_v = index(v)||value(v)$
2. If v has left child L and right child R, then $\mathsf{C}_v = \mathsf{H}(\mathsf{C}_L||\mathsf{C}_R)$.

The digest of the Merkle tree is the Merkle hash of the root node. The proof for a leaf comprises hashes along the path from the leaf to the root. It can be verified by using hashes in the proof to recompute the root digest C.

2.1 Vector Commitments (VCs)

We formalize VCs below, similar to Catalano and Fiore [3].

Definition 2.2 (VC). *A VC scheme is a set of* PPT *algorithms:*

$\underline{\mathsf{Gen}(1^\lambda,n)\to\mathsf{pp}}$: Given security parameter λ and maximum vector size n, outputs randomly-generated public parameters pp.

$\underline{\mathsf{Com}_{\mathsf{pp}}(\mathbf{a})\to\mathsf{C}}$: Outputs digest C of $\mathbf{a}=[a_0,\ldots,a_{n-1}]$, where $a_i \in \{0,1\}^{2\lambda}, \forall i\in[n]$.

$\underline{\mathsf{Open}_{\mathsf{pp}}(i,\mathbf{a})\to\pi_i}$: Outputs a proof π_i for position i in $\mathbf{a}$.

$\underline{\mathsf{OpenAll}_{\mathsf{pp}}(\boldsymbol{a})\to(\pi_0,\ldots,\pi_{n-1})}$: Outputs all proofs π_i for $\mathbf{a}$.

$\underline{\mathsf{Agg}_{\mathsf{pp}}(I,(a_i,\pi_i)_{i\in I})\to(\pi_I,\Lambda_I)}$: Combines individual proofs π_i for values a_i into an aggregated proof π_I and batch-proof data structure Λ_I.

$\underline{\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C},I,(a_i)_{i\in I},\pi_I)\to\{0,1\}}$: Verifies proof π_I that each position $i\in I$ has value a_i against digest C.

$\underline{\mathsf{UpdDig}_{\mathsf{pp}}(u,\delta,\mathsf{C},\mathsf{aux})\to\mathsf{C}'}$: Updates digest C to C′ to reflect position u changing by δ given auxiliary input aux.

$\underline{\mathsf{UpdProof}_{\mathsf{pp}}(u,\delta,\pi_i,\mathsf{aux})\to\pi_i'}$: Updates proof π_i to π_i' to reflect position u changing by δ given auxiliary input aux.

$\underline{\mathsf{UpdBatchProof}_{\mathsf{pp}}(u,\delta,\pi_I,\Lambda_I,\mathsf{aux})\to(\pi_I',\Lambda_I')}$: Updates proof π_i to π_i' and the witness Λ_I to Λ_I' to reflect position u changing by δ given auxiliary input aux.

$\underline{\mathsf{UpdAllProofs}_{\mathsf{pp}}(u,\delta,\pi_0,\ldots,\pi_{n-1})\to(\pi_0',\ldots,\pi_{n-1}')}$: Updates all proofs π_i to π_i' to reflect position u changing by δ.

We assume that all algorithms have access to pp of the scheme. Observe that a Merkle tree can be considered as a VC scheme.

Correctness and soundness. We define VC correctness in Definition 2.3 and VC soundness in Definition 2.4.

Definition 2.3 (VC Correctness). *A VC is correct, if for all $\lambda\in\mathbb{N}$ and $n=\mathsf{poly}(\lambda)$, for all $\mathsf{pp}\leftarrow\mathsf{Gen}(1^\lambda,n)$, for all vectors $\boldsymbol{a}=[a_0,\ldots,a_{n-1}]$, if $\mathsf{C}=\mathsf{Com}_{\mathsf{pp}}(\boldsymbol{a})$, $\pi_i=\mathsf{Open}_{\mathsf{pp}}(i,\boldsymbol{a}),\forall i\in[0,n)$ (or from $\mathsf{OpenAll}_{\mathsf{pp}}(\boldsymbol{a})$), and $\pi_I=\mathsf{Agg}_{\mathsf{pp}}(I,(a_i,\pi_i)_{i\in I}),\forall I\subseteq[n]$ then, for any polynomial number of updates (u,δ) resulting in a new vector $\boldsymbol{a}'$, if C' is the updated digest obtained via calls to $\mathsf{UpdDig}_{\mathsf{pp}}$, π_i' proofs obtained via calls to $\mathsf{UpdProof}_{\mathsf{pp}}$ or $\mathsf{UpdAllProofs}_{\mathsf{pp}}$ for all i, and π_I' proofs obtained via calls to $\mathsf{UpdBatchProof}_{\mathsf{pp}}$ for all subsets I then:*

1. $\Pr[1\leftarrow\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}',\{i\},a_i',\pi_i')]=1,\forall i\in[n]$
2. $\Pr[1\leftarrow\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}',I,\ (a_i')_{i\in I},\ \mathsf{Agg}_{\mathsf{pp}}(I,(a_i',\pi_i')_{i\in I}))]=1$, $\forall I\subseteq[n]$.

Observation: At a high-level, correctness says that proofs created via Open or OpenAll verify successfully via Ver, even in the presence of updates and aggregated proofs.

Definition 2.4 (VC Soundness). $\forall$ PPT *adversaries* $\mathcal{A}$,

$$\Pr\left[\begin{array}{c}\mathsf{pp}\leftarrow\mathsf{Gen}(1^\lambda,n), \\ (\mathsf{C},I,J,(a_i)_{i\in I},(a_j')_{j\in J},\pi_I,\pi_J')\leftarrow\mathcal{A}(1^\lambda,\mathsf{pp}): \\ 1\leftarrow\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C},I,(a_i)_{i\in I},\pi_I)\ \wedge \\ 1\leftarrow\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C},J,(a_j')_{j\in J},\pi_J')\ \wedge \\ \exists k\in I\cap J \text{ s.t. } a_k\neq a_k'\end{array}\right]\leq\mathsf{negl}(\lambda)$$

Observation: Soundness says that no adversary can output two *inconsistent proofs* for different values $a_k \neq a'_k$ at position k with respect to an adversarially-produced digest d. Note that such a definition allows the digest C to be produced adversarially.

3 Recproofs

In this section, we present the Recproofs VC scheme, which extends Merkle trees to provide updatable batch proofs. Recall that we assume, $n = 2^\ell$, where ℓ is the height of the tree.

To compute the commitment, C, of a vector $\mathbf{a} = [a_0, \ldots, a_{n-1}]$ in our scheme, we compute the Merkle tree digest of $\mathbf{a}$. The proof of opening for an index i in the vector is the Merkle membership proof of leaf a_i. We present the algorithms of Recproofs in Fig. 2.

We now present the algorithm to aggregate proofs (§3.2) in our scheme. Specifically, we first present *canonical hashing* algorithm (§3.1), which we require to compute the batch proof in our scheme.

3.1 Canonical hashing

The *canonical hashing* is a deterministic algorithm to compute a digest of subset I of k leaves from a set of 2^ℓ leaves. We define the canonical hash of a node v of Merkle tree T *with respect to a subset* I, denoted $d(v,I)$, recursively as:

1. If v is a leaf node we distinguish two cases: If v's index is in I, then $d(v,I) := index(v)||value(v) = \mathsf{C}_v$, otherwise $d(v,I) := 0$.

2. If v has left child L and right child R, then $d(v,I) := \mathsf{H}(d(L,I)||d(R,I))$, if $d(L,I) \cdot d(R,I) \neq 0$, otherwise $d(v,I) := d(L,I) + d(R,I)$.

Thus, the *canonical digest* of subset I (denoted as d_I or, simply, d) is the canonical hash of the root node of T for the subset I. Note that when the subset I is unambiguous from the context, we denote $d(v,I)$ as d_v.

3.2 Batch proofs

A batch proof is a single short proof that simultaneously proves the opening of multiple elements in the vector. In our scheme, a batch proof for some subset $I \subseteq [n]$, is a single recursive SNARK proof that proves the valid opening of all the elements in I. Before we describe the precise circuit, we present the NP statement for the batch proof:

"d is the root canonical digest with respect to *some* set of leaves of some Merkle tree of ℓ levels whose root Merkle digest is C."

Recursive SNARK circuit. We present the circuit $\mathcal{B}$ that the recursive SNARK proof verifies. We indicate below precisely what the public input of the circuit is and what the (private) witness is. Note that the pseudocode below contains the SNARK verification algorithm using key $\mathsf{vk}_\mathcal{B}$—namely, the **fixed** verification key (that does not change across recursive calls) that corresponds to a SNARK computed on $\mathcal{B}$.

Batch Proof, $\mathcal{B}$

Public input: C, d, ℓ

Witness: $(\mathsf{C}_L, d_L, \ell_L, \pi_L)$ and $(\mathsf{C}_R, d_R, \ell_R, \pi_R)$

Computation:

1. Check $\ell = \ell_L + 1$ and $\ell = \ell_R + 1$
2. Check $\mathsf{C} = \mathsf{H}(\mathsf{C}_L||\mathsf{C}_R)$
3. If $d_R \cdot d_L \neq 0$ check $d = \mathsf{H}(d_L||d_R)$
   else check $d = d_L + d_R$
4. If $\ell = 1$ and $d_L \neq 0$ check $d_L = \mathsf{C}_L$
5. If $\ell = 1$ and $d_R \neq 0$ check $d_R = \mathsf{C}_R$
6. If $\ell > 1$ and $d_L \neq 0$
   Check $\mathsf{Verify}(\mathsf{vk}_\mathcal{B}, (\mathsf{C}_L, d_L, \ell_L), \pi_L)$
7. If $\ell > 1$ and $d_R \neq 0$
   Check $\mathsf{Verify}(\mathsf{vk}_\mathcal{B}, (\mathsf{C}_R, d_R, \ell_R), \pi_R)$
8. Return true

Computing the witness. Given a subset of indices to batch, the witness required to compute the SNARK proof for circuit $\mathcal{B}$ can be calculated by accessing all the values in the VC or by accessing individual valid proofs of opening. For the sake of simplicity, in the procedure described below, we assume that all the vector values are available. However, we define a procedure to compute the batch proof from individual proofs in Fig. 2.

Given the subset I and the vector, we compute the witness as follows:

1. Compute the Merkle tree on all the 2^ℓ leaves. This outputs the Merkle hash C_v for all nodes v.

2. Compute the canonical hash $d(v,I)$ for all nodes v *with respect* to I.

3. Compute SNARK proofs from the lower levels of the recursion back for use in the higher levels.

Observation: Note that from the definition of canonical hashing and $\mathcal{B}$, we do not need Merkle hashes and canonical hashes of *all* the nodes in the tree. It is sufficient to calculate the canonical and Merkle hashes of all the nodes along the path from the leaf to the root and the Merkle hashes of all the siblings along the way to the root (as depicted in Fig. 1).

Computing the batch proof. In the public parameters, we run the setup of the SNARK: $(\mathsf{pk}_\mathcal{B}, \mathsf{vk}_\mathcal{B}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{B})$ for

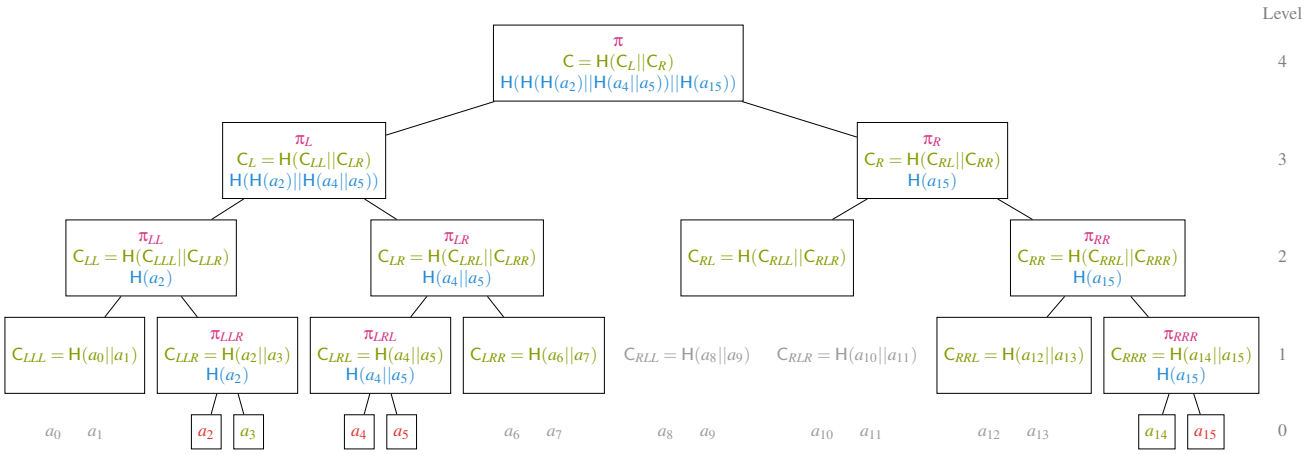


Figure 1: Batch proof data structure. Consider a vector of size 8 and subset $I = \{2,4,5,15\}$. Recall that every leaf stores *both* index and value. Every node v in the path from root to leaves of I, stores the Merkle hash, canonical hash with respect to I, and the recursive SNARK proof, and every sibling of v stores just the Merkle hash.

the circuit $\mathcal{B}$. Consider the set of indices I for which we are calculating the batch proof. Let $p_1, p_2, \ldots, p_{|I|}$ be the paths from the leaves in I to the root of the Merkle tree. Clearly, these paths are *not* disjoint. Let v_{ij}, for $j = 0, \ldots, \ell$, denote the j-th node of path p_i, starting from level 0. To compute the batch proof, we follow the procedure below. For all levels $m = 1, \ldots, \ell$ and for all paths p_i where $i = 1, \ldots, |I|$ do:

For all distinct nodes $v = v_{im}$ with Merkle hash C_v, canonical hash d_v, with left child L (of Merkle hash C_L, canonical hash d_L, proof π_L) and right child R (of Merkle hash C_R, canonical hash d_R, proof π_R), compute:

$$\pi_{v_{im}} \leftarrow \mathsf{Prove}(\mathsf{pk}_{\mathcal{B}}, (\mathsf{C}_v, d_v, m), (\mathsf{C}_L, d_L, m-1, \pi_L, \mathsf{C}_R, d_R, m-1, \pi_R))$$

The batch proof for subset I is the value $\pi_{v_{i\ell}}$. This implies that the recursive SNARK proof computed at the root of the Merkle tree serves as the batch proof. The *batch proof data structure* for a subset I, Λ_I,consist of all the Merkle hash values, canonical hash values, and recursive SNARK proofs computed along the nodes from leaves to the root, and the Merkle hashes of all the siblings on the paths from leaves of the subset to the root (as depicted in Fig. 1).

Complexity. Our approach has parallel complexity ℓ, independent of $|I|$, whereas naive approaches have parallel complexity $\ell + \log |I|$. This is because we are folding the canonical hashing computation inside the Merkle verification.

Security analysis. Here we outline the security argument to demonstrate the soundness of both individual and aggregated Recproofs.

Theorem 3.1 (Individual Recproofs are sound)**.** *Our* individual $\log n$*-sized (non-aggregated) proofs from Fig. 2 are sound as per Definition 2.4 assuming the collision resistance of the hash function.*

Proof sketch for Thm. 3.1. Since the individual Recproofs is a Merkle proof of opening of an index in the vector, the security argument of individual Recproofs directly follows from the security of [12]. Say if an adversary $\mathcal{A}$, returns a digest, index i, and proof π, π' (both accepted by the verifier). Then it would require the adversary to find a collision in the hash function, which is assumed to be computationally infeasible. □

Theorem 3.2 (Batch Recproofs are sound)**.** *Our* batch *proofs from §3.2 and Fig. 2 are sound as per Definition 2.4 under the knowledge-soundness of the SNARK (Definition 2.1) and collision resistance of the hash function.*

Proof sketch for Thm. 3.2. To prove the security argument of the batch proof: First we argue the soundness of the canonical digest with respect to a subset I (chosen by the adversary). That is, similar to Merkle digests, we argue that canonical digest with respect to a subset is secure. Second we argue that knowledge-soundness of the SNARK allows us to recursively extract Merkle hash and canonical hash of all the nodes used in computing the batch proof. Finally, we argue that computing two valid batch proofs that open an index to different values (under the same C) is similar to breaking the soundness of individual Recproofs (Thm. 3.1). We defer the proofs to the extended version of the paper. □

3.3 Updates

Updating batch proofs. Note that a significant feature of our construction is the support for *dynamic batch proofs*, i.e., the fact that one can update a batch proof (of a subset I), when any leaf value changes (independently of whether a leaf belongs to I or not). This feature has applications in the DeFi space, e.g., when one wishes to maintain a proof for a moving average for a contract variable (such as token price), as new blocks are being created, without recomputing the new proof from scratch. The main idea for maintaining a dynamic batch proof is the following:

$\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}$: Let pp contain the following:
- A collision-resistant hash function H.
- $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{B})$

$\mathsf{Com}_{\mathsf{pp}}(\mathbf{a}) \to \mathsf{C}$: Return the digest of the Merkle tree.

$\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}) \to \pi_i$: Return the Merkle hash of the nodes that are required to reconstruct the path from a_i to root.

$\mathsf{OpenAll}_{\mathsf{pp}}(\boldsymbol{a}) \to (\pi_0, \ldots, \pi_{n-1})$: Return the Merkle tree.

$\mathsf{Agg}_{\mathsf{pp}}(I, (a_i, \pi_i)_{i \in I}) \to (\pi_I, \Lambda_I)$:
- Compute C using any π_i
- Compute the partial Merkle hash tree T' using π_i's
- For every node v in T':
  - Compute the canonical hash $d(v, I)$
  - Compute the recursive SNARK proof π_v (as described in §3.2)
  - Augment v with $d(v, I)$ and π_v
- Return the recursive SNARK proof computed at the root as π_I and augmented tree T' as Λ_I

$\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i)_{i \in I}, \pi_I) \to \{0, 1\}$:
- Compute d_I
- Return $\mathsf{Verify}(\mathsf{vk}_{\mathcal{B}}, (\mathsf{C}, d_I, \ell), \pi_I)$

$\mathsf{UpdDig}_{\mathsf{pp}}(u, \delta, \mathsf{C}, \mathsf{aux}) \to \mathsf{C}'$:
- Parse aux as π_u
- Return the root of the Merkle hash path recomputed after updating $a_u + \delta$

$\mathsf{UpdProof}_{\mathsf{pp}}(u, \delta, \pi_i, \mathsf{aux}) \to \pi_i'$:
- Parse aux as π_u
- Recompute the Merkle hash path after updating $a_u + \delta$
- Update affected portions of π_i with the recomputed hash path

$\mathsf{UpdBatchProof}_{\mathsf{pp}}(u, \delta, \pi_I, \Lambda_I, \mathsf{aux}) \to (\pi_I', \Lambda_I')$:
- Parse aux as π_u
- Recompute the Merkle hash path after updating $a_u + \delta$
- For every node v in Λ_I affected by the update:
  - Update the Merkle hash value
  - Recompute the canonical hash $d'(v, I)$, if necessary
  - Recompute the recursive SNARK proof π_v' (as described in §3.3)

$\mathsf{UpdAllProofs}_{\mathsf{pp}}(u, \delta, \pi_0, \ldots, \pi_{n-1}) \to (\pi_0', \ldots, \pi_{n-1}')$:
- Parse aux as π_u
- Recompute the Merkle hash path after updating $a_u + \delta$

Figure 2: Algorithms for Recproofs

1. After the computation of the batch proof, we do not discard the batch proof data structure, which contains the recursive SNARK proofs, Merkle hashes, and canonical hashes in a tree-like data structure.

2. Whenever a leaf value changes, we distinguish two cases: a) the element belongs to the batch, in which case the canonical hash will change; b) the element does not belong to the batch, in which case the canonical hash will not change. But in both cases the recursive SNARK proofs, Merkle hashes, and the batch proof will change. To update it efficiently, we identify which nodes of the batch data structure are affected, and we spend $\log n$ time updating the batch proof data structure, which in the end will give the updated dynamic batch proof.

With this approach, we can update a batch proof of k elements in time $O(\log n)$, instead of spending time $O(k \log n)$ that previous approaches would require.

Updating the batch. Say an element a_j does not belong to the batch I. Given the valid proof π_j and the batch proof data structure Λ_I, it is possible to compute the batch proof of the set $I' = I \cup \{j\}$. To update the batch proof efficiently, we identify the nodes of the batch proof data structure that are affected. Then, we compute and update the Merkle tree hashes, canonical hashes, and the recursive SNARK proof for all the nodes along the path from a_j to the root of the batch proof data structure. Similarly, we can remove an element a_j from the batch, by pruning the batch proof data structure and recomputing the values stored in the nodes of the affected portion of the tree.

Updating digests and individual proofs. Say a_i changes to a_i'. With the valid proof of a_i, π_i, under commitment C, it is possible to compute the updated commitment C', by recomputing the Merkle hashes from a_i' to the root of the Merkle tree. The updated Merkle hashes along the path to the root can also be used to update any proof by updating just the portions of the proof that intersect.

3.4 Optimized recproofs circuit

Recall that in SNARKs, the prover incurs the overhead of both branches of the conditionals regardless of the branch that is executed. Observe that the prover incurs the overhead of the conditionals in steps 6 and 7 (§3.2) in $\mathcal{B}$ even when computing the SNARK proofs at level 1 do not involve the expensive SNARK verification. To remove this overhead, we decompose $\mathcal{B}$ into two smaller circuits: One for proof computation in the base case (denoted as $\mathcal{B}''$) and the other for the rest of the computation (denoted as $\mathcal{B}'$). We describe the decomposed circuits here:

Base case, $\mathcal{B}''$

Public input: (C, d)

Witness: (C_L, d_L) and (C_R, d_R)

Computation:

1. Check $\mathsf{C} = \mathsf{H}(\mathsf{C}_L || \mathsf{C}_R)$
2. If $d_R \cdot d_L \neq 0$ check $d = \mathsf{H}(d_L || d_R)$ else check $d = d_L + d_R$
3. If $d_L \neq 0$ check $d_L = \mathsf{C}_L$
4. If $d_R \neq 0$ check $d_R = \mathsf{C}_R$
5. Return true

Optimized batch proof, $\mathcal{B}'$

Public input: (C, d)

Witness: $(\mathsf{C}_L, d_L, \pi_L)$, $(\mathsf{C}_R, d_R, \pi_R)$, and vk

Computation:

1. Check $\mathsf{C} = \mathsf{H}(\mathsf{C}_L || \mathsf{C}_R)$
2. If $d_R \cdot d_L \neq 0$ check $d = \mathsf{H}(d_L || d_R)$
   else check $d = d_L + d_R$
3. Check if vk is either $\mathsf{vk}_{\mathcal{B}''}$ or $\mathsf{vk}_{\mathcal{B}'}$
4. If $d_L \neq 0$ check $\mathsf{Verify}(\mathsf{vk}, (\mathsf{C}_L, d_L), \pi_L)$
5. If $d_R \neq 0$ check $\mathsf{Verify}(\mathsf{vk}, (\mathsf{C}_R, d_R), \pi_R)$
6. Return true

Note that one key difference between $\mathcal{B}$ and $\mathcal{B}'$ is that $\mathcal{B}'$ takes the verification key as input to the circuit to accommodate the fact that we are using two circuits.

Computing the witness and batch proof. The procedure to compute the witness for the optimized circuits is identical to the procedure explained in §3.2. However, in the public parameters, we run the setup for both circuits $\mathcal{B}''$ and $\mathcal{B}'$ as follows: $(\mathsf{pk}_{\mathcal{B}''}, \mathsf{vk}_{\mathcal{B}''}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{B}'')$ and $(\mathsf{pk}_{\mathcal{B}'}, \mathsf{vk}_{\mathcal{B}'}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{B}')$ for the circuit $\mathcal{B}''$ and $\mathcal{B}'$, respectively.

Consider the set of indices I for which we are calculating the batch proof. Let $p_1, p_2, \ldots, p_{|I|}$ be the paths from the leaves in I to the root of the Merkle tree. Clearly, these paths are not disjoint. Let v_{ij}, for $j = 0, \ldots, \ell$, denote the j-th node of path p_i, starting from level 0. To compute the batch proof, we follow the procedure below. For all levels $m = 1, \ldots, \ell$ and for all paths p_i where $i = 1, \ldots, |I|$ do:

For all distinct nodes $v = v_{im}$ with Merkle hash C_v, canonical hash d_v, with left child L (of Merkle hash C_L, canonical hash d_L, proof π_L) and right child R (of Merkle hash C_R, canonical hash d_R, proof π_R), compute:

If $m = 1$

$\pi_{v_{im}} \leftarrow \mathsf{Prove}(\mathsf{vk}_{\mathcal{B}''}, (\mathsf{C}_v, d_v), (\mathsf{C}_L, d_L, \mathsf{C}_R, d_R))$

If $m = 2$

$\pi_{v_{im}} \leftarrow \mathsf{Prove}(\mathsf{vk}_{\mathcal{B}'}, (\mathsf{C}_v, d_v), (\mathsf{C}_L, d_L, \pi_L, \mathsf{C}_R, d_R, \pi_R, \mathsf{vk}_{\mathcal{B}''}))$

Otherwise

$\pi_{v_{im}} \leftarrow \mathsf{Prove}(\mathsf{vk}_{\mathcal{B}'}, (\mathsf{C}_v, d_v), (\mathsf{C}_L, d_L, \pi_L, \mathsf{C}_R, d_R, \pi_R, \mathsf{vk}_{\mathcal{B}'}))$

The batch proof for subset I is the value $\pi_{v_{i\ell}}$.

3.5 Batch Proofs for q-ary Trees

We now present the algorithm to batch proofs in q-ary trees. First, we present the generalized *canonical hashing* algorithm and then present the batch proof computation circuit. For simplicity, we assume that the q-ary tree is balanced of height ℓ. However, we remark that our approach can be generalized to incorporate Merkle Patricia Tries (MPT) in practice [1].

Canonical hashing. We describe the *canonical hashing* algorithm to compute a digest of subset I of k leaves from a set of $n = q^\ell$ leaves. We define the canonical hash of a node v of a q-ary Merkle tree T *with respect to a subset* I, denoted $d(v, I)$, recursively as:

1. If v is a leaf node we distinguish two cases: If v's index belongs in I, then $d(v, I) := index(v) || value(v) = \mathsf{C}_v$, otherwise $d(v, I) := 0$.
2. If v has children $w_1, \ldots, w_q$, then $d(v, I) := \mathsf{H}(d(w_1, I) || \ldots || d(w_q, I))$, if at least two children are non-zero, otherwise $d(v, I) = \sum_{b=1}^{q} d(w_b, I)$.

Thus, the *canonical digest* of subset I is the canonical hash of the root node of T for the subset I.

Recursive SNARK circuit. We present the circuit Q for the same NP statement in §3.2, however, for a q-ary tree.

Batch Proof, Q

Public input: C, d, ℓ

Witness: $(\mathsf{C}_b, d_b, \ell_b, \pi_b)_{b \in [1,q]}$

Computation:

1. Check $\mathsf{C} = \mathsf{H}(\mathsf{C}_1 || \ldots || \mathsf{C}_q)$
2. If $(d_b \neq 0 \wedge d_{b'} \neq 0)_{\exists b, b' \in [1,q] : b \neq b'}$
   Check $d = \mathsf{H}(d_1 || \ldots || d_q)$
   else check $d = \sum_{b=1}^{q} d_b$
3. For all $b \in [1, q]$
   (a) Check $\ell = \ell_b + 1$
   (b) If $\ell = 1$ and $d_b \neq 0$ check $d_b = \mathsf{C}_b$
   (c) If $\ell > 1$ and $d_b \neq 0$
   Check $\mathsf{Verify}(\mathsf{vk}_Q, (\mathsf{C}_b, d_b, \ell_b), \pi_b)$
4. Return true

Computing the witness and batch proof. The procedure to compute the witness and batch proof in q-ary trees closely resembles the batch proof computation explained in §3.2. For all distinct nodes v at level m in the q-ary tree with Merkle hash C_v, canonical hash d_v, with q children (of Merkle hash C_b, canonical hash d_b, proof π_b for all $b \in [1, q]$), compute:

$$\pi_v \leftarrow \mathsf{Prove}(\mathsf{pk}_Q, (\mathsf{C}_v, d_v, m), (\mathsf{C}_b, d_b, m-1, \pi_b)_{b \in [1,q]})$$

The batch proof will be the value π_r, where r is the root node of the q-ary tree.

Complexity. Our approach has parallel complexity ℓ, independent of $|I|$, whereas naive approaches have parallel complexity $\ell + \log_q |I|$. This is because we are folding the canonical hashing computation inside the Merkle verification.

4 Verifiable MapReduce

Our approach to batch proof can be extended to prove the correctness of MapReduce-style [6] computation on dynamic

data. The maintainability property of our construction allows us to efficiently recompute the proof of correctness even when the underlying data changes. This contrasts the need for recomputing from scratch in naive approaches.

Let $\mathcal{D}$ and $\mathcal{R}$ denote the domain of the input and output of the computation, respectively. We consider the following abstraction for MapReduce:

- $\mathsf{Map} : \mathcal{D} \to \mathcal{R}$
- $\mathsf{Reduce} : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$

We remark that the Reduce operation can also take just a single input. In such cases, we assume the existence of an identity element, e, for the Reduce operation.

Assume we want to execute the MapReduce computation on a subset $I \subseteq [n]$ of the memory slots which have d as their canonical digest. We present the recursive algorithm that checks the validity of the following NP statement:

"out is the output of the MapReduce computation on some set of leaves which (i) have d as their root canonical index digest; (ii) belong to some Merkle tree of ℓ levels whose root Merkle digest is C."

We present the circuit $\mathcal{M}$ that verifies the correctness of the MapReduce computation.

MapReduce, $\mathcal{M}$

Public input: $(\mathsf{out}, \mathsf{C}, d, \ell)$

Witness: $(\mathsf{out}_b, \mathsf{C}_b, d_b, \ell_b, \pi_b)_{b\in\{L,R\}}$

Computation:

1. Check $\mathsf{C} = \mathsf{H}(\mathsf{C}_L || \mathsf{C}_R)$
2. If $d_R \cdot d_L \neq 0$
   Check $d = \mathsf{H}(d_L || d_R)$
   else
   Check $d = d_L + d_R$
3. Check $\mathsf{out} = \mathsf{Reduce}(\mathsf{out}_L, \mathsf{out}_R)$
4. For all $b \in \{L, R\}$
   (a) Check $\ell = \ell_b + 1$
   (b) If $\ell = 1$ and $d_b \neq 0$
   Parse C_b as $index||value$
   Check $d_b = \mathsf{C}_b$ and $\mathsf{out}_b = \mathsf{Map}(value)$
   (c) If $\ell > 1$ and $d_b \neq 0$
   Check $\mathsf{Verify}(\mathsf{vk}_{\mathcal{M}}, (\mathsf{out}_b, \mathsf{C}_b, d_b, \ell_b), \pi_b)$
5. Return true

Computing the witness and MapReduce proof. The procedure to compute the witness for the $\mathcal{M}$ is similar to the procedure explained in §3.2. However, we additionally compute the output of the MapReduce computation along with canonical hash $d(v, I)$ for nodes along the path from leaves in the subset to the root of the tree. Whenever, the canonical hash of a node is 0, we set the result of the MapReduce as identity element e. For all distinct nodes v at level m in the tree with Merkle hash C_v, canonical hash d_v, and MapReduce result out_v (of Merkle hash C_b, canonical hash d_b, proof π_b, and MapReduce result out_b for all $b \in \{L, R\}$), compute:

$$\pi_v \leftarrow \mathsf{Prove}(\mathsf{pk}_{\mathcal{M}}, (\mathsf{out}_v, \mathsf{C}_v, d_v, m), (\mathsf{out}_b, \mathsf{C}_b, d_b, m-1, \pi_b)_{b\in\{L,R\}})$$

The proof of correct MapReduce computation will be the value π_r, where r is the root node of the tree.

5 Applications

In this section, we specifically discuss how the ideas of batch proofs in Recproofs can be used to prove the correctness of MapReduce computation over large amounts of dynamic on-chain state data. Recproofs are generalizable to the Merkle-Patricia Tries used by popular blockchains, such as Ethereum, to store smart contract states. The updatability of Recproofs makes them a powerful primitive for proving expensive and long-running computations on the continually updating data structures used by blockchains.

Aggregated public keys. Consider the following setting: Say BLS public keys of n validators are stored in the memory of a smart contract. Now the goal is to calculate the aggregated public key of a subset of validators, denoted as I, and the cardinality of this subset I to establish the fraction of validators that have signed the message. However, subset I can change across blocks. The problem of computing aggregated public key and the cardinality on-chain is useful in emerging real-world blockchain systems (E.g., Proofs of Ethereum Beacon Chain consensus or EigenLayer restaking). In these systems, validators attest to the results of some specific computational task, and a new tasks can arrive periodically. An existing approach is to attach a SNARK proof along with the aggregated public key and the cardinality of the attestor subset. However, this requires recomputing the SNARK proof *from scratch* every time when the subset changes. Additionally, this also requires computing a new proof *from scratch* whenever the set changes even if the subset stays the same..

MapReduce proofs for aggregated public keys. At a high-level, we need to prove the following NP statement: "(k, apk) is the output of the MapReduce computation on some set of leaves which belong to some Merkle tree of $n = 2^\ell$ BLS public keys whose root Merkle digest is C." This statement can be efficiently proved using batching techniques from Recproofs even when the subset changes.

Let $\mathbb{G}$ be the elliptic curve group for the BLS public keys, $\mathbb{Z}$ be the set of integers. Let the input and output domain of MapReduce be $\mathbb{G}$ and $\mathbb{Z} \times \mathbb{G}$, respectively. Let $e = (0, 1) \in \mathbb{Z} \times \mathbb{G}$ denote the identity element in $\mathcal{R}$. We define the MapReduce functions as follows:

- Map: Takes a public key g^{sk}, returns $(1, g^{\mathsf{sk}})$, if sk has signed the message, else e.
- Reduce: Takes two elements (a, g^{sk_1}) and (b, g^{sk_2}) from $\mathcal{R}$, and returns $(a+b, g^{\mathsf{sk}_1+\mathsf{sk}_2})$.

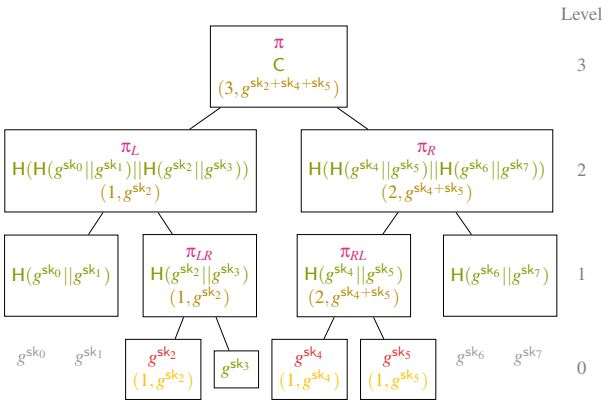


Figure 3: Batch proof data structure to compute the aggregated BLS public key and cardinality of the attestor set. Consider a vector of size 8 and attestor subset $I = \{2,4,5\}$. Recall that every leaf stores *both* index and value, however, we omit this detail (and canonical hashes of every node) for simplicity. Every node v in the path from root to leaves of I, stores the Merkle hash, the recursive SNARK proof, and results of Map or Reduce operation, and every sibling of v stores just the Merkle hash.

To prove the correctness of the aggregate public key and subset cardinality, we compute the correctness proof for the MapReduce computation using the circuit $\mathcal{M}$ described in §4. We depict the maintainable data structure for the MapReduce computation in Fig. 3. Thus, any verifier can check validity of this proof of correctness using the reported apk, k, n. Whenever the attestor subset changes, we can simply update the path from leaf to root to compute the new proof of correctness in $m \log n$ time (using the data structure in Fig. 3), instead of recomputing from scratch, where m is the number of updates to the subset I.

Digest translation. Digest translation is an emerging solution where a cryptographic proof argues that the Merkle digests C_1 and C_2 computed using hash functions H_1 (e.g., SHA-2) and H_2 (e.g., Poseidon), respectively, corresponds to the same n values, which can change over time. This cryptographic proof is useful in zk-rollups as any inclusion proof based on SHA-2 is inefficient to verify inside the zk-VM. To get around this limitation every inclusion proof submitted to the zk-rollup can be computed using a Poseidon hash function and all inclusion proofs sequenced in a batch would additionally *just need a* proof of digest translation to verify against the Merkle digest computed using SHA-2. An approach is to have a recursive SNARK proof that takes the proof of digest translation from the previous state, inclusion proofs under both hash functions, and the updated value and digest to compute an updated proof of digest translation using recursive SNARKs. However, this approach is not parallelizable as each update to the tree has to be computed sequentially in the proof one at time.

MapReduce proofs for digest translation. Say a Merkle is constructed using hash function H_1, and let the Merkle digest be C_1. To compute a digest translation proof to argue that C_2 is the Merkle digest when computed using H_2, we need to prove the following NP statement: "C_2 is the output of the MapReduce computation on all leaves which belong to some Merkle tree of ℓ levels whose root Merkle digest is C_1." This statement can be efficiently proved using the MapReduce proofs (§4) even when the subset changes.

Let $\mathcal{H}$ and $\mathcal{U}$ denote the hash space of H_2 and domain of leaf values of the tree, respectively. Let the input and output domain of MapReduce be $\mathcal{U}$ and $\mathcal{H}$, respectively. Let $e = 0 \in \mathcal{H}$ denote the identity element in $\mathcal{R}$.

We define the MapReduce functions as follows:

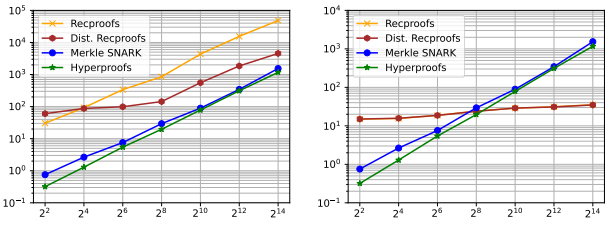
- Map: An identity function. Takes a leaf data and returns the same. Recall that for simplicity we assume that both $\mathcal{U}$ and $\mathcal{H}$ are $\{0,1\}^{2\lambda}$.
- Reduce: Takes two elements $a, b \in \mathcal{H}$ and returns $c := \mathsf{H}_2(a||b) \in \mathcal{H}$.

We set I as the entire set of indices $[n]$, as this new digest C_2 is computed over all leaves. To prove the correctness C_2, we compute the correctness proof for the MapReduce computation using the circuit $\mathcal{M}$ described in §4. Thus, any verifier can check validity of this MapReduce proof using the reported $\mathsf{C}_2, \mathsf{C}_1$. Whenever the values in the vector changes, we can simply update the path from leaf to root to compute the new proof of correctness in $m \log n$ time (using the data structure similar to Figs. 1 and 3), instead of recomputing from scratch, where m is the number of updates. Additionally, our updates are parallelizable as multiple updates can be applied one level at time. This approach can be applied to digest transformations involving both the changing of hash functions and the changing of the serialization structures used for leaf and branch elements.

Observation. Note that in both of the discussed applications, the public statement doesn't require the inclusion of the canonical digest. Thus demonstrating knowledge of the subset is adequate for these use-cases.

Other applications. Our MapReduce proofs can applied to real-world applications involving decentralized finance (DeFi) that require computation over on-chain states that spans multiple concurrent blocks. Examples of these applications include calculating moving averages of asset prices, lending market deposit rates, credit scores or airdrop eligibility. In these applications, a computation must be applied across the states of a contract or group of contracts for the n most recent blocks of a given blockchain, where n is a fixed number. These computations must then be updated continually whenever a new blocks is added to the blockchain. The natural updatability of Recproofs allows it to be extended to use-cases where a proof must be generated across tens of thousands of consecutive blocks of historical data.

For example, an on-chain options protocol may want to price an option using the volatility of an asset over the past n blocks on a decentralized exchange on Ethereum. The proof of the volatility must then be updated every 12 seconds as new blocks are added. The naive approach would require computing a proof of the volatility across the past n blocks *from*



(a) Aggregation time (seconds) (b) Update time (seconds)

Figure 4: The x-axis is # of proofs being aggregated. We extrapolate the Merkle SNARK numbers for batch size beyond 2^{12} due to large memory overhead. We use the 128-bit variant of Poseidon.

scratch, every 12 seconds. With Recproofs, this computation would only require updating the portion of the proof associated with the computation done on the oldest block with a proof of computation done on the new block.

6 Evaluation

In this section, we present our initial *preliminary* evaluation results. We defer the full performance analysis of our algorithms to the extended version of the paper. We compare the performance of our batch update and aggregation technique against two baselines (§6.1): (1) Naive Merkle proof aggregation using SNARKs [14] and (2) Inner-product arguments based aggregation in Hyperproofs [16].

Our implementation is based on Rust (nightly-2023-03-06) and we use Plonky2 [15] for our batch proof construction. Thus a field element corresponds to a 64-bit value from the Goldilocks field. We run our experiments on Amazon EC2 c5a.8xlarge instance, which has 32 cores and 64 GiB RAM. Except the SNARK provers (both Bellman and Plonky2), our experiments are single-threaded and report the average running times of 10 runs.

Recall that each leaf of Recproofs tree is a concatenation of the index and value. In our implementation, each vector element is a random 256 bit value and we allocate 64 bits to the index. Thus, each leaf is 40 bytes long. The Merkle tree in our construction uses Poseidon hash function [9].

6.1 Batch updates and aggregation.

In this subsection, we present the performance of our batch update and aggregation. Recall that aggregation combines multiple individual proof into a single batch proof.

Experimental setup. We set the height of the tree to $\ell = 29$ and study the performance of our scheme for varying batch sizes $k = \{2^2, 2^4, \ldots, 2^{14}\}$. For baseline comparison, we use the following:

1. Merkle SNARKs: We use a fork of the Rust implementation by Ozdemir *et al.* that was used in Hyperproofs to benchmark Merkle SNARKs [13, 14, 16]. We remark that the prover uses the standard parallelism available in Bellman to compute the Groth16 proofs.

2. Hyperproofs: We use the golang based implementation that was provided in Hyperproofs [16].

We compare the performance of Recproofs in the following settings:

1. Standard: In this implementation of aggregation, each node of the batch proof data structure is constructed sequentially.

2. Distributed proof generation: In this implementation of aggregation, we distribute the prover effort to construct a node in the batch proof data structure to a cluster of machines. That is, for every level in the batch proof data structure, a leader distributes the task of computing the recursive SNARK proofs to multiple followers. Thus, multiple followers can, in parallel, work on a level of the batch proof data structure. In our setting, we use a cluster of 100 machines (16 Core, 16 GiB RAM).

In each run, we randomly generate a Merkle tree and select a random set of leaves to batch/update.

Prover time. The tree structure of our construction presents avenues for parallelism. Thus by distributing the proof generation, as shown in Fig. 4(a), we observe a 3.4× to 10.6× faster prover time than the standard implementation of Recproofs for batch sizes between 2^6 to 2^{14}.

Groth16 [10] based Merkle SNARKs aggregation is around 3× faster than Recproofs for batch size 2^{14}. The main limitation of this approach is that the batch size is predetermined during the setup of the SNARK. Thus any change to batch size requires a new Groth16 setup. We discuss this in detail in batch updates. The aggregation of Hyperproofs outperforms other approaches. However, Hyperproofs trades this for a large proof size and increased verification times.

Verification time and proof size. The batch proofs in our scheme is 45.13 KiB. This single Plonky2 proof can verify the entire batch. To verify a batch proof, the verifier first needs to compute the canonical digest of the batch. Then the verifier invokes the Plonky2 verifier with the computed canonical digest and the digest of the vector to check the validity of the proof. In our experiments, for a batch size of 2^{10} proofs, it takes 2.21 ms to compute the canonical digest and 6.55 ms to verify the Plonky2 proof. However, a batch proof in Hyperproofs is 52 KiB and takes around 11.08 seconds to verify. We defer the gas cost analysis to the extended version of the paper.

Batch updates. In our experiments, we randomly sample an element from the batch to update. Here we present the prover cost to update a batch proof. Updating a batch proof in Recproofs involves re-computing the recursive SNARK proofs along the path from leaf to the root. Thus, it is possible

to update the batch proof in logarithmic time. We observe that the cost of updating a single proof within a batch of 2^{14} values is 35.46 seconds.

However, both Merkle SNARKs and Hyperproofs do not support updatable batch proofs. Thus, both these schemes have to recompute the batch proof *from scratch* whenever an element in the batch changes. For a batch size of 2^{14} values, Merkle SNARKs and Hyperproofs require 25.62 and 19.45 minutes, respectively, to recompute the batch proof. Thus, as show in Fig. 4(b), Recproofs is around 43× and 33× faster than Merkle SNARKs and Hyperproofs, respectively when the batch is 2^{14}.

Besides updating an element inside the batch, our construction can also efficiently *update the size of the batch*. In contrast, both Merkle SNARKs and Hyperproofs require an apriori bound on the maximum size of the batch. Additionally, Merkle SNARKs incur proving cost proportional to the maximum batch size regardless of the number of elements in the batch. Whenever the batch size is insufficient, Merkle SNARKs require a setup with new "powers-of-tau" and circuit specific parameters. Recproofs does not suffer this limitation, allowing for flexibility in adjusting the batch size as required.

We estimate the cost of running a fresh Groth16 setup and computing a SNARK proof whenever the batch size is insufficient in Merkle SNARKs. The bellman-bignat [13] repository used in the our benchmarks return the cost of initialization, parameters generation, and circuit synthesis along with the prover overhead. For a batch size 2^{14}, the setup overhead is estimated around 54 minutes and the prover overhead is 25 minutes. However, in Recproofs the updates can be performed in 35.46 seconds. Thus, whenever a prover requires a new Groth16 circuit to accommodate larger batch sizes. say 2^{14}, we estimate that our performance will likely be up to 135× faster.

Acknowledgements

We'd like to thank the Lagrange Labs Engineering Team for their work in identifying and exploring applications of Recproofs to blockchain protocols. We also extend our gratitude to Rahul Ghangas for his assistance with the implementation of Recproofs. Rahul was contracted by Lagrange Labs to provide implementation support for Recproofs.

References

[1] https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/.

[2] Securely scaling zk interoperability. https://www.lagrange.dev/.

[3] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public-Key Cryptography - PKC 2013*, 2013.

[4] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang. Edrax: A Cryptocurrency with Stateless Transaction Validation. Cryptology ePrint Archive, Report 2018/968, 2018.

[5] Alessandro Chiesa, Eran Tromer, and Madars Virza. Cluster Computing in Zero Knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 371–403, 2015.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[7] Sai Deng and Bo Du. zkTree: A Zero-Knowledge Recursion Tree with ZKP Membership Proofs. Cryptology ePrint Archive, Paper 2023/208, 2023.

[8] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating Proofs for Multiple Vector Commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 2007–2023, 2020.

[9] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[10] Jens Groth. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, 2016.

[11] Jing Liu and Liang Feng Zhang. Matproofs: Maintainable matrix commitment with efficient aggregation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2041–2054, 2022.

[12] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435, pages 218–238, 1989.

[13] Alex Ozdemir. bellman-bignat, 2020. https://github.com/alex-ozdemir/bellman-bignat.

[14] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling Verifiable Computation Using Efficient Set Accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2075–2092. USENIX Association, August 2020.

[15] Plonky2 Polygon. Fast recursive arguments with plonk and fri. https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf.

[16] Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3001–3018, Boston, MA, August 2022. USENIX Association.

[17] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable Subvector Commitments for Stateless Cryptocurrencies. In Clemente Galdi and Vladimir Kolesnikov, editors, *Security and Cryptography for Networks*, pages 45–64, 2020.

[18] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards Scalable Threshold Cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 877–893, May 2020.

[19] Weijie Wang, Annie Ulichney, and Charalampos Papamanthou. BalanceProofs: Maintainable vector commitments with fast aggregation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4409–4426, Anaheim, CA, August 2023. USENIX Association.

[20] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 863–880, 2017.