Reckle Trees: Updatable Merkle Batch Proofs with Applications

Charalampos Papamanthou*[1,2], Shravan Srinivasan[1], Nicolas Gailly[1],
Ismael Hishon-Rezaizadeh[1], Andrus Salumets[1], Stjepan Golemac[1]

[1]Lagrange Labs
[2]Yale University

ABSTRACT

We propose Reckle trees, a new vector commitment based on succinct RECursive arguments and MerKLE trees. Reckle trees' distinguishing feature is their support for succinct batch proofs that are *updatable*—enabling new applications in the blockchain setting where a proof needs to be computed and efficiently maintained over a moving stream of blocks. Our technical approach is based on embedding the computation of the batch hash inside the recursive Merkle verification via a hash-based accumulator called *canonical hashing*. Due to this embedding, our batch proofs can be updated in logarithmic time, whenever a Merkle leaf (belonging to the batch or not) changes, by maintaining a data structure that stores previously-computed recursive proofs. Assuming enough parallelism, our batch proofs are also computable in $O(\log n)$ parallel time—independent of the size of the batch. As a natural extension of Reckle trees, we also introduce Reckle+ trees. Reckle+ trees provide updatable and succinct proofs for certain types of Map/Reduce computations. In this setting, a prover can commit to a memory M and produce a succinct proof for a Map/Reduce computation over a subset I of M. The proof can be efficiently updated whenever I or M changes.

We present and experimentally evaluate two applications of Reckle+ trees, *dynamic digest translation* and *updatable BLS aggregation*. In dynamic digest translation we are maintaining a proof of equivalence between Merkle digests computed with different hash functions, e.g., one with a SNARK-friendly Poseidon and the other with a SNARK-unfriendly Keccak. In updatable BLS aggregation we maintain a proof for the correct aggregation of a t-aggregate BLS key, derived from a t-subset of a Merkle-committed set of individual BLS keys. Our evaluation using Plonky2 shows that Reckle trees and Reckle+ trees have small memory footprint, significantly outperform previous approaches in terms of updates and verification time, enable applications that were not possible before due to huge costs involved (Reckle trees are up to 200 times faster), and have similar aggregation performance with previous implementations of batch proofs.

1 INTRODUCTION

A *Merkle tree* [15] is a seminal cryptographic data structure that enables a party to commit to a memory M of n slots via a succinct digest d. A third party with access to d can verify correctness of any memory slot $\mathsf{M}[i]$ via a $\log n$-sized and efficiently-computable proof π_i. Merkle trees can be used to verify untrusted storage efficiently, and have found many applications particularly in the blockchain space, such as in Ethereum state compression via Merkle Patricia Tries (MPTs) [1], stateless validation [6], zero-knowledge proofs [24] as well as in verifiable cross-chain computation [2].

Many applications that work with Merkle trees require the use of a *Merkle batch proof*. A Merkle batch proof π_I is a single proof that can be used to prove multiple memory slots $\{\mathsf{M}[i]\}_{i\in I}$ at once. When the slots are consecutive, Merkle batch proofs (also called Merkle range proofs) have logarithmic size, independent of the batch size $|I|$. In the general case of arbitrary set I though, a Merkle batch proof comprises $O(|I|\log n)$ hashes. For blockchain applications that must prove thousands of transactions at once, the lack of succinctness of Merkle batch proofs tends to become an issue.

Updatable Merkle batch proofs. One distinguishing feature of Merkle trees is their support for extremely fast updates: If a memory slot $\mathsf{M}[j]$ of the committed memory M changes, a batch proof π_I (as well as the whole Merkle tree) can be updated in logarithmic time with a simple algorithm. This is particularly useful for applications. For instance, when new Ethereum blocks are created and new memory is allocated for use by smart contracts, Ethereum nodes can update their local MPTs (which are q-ary unbalanced Merkle trees), very fast.

Unfortunately (and as mentioned before), while Merkle trees support blazingly-fast updates of batch proofs, their batch proofs are *not succinct*, i.e., their size depends on $|I|$. The motivation and initial focus of this paper is on this very issue.

Can we build succinct Merkle batch proofs that are efficiently updatable?

This question *relates* to the notion of *updatable SNARKs* (e.g.,[12]), that, to the best of our knowledge, we put forth for the first time. An updatable SNARK is a SNARK that is equipped with an additional algorithm $\pi' \leftarrow \mathsf{Update}((x,w),(x',w'),\pi)$. Algorithm Update takes as input a true public statement x along with its witness w and its verifying proof π as well as an updated true public statement x' along with the updated witness w'. It outputs a verifying proof π' for x' without running the prover algorithm from scratch and ideally in time proportional to the distance (for some definition of distance) of (x,w) and (x',w').

While we are not solving the problem in its generality, we provide constructions that handle important classes of functions, such as batch proofs and Map/Reduce-style computations over Merkle trees (See applications in Section 1.2.)

1.1 Our contribution: Reckle Trees

We introduce Reckle Trees, a new vector commitment scheme [5] that supports *updatable* and *succinct* batch proofs using RECursive

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

SNARKs* and MerKLE trees (A vector commitment is a cryptographic abstraction we use for "verifiable storage" and which can be implemented, for example, by both Reckle trees and Merkle trees.)

Reckle Trees can work under a fully-dynamic setting for batch proofs: Assume a Reckle batch proof π_I for a subset I of memory slots has been computed. Reckle Trees can support the following updates to π_I in logarithmic time: (a) change the value of element $\mathsf{M}[i]$, where $i \in I$; (b) change the value of element $\mathsf{M}[j]$, where $j \notin I$; (c) extend the index set I to $I \cup w$ so that $\mathsf{M}[w]$ is also part of the batch; (d) remove index w from I so that $\mathsf{M}[w]$ is not part of the batch; (e) remove or add an element from the memory altogether (In this case, Reckle Trees can rebalance following the same rules of standard data structures such as red-black trees or AVL trees.)

Updating a batch proof π_I in Reckle Trees is achieved through a batch-specific data structure Λ_I that stores recursively-computed SNARKs proofs. Importantly, Reckle Trees, just as Merkle trees, are naturally parallelizable. Assuming enough parallelism, any number of updates $T > 1$ can be performed in $O(\log n)$ parallel time. As we will see, our massively-parallel Reckle Trees implementation achieves up to 270× speedup over our sequential implementation. Reckle Trees also have particularly low memory requirements: While they can make statements about n leaves, their memory requirements (excluding the underlying linear-size Merkle tree) scale logarithmically with n. Finally, Reckle trees have the flexibility of not being tied to a specific SNARK: If a faster recursive SNARK implementation is introduced in the future (e.g., folding schemes [13]), Reckle trees can use the faster technology seamlessly.

Our main technical approach. Let I be the set of Merkle leaf indices for which we wish to compute the batch proof. Starting from the leaves $l_1, \ldots, l_{|I|}$ that belong to the batch I, Reckle Trees run the SNARK recursion on the respective Merkle paths $p_1, \ldots, p_{|I|}$, merging the paths whenever common ancestors of the leaves in I are encountered. While the paths are being traversed, Reckle Trees not only verify that the elements in I belong to the Merkle tree but they also compute a "batch" hash for the elements in I, eventually making this batch hash part of the public statement. The batch hash is computed via *canonical hashing*, a deterministic and secure way to represent any subset of $|I|$ leaves succinctly (see Fig. 1). While we could have used any number-theoretic accumulator (e.g., [3]) (or even the elements in the batch themselves), we choose canonical hashing to avoid encoding algebraic statements within our circuits (and to ensure our circuits' size never depends on the size and topology of the batch). If the final recursive proof verifies, that means that the batch hash corresponds to a valid subset of elements in the tree, and can be recomputed using the actual claimed batch as an input. In summary, the main difference with the Merkle tree construction is that every node v in a Reckle tree, in addition to storing a Merkle hash C_v, can also store a recursively-computed SNARK proof π_v, depending on whether any of v's descendants belongs to the batch I in question or not (For nodes that have no descendants in the batch, there is no need to store a SNARK proof.) Our approach can be easily extended to unbalanced q-ary Merkle trees (that model Ethereum MPTs) as we show in Section 3.8.

Map/Reduce proofs with Reckle+ Trees. Reckle trees, just like Merkle trees, can only be used to prove memory content, but no computation over it. However one might want to compute an updatable proof for some arbitrary computation over the Merkle leaves, e.g., counting the number of Merkle leaves v satisfying an arbitrary function $f(v) = 1$. For example, smart contracts could benefit by accessing historic chain MPT data to compute useful functions such as price volatility or BLS aggregate keys. Instead, such applications are currently enabled by the use of blockchain "oracles" that have to be blindly trusted to expose the correct output to the smart contract.

We introduce Reckle+ Trees, a natural extension of Reckle Trees that support updatable verifiable computation over Merkle leaves. With Reckle+ Trees, a prover can commit to a memory M, and provide a proof of correctness for Map/Reduce computation on any subset of M. This is technically achieved by encoding, in the recursive circuit, not only the computation of the canonical hash and the Merkle hash (as we would do in the case of batch proofs), but also the logic of the Map and the Reduce functions. The final Map/Reduce proof can be easily updated whenever the subset changes, without having to recompute it from scratch.

1.2 Applications

We use Reckle+ Trees to enable (and experimentally evaluate), for the first time, the following applications in the blockchain space.

Dynamic digest translation. Most blockchains such as Ethereum employ MPTs that use hash functions such as SHA-256 or Keccak. Unfortunately, these hash functions are particularly SNARK-unfriendly, meaning that they generate a large number of constraints when turned into circuits so that they can be used by SNARKs. Due to this, it becomes difficult and slow to prove any meaningful computation (say, to a smart contract that cannot execute the computation due to limited computational resources) over Ethereum MPT data. On the other hand, there are particularly SNARK-friendly hash functions such as Poseidon [11] that can generate up to 100× less constraints than SHA-256 or Keccak, leading to tremendous savings in prover time. Reconfiguring the whole Ethereum blockchain to use Poseidon so that we can produce faster proofs cannot work, of course—we just need to work with Keccak. Our main idea is to enable a *digest translation* service that can provide *proofs equivalence* between a Keccak-based digest and a Poseidon-based digest, i.e., that both digests are computed over the same set of leaves. These proofs of equivalence should be easily *updatable* when new blocks are generated. After we have an equivalence proof, we can then compute a SNARK proof on Poseidon-hashed data much faster.

It turns out that Reckle+ Trees can be used for digest translation. The "Map" computation is applied at the leaves as the identity function, and the "Reduce" computation is applied at the internal nodes producing both Keccak and Poseidon hashes of their children (Note that for this application we use all the leaves as the batch index set I.) As new blocks are produced, and some of the Merkle leaves change, this equivalence proof can be updated fast, always

*A recursive SNARK is a SNARK (e.g., [12]) that can call its verification algorithm within its circuit. While all SNARKs are recursive (in theory), certain SNARKs have been optimized for recursion, e.g., via the use of special curves. Our implementation is using Plonky2 [18], but our framework can use any recursive SNARK.

being ready to be consumed by a SNARK working with Poseidon-hashed data.

Updatable BLS key aggregation. Light clients [9] are used to check the "local correctness" of a block. That is, given a block header h and an alleged set of transactions, a light client would check that these transactions indeed correspond to the specific header h. However, a light client does not have the security of a full node, since it does not check the specific block header h is correct, by going back to genesis. Therefore, the block header that a light client is checking could in principle be bogus. One way to address this issue is to have a certain number of (Ethereum) signers, picked from a fixed set of validators, sign the block header (These signers can be staked and slashed in case they sign a bogus header, offering cryptoeconomic security.)

In this scenario, the light clients, instead of just receiving the header h, they would receive an aggregate signature $sig(h)$ allegedly signed by a set of at least t signers from a Merkle-committed set of validators with digest d. To verify this aggregate signature, a light client would need an aggregate key apk and a proof for the following public statement (apk, t, d)

"apk *is the aggregate BLS public key from* $\geq t$ *BLS keys derived from the set of BLS keys committed to by* d."

Reckle+ Trees can be used to produce highly parallelizable and updatable proofs for the statement (apk, t, d). Here, the "Map" function selects the leaf public key and sets its counter to 1 if the specific leaf participates in the set of signers, and the "Reduce" function multiplies the aggregate keys of its children, adding their counters accordingly. Note here that the updatability property of Reckle+ Trees becomes crucial since as new blocks are being produced, the set of signers (as well as the set of validators) could change, in which case computing the proof for the public statement (apk', t, d') could be much faster than recomputing the proof from scratch—especially when the delta between the old and new signer/validator sets is small.

1.3 Evaluation

We implement*and evaluate Reckle trees and Reckle+ trees using Plonky2 [18]. We find that Reckle trees have small memory footprint, significantly outperform previous approaches in terms of updates and verification (Reckle trees updates take 16.61s and verification is around 18ms) and perform similarly with other approaches in terms of aggregation time. In terms of applications, we estimate that Reckle+ trees can speed up digest translation up to 200× (The 200× is a figure that we had to extrapolate, since other approaches could not execute for large parameters, due to very large memory requirements.) Finally, Reckle trees' proof size is 112 KiB, independent of the batch size and the size of the vector. We also note that while there are few implementations of batch proofs that we can compare to, there are no implementations of Merkle computation proofs.

1.4 Related work

We now describe some related approaches could be used to build batch proofs and Merkle computation proofs.

*Our circuits are available at: https://github.com/Lagrange-Labs/reckle-trees

Recursive batch proofs via tree-of-proofs approach. Deng and Du [8] recently proposed an application of recursive SNARKs in computing succinct Merkle batch proofs—using a "tree of proofs" approach: Suppose one wants to compute a Merkle batch proof π_I for an index set I. Their main idea has two steps. In the first step a SNARK circuit is built for verifying a single Merkle proof. This SNARK is executed $|I|$ times, outputting a SNARK proof p_i for every index $i \in I$. Then a binary tree is built with all proofs p_i as leaves. For every node v of the binary tree a recursive SNARK is executed (outputting a proof p_v) that verifies the proofs coming from v's children. This process continues up to the root and the batch proof is defined as the final recursive proof p_r of the root. Unfortunately, the produced batch proof is not *updatable*. If an element of the tree or the batch changes, all proofs at the leaves will be affected and therefore the whole procedure must be executed from scratch, requiring computational work that is proportional to the size of the batch. In addition, in such "tree of proofs" approach, if two indices share common structure in their Merkle proofs (e.g., successive indices), the same hash computations will be repeating within the two circuits corresponding to those leaves, unnecessarily consuming computational resources.

Succinct batch proofs via vector-commitment approach. Succinct batch proofs can be computed using *vector commitments* [5, 10, 14, 20–23]. Vector commitments are typically algebraic constructions as opposed to hash-based Merkle trees. With vector commitments a batch proof for $|I|$ elements has size either optimal $O(1)$ (e.g., [21]) or logarithmic (e.g., [20]), but always independent of the batch size $|I|$. However, while vector commitments achieve optimal batch proof sizes, they face other challenges. In particular, the majority of vector commitments are not updatable: As opposed to Merkle trees, whenever a single memory slot changes, $\Omega(n)$ time is required to update all individual proofs, which can be a bottleneck for many applications. While there are some vector commitments (e.g., [20, 23]) that can update proofs in logarithmic time (while having succinct batch proofs), those suffer from increased concrete batch proof sizes, large public parameters and high aggregation and verification times (At the same time, *their batch proofs are not updatable*.) For example, a Hypeproof [20] batch proof for a thousand memory slots requires access to gigabytes of data to be generated and approximately 17 seconds to be verified.

Merkle computation proofs via Merkle-SNARK approach. Vector commitments can handle only memory content. To enable arbitrary computation over Merkle leaves one of course can building a SNARK (e.g., [12]) that verifies Merkle proofs and then performs computation via a monolithic circuit, but this is typically very expensive (Such an approach also leaves very little space for massive parallelism since proof computation can be parallelized only to the extent that the the prover of underlying SNARK can be parallelized.) For instance, computing a SNARK-based proof that verifies a thousand memory slots on a Poseidon-based 30-deep Merkle tree can take up to 20 minutes [20]—and this excludes any computation that one might wish to perform on the leaves. To make things worse, any updates to the memory M would require to pay that very cost again!

2 PRELIMINARIES

We now provide necessary notation and definitions for SNARKs, Merkle Trees and vector commitments. We will be extending the vector commitment definition to account for updatable batch proofs.

Let λ be the security parameter and $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{2\lambda}$ 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 Arguments of Knowledge [12]. 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 Π = (Setup, Prove, Verify) defined as follows:

- $\mathsf{Setup}(1^\lambda,\mathcal{R}) \to (\mathsf{pk},\mathsf{vk})$: On input security parameter λ and the binary relation $\mathcal{R}$, it 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, it 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 statement or 0 for rejecting it.

It also satisfies the following properties:

- *Completeness*: For all $(x,w) \in \mathcal{R}$, the following holds:
$$\Pr\left(\mathsf{Verify}(\mathsf{vk},x,\pi)=1 \,\middle|\, \begin{array}{c}(\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda,\mathcal{R}) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk},x,w)\end{array}\right) = 1\,.$$
- *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:
$$\Pr\left(\begin{array}{c}\mathsf{Verify}(\mathsf{vk},x,\pi)=1 \\ \wedge\ \mathcal{R}(x,w)=0\end{array}\,\middle|\, \begin{array}{c}(\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda,\mathcal{R}) \\ ((x,\pi);w) \leftarrow \mathcal{A} \parallel_{\mathcal{X}_\mathcal{A}} ((\mathsf{pk},\mathsf{vk}))\end{array}\right).$$
(The notation $((x,\pi);w) \leftarrow \mathcal{A} \parallel_{\mathcal{X}_\mathcal{A}} ((\mathsf{pk},\mathsf{vk}))$ means the following: After the adversary $\mathcal{A}$ outputs (x,π), we can run the extractor $\mathcal{X}_\mathcal{A}$ on the adversary's state to output w. The intuition is that if the adversary outputs a verifying proof, then it must know a satisfying witness that can be extracted by looking into the adversary's state.)
- *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 $n = 2^\ell$ slots. A Merkle tree [15] 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 $\mathsf{C}_v = index(v)||value(v)$ (For simplicity of notation, we use λ bits for index and λ bits for value.)
(2) If v's left child is L and v's right child is 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.

BLS signatures. BLS signatures [4] are signatures that can be easily aggregated. For BLS signatures we need a bilinear map $e : \mathbb{G}\times\mathbb{G} \to \mathbb{G}_T$ over elliptic curve groups and a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{G}$. Groups $\mathbb{G}$ and $\mathbb{G}_T$ have prime order p. The secret key is $sk \in \mathbb{Z}_p$ and the public key is $g^{sk} \in \mathbb{G}$, where g is the generator of $\mathbb{G}$. To sign a message $m \in \{0,1\}^*$ we output the signature $\mathsf{H}(m)^{sk} \in \mathbb{G}$. To verify a signature $s \in \mathbb{G}$ on message $m \in \{0,1\}^*$ given public key $PK \in \mathbb{G}$, the verifier checks whether $e(s,g) = e(\mathsf{H}(m),PK)$. Given public keys $\{g^{sk_i}\}_{i=1,\ldots,t}$ one can compute an aggregate public key $\prod_{i=1}^t g^{sk_i}$. To verify a t-multisignature S on a message m given aggregate public key APK the verifier checks whether $e(S,g) = e(\mathsf{H}(m),APK)$. If APK is provably the product of t individual BLS public keys PK, we can be assured that t parties (in particular the owners of the respective secret keys) signed the message.

Vector Commitments (VCs). A vector commitment is a set of algorithms that allow one to commit to a vector of n slots so that later one can open the commitment to values of individual memory slots. One of the most popular implementations of vector commitments are Merkle trees [15]. Vector commitments typically enable more advanced properties than Merkle trees, such as batch proofs (succinct proofs of multiple values). We formalize VCs below, similar to Catalano and Fiore [5]. We extend the definition with batch proofs updatability, to capture the new properties introduced byReckle Trees.

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

- $\overline{\mathsf{Gen}(1^\lambda,n) \to \mathsf{pp}}$: Outputs public parameters pp.
- $\overline{\mathsf{Com}_{\mathsf{pp}}(\mathbf{a}) \to \mathsf{C}}$: Outputs digest C.
- $\overline{\mathsf{Open}_{\mathsf{pp}}(i,\mathbf{a}) \to \pi_i}$: Outputs proof π_i.
- $\overline{\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{a}) \to (\pi_0,\ldots,\pi_{n-1})}$: Outputs all proofs π_i.
- $\overline{\mathsf{Agg}_{\mathsf{pp}}((a_i,\pi_i)_{i\in I}) \to (\pi_I,\Lambda_I)}$: Outputs a batch proof π_I and a batch-proof data structure Λ_I.
- $\overline{\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C},(a_i)_{i\in I},\pi_I) \to \{0,1\}}$: Verifies batch proof π_I against C.
- $\overline{\mathsf{UpdDig}_{\mathsf{pp}}(u,\delta,\mathsf{C},\mathsf{aux}) \to \mathsf{C}'}$: Updates digest C to C′ to reflect position u changing by δ given auxiliary input aux.
- $\overline{\mathsf{UpdProof}_{\mathsf{pp}}(u,\delta,\pi_i,\mathsf{aux}) \to \pi_i'}$: Updates proof π_i to π_i' to reflect position u changing by δ given auxiliary input aux.
- $\overline{\mathsf{UpdBatchProof}_{\mathsf{pp}}(u,\delta,\pi_I,\Lambda_I,\mathsf{aux}) \to (\pi_I',\Lambda_I')}$: Updates batch proof π_I to π_I' and the batch-proof data structure Λ_I to Λ_I' to reflect position u changing by δ given auxiliary input aux.
- $\overline{\mathsf{UpdAllProofs}_{\mathsf{pp}}(u,\delta,\pi_0,\ldots,\pi_{n-1}) \to (\pi_0',\ldots,\pi_{n-1}')}$: Updates all proofs π_i to π_i' to reflect position u changing by δ.

We define VC correctness and soundness in Definitions 2.3 and 2.4, respectively.

Definition 2.3 (VC Correctness). A VC scheme is correct, if for all $\lambda \in \mathbb{N}$ and $n = \text{poly}(\lambda)$, for all $\text{pp} \leftarrow \text{Gen}(1^\lambda, n)$, for all vectors $\mathbf{a} = [a_0, \ldots, a_{n-1}]$, if $\text{C} = \text{Com}_{\text{pp}}(\boldsymbol{a})$, $\pi_i = \text{Open}_{\text{pp}}(i, \boldsymbol{a})$, $\forall i \in [0, n)$ (or from $\text{OpenAll}_{\text{pp}}(\mathbf{a})$), and $\pi_I = \text{Agg}_{\text{pp}}((a_i, \pi_i)_{i \in I})$, $\forall I \subseteq [n]$ 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 $\text{UpdDig}_{\text{pp}}$, π'_i proofs obtained via calls to $\text{UpdProof}_{\text{pp}}$ or $\text{UpdAllProofs}_{\text{pp}}$ for all i, and π'_I are proofs obtained via calls to $\text{UpdBatchProof}_{\text{pp}}$ for all subsets I then:

(1) $\Pr[1 \leftarrow \text{Ver}_{\text{pp}}(\text{C}', \{i\}, a'_i, \pi'_i)] = 1, \forall i \in [n]$.
(2) $\Pr[1 \leftarrow \text{Ver}_{\text{pp}}(\text{C}', I, (a'_i)_{i \in I}, \pi'_I)] = 1, \forall I \subseteq [n]$.

At a high-level, correctness ensures 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 $\mathcal{A}$,

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

At a high level, soundness ensures 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 C.

3 RECKLE TREES

In this section, we present Reckle Trees, a vector commitment 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. Computing the commitment C of vector $\mathbf{a} = [a_0, \ldots, a_{n-1}]$ in Reckle Trees is straightforward: Just compute the Merkle tree digest of **a** as we described in Section 2 (For our basic implementation we use the Poseidon [11] hash function.) Similarly, the proof of opening for an index i in the vector is the Merkle membership proof of leaf a_i. Reckle Trees have a new algorithm to aggregate Merkle proofs and compute a batch proof, using recursive SNARKs. Like we previously said, our algorithm outputs batch proofs that can be updated in logarithmic time.

We now present the algorithm to aggregate proofs. An important component of our algorithm is the notion of *canonical hashing* Section 3.1.

3.1 Canonical hashing

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 $d(v, I) := index(v)||value(v) = \text{C}_v$, otherwise $d(v, I) := 0$.
(2) If v has left child L and right child R, then

$$d(v, I) := \text{H}(d(L, I)||d(R, I))\,,$$

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.

In Fig. 1, we show a Merkle tree of 16 leaves. At every node we show with blue the canonical hash of that node with respect to the subset $\{2, 4, 5, 15\}$. For the nodes with no blue hash, their canonical hash is 0.

3.2 Recursive circuit

A batch proof is a single short proof that simultaneously proves that a specific subset I of elements belongs in the vector. In our scheme, we implement the batch proof using recursion. In particular, our precise circuit verifies, recursively, that the following NP statement (C, d) is true:

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

Note that if we have a proof for the statement above, then we can easily prove that a specific subset of elements $\{a_i\}_{i \in I}$ belongs to the Merkle tree by just locally recomputing the root canonical digest. In Fig. 2 we present the recursive circuit $\mathcal{B}$ for the statement above. We indicate precisely what the public input of the circuit is and what the (private) witness is. Note that because the circuit is recursive (calling itself), the public input *must contain* a verification key that will be used by the verification call inside the circuit, otherwise, the prover could potentially use an arbitrary verification key (The verification key cannot be hardcoded either since it leads to circularity issues.)

We note that circuit $\mathcal{B}$ works for arbitrary Merkle trees, both balanced and unbalanced.

3.3 Batch proof

We now show how to aggregate individual Merkle proofs π_i for an arbitrary set of indices $i \in I$ so that to compute the batch proof. The first thing that the prover needs to do is to compute the witnesses that are required to run the SNARK proof. To do that, the prover performs the following.

(1) The prover computes the tree T_I formed by proofs $\{\pi_i\}_{i \in I}$. For each node v of T_I we store the respective value C_v that that can be found in or computed from the proofs $\{\pi_i\}_{i \in I}$. For example, in Fig. 1, where $I = \{2, 4, 5, 15\}$, the nodes of tree T_I are drawn as rectangles.
(2) Let now $T'_I \subset T_I$ be the subtree of T_I that contains just the nodes on the paths from I to the root. The prover computes the canonical hash $d(v, I)$ for all nodes $v \in T'_I$ *with respect* to I. By the definition of the canonical hash, all nodes not included T'_I will have canonical hash equal to 0. For example, in Fig. 1 we indicate the canonical hash with blue. All nodes without a blue hash have canonical hash equal to 0.

After we have computed the witness, we proceed with computing the recursive SNARK proofs that eventually will output the batch proof. We first produce the public parameters by running the setup of the SNARK $(\text{pk}_{\mathcal{B}}, \text{vk}_{\mathcal{B}}) \leftarrow \text{Setup}(1^\lambda, \mathcal{B})$ for the circuit $\mathcal{B}$. Consider now the set of indices I for which we are calculating the

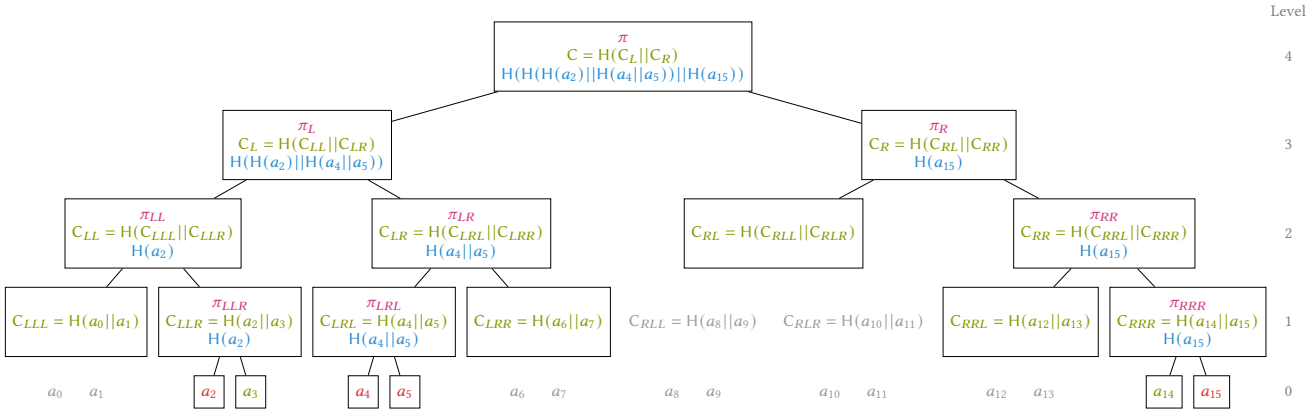


Figure 1: Batch proof data structure. Consider a vector of size 8 and subset $I = \{2, 4, 5, 15\}$. Recall that every leaf stores *both* index and value. Every node v in the path from root to leaves of I, stores the Merkle hash, canonical hash with respect to I, and the recursive SNARK proof, and every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

Circuit $\mathcal{B}$

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

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

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 $\mathsf{Verify}(\mathsf{vk}, (\mathsf{vk}, \mathsf{C}_L, d_L), \pi_L)$ or $\mathsf{C}_L = d_L$;
(4) If $d_R \neq 0$ check $\mathsf{Verify}(\mathsf{vk}, (\mathsf{vk}, \mathsf{C}_R, d_R), \pi_R)$ or $\mathsf{C}_R = d_R$;
(5) Return true;

Figure 2: Batch proof recursive circuit for Reckle Trees.

batch proof and let T'_I be the subtree as defined above. Let V_l be the nodes of T'_I at level $l = 1, \ldots, \ell$. To compute the batch proof, we follow the procedure below.

for all levels $l = 1, \ldots, \ell$, **for** all nodes $v \in V_l$

- Let L be v's left child and R be v's right child in T_I;
- Set π_v to be the output of

$$\mathsf{Prove}(\mathsf{pk}_{\mathcal{B}}, (\mathsf{C}_v, d_v, \mathsf{vk}_{\mathcal{B}}), ((\mathsf{C}_L, d_L, \pi_L), (\mathsf{C}_R, d_R, \pi_R)) \quad (1)$$

The final batch proof for index set I will be π_r, where r is the root of T_I. Note that π_r proves the statement $(\mathsf{C}, d(r, I))$ is true, where $(\mathsf{C}, d(r, I))$ is defined in Section 3.2.

Computing the batch proof has parallel complexity ℓ, independent of $|I|$. This is because we perform the canonical hashing computation inside the Merkle verification.

How to ensure Merkle leaves are used as witness. We note here that the circuit in Figure 2 does not necessarily consider the leaves of the Merkle tree. For example, given a Merkle tree of 10 levels, we can compute a valid batch proof using the circuit in Figure 2 even if we discard some of the levels of the tree and just using only, say, the first three levels. Note that this is not an attack since what we prove is consistent with the public statement we put forth. If we wish our proof to always consider the leaves of the tree we need to assume there is a function $\mathsf{leaf}()$ that distinguishes between a Merkle leaf and a Merkle hash. For example, this function might be checking whether the node in question has a particular format, e.g., a bit indicating whether it is a leaf or not (For instance, in Ethereum MPTs, leaves always have a particular format.) In presence of the $\mathsf{leaf}()$ function we can always force our prover to consider leaves by changing lines (3) and (4) in Figure 2 as follows.

If $d_L \neq 0$ check $\mathsf{Verify}(\mathsf{vk}, (\mathsf{vk}, \mathsf{C}_L, d_L), \pi_L) \vee \mathsf{C}_L = d_L \wedge \mathsf{leaf}(\mathsf{C}_L)$;

and

If $d_R \neq 0$ check $\mathsf{Verify}(\mathsf{vk}, (\mathsf{vk}, \mathsf{C}_R, d_R), \pi_R) \vee \mathsf{C}_R = d_R \wedge \mathsf{leaf}(\mathsf{C}_R)$;

Note that ensuring the leaves are used as witness will be needed in our applications section, to compute some function on the leaves and prove the result of this computation (See Section 4.1.)

3.4 Updating the batch proof

In order to update a batch proof (of a subset I), when some leaves are changing, we use the *batch data structure*. The batch data structure for a subset I, Λ_I, consists of all the Merkle hash values, canonical hash values, and recursive SNARK proofs (as computed before) along the nodes (and their siblings) from the leaf nodes in I to the root, as depicted in Fig. 1). It is now natural to maintain a dynamic batch proof as follows: Whenever a leaf value a_j changes to a'_j, we distinguish two cases:

(1) If the leaf's index j belongs to I, then all canonical hashes, Merkle hashes and SNARK proofs of all nodes from leaf j to the root must be updated;
(2) If the leaf's index j does not belong to I, then at least one Merkle hash of a node v' of the batch data structure will change that will cause other Merkle hashes and related SNARK proofs to change, *along the path from v' to the root.*

A similar approach can be used when we are changing the size of the batch, either by adding or removing elements. In both cases, all updates can be performed in $O(\log n)$ time since the height of the batch data structure is $\log n$. Note that the update time is independent of $|I|$, as opposed to previous approaches where the

$\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}$: Let pp contain the following:

- The size of the vector n.
- A collision-resistant hash function H.
- $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{B})$, where $\mathcal{B}$ is the circuit in Fig. 2.

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

$\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}) \to \pi_i$: Return the Merkle proof for element a_i.

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

$\mathsf{Agg}_{\mathsf{pp}}((a_i, \pi_i)_{i \in I}) \to (\pi_I, \Lambda_I)$:

- Using $\{\pi_i\}_{i \in I}$ and I, compute the Merkle hash C_v and the canonical digest $d(v, I)$ for every node in $v \in T_I$, where T_I is defined in Section 3.3.
- Let T'_I be the subtree of T_I as defined in Section 3.3.
- For all nodes v in T'_I compute π_v as in Eq. (1).
- Return π_I as π_r where r is the root of T_I and Λ_I as T_I along with the values

$$\{(\mathsf{C}_v, d(v, I), \pi_v)\}_{v \in T_I}\,.$$

$\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, (a_i)_{i \in I}, \pi_I) \to \{0, 1\}$:

- Using $(a_i)_{i \in I}$, compute d_I as in Section 3.1.
- Return $\mathsf{Verify}(\mathsf{vk}_{\mathcal{B}}, (\mathsf{vk}_{\mathcal{B}}, \mathsf{C}, d_I), \pi_I)$.

$\mathsf{UpdDig}_{\mathsf{pp}}(u, \delta, \mathsf{C}, \mathsf{aux}) \to \mathsf{C}'$:

- Parse aux as π_u.
- Return the Merkle root after updating to $a_u + \delta$.

$\mathsf{UpdProof}_{\mathsf{pp}}(u, \delta, \pi_i, \mathsf{aux}) \to \pi'_i$:

- Parse aux as π_u.
- Recompute the Merkle path after updating to $a_u + \delta$.
- Update affected portions of π_i.

$\mathsf{UpdBatchProof}_{\mathsf{pp}}(u, \delta, \pi_I, \Lambda_I, \mathsf{aux}) \to (\pi'_I, \Lambda'_I)$:

- Parse aux as π_u.
- Recompute the Merkle path after updating to $a_u + \delta$.
- For every node v in Λ_I affected by update do the following.
  - Update the Merkle hash value.
  - Recompute the canonical hash $d(v, I)$ if necessary.
  - Recompute the SNARK proof π_v as in Eq. (1).
  - Return the updated π_I and Λ_I.

$\mathsf{UpdAllProofs}_{\mathsf{pp}}(u, \delta, \pi_0, \ldots, \pi_{n-1}) \to (\pi'_0, \ldots, \pi'_{n-1})$:

- Parse aux as π_u.
- Recompute the Merkle path after updating to $a_u + \delta$.

Figure 3: Algorithms for Reckle Trees. We use a collision-resistant hash function H and a SNARK (Setup, Prove, Verify) from Definition 2.1 as black boxes.

update requires work proportional to $|I|$. In Fig. 3 we present the detailed implementation of theReckle Trees algorithms.

3.5 Security proof

In this section we prove that theReckle Trees VC scheme, whose detailed implementation is described in Fig. 3 satisfies soundness according to Definition 2.4. Correctness, per Definition 2.3, follows easily by inspection.

Theorem 3.1 (Soundness of Reckle Trees). *Reckle Trees from Fig. 3 are sound per Definition 2.4 assuming collision resistance of the hash function* H *and knowledge soundness of the underlying SNARK* (Setup, Prove, Verify).

Proof. Deferred to Appendix A. □

3.6 Optimized Reckle Trees circuit

While the circuit of Fig. 2 is simple, there are two issues that could create significant overhead during implementation.

(1) Recall that the circuit in Fig. 2 takes as input SNARK proofs π_R and π_L that need to be verified. When using a SNARK whose proof size depends on the size of computation (such as in Plonky2 [18] that we use in our implementation), the circuit size changes during each recursive call. One could address this issue by having an upper bound on the input of the circuit, but this could create additional overhead.
(2) For a tree of n leaves, the circuit in Fig. 2 needs to be called $n-1$ times, equal to the number of internal nodes, leading to $n-1$ recursive calls. However note that there is no need to recurse on the leaf nodes—this is the base case of the recursion. Given recursion is the most expensive operation, we would like to reduce the number of recursive calls.

Our approach to address both problems above is to consider $\log n$ different circuits, one per each lever of the tree. Then the size of the circuit is fixed at each level and we can ensure the leaf circuit is quite simple and does not contain any recursive calls (The number of recursive calls will be $n/2-1$.) We show the new circuits in Fig. 4. Note that the circuit $\mathcal{B}_i$ always hardcodes the key vk_{i-1} of circuit $\mathcal{B}_{i-1}$. The new circuit is shown in Fig. 4.

3.7 Reducing recursive calls with bucketing

Let r be the concrete cost of recursion and h be the concrete cost of hashing. The approximate concrete parallel complexity (both aggregation and update) of Reckle Trees is

$$f = (\ell - 1) \cdot 2r + \ell \cdot 2h$$

since at every level of the tree we require two recursive calls and two hashes, except for the last level where we just do two hashes (For this simplistic analysis we do not account for conditionals.)

In our implementation we have noticed that the overhead of recursion is the dominant cost in our circuits, in terms of number of constraints in Plonky2 [18].

To reduce the number of recursive call as much as possible, we bucket $p = 2^q$ leaves together into a monolithic circuit. In this way we have the following concrete parallel complexity

$$g = (\ell - q) \cdot (2r + 2h) + (2^q - 1) \cdot 2h\,.$$

This is because for the last q levels of the tree, the bucketing approach will have to compute more hashes ($2^q - 1$, as opposed to q) since we are using a monolithic circuit.

We want to find the q such that the difference between these parallel complexities be maximum, i.e., we want to maximize the

Circuit $\mathcal{B}_0$

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;

Circuit $\mathcal{B}_i$ ($i = 1, \ldots, \ell - 1$)

Public input: (C, d)

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

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 $\mathsf{Verify}(\mathsf{vk}_{i-1}, (\mathsf{C}_L, d_L), \pi_L)$;
(4) If $d_R \neq 0$ check $\mathsf{Verify}(\mathsf{vk}_{i-1}, (\mathsf{C}_R, d_R), \pi_R)$;
(5) Return true;

Figure 4: Optimized batch proof circuits for Reckle Trees.

Circuit $\mathcal{Q}_k$ ($k = 0, \ldots, q$)

Public input: $\mathsf{C}, d, \mathcal{V}$

Witness 1: $(\mathsf{C}_1, d_1), \ldots, (\mathsf{C}_q, d_q)$

Witness 2: $(\mathsf{C}_{x_1}, d_{x_1}), \ldots, (\mathsf{C}_{x_k}, d_{x_k})$

Witness 3: $(\mathsf{vk}_{x_1}, \pi_{x_1}), \ldots, (\mathsf{vk}_{x_k}, \pi_{x_k})$

Computation:

(1) Check $\mathsf{vk}_{x_1}, \ldots, \mathsf{vk}_{x_k}$ are in $\mathcal{V}$;
(2) Check
$$(\mathsf{C}_{x_1}, d_{x_1}), \ldots, (\mathsf{C}_{x_k}, d_{x_k})$$
have all non-zero d_i's and that all are a subset of
$$(\mathsf{C}_1, d_1), \ldots, (\mathsf{C}_q, d_q)\,;$$
(3) Check $\mathsf{C} = \mathsf{H}(C_1||C_2||\ldots||C_q)$;
(4) Check $d = \mathsf{H}(d_{x_1}||d_{x_2}||\ldots||d_{x_k})$;
(5) Check $\mathsf{Verify}(\mathsf{vk}_1, (\mathsf{C}_{x_1}, d_{x_1}), \pi_{x_1})$ or $\mathsf{C}_{x_1} = d_{x_1}$;
(6) …
(7) Check $\mathsf{Verify}(\mathsf{vk}_{x_k}, (\mathsf{C}_{x_k}, d_{x_k}), \pi_k)$ or $\mathsf{C}_{x_k} = d_{x_k}$;
(8) Return true;

Figure 5: Circuit for batch proof in q-ary tree Reckle tree. Note that the circuit is parameterized for $k = 0, \ldots, q$.

function $|f - g|$ with respect to q. For an implementation, where $r = 450$ ms and $h = 15$ ms we have found that the optimal $q = 5.48$ (for $\ell = 27$).

3.8 q-ary Reckle Trees

We can extend Reckle Trees to q-ary trees. q-ary Reckle trees model batch proof computation for MPT trees (Merkle Patricia Tries) that are used in Ethereum and other blochchain projects. Every node in a q-ary tree has degree at most q (In Ethereum MPT, $q = 16$.) As with Merkle trees, we define the Merkle hash C_v of a node in a q-ary tree as

$$\mathsf{H}(\mathsf{C}_1||\ldots||\mathsf{C}_q)\,,$$

where C_i is the Merkle hash of its i-th child. If some child is missing (and therefore the degree is less than q), we set the Merkle hash of this child to be *null* (We can also define the Merkle hash of a node to be the hash of the sorted list that contains the hashes of only those children that are present, but we avoid this in sake of simplicity.) The canonical hash d_v of a node v with respect to a subset of leaves I is naturally defined as the hash of the the sorted list of children that are ancestors of I. The circuit $\mathcal{Q}_k$ is given in Fig. 5.

Note that $\mathcal{Q}_k$ is parameterized by $k = 0, \ldots, q$, leading to a total of $q + 1$ circuits. We do that to represent a node with $0, 1, \ldots, q$ active children, with respect to the batch I. This allows to avoid executing q recursive calls even when there are fewer than q active children. Note also that the circuit takes also as input the set of all verification key $\mathcal{V} = \{\mathsf{vk}_0, \ldots, \mathsf{vk}_q\}$ to ensure that that prover always uses as verification key in the recursive call a key from a correct, predefined set of keys (We can implement that efficiently by providing a Merkle digest $d(\mathcal{V})$ as public input along with Merkle proofs for the verification keys.)

In $\mathcal{Q}_k$, **Witness 1** contains the Merkle hashes and canonical hashes of v's children, **Witness 2** is the alleged subset of **Witness 1** consisting of active nodes with respect to the batch, and **Witness 3** contains the SNARK proofs and verification keys that will used in the recursive calls. By applying circuit $\mathcal{Q}_k$ from the leaves of the batch I to the root, deciding which k to use based on the number of active children that the specific node has, we can prove the following statement.

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

Recall that as with circuit in Fig. 2, circuit $\mathcal{Q}_k$ takes as input SNARK proofs that need to be verified. Since our underlying SNARK is Plonky2 [18], these proofs will have different sizes. As we said, one could address this issue either by having an upper bound on the input or, in the case where the q-ary tree has a fixed shape (e.g., a full q-ary tree) by hardcoding the respective verification keys, as we did in Fig. 4 for the full binary tree. We leave the implementation of circuits for q-ary trees as future work.

4 RECKLE+ TREES FOR VERIFIABLE MAP/REDUCE

In this section we present Reckle+ Trees, an extension of Reckle Trees can be use to prove the correctness of Map/Reduce-style [7] computations on data committed to by a Merkle tree. Reckle+ Trees provide Map/Reduce proofs that are easily updatable and can be computed in parallel. Let $\mathcal{D}$ and $\mathcal{R}$ denote the domain of the input and output of the computation, respectively. We consider the following abstraction for Map/Reduce:

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

We remark that the Reduce operation can also take just a single input. Assume we want to execute the Map/Reduce computation on a subset $I \subseteq [n]$ of the memory slots which have d as their

Circuit $\mathcal{M}$

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

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

Computation:

(1) Check $\mathsf{C} = \mathsf{H}(\mathsf{C}_L||\mathsf{C}_R)$;
(2) If $d_R \cdot d_L \neq 0$ check ($d = \mathsf{H}(d_L||d_R)$ and out = $\text{Reduce}(\text{out}_L, \text{out}_R)$) else check ($d = d_L + d_R$ and out = $\text{out}_L + \text{out}_R$);
(3) If $d_L \neq 0$ check $\mathsf{Verify}(\mathsf{vk}, (\mathsf{vk}, \mathsf{C}_L, d_L), \pi_L)$ or ($\mathsf{C}_L = d_L$ and $\text{out}_L = \mathsf{Map}(\mathsf{C}_L)$);
(4) If $d_R \neq 0$ check $\mathsf{Verify}(\mathsf{vk}, (\mathsf{vk}, \mathsf{C}_R, d_R), \pi_R)$ or ($\mathsf{C}_R = d_R$ and $\text{out}_R = \mathsf{Map}(\mathsf{C}_R)$);
(5) Return true;

Figure 6: Map/Reduce circuit for Reckle+ Trees.

Circuit $\mathcal{D}_0$

Public input: $(\mathsf{C}_K, \mathsf{C}_P)$

Witness: $(\mathsf{C}_{KL}, \mathsf{C}_{PL})$, $(\mathsf{C}_{PR}, \mathsf{C}_{KR})$

Computation:

(1) Check $\mathsf{C}_K = \mathsf{H}_K(\mathsf{C}_{KL}||\mathsf{C}_{KR})$;
(2) Check $\mathsf{C}_P = \mathsf{H}_P(\mathsf{C}_{PL}||\mathsf{C}_{PR})$;
(3) Return true;

Circuit $\mathcal{D}_i$ ($i = 1, \ldots, \ell - 1$)

Public input: $(\mathsf{C}_K, \mathsf{C}_P)$

Witness: $(\mathsf{C}_{KL}, \mathsf{C}_{PL}, \pi_L)$, $(\mathsf{C}_{PR}, \mathsf{C}_{KR}, \pi_R)$

Computation:

(1) Check $\mathsf{C}_K = \mathsf{H}_K(\mathsf{C}_{KL}||\mathsf{C}_{KR})$;
(2) Check $\mathsf{C}_P = \mathsf{H}_P(\mathsf{C}_{PL}||\mathsf{C}_{PR})$;
(3) Check $\mathsf{Verify}(\mathsf{vk}_i, (\mathsf{C}_{KL}, \mathsf{C}_{PL}), \pi_L)$;
(4) Check $\mathsf{Verify}(\mathsf{vk}_i, (\mathsf{C}_{KR}, \mathsf{C}_{KL}), \pi_L)$;
(5) Return true;

Figure 7: Optimized digest translation circuit.

canonical digest. We present the recursive algorithm that checks the validity of the following NP statement:

"out *is the output of the Map/Reduce computation on some set of leaves which (i) have* d *as their root canonical digest; (ii) belong to some Merkle tree whose Merkle root is* C."

In Fig. 6, we present the circuit $\mathcal{M}$ that verifies the correctness of the Map/Reduce computation. In the following, we present two applications of Reckle+ Trees. We do not give the optimized circuit for simplicity, it can easily be transformed into $\log n$ circuits in the same way that $\mathcal{B}$ (Fig. 2) was transformed into $\mathcal{B}_0$ and $\mathcal{B}_i$ (Fig. 4).

4.1 Applications

Reckle+ Trees can enable powerful applications by allowing us to prove the correctness of Map/Reduce computation over large amounts of dynamic on-chain state data. Reckle Trees are generalizable to the Merkle-Patricia Tries used by popular blockchains, such as Ethereum, to store smart contract states. We now describe

Circuit $\mathcal{A}_0$

Public input: $(\mathsf{C}, \mathsf{apk}, cnt)$

Witness: $(\mathsf{C}_L, \mathsf{apk}_L, cnt_L)$, $(\mathsf{C}_R, \mathsf{apk}_R, cnt_R)$

Computation:

(1) Check $\mathsf{C} = \mathsf{H}(\mathsf{C}_L||\mathsf{C}_R)$;
(2) Check $\mathsf{apk} = \mathsf{apk}_L \cdot \mathsf{apk}_R$;
(3) Check $cnt = cnt_L + cnt_R$;
(4) If $\mathsf{apk}_L \neq 1_{\mathbb{G}}$ check $\mathsf{C}_L = \mathsf{apk}_L$ and $cnt_L = 1$ else check $cnt_L = 0$;
(5) If $\mathsf{apk}_R \neq 1_{\mathbb{G}}$ check $\mathsf{C}_R = \mathsf{apk}_R$ and $cnt_R = 1$ else check $cnt_R = 0$;
(6) Return true;

Circuit $\mathcal{A}_i$ ($i = 1 \ldots, \ell - 1$)

Public input: $(\mathsf{C}, \mathsf{apk}, cnt)$

Witness: $(\mathsf{C}_L, \mathsf{apk}_L, cnt_L, \pi_L)$, $(\mathsf{C}_R, \mathsf{apk}_R, cnt_R, \pi_R)$

Computation:

(1) Check $\mathsf{C} = \mathsf{H}(\mathsf{C}_L||\mathsf{C}_R)$;
(2) Check $\mathsf{apk} = \mathsf{apk}_L \cdot \mathsf{apk}_R$;
(3) Check $cnt = cnt_L + cnt_R$;
(4) If $\mathsf{apk}_L \neq 1_{\mathbb{G}}$ check $\mathsf{Verify}(\mathsf{vk}_i, (\mathsf{C}_L, \mathsf{apk}_L, cnt_L), \pi_L)$ else check $cnt_L = 0$;
(5) If $\mathsf{apk}_R \neq 1_{\mathbb{G}}$ check $\mathsf{Verify}(\mathsf{vk}_i, (\mathsf{C}_R, \mathsf{apk}_R, cnt_R), \pi_R)$ else check $cnt_R = 0$;
(6) Return true;

Figure 8: Optimized BLS aggregation circuit.

in detail two such applications, digest translation and BLS key aggregation, which we later fully evaluate in Section 5.

Dynamic digest translation. As we mentioned in the introduction, in digest translation we wish to compute a cryptographic proof for the following public statement $(\mathsf{C}_K, \mathsf{C}_P)$.

"$(\mathsf{C}_K, \mathsf{C}_P)$ *is the pair of Keccak/Poseidon Merkle digests on some same set of leaves.*"

This is useful since it allows us to extensively work with SNARK-friendly hashes (such as Poseidon) to compute SNARK proofs, while still being able to verify such Poseidon-based proofs (over dynamic data) against legacy SNARK-unfriendly hashes, such as SHA-256 or Keccak. All we have to do is to attach the proof of equivalence $(\mathsf{C}_K, \mathsf{C}_P)$ for the statement above.

In particular, since digest translation is an application of our general Map/Reduce framework, we can instantiate the Map/Reduce functions as follows:

- Map: It is the identity function. It takes as input a leaf and outputs the same leaf.
- Reduce: Takes as input two children and outputs the Poseidon hash of these children (We assume the initial Merkle tree is built with the Keccak digest.)

Note that while we describe digest translation for Keccak and Poseidon, we have the flexibility to implement it for an arbitrary pair of hash functions. Finally, note that since for digest translation our "batch" is the set of all the leaves, we do not need to compute

the canonical digest. The optimized circuit for digest translation is shown in Fig. 7.

Alternative ways to implement digest translation. While we will be using Reckle+ Trees to implement digest translation, we want to emphasize that are other approaches to do so. One way to do it is to build a monolithic circuit that builds both a Poseidon and a Keccak tree. Another 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 values 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. At the same time, updating the proofs cannot be done with a circuit that is proportional to the number of updates, since the shape of circuit depends on which leaves are updates (Anticipating all possible updates by computing different verification keys will lead to an exponential number of keys.) Due to the exponential issue of the second approach, we use the monolithic approach as the baseline in our implementation. As we will see, with the monolithic approach, we run into significant memory issues quite fast.

Updatable BLS key aggregation. Consider the following setting: BLS public keys of n validators are stored in the memory of a smart contract. 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 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 initial set changes even if the subset stays the same.

Reckle+ Trees enable us to prove that an alleged aggregate BLS public key apk is the product of a subset of t individual BLS keys from a set of BLS keys stored at the leaves of a Merkle tree whose digest is d. At the same time, this proof will be updatable in case this set changes. For the BLS aggregation application, we define the Map/Reduce functions as follows:

- Map: Takes a public key g^{sk} as input, and returns $(1, g^{\mathsf{sk}})$, if sk has signed the message, else $(0, 1_{\mathbb{G}})$, where $1_{\mathbb{G}}$ is the identity of group $\mathbb{G}$.
- Reduce: Takes two elements $(cnt_L, g^{\mathsf{sk}_L})$ and $(cnt_R, g^{\mathsf{sk}_R})$, and returns $(cnt_L + cnt_R, g^{\mathsf{sk}_L} \times g^{\mathsf{sk}_R})$.

The circuit for BLS key aggregation is shown in Fig. 8 and the Reckle+ Trees data structure for BLS key aggregation is shown in Fig. 9. Note that, just like in digest translation, we do not need to comptute canonical hashing for the BLS key aggregation.

Other applications. Our Map/Reduce framework can be 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 new blocks are added to the blockchain. The natural updatability of Reckle Trees allows it to be extended to use-cases where a proof must be generated across tens of thousands of consecutive blocks of historical data.

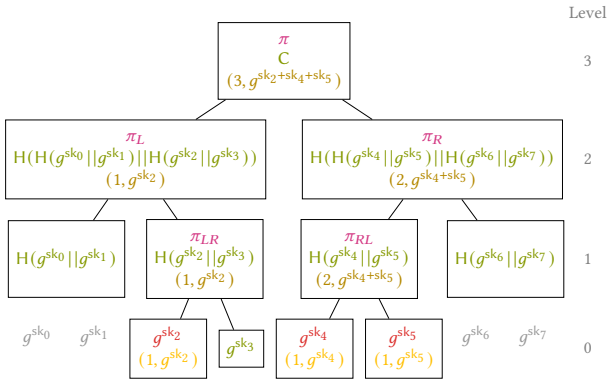


Figure 9: Reckle+ Trees data structure to compute the aggregated BLS public key and cardinality of the attestor set. Consider a vector of size 8 and attestor subset $I = \{2, 4, 5\}$. Every node v in the path from root to leaves of I, stores the Merkle hash, the recursive SNARK proof, and results of Map or Reduce operation, and every sibling of v stores just the Merkle hash.

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 scratch*, every 12 seconds. With Reckle Trees, 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.

5 EVALUATION

In this section, we evaluate the performance of Reckle Trees and the applications enabled by Reckle+ Trees. Recall that Reckle trees are naturally parallelizable regardless of the underlying proof system used to compute the recursive SNARK proofs. To realize the benefits of construction, we design and implement a large scale distributed system for distributed proof generation.

5.1 Distributed proof generation

In this subsection, we describe the architecture (Fig. 10) of our distributed system, designed to enable efficient proof generation.

At its core, our distributed system is driven by real-time events, which enables seamless addition of computation resources without degrading the performance. Logically, our system has five main

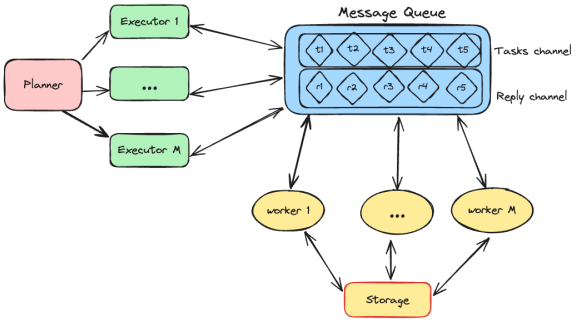


Figure 10: Distributed system architecture.

components: Planner, Executor, Worker, Message Queue, and Storage. The Planner is responsible for dividing a task into subtasks and generating a computational graph structure that describes the dependencies between the subtasks. Using the computational graph from the Planner, the Executor schedules the subtask for execution by injecting them in the Message Queue and retrieving results of the completed subtasks. The Workers are responsible for fetching subtasks from the message queue, completing the subtasks, and placing the results in the message queue.

Our implementation is in Rust language and we use Redis server for messaging. We deployed our distributed system on Kubernetes cluster inside AWS datacenter. Our system also heavily relies on AWS S3 storage to store intermediate job results. We use AWS c7i.8xlarge EC2 instances (32vCPU, 64 GiB memory) as worker nodes.

5.2 Batch updates and aggregation

In this subsection, we present the performance of aggregation, batch update, and batch proof verification. Specifically, we show that batch updates in our construction is 11× to 15× faster than baseline, and that our our distributed systems can achieve up to 270× performance improvement over the sequential implementation of Reckle Trees.

Recall that our construction extends Merkle trees. Thus, we compare the performance of our batch operation with prior known Merkle proof aggregation using SNARKs [17] and Inner-product arguments based aggregation in Hyperproofs [20], which improves upon the Merkle SNARKs aggregation techniques.

Our implementation is in Rust, and we use Plonky2 [18] for our batch proof construction. Thus, a field element corresponds to a 64-bit value from the Goldilocks field. We run our all experiments on Amazon EC2 c7i.8xlarge instance. Both our implementation and the baselines make full use of the parallelism offered by the underlying framework in all our experiments. However, our distributed implementation additional uses the natural parallelism offered by our construction. The Merkle tree in our construction and the baseline uses the Poseidon hash function [11].

Experimental setup. We set the vector size to $n = 2^{27}$ and study the performance of our scheme for varying batch sizes $k = \{2^2, 2^4, \ldots, 2^{12}\}$. In each run, we randomly generate a Merkle tree and select a random set of leaves to batch/update.

For baseline experiments, we use the following:

(1) Merkle proof aggregation using Groth16 SNARKs [12, 17] by Ozdemir *et al.* Specifically, we use the fork of the Rust implementation that was used in Hyperproofs to benchmark Merkle SNARKs [16, 17, 20].
(2) Inner-product arguments based aggregation in Hyperproofs: We use the golang based implementation that was provided in Hyperproofs [20].

Note that both these constructions are not easily amenable to distributed proving. Thus we run baseline experiments on a single machine, but we exploit all the parallelism offered by the underlying framework.

We implement the bucket variant of Reckle Trees, with bucket size 2^5 and compare the performance against the above baselines in the following settings:

(1) Standard: In this implementation of aggregation, each node of the batch proof data structure is constructed sequentially on the same machine.
(2) Distributed proof generation: In this implementation of aggregation, we use the distributed system from Section 5.1 to compute the SNARK proofs in the batch tree data structure. Specifically, in our experiments, we use a cluster of 192 workers, where each worker is responsible for computing at most three Plonky2 proofs simultaneously.

Public parameters. The proving keys in Hyperproofs and Merkle SNARKs are approximately around 12 GiB and 6 GiB, respectively. This is because, in Hyperproofs, the size of public parameters is linear in the size of the vector. Whereas, in the Merkle SNARKs it is linear in the size of the batch and depth of the Merkle tree. However, in Reckle Trees, the proving is 3.62 GiB in total, as the prover key in our scheme depends only on the depth of the Reckle tree.

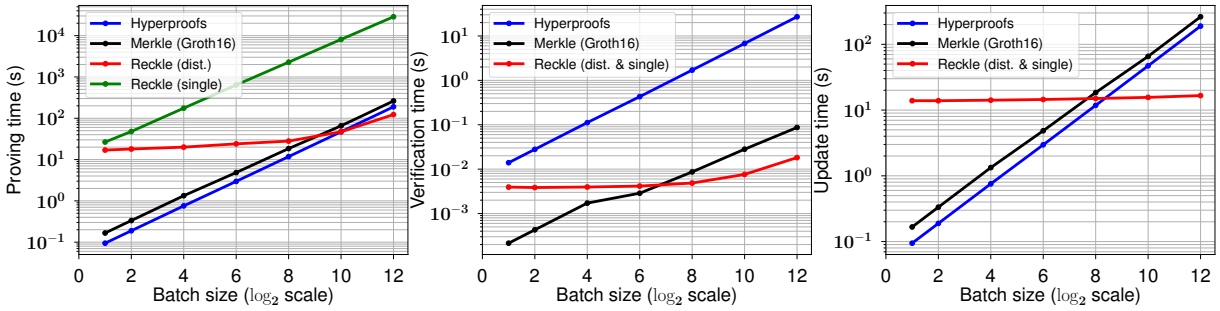
The verification key in our approach is at most 1.85 KiB. However, in Hyperproofs and Merkle SNARKs, the verification keys are in the order of around 10 MiB.

Prover time. In Fig. 11a, we show that the distributed implementation of Reckle Trees is 3.15× to 270× faster than the sequential implementation.

The distributed version of Reckle Trees has faster proving time (123 seconds) when compared with Groth16 [12] based Merkle SNARKs aggregation (4.44 minutes) and Hyerproofs (3.15 minutes). But, the sequential implementation of Reckle Trees is substantially slower than the baselines. However, as we argue in the previous sections, this is a one-time cost, but allows fast updates enabling many potential applications.

The aggregation of Hyperproofs outperforms other approaches in Fig. 11a. However, Hyperproofs trades this for a large proof size and increased verification times.

Verification time and proof size. The batch proof in our scheme is 112 KiB. 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^{12} proofs, it takes 14.29 ms to compute the canonical digest and 3.91 ms to verify the Plonky2 proof. We include the cost of computing canonical digest in Fig. 11b. However, a batch



(a) Aggregation **(b) Verification** **(c) Updates**

Figure 11: The x-axis is # of proofs being aggregated. We use the 128-bit variant of Poseidon.

proof in Hyperproofs is 52 KiB and takes around 27.03 seconds to verify. Similarly, a batch proof in Merkle SNARKs is 192 bytes and takes around 87 ms to verify. Since Merkle SNARKs use Groth16 proof system, the proof verification involves constant number of pairings and a multi-exponentiation of size $2k + 1$. We run the Merkle SNARKs verifier implemented in [20] and report the results in Fig. 11b.

Batch updates. In our experiments, we randomly sample an element from the batch to update. Updating a batch proof in Reckle Trees involves re-computing the recursive SNARK proofs just 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^{12} values is 16.61 seconds.

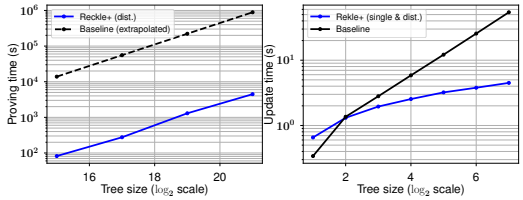
However, both Merkle SNARKs and Hyperproofs do not support updatable batch proofs. Thus, both these schemes have to recompute the batch proof *from scratch* whenever an element in the batch changes. For a batch size of 2^{12} values, Merkle SNARKs and Hyperproofs require 4.44 and 3.15 minutes, respectively, to recompute the batch proof. Thus, as shown in Fig. 11c, Reckle Trees is around 11× and 15× faster than Hyperproofs and Merkle SNARKs, respectively when the batch is 2^{12}.

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. Reckle Trees does not suffer this limitation, allowing for flexibility in adjusting the batch size as required.

5.3 Applications

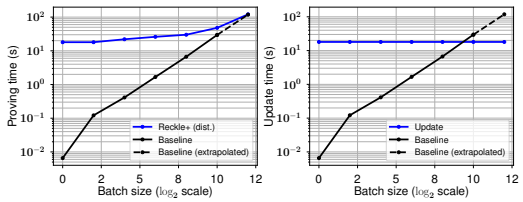
In this subsection, we evaluate the performance of Reckle+ Trees for the digest translation and BLS key aggregation.

Experimental setup. For both digest translation and BLS public key aggregation, we implement the Reckle+ Trees and the baseline circuits in Plonky2 [18]. However, we use the distributed system from Section 5.1 with identical configuration, instance types, and the number of workers for our experiments in this section.



(a) Aggregation **(b) Updates**

Figure 12: Prover and update cost for dynamic digest translation. We extrapolate baselines due to insufficient memory for the SNARK prover. For height of tree $[1, 7]$, the baseline proving times are $[0.34, 1.36, 2.80, 5.86, 12.15, 25.59, 54.21]$ seconds, respectively.



(a) Aggregation **(b) Updates**

Figure 13: Prover and update cost for the BLS application. We extrapolate baseline due to insufficient memory.

For digest translation, the baseline circuit is a monolithic circuit that recomputes the Merkle tree inside the circuit using both Poseidon and Keccak hash functions. Since Plonky2 does not support distributed proving, we run the baseline on a single machine. We observe that even for trees with 2^8 leaves, the baseline implementation *runs out of memory* (64 GiB) while computing the Plonky2 proof. Thus we extrapolate the performance of the baseline implementation for larger tree sizes. However, our distributed variant of Reckle+ Trees scales even for 8 million leaves!

For the BLS public key aggregation (Reckle+ Trees and the baseline), we assume that the public keys are stored in a Merkle tree of height 21. To implement Reckle+ Trees, we use a fork of Plonky2-BN254 library [19], which implements the BN254 group operations non-natively on Goldilocks field. However, to implement

the baseline, we repurpose Reckle+ Trees's circuit and additionally include the cost of hashing operations to simulate the membership proof verification. Similar to digest translation, we observe that the baseline implementation runs out of memory while computing a Plonky2 for modestly subset sizes. Thus, we extrapolate values for comparison.

Prover. We observe that distributed Reckle+ Trees takes around 1.25 hours to compute the batch data structure for the digest translation application. However, we estimate (Fig. 12a) that Reckle+ Trees is 200× faster than the baseline implementation. Similarly, for BLS public key aggregation, we estimate Reckle+ Trees to be faster than the baseline for larger batch sizes.

Updates. For the digest translation application, the cost of a single update is logarithmic in the capacity of the tree. However, the baseline implementation requires work linear in the capacity of the tree. In our experiments, we observe that for tree height of 7, the baseline approach requires 54.21 seconds to update a single leaf. However, Reckle+ Trees requires 4.49 seconds, which is 12× faster than the baseline. Similarly, in the BLS application, we expect Reckle+ Trees to be 6× faster than the baseline. We observe comparable verification times (3-6ms) and proof sizes (110-120KiB) for the baseline and Reckle+ Trees in both applications.

ACKNOWLEDGEMENTS

We'd like to thank the Lagrange Labs Engineering Team for their work in identifying and exploring applications of Reckle Trees and Reckle+ Trees to blockchain protocols.

REFERENCES

[1] [n. d.]. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/.
[2] [n. d.]. Securely Scaling ZK Interoperability. https://www.lagrange.dev/.
[3] Josh Benaloh and Michael de Mare. 1993. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In *Eurocrypt 93*. International Association for Cryptologic Research.
[4] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short Signatures from the Weil Pairing. *Journal of Cryptology* 17 (2001), 297–319.
[5] Dario Catalano and Dario Fiore. 2013. Vector Commitments and Their Applications. In *Public-Key Cryptography - PKC 2013*.
[6] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang. 2018. Edrax: A Cryptocurrency with Stateless Transaction Validation. Cryptology ePrint Archive, Report 2018/968.
[7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. 137–150.
[8] Sai Deng and Bo Du. 2023. zkTree: A Zero-Knowledge Recursion Tree with ZKP Membership Proofs. Cryptology ePrint Archive, Paper 2023/208. https://eprint.iacr.org/2023/208
[9] Ethereum.org. [n. d.]. Nodes and clients. https://ethereum.org/en/developers/docs/nodes-and-clients#light-node
[10] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. 2020. Pointproofs: Aggregating Proofs for Multiple Vector Commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. 2007–2023.
[11] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association.
[12] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*. 305–326.
[13] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. [n. d.]. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 13510)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). 359–388.
[14] Jing Liu and Liang Feng Zhang. 2022. Matproofs: Maintainable Matrix Commitment with Efficient Aggregation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. 2041–2054.
[15] Ralph C. Merkle. 1989. A Certified Digital Signature. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, Gilles Brassard (Ed.), Vol. 435. 218–238.
[16] Alex Ozdemir. 2020. bellman-bignat. https://github.com/alex-ozdemir/bellman-bignat.
[17] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. 2020. Scaling Verifiable Computation Using Efficient Set Accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2075–2092.
[18] Plonky2 Polygon. [n. d.]. Fast recursive arguments with PLONK and FRI. https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf.
[19] Qope. 2023. plonky2-bn254. https://github.com/qope/plonky2-bn254.
[20] Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. 2022. Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3001–3018.
[21] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. 2020. Aggregatable Subvector Commitments for Stateless Cryptocurrencies. In *Security and Cryptography for Networks*, Clemente Galdi and Vladimir Kolesnikov (Eds.). 45–64.
[22] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. 2020. Towards Scalable Threshold Cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 877–893.
[23] Weijie Wang, Annie Ulichney, and Charalampos Papamanthou. 2023. BalanceProofs: Maintainable Vector Commitments with Fast Aggregation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4409–4426.
[24] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy (SP)*. 863–880.

A SOUNDNESS PROOF

Proof of Theorem 3.1. Following the notation from Definition 2.4, suppose the adversary outputs a commitment C, two element sets $\{a_i\}_{i\in I}$ and $\{a'_j\}_{j\in J}$, two batch proofs π_I and π_J such that the canonical digest of $\{a_i\}_{i\in I}$ is $d(I)$, the canonical digest of $\{a'_j\}_{j\in J}$ is $d(J)$ and

$$1 \leftarrow \mathit{SNARK}.\mathsf{Verify}(\mathsf{vk}_{\mathcal{B}}, (\mathsf{vk}_{\mathcal{B}}, \mathsf{C}, d(I)), \pi_I))$$

and

$$1 \leftarrow \mathit{SNARK}.\mathsf{Verify}(\mathsf{vk}_{\mathcal{B}}, (\mathsf{vk}_{\mathcal{B}}, \mathsf{C}, d(J)), \pi_J))$$

while there exists $k \in I \cap J$ such that $a_k \neq a'_k$.

Due to SNARK knowledge soundness, we can extract the batch-proof data structures Λ_I and Λ_J. Since $k \in I \cap J$, Λ_I and Λ_J will both contain path p_k in common. Since both recursive proofs verified, and unless the adversary is able to break collision resistance, all nodes v on these paths (along with their sibling nodes) must have the same Merkle hash values C_v. However, the last node that is extracted on Λ_I's p_k path has to be a_k (by collision resistance with respect to canonical digest $d(I)$) and the last node that is extracted on Λ_J's p_k path has to be a'_k (by collision resistance with respect to canonical digest $d(J)$). By assumption, $a_k \neq a'_k$. This is a contradiction. Therefore, soundness holds. □