Vector Commitments with Efficient Updates

Ertem Nusret Tas
Stanford University, CA, USA

Dan Boneh
Stanford University, CA, USA

Abstract

Dynamic vector commitments that enable local updates of opening proofs have applications ranging from verifiable databases with membership changes to stateless clients on blockchains. In these applications, each user maintains a relevant subset of the committed messages and the corresponding opening proofs with the goal of ensuring a succinct global state. When the messages are updated, users are given some global update information and update their opening proofs to match the new vector commitment. We investigate the relation between the size of the update information and the runtime complexity needed to update an individual opening proof. Existing vector commitment schemes require that either the information size or the runtime scale *linearly* in the number k of updated state elements. We construct a vector commitment scheme that asymptotically achieves both length and runtime that is *sublinear* in k, namely k^{ν} and $k^{1-\nu}$ for any $\nu \in (0, 1)$. We prove an information-theoretic lower bound on the relation between the update information size and runtime complexity that shows the asymptotic optimality of our scheme. While in practice, the construction is not yet competitive with Verkle commitments, our approach may point the way towards more performant vector commitments.

2012 ACM Subject Classification Security and privacy

Keywords and phrases Vector commitments, stateless clients

Digital Object Identifier 10.4230/LIPIcs.AFT.2023.29

Related Version *Full Version*: `https://arxiv.org/abs/2307.04085`

Funding This work was partially funded by NSF, DARPA, the Simons Foundation, and NTT Research. Additional support was provided by the Stanford Center for Blockchain Research. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

1 Introduction

A Vector Commitment (VC) scheme [14, 24, 13] enables a committer to succinctly commit to a vector of elements. Later, the committer can generate an *opening proof* to prove that a particular position in the committed vector is equal to a certain value. VCs have found many applications in databases and blockchains [26, 39] as they enable a storage system to only store a commitment to the vector instead of the entire vector. The data itself can be stored elsewhere along with opening proofs. In a multiuser system, every user might store only one position of the vector along with the opening proof for that position.

Dynamic VCs [13] are vector commitments that support updates to the vector. Suppose the committed vector is of length N and some $k < N$ positions in the vector are updated, so that a new vector commitment is published. Then, every user in the system will need to update their local opening proof to match the updated commitment, and this is done with the help of some global *update information* U that is broadcast to all users. This information is typically generated and published by a manager who maintains the entire vector. Applications of dynamic VCs include verifiable databases, zero-knowledge sets with

Table 1 Comparison of different VCs. $|U|$ denotes the length of the update information. T denotes the runtime of a single proof update. $|G|$ and $|H|$ denote the size of a single group element and a single hash value, respectively. T_G and T_f denote the time complexity of a single group operation and a single function evaluation for the hash function used by the VC. The last column PP is 'Y' if the proof update requires pre-processing to generate a global and fixed table of auxiliary data needed for proof updates.

Vector Commitment	$\|U\|$	T	PP
Merkle tree [25]	$\tilde{\Theta}(k)\ \|H\|$	$\tilde{O}(1)$	N
Hyperproofs [33]	$\tilde{\Theta}(k)\ \|G\|$	$\tilde{O}(1)$	N
Verkle tree [11]	$\tilde{\Theta}(k)\ \|G\|$	$\tilde{O}(1)\ \|H\|\ T_G$	Y
This work with $\nu \in [0,1]$	$\tilde{\Theta}(k^{\nu})\|H\|$	$\tilde{\Theta}(k^{1-\nu})\ T_f$	N
KZG commitments [19]	$\tilde{O}(1)$	$\tilde{\Theta}(k)\ T_G$	Y
RSA accumulators and VCs [7, 8]	$\tilde{O}(1)$	$\tilde{\Theta}(k)\ T_G$	N
Bilinear accumulators [27, 34]	$\tilde{O}(1)$	$\tilde{\Theta}(k)\ T_G$	N

frequent updates [13] and stateless clients for blockchains [9]. The challenge is to design a VC scheme that minimizes the size of the update information U as well as the computation work by each user to update their local opening proof.

For example, consider stateless clients on a blockchain as an important application for dynamic VCs. The state of the chain can be represented as a vector of length N, where position i corresponds to the state of account number i. Every user will locally maintain its own state (corresponding to some position in the vector) along with an opening proof that enables the user to convince a third party as to its current state. Whenever a new block is published, the state of the chain changes. In particular, suppose k out of the N positions in the vector need to be updated. The block proposer will publish the *update information* U along with the new block, and every user will update their opening proof to match the new committed state of the chain. Thus, users can ensure that their opening proofs are up to date with respect to the latest committed state of the chain.

We stress that in this application, the data being updated, namely the updated positions and diffs, is published as part of the block. The update information U only contains additional information that is needed to update the opening proofs. When we refer to the size of U, we refer to its size, excluding the updated data (i.e., excluding the updated positions and diffs).

In this paper, we investigate the trade-off between the length $|U|$ of the update information and the time complexity of proof updates. Dynamic VCs can be grouped into two categories in terms of these parameters (Table 1). Tree-based VCs [25, 33] enable users to update their proofs in time $O(\text{polylog}\, N)$. Each opening proof typically consists of polylog (N) inner nodes, and the update information U contains the changes in the inner nodes affected by the message updates. Each user calculates its new opening proof by downloading the relevant inner nodes published as part of U. When k positions are updated, a total of $O(k \log(N))$ inner nodes in the tree are affected in the worst case. Thus, when each inner node has length $\Theta(\lambda)$, proportional to the security parameter λ, the update information consists of $O(k \log(N)\lambda)$ bits.

In contrast, algebraic VCs [19, 7, 8, 27, 34] enable users to update their opening proofs with only knowledge of the updated data. They do not require any additional update information U to be published beyond the indices and the "diffs" of the updated data. Thus, the length of the update information needed to update the opening proofs is $O(1)$. However, algebraic VCs typically require each user to read all of the changed messages and incorporate the effect of these changes on their proofs, resulting in $\Theta(k)$ work per proof update.

To summarize, while tree-based VCs support efficient calculation of the new opening proofs by publishing a large amount of update information, linear in k, algebraic VCs do not require any additional update information beyond the updated data, but suffer from a large runtime for proof updates, linear in k. We formalize the dichotomy of VCs in Section 3.

1.1 Our Results

We propose a family of VCs that can support *sublinear update*, where both the length $|U|$ of the update information and the complexity of proof updates are sublinear in k. More specifically, our VCs can attain $|U| = \Theta(k^{\nu}\lambda)$, $\nu \in (0,1)$, with a proof update complexity of $\Theta(k^{1-\nu})$ operations. Our candidate construction with sublinear update is a *homomorphic Merkle tree*, first developed by [29, 32], where each inner node can be expressed as a *sum* of the *partial digests* of the messages underneath (Section 4). The algebraic structure of these trees enable each user to calculate the *effect* of a message update on any inner node without reading other inner nodes or messages. We identify homomorphic Merkle tree constructions based on lattices, from the literature [30, 29, 32].

In Section 4, we provide the update algorithms (Alg. 1) for homomorphic Merkle trees, parameterized by $\nu \in (0,1)$. Our algorithm identifies a special subset of size $\tilde{\Theta}(k^{\nu})$ of the inner nodes affected by the message updates, and publish their new values as U; so that the users need not calculate these values. These inner nodes are selected carefully to ensure that any inner node *outside* of U is affected by at most $\Theta(k^{1-\nu})$ updated messages. Thus, to modify its opening proof, each user has to calculate the partial digests of at most $\Theta(k^{1-\nu})$ updated messages per inner node within its proof (that consists of $\Theta(\log(N))$ inner nodes). Moreover, to calculate these partial digests, the user only needs the "diffs" of the updated messages. This brings the asymptotic complexity of proof updates to $\tilde{\Theta}(k^{1-\nu})$ operations, while achieving an update information size of $\tilde{\Theta}(k^{\nu}\lambda)$ as opposed to $\tilde{\Theta}(k\lambda)$ on Merkle trees using SHA256.

In Section 6, we prove an information theoretic lower bound on the size of the update information given an upper bound on the runtime complexity of proof updates. The bound implies the asymptotic optimality of our scheme with sublinear update. Its proof is based on the observation that if the runtime complexity is bounded by $O(k^{1-\nu})$, a user that wants to update its proof cannot read beyond $O(k^{1-\nu})$ updated messages. Then, to calculate the effect of the remaining $k - O(k^{1-\nu})$ messages on its opening proof, the user has to download parts of the structured update information U. Finally, to obtain the lower bound on $|U|$, we use Shannon entropy and lower bound the number of bits, namely $O(k^{\nu}\lambda)$, required to capture the total information that will be downloaded by the users; while maintaining the security of the VC with parameter λ.

1.2 Applications

We identify three main applications for VCs with sublinear update.

1.2.1 Stateless clients for Ethereum

Ethereum is the largest decentralized general purpose computation platform by market cap. Ethereum state (*e.g.*, user accounts) is currently stored in the form of a Merkle tree [5] and grows approximately by half every year [10]. Stateless clients [9, 10] were proposed to mitigate the problem of state bloat and prevent the state storage and maintenance from becoming a bottleneck for decentralization. Stateless clients maintain an opening proof to their account balances within the Ethereum state, thus can effortlessly prove the inclusion of their accounts within the latest state. This enables the other Ethereum clients to verify the transactions that come with opening proofs without having to download the full state and check the validity of the claimed account balances. Since block verification now requires downloading the proofs for the relevant state elements, Verkle trees [20, 11, 16] were proposed as a replacement for Merkle trees due to their short proof size.

Each new Ethereum block contains transactions that update the state elements and their opening proofs. Archival nodes and block producers still maintain the full state so that they can inform the stateless clients about their new opening proofs [10]. For this purpose, block producers must broadcast enough information to the clients over the peer-to-peer gossip network of Ethereum[1]. As minimizing the proof size was paramount to decentralizing verification for blocks, minimizing the update information size becomes necessary for decentralizing the role of the block producer who has to disseminate this information. However, reducing the length of the update information must not compromise the low overhead of stateless clients by requiring larger number of operations per proof update. Therefore, the ideal VC scheme for stateless clients must strike a delicate balance between the size of the update information and the runtime complexity of proof updates.

In Section 5, we provide the update algorithms for Verkle trees given their role in supporting stateless clients. We observe that Verkle trees do not support sublinear update, and fall under the same category as tree-based VCs with update information length $\tilde{\Theta}(k\lambda)$. Despite this fact, Verkle trees are highly practical in terms of updates. In Section 5.5, we estimate that the update information size after a typical Ethereum block does not exceed $|U| \approx 100$ kBytes (compared to the typical block size of < 125 kBytes). Moreover, each Verkle proof can be updated within approximately less than a second on commodity hardware. In contrast, even the most efficient homomorphic Merkle tree construction [32] requires an update information size of 110.88 MBytes and an update time of 32.6 seconds when the trade-off parameter ν is $1/2$, despite its asymptotic optimality (*cf.* Section 4.4). The large update information size is due to the lattice-based construction of these VCs. Despite their advantage in terms of concrete performance, unlike these lattice-based constructions, Verkle trees are not secure against quantum computers. Designing dynamic VCs that are asymptotically optimal, practically efficient and post-quantum resilient remains an open problem.

1.2.2 Databases with frequent membership changes

VCs with sublinear update can support databases with frequent membership changes. When a user first registers, a message is updated to record the membership of the user. The user receives this record and its opening proof, using which it can later anonymously prove its membership. When the user leaves the system, the message is once again updated to delete the record. In all these steps, membership changes result in updates to the opening proofs of other members. When these changes are frequent, it becomes infeasible to distribute new proofs after each change. VCs with sublinear update offer an alternative and efficient way to update the opening proofs of the users in the event of such changes.

[1] Block producers can enable the clients to succinctly verify the correctness of this information via SNARK proofs, thus still keeping the verification cost of blocks small.

1.3 Related Work

There are many VC constructions, each with different guarantees regarding the proof, commitment and public parameter sizes, verification time, updatability and support for subvector openings [14, 24, 13, 37, 29, 23, 21, 18, 17, 6, 40, 8, 34, 33, 19, 11] (cf [28] for an SoK of VCs). First formalized by [13], almost all VCs allow some degree of updatability. Whereas [29, 6, 8, 34] enable updating the commitment and the opening proofs with only the knowledge of the old and the new messages, most VCs require some structured update information beyond the messages when the users do not have access to the internal data structures. Among the lattice-based accumulators, vector commitments and functional commitments [18, 22, 31, 29, 32, 30, 38], constructions amenable to sublinear update are presented in [30, 29, 32, 31]. Homomorphic Merkle trees were formalized and instantiated by [30, 29, 32] in the context of streaming authenticated data structures and parallel online memory checking. The construction presented in [31, Section 3.4] offers an alternative VC with sublinear update as it is not a Merkle tree, yet has the property that each inner node can be expressed as a *sum* of the partial digests of individual messages.

An alternative design to support stateless clients is the agregatable subvector commitment (aSVC) scheme [36], which is a VC that enables aggregating multiple opening proofs into a succinct subvector proof. It enables each user to update its opening proof with the knowledge of the transactions in the blocks, and block producers to prove the validity of these transactions succinctly by aggregating the proofs submitted by the transacting users. As the scheme is based on KZG commitments, no update information is needed, yet, the update time complexity is linear in the number of transactions per block.

For dynamic accumulators that support additions, deletions and membership proofs, Camacho and Hevia proved that after k messages are deleted, $\Omega(k)$ bits of data must be published to update the proofs of the messages in the initial accumulated set [12, Theorem 1]. Their lower bound is information-theoretic and follows from a compression argument. Christ and Bonneau subsequently used a similar method to prove a lower bound on the global state size of a *revocable proof system* abstraction [15]. As revocable proof systems can be implemented by dynamic accumulators and vector commitments, their lower bound generalizes to these primitives, *i.e.*, after k messages are updated in a dynamic VC, at least $\Omega(k)$ bits of data must be published to update the opening proofs (*cf.* the full version of the paper [35, Appendix A] for the proof). They conclude that a stateless commitment scheme must either have a global state with linear size in the number of accounts, or require a near-linear rate of local proof updates. In our work, we already assume a linear rate of local proof updates, *i.e.*, after every Ethereum block or k messages in our parameterization, and that the message updates are publicized by the blockchain. We instead focus on the trade-off between the global structured update information size (beyond the published messages) and the runtime complexity of proof updates.

2 Preliminaries

2.1 Notation

We denote the security parameter by λ. An event is said to happen with *negligible probability*, if its probability, as a function of λ, is $o(1/\lambda^d)$ for all $d > 0$. An event happens with *overwhelming probability* if it happens except with negligible probability.

We denote the set $\{0, 1, 2, .., N-1\}$ by $[N]$. When $y = O(h(x) \operatorname{polylog}(x))$, we use the shorthand $y = \tilde{O}(h(x))$ (similarly for $\Theta(.)$ and $\tilde{\Theta}(.)$). The function $H(.) \colon \mathcal{M} \to \{0,1\}^\lambda$ represents a collision-resistant hash function. We denote the binary decomposition of an integer x by $\mathrm{bin}(x)$, and for $c > 2$, its base c decomposition by $\mathrm{bin}_c(x)$. A vector of N elements $(n_0, .., n_{N-1})$ is shown as $(n_i)_i$. The notation $\mathbf{x}[i{:}j]$ denotes the substring starting at the i^{th} index and ending at the j^{th} index within the sequence $\mathbf{x}$. The indicator function 1_P is equal to one if the predicate P is true, otherwise, it is zero. In the subsequent sections, k will be used to denote the number of updated messages.

For a prime p, let $\mathbb{F}_p$ denote a finite field of size p. We use $\mathbb{G}$ to denote a cyclic group of prime order p with generator g. The Lagrange basis polynomial for a given $x \in \mathbb{F}_p$ is denoted as $L_x(X)$:

$$L_x(X) = \prod_{\substack{i \in \mathbb{F}_p \\ i \neq x}} \frac{X - i}{x - i}$$

We will use $|G|$ and $|H|$ to denote the maximum size of the bit representation of a single group element and a single hash value respectively. We will use T_G and T_f to denote the time complexity of a single group operation and a single function evaluation for the hash functions in Section 4.1.

2.2 Vector Commitments

A vector commitment (VC) represents a sequence of messages such that each message can be proven to be the one at its index via an *opening proof.* A dynamic vector commitment allows updating the commitment and the opening proofs with the help of an *update information* when the committed messages are changed.

▶ **Definition 1** (from [13]). *Dynamic (updateable) vector commitments can be described by the following algorithms:*

$\textsc{KeyGen}(1^\lambda, N) \to pp$: *Given the security parameter λ and the size $N = \mathrm{poly}(\lambda)$ of the committed vector, the key generation algorithm outputs public parameters pp, which implicitly define the message space $\mathcal{M}$.*

$\textsc{Commit}_{pp}(m_0, .., m_{N-1}) \to (C, \mathsf{data})$: *Given a sequence of N messages in $\mathcal{M}$ and the public parameters pp, the commitment algorithm outputs a commitment string C and the data* data *required to produce the opening proofs for the messages. Here,* data *contains enough information about the current state of the VC's data structure (*i.e.*, the current list of committed messages) to help generate the opening proofs.*

$\textsc{Open}_{pp}(m, i, \mathsf{data}) \to \pi_i$: *The opening algorithm is run by the committer to produce a proof π_i that m is the i^{th} committed message.*

$\textsc{Verify}_{pp}(C, m, i, \pi_i) \to \{0, 1\}$: *The verification algorithm accepts (*i.e.*, outputs 1) or rejects a proof. The security definition will require that π_i is accepted only if C is a commitment to some $(m_0, .., m_{N-1})$ such that $m = m_i$.*

$\textsc{Update}_{pp}(C, (i, m_i)_{i \in [N]}, (i, m'_i)_{i \in [N]}, \mathsf{data}) \to (C', U, \mathsf{data}')$: *The algorithm is run by the committer to update the commitment C when the messages $(m_{i_j})_{j \in [k]}$ at indices $(i_j)_{j \in [k]}$ are changed to $(m'_{i_j})_{j \in [k]}$. The other messages in the vector are unchanged. It takes as input the old and the new messages, their indices and the data variable* data*. It outputs a new commitment C', update information U and the new data variable* data'*.*

$\textsc{ProofUpdate}_{pp}(C, p((i, m_i)_{i \in [N]}, (i, m'_i)_{i \in [N]}), \pi_j, m', i, U) \to \pi'_j$: *The proof update algorithm can be run by any user who holds a proof π_j for some message at index j and a (possibly) new message m' at that index. It allows the user to compute an updated proof π'_j (and the updated commitment C') such that π'_j is valid with respect to C', which contains*

m'_i, $i \in N$, as the new messages at the indices $i \in N$ (and m' as the new message at index i). Here, $p(.)$ specifies what portion of the old and the new messages is sufficient to update the opening proof. For instance, the proof update algorithm often does not need the old and the new messages in the open; but can carry out the proof update using only their differences. In this case, $p((i, m_i)_{i\in[N]}, (i, m'_i)_{i\in[N]}) = (i, m'_i - m_i)_{i\in N}$.

Correctness of a VC requires that $\forall N = \text{poly}(\lambda)$, for all honestly generated parameters $pp \leftarrow \textsc{KeyGen}(1^\lambda, N)$, given a commitment C to a vector of messages $(m_0, .., m_{N-1}) \in \mathcal{M}^N$, generated by $\textsc{Commit}_{pp}$ (and possibly followed by a sequence of updates), and an opening proof π_i for a message at index i, generated by $\textsc{Open}_{pp}$ or $\textsc{ProofUpdate}_{pp}$, it holds that $\textsc{Verify}_{pp}(C, m_i, i, \pi_i) = 1$ with overwhelming probability.

Security of a VC is expressed by the position-binding property:

▶ **Definition 2** (Definition 4 of [13]). *A VC satisfies position-binding if $\forall i \in [N]$ and for every PPT adversary $\mathcal{A}$, the following probability is negligible in λ:*

$$\Pr\left[\begin{smallmatrix} \textsc{Verify}_{pp}(C,m,i,\pi_i)=1\wedge \\ \textsc{Verify}_{pp}(C,m',i,\pi'_i)=1\wedge m\neq m' \end{smallmatrix} : \begin{smallmatrix} pp\leftarrow \textsc{KeyGen}(1^\lambda,N) \\ (C,m,m',\pi_i,\pi'_i)\leftarrow \mathcal{A}(pp) \end{smallmatrix}\right]$$

We relax the *succinctness* assumption of [13] and denote a value to be succinct in x if it is $\text{polylog}(x)$.

Many VC constructions also satisfy the hiding property: informally, no PPT adversary $\mathcal{A}$ should be able to distinguish whether the VC was calculated for a vector $(m_0, .., m_{N-1})$ or a vector $(m'_0, .., m'_{N-1}) \neq (m_0, .., m_{N-1})$. In this work, we do not consider the hiding property since it is not explicitly required by our applications, and VCs can be made hiding by combining them with a hiding commitment [13].

2.3 KZG Polynomial Commitments

The KZG commitment scheme [19] commits to polynomials of degree bounded by ℓ using the following algorithms:

KeyGen$(1^\lambda, \ell) \rightarrow pp$**:** outputs $pp = (g, g^\tau, g^{(\tau^2)}, .., g^{(\tau^\ell)})$ as the public parameters, where g is the generator of the cyclic group $\mathbb{G}$ and τ is a trapdoor ($pp[i] = g^{\tau^i}$).

Commit$(pp, \phi(X)) \rightarrow (C, \mathsf{data})$**:** The commitment to a polynomial $\phi(X) = \sum_{i=0}^{\ell-1} a_i X^i$ is denoted by $[\phi(X)]$, and is computed as $[\phi(X)] = \prod_{i=0}^{\ell} (pp[i])^{a_i}$. The commitment algorithm outputs $C = [\phi(X)]$ and $\mathsf{data} = \phi(X)$.

Open$_{pp}(m, i, \mathsf{data}) \rightarrow \pi$**:** outputs the opening proof π_i that $\phi(i) = m$, calculated as the commitment to the quotient polynomial $(\phi(X) - \phi(i))/(X - i)$.

Verify(C, m, i, π) accepts if the pairing check $e\left(C/g^m, g\right) = e\left(\pi, pp[1]/g^i\right)$ holds.

We refer to [19] for the security analysis of this scheme.

2.4 Merkle Trees

Merkle Tree is a vector commitment using a collision-resistant hash function. In a Merkle tree, hashes of the committed messages constitute the leaves of a c-ary tree of height $h = \log_c(N)$, where each inner node is found by hashing its children. The depth of the root is set to be 0 and the depth of the leaves is $\lceil \log_c(N) \rceil$. The commitment function outputs the Merkle root as the commitment C and the Merkle tree as data. The opening proof for a message m_x at some index x is the sequence of $h(c-1)$ hashes consisting of the siblings of the inner nodes on the path from the root to the hash of the message m_x. We hereafter consider binary Merkle trees ($c = 2$) and assume $N = c^h = 2^h$ unless stated otherwise. Let $u_{b_0,b_1,..,b_{i-1}}$, $b_j \in \{0, 1\}$,

$j \in [i]$, denote an inner node at depth $i-1$ that is reached from the root by choosing the left child at depth j if $b_j = 0$ and the right child at depth j if $b_j = 1$ ($b_0 = \perp$ and $u_\perp$ is the root). By definition, for a message m_x at index x, $H(m_x) = u_{\perp,\mathrm{bin}(x)}$.

2.5 Verkle Trees

A Verkle tree [11, 16] is similar to a Merkle tree except that each inner node is calculated as the hash of the KZG polynomial commitment to its children. Let $b_j \in [c]$, $j = 1, .., h$, denote the indices of the inner nodes on the path from the root to a leaf at index x, $\mathrm{bin}_c(x) = (b_1, .., b_h)$, relative to their siblings. Define $f_{b_0,..,b_j}$, $j \in [h]$, as the polynomials determined by the children of the inner nodes on the path from the root to the leaf, where $f_{b_0} = f_\perp$ is the polynomial determined by the children of the root. Let $C_{b_0,..,b_j} = [f_{b_0,..,b_j}]$, $j \in [h]$, denote the KZG commitments to these polynomials. By definition, $u_{b_0,..,b_j} = H(C_{b_0,..,b_j})$, and the value of the polynomial $f_{b_0,..,b_j}$ at index b_{j+1} is $u_{b_0,..,b_{j+1}}$ for each $j \in [h]$. Here, $u_{b_0} = H(C_{b_0})$ is the root of the tree, and $u_{b_0,..,b_h}$ equals the hash $H(m_x)$ of the message at index x. For consistency, we define $C_{b_0,..,b_h}$ as m_x. For example, given $h = 3$ and $c = 4$, the inner nodes from the root to the message m_{14} have the indices $b_0 = 0$, $b_1 = 3$ and $b_2 = 2$, and they are committed by the polynomials $f_\perp$, $f_{\perp,0}$ and $f_{\perp,0,3}$ respectively.

The commitment function $\textsc{Commit}_{pp}(m_0, .., m_{N-1})$ outputs the root u_{b_0} as the commitment C and the Verkle tree itself as data.

The Verkle opening proof for the message m_x, $\mathrm{bin}(x) = (b_1, .., b_h)$, consists of two parts: (i) the KZG commitments $(C_{b_0,b_1}, .., C_{b_0,..,b_{h-1}})$ on the path from the root to the message, and (ii) a Verkle multiproof. The goal of the Verkle multiproof is to show that the following evaluations hold for the inner nodes from the root to the message: $f_{b_0,..,b_j}(b_{j+1}) = u_{b_0,..,b_{j+1}} = H(C_{b_0,..,b_{j+1}})$, $j \in [h]$. It has two components: (i) the commitment $[g(X)]$ and (ii) the opening proof π' for the polynomial $h(X) - g(X)$ at the point $t = H(r, [g(X)])$, where

$$g(X) = \sum_{j=0}^{h-1} r^j \frac{f_{b_0,..,b_j}(X) - u_{b_0,..,b_{j+1}}}{X - b_{j+1}}, \qquad h(X) = \sum_{j=0}^{h-1} r^j \frac{f_{b_0,..,b_j}(X)}{t - b_{j+1}},$$

and $r = H(C_{b_0}, .., C_{b_0,..,b_{h-1}}, u_{b_0,b_1}, .., u_{b_0,..,b_h}, b_1, .., b_h)$. Thus, $\textsc{Open}_{pp}(m, i, \mathsf{data})$ outputs $((C_{b_0,b_1}, .., C_{b_0,..,b_{h-1}}), ([g(X)], \pi'))$.

To verify a Verkle proof $\pi = ((C_{b_0,b_1}, .., C_{b_0,..,b_h}), (D, \pi'))$, algorithm $\textsc{Verify}_{pp}(C, m, x, \pi)$ first computes r and t using $u_{b_0,..,b_j} = H(C_{b_0,..,b_j})$, $j \in [h]$, and $u_{b_0,..,b_h} = H(m)$. Then, given the indices $\mathrm{bin}(x) = (b_1, .., b_h)$ and the commitments $(C_{b_0,b_1}, .., C_{b_0,..,b_h})$, it calculates

$$y = \sum_{j=0}^{h-1} r^j \frac{C_{b_0,..,b_j}}{t - b_{j+1}} \qquad E = \sum_{j=0}^{h-1} \frac{r^j}{t - b_{j+1}} C_{b_0,..,b_j}.$$

Finally, it returns true if the pairing check $e(E - D - [g(X)], [1]) = e(\pi', [X - t])$ is satisfied.

As the degree c of a Verkle tree increases, size of the opening proofs and the runtime of the verification function decreases in proportion to the height $h = \log_c N$ of the tree. This enables Verkle trees to achieve a short opening proof size for large number of messages (as in the case of the Ethereum state trie) by adopting a large degree (*e.g.*, $c = 256$). In comparison, each Merkle proof consists of $(c-1)\log_c N$ inner nodes, which grows linearly as c increases.

3 Formalizing the Dichotomy of VCs

We first analyze the trade-off between the number of operations required by proof updates and the size of the update information U by inspecting different types of dynamic VCs.

Recall that the number of updated messages is $k \leq N$.

3.1 Updating KZG Commitments and Opening Proofs

In the subsequent sections, we assume that each user has access to a dictionary of KZG commitments to the Lagrange basis polynomials $L_i(X)$, $i \in \mathbb{F}_p$, and for each polynomial, its opening proofs at each point $j \in \mathbb{F}_p$, $j < N$. With the help of this table, one can instantiate a KZG based VC to the messages $(m_i)_{i\in[N]}$, by treating them as the values of the degree N polynomial $\phi(X)$ at inputs $i \in \mathbb{F}_p$, $i < N$. We next analyze the complexity of the update information and the proof updates in this VC. The update and proof update algorithms are described in [35, Appendix F].

3.1.1 Update Information

Suppose the vector $(i, m_i)_{i\in[N]}$ is updated at some index i such that $m'_i \leftarrow m_i + \delta$ for some $\delta \in \mathbb{F}_p$. Then, the polynomial $\phi(X)$ representing the vector is replaced by $\phi'(X)$ such that $\phi'(X) = \phi(X)$ if $X \neq i$, and $\phi'(i) = \phi(i) + \delta$ at $X = i$. Thus, the new KZG commitment C' to $\phi'(X)$ is constructed from the commitment C to $\phi(X)$ as follows:

$$C' = [\phi'(X)] = [\phi(X) + \delta L_i(X)] = [\phi(X)][L_i(X)]^{\delta} = C \cdot [L_i(X)]^{\delta} = C \cdot [L_i(X)]^{m'_i - m_i}.$$

If the vector is modified at k different indices $i_1, ..., i_k$ from message m_{i_j} to m'_{i_j}, $j \in [k]$, then the new commitment $C' = [\phi'(X)]$ becomes

$$\begin{aligned}\left[\phi(X) + \sum_{j=1}^{k}(m'_{i_j} - m_{i_j})L_{x_{i_j}}(X)\right] &= [\phi(X)]\prod_{j=1}^{k}[L_{i_j}(X)]^{(m'_{i_j} - m_{i_j})} \\ &= C\prod_{j=1}^{k}[L_{i_j}(X)]^{(m'_{i_j} - m_{i_j})}.\end{aligned}$$

Thus, the commitment can updated given only the old and the new messages at the updated indices, besides the table.

3.1.2 Proof Update

Let π_x denote the opening proof of a polynomial $\phi(X)$ at a point (x, m_x). When k messages are updated, the new opening proof π'_x can be found as a function of the old proof π_x and the opening proofs $\pi_{i_j,x}$ of the Lagrange basis polynomials $L_{i_j}(X)$, $j \in [k]$, at the index x ($m'_x = m_x + \sum_{j=1}^{k}(m'_{i_j} - m_{i_j}) \cdot 1_{x=i_j}$ is the new value of m_x after the k updates). Namely, π'_x is

$$\left[\frac{\phi'(X) - m_x - \sum_{j=1}^{k}\delta_j \cdot 1_{x=i_j}}{X - x}\right] = \pi_x \prod_{j=1}^{k}\left[\frac{L_{i_j}(X) - L_{i_j}(x)}{X - x}\right]^{m'_{i_j} - m_{i_j}} = \pi_x \prod_{j=1}^{k} \pi_{i_j,x}^{m'_{i_j} - m_{i_j}}$$

Thus, the proof can updated given only the old and the new messages at the updated indices, besides the table. The update information is set to be the empty set, *i.e.*, $U = \emptyset$.

3.1.3 Complexity

The size of the update information is constant, *i.e.*, $\tilde{\Theta}(1)$. Each user can update its proof after k accesses to the dictionary, and in the worst case, $\Theta(k \log |\mathcal{M}|) = \tilde{\Theta}(k)$ group operations as $\log (m'_i - m_i) \leq \log |\mathcal{M}|$ for all $i \in [N]$.

3.2 Updating Merkle Trees and Opening Proofs

We next consider a Merkle tree and analyze the complexity of the update information size and the runtime for proof updates. A simple update scheme would be recalculating the new Merkle tree given all of the old messages or the old inner nodes of the Merkle tree, and the message updates. However, this implies a large complexity for the runtime of the proof update algorithm that scales as $\Omega(k)$ when users keep track of the inner nodes, and as $\Omega(N)$ when the users recalculate the tree from scratch at each batch of updates. Moreover, in many applications, the users do not have access to any messages or inner nodes besides those that are part of the Merkle proof held by the user. Hence, in the following sections, we describe update and proof update algorithms that reduce the runtime complexity of the proof updates at the expanse of larger update information (*cf.* the full version of the paper [35, Appendix F]).

3.2.1 Update Information

Suppose the vector $(i, m_i)_{i \in [N]}$ is updated at some index x, $(b_1, .., b_h) = \text{bin}(x)$, to m'_x. Then, the root $C = u_{b_0}$ and the inner nodes $(u_{b_0,b_1}, .., u_{b_0,b_1,..,b_h})$, $(b_1, .., b_h) = \text{bin}(i)$, must be updated to reflect the change at that index. Given the old inner nodes, the new values for the root and these inner nodes, denoted by $C' = u'_{b_0}$ and $(u'_{b_0,b_1}, .., u'_{b_0,b_1,..,b_h})$, are calculated recursively as follows:

$$u'_{b_0,b_1,..,b_h} \leftarrow H(m'_x),$$
$$u'_{b_0,b_1,..,b_j} \leftarrow \begin{cases} H(u'_{b_0,b_1,..,b_j,0}, u_{b_0,b_1,..,b_j,1}) \text{if } b_{j+1} = 0, \; j < h \\ H(u_{b_0,b_1,..,b_j,0}, u'_{b_0,b_1,..,b_j,1}) \text{if } b_{j+1} = 1, \; j < h \end{cases}$$

When the messages are modified at k different points i_j, $j \in [k]$, the calculation above is repeated k times for each update.

As the updated inner nodes are parts of the Merkle proofs, the update information consists of the new values at the inner nodes listed from the smallest to the largest depth in the canonical left to right order. For instance, $U = ((\bot, u'_\bot), (\bot 0, u'_0), (\bot 1, u'_1), (\bot 00, u'_{00}), (\bot 10, u'_{10}), ..)$ implies that the root $u_\bot$ and the inner nodes $u_{\bot 0}$, $u_{\bot 1}$, $u_{\bot 00}$ and $u_{\bot 10}$ were updated after k messages were modified at the leaves of the Merkle tree. We reference the updated inner nodes using their indices (*e.g.*, $U[b_0, b_1 .. b_j] = v$, when $(b_1 .. b_j, v) \in U$).

3.2.2 Proof Update

The Merkle proof π_x for a message at index x, $(b_1, .., b_h) = \text{bin}(x)$, is the sequence $(u_{\bar{b}_1}, u_{b_1,\bar{b}_2}, .., u_{b_1,b_2,..,\bar{b}_h})$. When k messages are updated, some of the inner nodes within the proof might have changed. A user holding the Merkle proof for index x can find the new values of these inner nodes by querying the update information with their indices.

3.2.3 Complexity

Upon receiving the update information U, each user can update its proof in $\Theta(\log^2(N) + |H| \log(N)) = \tilde{\Theta}(1)$ time by running a binary search algorithm to find the updated inner nodes within U that are part of its Merkle proof, and reading the new values at these nodes. Since modifying each new message results in $h = \log(N)$ updates at the inner nodes and some of the updates overlap, $|U| = \Theta(k \log(N/k)(\log(N) + |H|)) = \tilde{\Theta}(k)|H|$, as each updated inner node is represented by its index of size $\Theta(\log(N))$ and its new value of size $|H|$ in U.

3.3 Dichotomy of VCs

In the case of KZG commitments, $|U| = \tilde{\Theta}(1)$, and there is no information overhead on top of the message updates. For Merkle trees with an efficient proof update algorithm, $|U| = \tilde{\Theta}(k)|H|$, thus there is an extra term scaling in $\tilde{\Theta}(k)|H| = \tilde{\Theta}(k)\lambda$, since $|H| = \Omega(\lambda)$ for collision-resistant hash functions. In contrast, for KZG commitments, each user has to do $\tilde{\Theta}(k)$ group operations to update its opening proof; whereas in Merkle trees, each user can update its proof in $\tilde{\Theta}(1)$ time, which does not depend on k. Hence, KZG commitments outperform Merkle trees in terms of the update information size, whereas Merkle trees outperform KZG commitments in terms of the time complexity of proof updates. Table 1 generalizes this observation to a dichotomy between algebraic VC schemes and tree-based ones favoring shorter runtimes for proof updates. The algebraic and tree-based ones outperform each other in terms of the update information size and runtime complexity respectively.

4 Vector Commitments with Sublinear Update

We would like to resolve the separation in Table 1 and obtain a vector commitment, where both the size of the update information and the complexity of proof updates have a sublinear dependence on k. In particular, $|U| = \tilde{\Theta}(g_1(k)\lambda)$ in the worst case, and the proof update algorithm requires at most $\tilde{\Theta}(g_2(k))$ operations, where both $g_1(k)$ and $g_2(k)$ are $o(k)$. We say that such a VC supports *sublinear update*.

In this section, we describe a family of VCs with sublinear update, parameterized by the values $\nu \in (0,1)$ and characterized by the functions $(g_1, g_2) = (k^\nu, k^{1-\nu})$.

4.1 Homomorphic Merkle Trees

We first introduce homomorphic Merkle trees where messages placed in the leaves take values in a set $\mathcal{M}$. We will use two collision-resistant hash functions $\tilde{f} \colon \mathcal{D} \times \mathcal{D} \to \mathcal{R}$ and $f \colon \mathcal{M} \to \mathcal{R}$, where both $\mathcal{M}$ and $\mathcal{D}$ are vector spaces over some field $\mathbb{F}$, and $\mathcal{R}$ is an arbitrary finite set. We will also need an injective mapping $g : \mathcal{R} \to \mathcal{D}$, which need not be efficiently computable. We use $g^{-1} : \mathcal{D} \to \mathcal{R}$ to denote the inverse of g, meaning that $g^{-1}(g(x)) = x$ for all $x \in \mathcal{R}$. We require that g^{-1} be efficiently computable.

Now, for $j \in [h]$, where h is the height of the tree, every node $u_{b_0,..,b_j} \in \mathcal{D}$ of the homomorphic Merkle tree is characterized by the following expressions:

a leaf node: $\quad g^{-1}(u_{b_0,\mathrm{bin}(i)}) = f(m_i)$

an internal node: $\quad g^{-1}(u_{b_0,..,b_j}) = \tilde{f}(u_{b_0,..,b_j,0},\ u_{b_0,..,b_j,1})$ for $j < h$

The homomorphic property of the Merkle tree refers to the fact that there are efficiently computable functions

$$h_{i,j} : \mathcal{D} \to \mathcal{D} \qquad \text{for } i \in [N] \text{ and } j \in [h],$$

such that every inner node $u_{b_0,..,b_j} \in \mathcal{D}$ can be expressed as

$$u_{b_0} = \sum_{i \in [N]} h_{i,0}(m_i)$$

$$u_{b_0,..,b_j} = \sum_{i\colon \mathrm{bin}(i)[0:j-1]=(b_1,..,b_j)} h_{i,j}(m_i).$$

We refer to the function $h_{i,j}$ as a *partial digest function* and refer to $h_{i,j}(m_i)$ as the *partial digest* of m_i. In a homomorphic Merkle tree, every internal node is the sum of the partial digests of the leaves under that node. We will show in Section 4.3 that each function $h_{i,j}$ can be expressed as an iterated composition of the functions f and $\tilde{f}$. Evaluating $h_{i,j}$ requires evaluating the functions f and $\tilde{f}$ exactly $h-j$ times.

Opening proof for a message consists of *both* children of the internal nodes on the path from the message to the root (as opposed to Merkle opening proofs that contain only the siblings of the internal nodes on the path). For instance, the opening proof for the message m_i at leaf index i, with $\text{bin}(i) = (b_1, .., b_h)$, is $(i, (u_{b_0,..,b_j,0}, u_{b_0,..,b_j,1})_{j=0,..,h-1})$. Opening proofs are verified using the functions f and $\tilde{f}$ (not by using the functions $h_{i,j}$). To verify an opening proof $(i, (u_{b_0,..,b_j,0}, u_{b_0,..,b_j,1})_{j=0,..,h-1})$ for a message m_i with respect to the root u_{b_0}, the verifier checks if the following equalities hold:

$$\begin{aligned}
&\text{for the leaf:} && g^{-1}(u_{b_0,\text{bin}(i)}) = f(m_i)\\
&\text{for the internal nodes:} && g^{-1}(u_{b_0,..,b_j}) = \tilde{f}(u_{b_0,..,b_j,0},\ u_{b_0,..,b_j,1}) \text{ for } j = h-1, .., 0.
\end{aligned}$$

If so, it accepts the proof, and otherwise it outputs reject.

As an example, consider a homomorphic Merkle tree that commits to four messsages m_0, m_1, m_2, m_3. Then, its root $u_\perp$ and inner nodes $u_{\perp,0}$, $u_{\perp,1}$, $u_{\perp,0,0}$, $u_{\perp,0,1}$, $u_{\perp,1,0}$, $u_{\perp,1,1}$ can be calculated as follows:

$$\begin{aligned}
u_\perp &= h_{0,0}(m_0) + h_{1,0}(m_1) + h_{2,0}(m_2) + h_{3,0}(m_3)\ ; & u_{\perp,0,0} &= h_{0,2}(m_0)\\
u_{\perp,0} &= h_{0,1}(m_0) + h_{1,1}(m_1)\ ; & u_{\perp,0,1} &= h_{1,2}(m_1)\\
u_{\perp,1} &= h_{2,1}(m_2) + h_{3,1}(m_3)\ ; & u_{\perp,1,0} &= h_{2,2}(m_2)\\
& & u_{\perp,1,1} &= h_{3,2}(m_3)
\end{aligned}$$

The opening proof for m_3 is given by $(3, ((u_{\perp,0}, u_{\perp,1}), (u_{\perp,1,0}, u_{\perp,1,1})))$, and verified by checking the following equations:

$$\begin{aligned}
&\text{for } u_{\perp,1,1}\text{:} && g^{-1}(u_{\perp,1,1}) = f(m_i)\\
&\text{for } u_{\perp,1}\text{:} && g^{-1}(u_{\perp,1}) = \tilde{f}(u_{\perp,1,0},\ u_{\perp,1,1})\\
&\text{for } u_{\perp}\text{:} && g^{-1}(u_{\perp}) = \tilde{f}(u_{\perp,0},\ u_{\perp,1})
\end{aligned}$$

It now follows that when a message m_i is updated to m'_i, each inner node on the path from the leaf to the root can be updated from $u_{b_0,..,b_j}$ to $u'_{b_0,..,b_j}$ using the functions $h_{i,j}$ as follows:

$$u'_{b_0,..,b_j} \quad = \quad h_{i,j}(m'_i) + \sum_{\substack{x \neq i:\\ \text{bin}(x)[0:j-1]=(b_1,..,b_j)}} h_{x,j}(m_x) \quad = \quad u_{b_0,..,b_j} + h_{i,j}(m'_i) - h_{i,j}(m_i)$$

When the partial digest functions are linear in their input, the expression $h_{i,j}(m'_i) - h_{i,j}(m_i)$ can be written as $h_{i,j}(m'_i) - h_{i,j}(m_i) = \text{sign}(m'_i - m_i)h_{i,j}(|m'_i - m_i|)$. This lets us calculate the updated internal node using only the knowledge of the message diff $m'_i - m_i$. We provide examples of homomorphic Merkle tree constructions in Section 4.3 with linear partial digest functions $h_{i,j}$. Homomorphic Merkle proofs in these constructions consist of the two siblings of the inner nodes on the path from the proven message to the root and the vector commitment itself is given by $g^{-1}(b_\perp)$ (Section 4.3).

Unlike in Section 3.2, homomorphic Merkle trees enable calculating the new inner nodes after message updates using *only* the new and the old updated messages, in particular using only their difference. Hence, we can construct a tree that achieves the same complexity for the update information size as algebraic VCs, albeit at the expanse of the proof update complexity, *without requiring the users to keep track of the old messages or to calculate the tree from scratch given all messages.* This is in contrast to Merkle trees based on SHA256. The update and proof update algorithms of such a homomorphic Merkle tree with no structured update information and the same asymptotic complexity as algebraic VCs is described in the full version of the paper [35, Appendix B]. Since the homomorphic Merkle trees can achieve both extremes in terms of update information size and update runtime (Table 1), with a smart structuring of the update information, they can support sublinear update. We show how in the next subsection.

4.2 Structuring the Update Information

Algorithm 1 Algorithms for a homomorphic Merkle tree. Each user knows the total number of leaves N. The recursive algorithm UPDATENODE, parameterized by $\nu \in [0,1]$, takes an index as input, and checks if the new value of the node at that index is to be published as part of the update information U. If so, it appends the new value to U, and recursively calls itself on the children of the node. Not all of U and $(i, m'_i - m_i)_{i\in[N]}$ are passed to the proof update algorithm and its relevant parts are read selectively to keep the runtime at a minimum.

```
1: algorithm UPDATE($C, (i, m'_i - m_i)_{i\in[N]}, \mathcal{T}$)
2:     $U \leftarrow$ EMPTY()
3:     algorithm UPDATENODE(idx)
4:         $b_0, .., b_d \leftarrow$ idx
5:         $\mathcal{S} \leftarrow \{j \in [k] \colon 1_{\mathrm{bin}(i_j)[0:d]=(b_1,..,b_d)}\}$
6:         if $|\mathcal{S}| > k^{1-\nu}$
7:             $\mathcal{T}[b_0, .., b_d] \leftarrow \mathcal{T}[b_0, .., b_d] + \sum_{j\in\mathcal{S}} \mathrm{SIGN}(m'_{i_j} - m_{i_j}) h_{i_j,d}(|m'_{i_j} - m_{i_j}|)$
8:             $U[b_0, b_1, .., b_d] \leftarrow \mathcal{T}[b_0, b_1, .., b_d]$
9:             UPDATENODE($(b_0, .., b_d, 0)$)
10:            UPDATENODE($(b_0, .., b_d, 1)$)
11:        end if
12:    end algorithm
13:    UPDATENODE($(b_0)$)
14:    $C' \leftarrow \mathcal{T}[b_0]$
15:    return $(C', U, \mathcal{T})$
16: end algorithm
17: algorithm PROOFUPDATE($C, \pi_x, m'_x, x, U$)
18:     $\pi'_x \leftarrow \{\}$
19:     $(b_1, .., b_h) \leftarrow \mathrm{bin}(x)$
20:     for $d = h, .., 1$
21:         if $(b_0, b_1 .. \bar{b}_d) \in U$
22:             $\pi'_x[b_0, b_1 .. \bar{b}_d] \leftarrow U[b_0, b_1 .. \bar{b}_d]$
23:         else
24:             $\mathcal{S} \leftarrow \{j \in [k] \colon 1_{\mathrm{bin}(i_j)[0:d]=(b_1,..,\bar{b}_d)}\}$
25:             $\pi'_x[b_0, .., \bar{b}_d] \leftarrow \pi_x[b_0, .., \bar{b}_d] + \sum_{j\in\mathcal{S}} \mathrm{SIGN}(m'_{i_j} - m_{i_j}) h_{i_j,d}(|m'_{i_j} - m_{i_j}|)$
26:         end if
27:     end for
28:     return $\pi'_x$
29: end algorithm
```

We now describe the new update and proof update algorithms that enable homomorphic Merkle trees to achieve sublinear complexity as a function of the parameter ν (Alg. 1).

4.2.1 Update Information

When the messages $(i_j, m_{i_j})_{j\in[k]}$ change to $(i_j, m'_{i_j})_{j\in[k]}$, the update information U is generated recursively using the following algorithm:

1. Start at the root u_{b_0}. Terminate the recursion at an inner node if there are $k^{1-\nu}$ or less updated messages under that node.
2. If there are more than $k^{1-\nu}$ updated messages with indices $\geq N/2$, *i.e.*, under the right child, then publish the new right child of the root as part of U, and apply the same algorithm to the subtree rooted at the right child, with u_{b_0} and N replaced by $u_{b_0,1}$ and $N/2$ respectively.
3. If there are more than $k^{1-\nu}$ updated messages with indices less than $N/2$, *i.e.*, under the left child, then publish the new left child of the root as part of U, and apply the same algorithm to the subtree rooted at the left child, with u_{b_0} and N replaced by $u_{b_0,0}$ and $N/2$ respectively.

The new values of the inner nodes included in U are again listed from the smallest to the largest depth in the canonical left to right order.

4.2.2 Proof Update

When the messages $(i_j, m_{i_j})_{j\in[k]}$ are updated to $(i_j, m'_{i_j})_{j\in[k]}$, a user first retrieves the inner nodes within its Merkle proof that are published as part of the update information. It then calculates the non-published inner nodes within the proof using the partial digests. For instance, consider a user with the proof $(u_{\bar{b}_1}, u_{b_1,\bar{b}_2}, .., u_{b_1,b_2,..,\bar{b}_h})$ for some message m_x, $(b_1, .., b_h) = \text{bin}(x)$. To update the proof, the user first checks the update information U and replaces the inner nodes whose new values are provided by U: $u'_{b_1,..,\bar{b}_d} \leftarrow U[b_1 .. \bar{b}_d]$, $d \in [h]$, if $U[b_1 .. \bar{b}_d] \neq \perp$. Otherwise, the user finds the new values at the nodes $u_{b_1,..,\bar{b}_d}$, $d \in [h]$, using the functions $h_{x,d}$:

$$u'_{b_1,..,b_{d-1},\bar{b}_d} = u_{b_1,..,b_{d-1},\bar{b}_d} + \sum_{j\in[k]} 1_{\text{bin}(i_j)[:d]=(b_1,..,\bar{b}_d)} \left(\text{sign}(m'_{i_j} - m_{i_j}) h_{i_j,d}(|m'_{i_j} - m_{i_j}|)\right)$$

4.2.3 Complexity

Finally, we prove bounds on the complexity given by these algorithms:

▶ **Theorem 3.** *Complexity of the update information size and the runtime of proof updates are as follows: $g_1(k) = k^\nu$ and $g_2(k) = k^{1-\nu}$.*

Proof. Let $\mathcal{U}$ denote the subset of the inner nodes published by the algorithm as part of U such that no child of a node $u \in \mathcal{U}$ is published. Then, there must be over $k^{1-\nu}$ updated messages within the subtree rooted at each node $u \in \mathcal{U}$. Since there are k updated messages, and by definition of $\mathcal{U}$, the subtrees rooted at the nodes in $\mathcal{U}$ do not intersect at any node, there must be less than $k/k^{1-\nu} = k^\nu$ inner nodes in $\mathcal{U}$. Since the total number of published inner nodes is given by $\mathcal{U}$ and the nodes on the path from the root to each node $u \in \mathcal{U}$, this number is bounded by $k^\nu \log(N) = \tilde{\Theta}(k^\nu)$. Hence, $|U| = \Theta(k^\nu \log(N)(\log(N) + |H|)) = \tilde{\Theta}(k^\nu)|H| = \tilde{\Theta}(k^\nu)\lambda$, which implies $g_1(k) = k^\nu$.

For each inner node in its Merkle proof, the user can check if a new value for the node was provided as part of U, and replace the node if that is the case, in at most $\Theta(\log(N) + |H|)$ time by running a binary search algorithm over U. On the other hand, if the new value of a node in the proof is not given by U, the user can calculate the new value after at most $k^{1-\nu}\log(N)$ function evaluations. This is because there can be at most $k^{1-\nu}$ updated messages within the subtree rooted at an inner node, whose new value was not published as part of U. This makes the total time complexity of a proof update at most

$$\Theta(\log(N)(\log(N) + |H| + k^{1-\nu}\log(N)T_f)) = \tilde{\Theta}(k^{1-\nu})T_f,$$

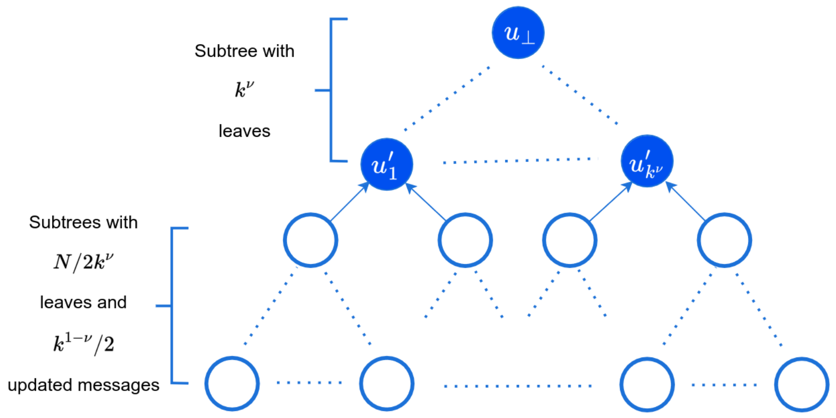

Figure 1 Homomorphic Merkle tree example. The new values of the inner nodes with solid blue color are published as part of the updated information.

which implies $g_2(k) = k^{1-\nu}$. ◀

To illustrate the proof above, consider the homomorphic Merkle tree in Figure 1 where k messages are updated. Suppose there are $k^{1-\nu}/2$ updated messages among the first $N/2k^\nu$ messages $m_0, .., m_{N/2k^\nu-1}$, another $k^{1-\nu}/2$ updated messages among the second $N/2k^\nu$ messages $m_{N/2k^\nu}, .. m_{2N/2k^\nu-1}$ and so on. In this case, the algorithm identifies the inner nodes within the subtree at the top of the tree (whose nodes are denoted in solid blue) and publishes their new values as part of the update information. This is because there are $k^{1-\nu}$ updated messages under each inner node and leaf of this subtree, denoted by u'_i, $i = 1, .., k^\nu$, whereas under the children of these leaf nodes there are less than $k^{1-\nu}$ updated messages. Thus, each user can update its opening proof by downloading the new values of the top $\log k^\nu$ inner nodes within its proof from the update information. There are at most $k^{1-\nu}/2$ updated messages under each of the remaining $\log N/k^\nu$ nodes in the proof; hence, the user can find their updated values in $\Theta(k^\nu \log N)$ time. Note that in this example, and in general when the updated messages are distributed uniformly among the leaves, the size of the update information becomes $\Theta(k^\nu)\lambda$ rather than $\Theta(k^\nu \log N)\lambda$.

4.3 Constructions for Homomorphic Merkle Trees

Homomorphic Merkle trees were proposed by [30, 29, 32]. They use lattice-based hash functions, and their collision-resistance is proven by reduction to the hardness of the gap version of the shortest vector problem (GAPSVP_γ), which itself follows from the hardness of the small integer solution problem. We next describe the construction introduced by [30], which is similar to those proposed by later works [29, 32]. Its correctness and security follow from [30, Theorem 4].

Let $L(\mathbf{M})$ denote the lattice defined by the basis vectors $\mathbf{M} \subset \mathbb{Z}_q^{k\times m}$ for appropriately selected parameters k, m, q, where $m = 2k \log q$. Consider vectors $u \in \{0, .., t\}^{k \log q}$, where t is a small integer. The (homomorphic) hash functions $f \colon \mathbb{Z}^{k \log q} \to L(\mathbf{M})$ and $\tilde{f} \colon \mathbb{Z}^{k \log q} \times \mathbb{Z}^{k \log q} \to L(\mathbf{M})$ used by [30] are defined as $f(x) = \mathbf{M}x$ and $\tilde{f}(x, y) = \mathbf{MU}x + \mathbf{MD}y$ respectively. Here, $\mathbf{U}$ and $\mathbf{D}$ are special matrices that double the dimension of the multiplied vector and shift it up or down respectively. The remaining entries are set to zero. For convenience, we define $\mathbf{L} = \mathbf{MU}$ and $\mathbf{R} = \mathbf{MD}$.

Since the domain and range of the hash functions are different, to ensure the Merkle tree's homomorphism, authors define a special mapping $g \colon \mathbb{Z}_q^k \to \mathbb{Z}_q^{k \log q}$ from the range of the hash functions to their domain. Here, $g(.)$ takes a vector $\mathbf{v} \in \mathbb{Z}_q$ as input and outputs *a* radix-2 representation for $\mathbf{v}$. However, as there can be many radix-2 representations of a vector, to help choose a representation that yields itself to homomorphism, authors prove the following result: for any $\mathbf{x}_1, \mathbf{x}_2, .., \mathbf{x}_t \in \mathbb{Z}_q$, there exists *a short* radix-2 representation $g(.)$ such that $g(\mathbf{x}_1 + \mathbf{x}_2 + .. + \mathbf{x}_t \mod q) = b(\mathbf{x}_1) + b(\mathbf{x}_2) + .. + b(\mathbf{x}_t) \mod q \in \{0, .., t\}^{k \log q}$, where the function $b\colon \mathbb{Z}_q^k \to \{0,1\}^{k \log q}$ returns the binary representation of the input vector. This equality enables the mapping $g(.)$ to *preserve* the hash functions' original homomorphic property. Then, given an inner node $u_{b_0,..,b_j}$ as input, the homomorphic Merkle tree uses the short radix-2 representation $g(.)$ that enforces the following equality: $u_{b_0,..,b_j} = g(\mathbf{L}u_{b_0,..,b_j,0} + \mathbf{R}u_{b_0,..,b_j,1} \mod q) = b(\mathbf{L}u_{b_0,..,b_j,0}) + b(\mathbf{R}u_{b_0,..,b_j,1}) \mod q$. Finally, this enables calculating the value of each inner node as a sum of the partial digests $h_{i,j}(.)$ of the messages m_i under the node $u_{b_0,..,b_j}$ (*i.e.*, $(m_i)_{\text{bin}(i)[0:j]=(b_0,..,b_j)}$) as outlined in Section 4.1, i.e.,

$$u_{b_0,..,b_j} = \sum_{i\colon \text{bin}(i)[0:j-1]=(b_1,..,b_j)} h_{i,j}(m_i),$$

where $h_{i,j}(.)$ is expressed in terms of the bits $\text{bin}(i)[j{:}h-1] = (b'_1, .., b'_{h-j})$:

$$h_{i,j}(m_i) = f_{b'_1}(f_{b'_2}(.. f_{b'_{h-j}}(b(f(m_i)))))$$

Here, $f_0(.)$ and $f_1(.)$ are defined as $b(\mathbf{L}.)$ and $b(\mathbf{R}.)$ respectively. Since $b(.)$, binary expansion, is a linear operation and matrix multiplication is linear, $h_{i,j}(.)$ is linear in its input.

4.4 A Concrete Evaluation

Suppose the Ethereum state is persisted using the homomorphic Merkle tree construction of [29, 32] with the trade-off parameter $\nu = 1/2$. We next estimate the size of the update information and the proof update time after observing an Ethereum block with ERC20 token transfers. Suppose the block has the target size of 15 million gas [4], and each token transfer updates the balance of two distinct accounts stored at separate leaves of the homomorphic Merkle tree. Since each ERC20 token transfer consumes approximately 65,000 gas, there are ~ 230 such transactions in the block, and the block updates $k = 460$ accounts.

Suppose the homomorphic Merkle tree has degree 2 and commits to $N = 256^3 = 2^{24}$ accounts. For comparison, $256^3 \approx 16.7$ million, matching in magnitude the total number of cumulative unique Ethereum addresses, which is 200 million as of 2023 [3]. Each opening proof consists of $2 \log(N) = 48$ inner nodes.

When 460 accounts are updated, in the worst case, the update information consists of $\lceil\sqrt{k}\rceil \log(N) = 528$ inner nodes. To evaluate its size, we use the parameters calculated by [32] for secure instantiations of the homomorphic Merkle trees from both their paper and [29]. Since the parameters for [29] result in a large inner node size on the order of hundreds of MBs, our evaluation takes the size of an inner node as that of [32], namely $|H| = 0.21$ MB (which is equal to the key size in [32]). This implies an update information size of $|U| = 110.88$ MBytes and an opening proof size of $|\pi| = 10.08$ MBytes.

As for update time, in the worst case, each user has to calculate the partial digests of 44 updated messages at each height of the homomorphic Merkle tree, *i.e.*, the effect of these updated messages on each inner node of its opening proof. Calculating the partial digest of a message at height h measured from the leaves requires h evaluations of the hash function. This implies a proof update complexity of $2\sum_{i=0}^{\log N-1} i \min(\lceil\sqrt{k}\rceil, 2^i) = 11,900$ hash evaluations. To find numerical upper bounds for the update time, we use the hash

Table 2 For different trade-off points between the update information size and proof update complexity, parameterized by ν, the table shows the number of published inner nodes $\lceil k^\nu \rceil \log(N)$, the total update information size $\lceil k^\nu \rceil \log(N)|H|$, the number of hash function evaluations per proof update $2\sum_{i=0}^{\log N-1} i \min(\lceil k^{1-\nu} \rceil, 2^i)$ and the proof update time $2\sum_{i=0}^{\log N-1} i \min(\lceil k^{1-\nu} \rceil, 2^i)T_f$. There are $N = 2^{24}$ accounts in total, $k = 460$ updates at the accounts, the inner nodes have size $|H| = 0.21$ Mbytes, and a hash function evaluation takes $T_f = 2.74$ ms.

ν	# inner nodes	$\lvert U \rvert$ (MBytes)	# hash evaluations	Time (s)
0	1	0.21	227,972	624.6
1/4	120	25.20	52,284	143.3
1/2	528	110.88	11,900	32.6
3/4	2400	504.00	2750	7.54
1	11040	2318.40	552	1.51

function evaluation times, namely $T_f = 26.84$ and $T_f = 2.74$ ms, published by [32] for both the hash function in [29] and their new and more performant function (these times are for commodity hardware; *cf.* [32] for the details). This gives an upper bound of 319.4 and 32.6 seconds for the update time using the hash functions in [29] and [32] respectively.

Based on the benchmarks for the practical hash function introduced in [32], Table 2 compares the number of published inner nodes $\lceil k^\nu \rceil \log(N)$, the total update information size $\lceil k^\nu \rceil \log(N)|H|$ (assuming that the size of each inner node is $|H|$ upper bounded by 0.21 MBytes), the number of hash function evaluations per proof update $2\sum_{i=0}^{\log N-1} i \min(\lceil k^{1-\nu} \rceil, 2^i)$ and the proof update time $2\sum_{i=0}^{\log N-1} i \min(\lceil k^{1-\nu} \rceil, 2^i)T_f$ (assuming that each hash evaluation takes less than $T_f = 2.74$ ms) at $\nu = 0, 1/4, 1/2, 3/4, 1$. The degree of the homomorphic Merkle tree and the opening proof size are fixed at 2 and 48 inner nodes ($|\pi| = 10.08$) respectively.

5 Updating Verkle Trees and Opening Proofs

We now describe the update and proof update functions for Verkle trees (see Algs. 2 and 3 in the full version of the paper [35, Section 5] for update and proof update algorithms respectively). Since Verkle trees were proposed to support stateless clients, we describe an update scheme that minimizes the runtime complexity of proof updates and does not require the users to download the updated messages or have access to old inner nodes. As Verkle trees do not support sublinear update, we numerically estimate the size of the update information and the complexity of proof updates in Section 5.5.

5.1 Update Information

Suppose the vector $(i, m_i)_{i\in[N]}$ is modified at some index x, $(b_1, .., b_h) = \text{bin}(x)$ to be m'_x. Since each inner node is the hash of a KZG commitment, the new inner nodes $u'_{b_0,..,b_j} = H(C'_{b_0,..,b_j})$, $j \in [h]$, can be found as a function of the old commitments at the nodes and the powers of the Lagrange basis polynomials as described in Section 3.1:

$$C'_{b_0,..,b_h} \leftarrow m'_x, \qquad C'_{b_0,..,b_j} \leftarrow C_{b_0,..,b_j}[L_{b_{j+1}}]^{(u'_{b_0,...,b_{j+1}} - u_{b_0,...,b_{j+1}})}$$

When k messages are updated, the above calculation is repeated k times for each update.

Update information U consists of the new values of the KZG commitments on the path from the updated messages to the Verkle root akin to the Merkle trees, ordered in the canonical top-to-bottom and left-to-right order.

5.2 Verkle Proofs

Let π_x denote the Verkle proof of some message m_x at index x, $(b_1,..,b_h) = \text{bin}(x)$: $\pi_x = ((C_{b_0,b_1},..,C_{b_0,..,b_{h-1}}),([g(X)],\pi))$. We define π_x^f as the opening proof for index x within polynomial f. We observe that the commitment $[g(X)]$ and the proof π can be expressed as functions of the opening proofs of the inner nodes $u_{b_0,b_1},..,u_{b_0,..,b_h}$ at the indices $b_1,..,b_h$ within the polynomials $f_{b_0},..,f_{b_0,..,b_{h-1}}$, respectively. Namely, $[g(X)]$ is

$$\left[\sum_{j=0}^{h-1} r^j \frac{f_{b_0,..,b_j}(X) - u_{b_0,..,b_{j+1}}}{X - b_{j+1}}\right] = \prod_{j=0}^{h-1} \left[\frac{f_{b_0,..,b_j}(X) - u_{b_0,..,b_{j+1}}}{X - b_{j+1}}\right]^{r^j} = \prod_{j=0}^{h-1} \left(\pi_{b_{j+1}}^{f_{b_0,..,b_j}}\right)^{r^j}$$

Similarly, the opening proof $\pi = \pi_t^{(h-g)}$ for index t within the polynomial $h(X) - g(X)$ can be expressed as follows (for details, see the full version of the paper [35, Appendix E]):

$$\begin{aligned}\left[\frac{h(X) - g(X) - (h(t) - g(t))}{X - t}\right] &= \prod_{j=0}^{h-1} \left[\frac{f_{b_0,..,b_j}(X) - u_{b_0,..,b_{j+1}}}{X - b_{j+1}}\right]^{\frac{r^j}{t-b_{j+1}}} \\ &= \prod_{j=0}^{h-1} \left(\pi_{b_{j+1}}^{f_{b_0,..,b_j}}\right)^{\frac{r^j}{t-b_{j+1}}}\end{aligned}$$

We assume that each user holding the Verkle proof π_x for some index x, $(b_1,..,b_h) = \text{bin}(x)$, also holds the opening proofs $\pi_{b_{j+1}}^{f_{b_0,..,b_j}}$, $j \in [h]$, *in memory*. As we will see in the next section, the user also holds the KZG commitments at the children of the inner nodes on the path from the root to the message m_x, *i.e.* $C_{b_0,..,b_j,i}$ for all $j \in [h]$ and $i \in [c]$ *in memory*. These opening proofs and KZG commitments are not broadcast as part of any proof; however, they are needed for the user to locally update its Verkle proof after message updates.

5.3 Proof Update

When the messages $(i_j, m_{i_j})_{j\in[k]}$ are updated to $(i_j, m'_{i_j})_{j\in[k]}$, to calculate the new Verkle proof π'_x, the user must obtain the new commitments $C'_{b_0},..,C'_{b_0,..,b_{h-1}}$ on the path from the root to message m_x, the new commitment $[g'(X)]$ and the new opening proof π' for the polynomial $h'(X) - g'(X)$ at index $t' = H(r', [g'(X)])$. Message updates change the commitments at the inner nodes, which in turn results in new polynomials $f_{b_0,..,b_j}$, $j \in [h]$. Suppose each polynomial $f_{b_0,..,b_j}$, $j \in [h]$, is updated so that

$$f'_{b_0,..,b_j}(X) = f_{b_0,..,b_j}(X) + \sum_{i=0}^{c-1} (f'_{b_0,..,b_j}(i) - f_{b_0,..,b_j}(i)) L_i(X),$$

where, by definition, $f'_{b_0,..,b_j}(i) - f_{b_0,..,b_j}(i) = u'_{b_0,..,b_j,i} - u_{b_0,..,b_j,i} = H(C'_{b_0,..,b_j,i}) - H(C_{b_0,..,b_j,i})$. Then, given the new and the old commitments $(C_{b_0,..,b_j,i}, C'_{b_0,..,b_j,i})$ for $i \in [c]$ and $j \in [h]$, the table of Lagrange basis polynomials, and using the technique in Section 3.1, the new opening proofs $\tilde{\pi}_{b_{j+1}}^{f_{b_0,..,b_j}}$ after the message updates can be computed as follows for $j \in [h]$:

$$\tilde{\pi}_{b_{j+1}}^{f_{b_0,..,b_j}} = \pi_{b_{j+1}}^{f_{b_0,..,b_j}} \prod_{i=0}^{c-1} \left[\frac{L_i(X) - L_i(b_{j+1})}{X - b_{j+1}}\right]^{(H(C'_{b_0,..,b_j,i}) - H(C_{b_0,..,b_j,i}))},$$

where $\left[\frac{L_i(X)-L_i(b_{j+1})}{X-b_{j+1}}\right]$ is the opening proof of the Lagrange basis polynomial $L_i(X)$ at index b_{j+1}. Once the new opening proofs are found, the new commitment $[g'(X)]$ and the new proof π' become

$$[g'(X)] = \prod_{j=0}^{h-1} \left(\tilde{\pi}^{f_{b_0,\dots,b_j}}_{b_{j+1}}\right)^{r'^j}, \qquad \pi' = \prod_{j=0}^{h-1} \left(\tilde{\pi}^{f_{b_0,\dots,b_j}}_{b_{j+1}}\right)^{\frac{r'^j}{t'-b_{j+1}}}$$

where $r' = H(C'_{b_0,b_1},..,C'_{b_0,..,b_{h-1}},u'_{b_0,b_1},..,u'_{b_0,..,b_h},b_1,..,b_h)$ and $t' = H(r',[g'(X)])$. Note that both r' and t' can be calculated by the user given the new KZG commitments $C'_{b_0,..,b_j,i}$ for all $i \in [c]$ and $j \in [h]$.

Finally, to retrieve the new KZG commitments $C'_{b_0,..,b_j,i}$ for all $i \in [c]$ and $j \in [h]$, the user inspects the commitments published as part of the update information U: $C'_{b_0,b_1,..,b_{j-1},i} \leftarrow U[b_0,b_1,..,b_{j-1},i]$ if $U[b_0,b_1,..,b_{j-1},i] \neq \perp$ and $C'_{b_0,b_1,..,b_{j-1},i} \leftarrow C_{b_0,b_1,..,b_{j-1},i}$ otherwise, for all $i \in [c]$ and $j \in [h]$.

In Verkle trees, the user cannot calculate the effect of an updated message on an arbitrary inner node without the knowledge of the inner nodes on the path from the message to the target node. For instance, suppose $U[b_0,b_1,..,b_{j-1},i] = \perp$ for some $i \in [c]$ and $j \in [h]$, and the user wants to calculate the effect of an update from m_x to m'_x on $C'_{b_0,..,b_{j-1},i,\tilde{b}_{j+1},..,\tilde{b}_h}$, $\text{bin}(x) = (b_1,..,b_{j-1},i,\tilde{b}_{j+1},..,\tilde{b}_h)$ and $\tilde{b}_j = i$. Then, for each $\ell \in \{j,..,h-1\}$, the user have to find

$$C'_{b_0,..,\tilde{b}_j,..,\tilde{b}_h} \leftarrow m'_x$$
$$C'_{b_0,..,\tilde{b}_j,..,\tilde{b}_\ell} \leftarrow C_{b_0,..,\tilde{b}_j,..,\tilde{b}_\ell}[L_{\tilde{b}_{\ell+1}}]^{(u'_{b_0,..,\tilde{b}_j,..,\tilde{b}_{\ell+1}}-u_{b_0,..,\tilde{b}_j,..,\tilde{b}_{\ell+1}})},$$

where $C'_{b_0,..,\tilde{b}_j,..,\tilde{b}_\ell}$ are the commitments on the path from the target commitment $C_{b_0,b_1,..,b_{j-1},i}$ to the message m_x. Hence, the user has to know the original commitments on the path from the message to the target commitment, *i.e.*, keep track of inner nodes, which contradicts with the idea of stateless clients. This shows the necessity of publishing all of the updated inner nodes as part of the update information.

5.4 Complexity

Suppose each KZG commitment is of size $|G|$ and each hash $H(C)$ of a KZG commitment, *i.e.* each inner node, has size $|H|$. Then, updating a single message results in one update at each level of the Verkle tree and requires $\Theta(h|H|)$ group operations. Thus, when k messages are updated, the new Verkle root can be found after $\Theta(kh|H|)$ group operations. As U consists of the published KZG commitments at the inner nodes and their indices, $|U| = \Theta(k \log_c(N)(\log(N)+|G|)) = \tilde{\Theta}(k)|G|$, which implies $g_1(k) = k$.

The user can replace each KZG commitment at the children of the inner nodes from the root to its message in $\Theta(\log(N)+|G|)$ time by running a binary search algorithm over U. Since there are ch such commitments to be updated, *i.e.*, $C_{b_0,..,b_j,i}$, $i \in [c]$ and $j \in [h]$, updating these commitments takes $\Theta(ch(\log(N)+|G|)) = \tilde{\Theta}(1)$ time.

Upon obtaining the new commitments $C'_{b_0,..,b_{j-1},i}$, $i \in [c]$, $j \in [h]$, with access to the table of Lagrange basis polynomials, the user can update each opening proof $\pi_{b_{j+1}}$ (for the function $f_{b_0,..,b_j}$), $j \in [h]$, with $\Theta(c|H|)$ group operations. Since there are h such proofs, updating them all requires $\Theta(ch|H|)$ group operations. Given the new proofs, computing the new commitment $[g'(X)]$ and proof π' requires $\Theta(h|H|)$ group operations. This makes the total complexity of updating a Verkle proof $\Theta(ch+2h)|H|T_G + \Theta(ch(\log_c(N)+|G|))$. For a constant c and $h = \log_c(N)$, this implies a worst-case time complexity of $\tilde{\Theta}(1)|H|T_G$ for Verkle proof updates, *i.e.*, $g_2(k) = 1$.

5.5 A Concrete Evaluation

We now estimate the size of the update information and the number of group operations to update an opening proof after observing an Ethereum block consisting of ERC20 token transfers. As in Section 4.4, suppose the block has the target size of 15 million gas [4], and each token transfer updates the balance of two distinct accounts stored at separate leaves of the Verkle tree. Then, there are ~ 230 such transactions in the block, and the block updates $k = 460$ accounts. We assume that the Verkle tree has degree 256 (*cf.* [11]) and commits to 256^3 accounts as in Section 4.4. Then, each proof consists of 2 KZG commitments, $C_{\perp,b_1}$ and $C_{\perp,b_1,b_2}$ and a multiproof consisting of the commitment $[g(X)]$ and proof π'. These components are elements of the pairing-friendly elliptic curve BLS12_381 and consist of $|G| = 48$ bytes [11]. This implies a proof size of $(\log_c(N) + 1)|G| = 192$ bytes (excluding the message at the leaf and its hash value; adding those makes it 272 bytes).

When 460 accounts are updated, in the worst-case, the update information has to contain $k\log_c(N)(\log(N) + |G|) = 460 \times 3 \times (24 + 48)$ Bytes, *i.e.*, 99.4 kBytes. This is comparable to the size of the Ethereum blocks, which are typically below 125 kBytes [2]. Hence, even though the update information of Verkle trees is linear in k, it does not introduce a large overhead beyond the block data. Note that the runtime of the proof updates are constant and do not scale in the number of updated messages k, or the Ethereum block size.

On the other hand, in the worst case, an opening proof can be updated after $c\log(c)|H| + 2\log_c(N)|H|$ group operations. Then, with $|H| = 256$, the number of bits output by SHA256, as many as $c\log_c(N)|H| + 2\log_c(N)|H| = (c+2)\log_c(N)|H| = 774 \times 2256 \approx 200,000$ elliptic curve multiplications might have to be made. Following the benchmarks published in [1] for the specified curve, these operations can take up to $(c+2)\log_c(N)$ 0.000665471 ns $= 0.52$ seconds on commodity hardware, given a runtime of 665471 nanoseconds per exponentiation of a group element with a message hash value. This is again comparable to the 12 second inter-arrival time of Ethereum blocks.

Table 3 compares the Verkle proof size $|\pi| = (\log_c(N) + 1)|G|$, update information size $|U| = k\log_c(N)(\log_c N + |G|)$, the upper bound $(c+2)\log_c N|H|$ on the number of group operations needed for a single proof update and the estimated time it takes to do these operations on a commodity hardware for different values of c, the Verkle tree degree, while keeping the number of accounts and the updated accounts fixed at 2^{24} and 460 respectively. The table shows the trade-off between the Verkle proof and update information size on one size and update complexity on the other.

Comparing Table 3 with Table 2 shows that the Verkle tree with any given degree c, $1 < c \leq 256$, significantly outperforms the existing homomorphic Merkle trees in Section 4.4 in terms of almost all of proof size, update information size and proof update time.

6 Lower Bound

Finally, we prove the optimality of our VC scheme with sublinear update by proving a lower bound on the size of the update information given an upper bound on the complexity of proof updates. The lower bound is shown for VCs that satisfy the following *proof-binding* property. It formalizes the observation that for many dynamic VCs (*e.g.*, Merkle trees [25], Verkle trees [11], KZG commitments [19], RSA based VCs [8]) including homomorphic Merkle trees (*cf.* the full version of the paper [35, Section 6]), the opening proof for a message at some index can often act as a commitment to the vector of the remaining messages.

Table 3 For different values of the tree degree c, the table shows the Verkle proof size which is $|\pi| = (\log_c(N)+1)|G|$; the update information size which is $|U| = k\log_c(N)(\log(N)+|G|)$; the number of group operations for a single proof update which is $(c+2)\log_c(N)|H|$; and the estimated time for a single proof update. We use $N = 2^{24}$ accounts in total, $k = 460$ updates at the accounts, a group element size of $|G| = 48$ bytes, and a hash size of $|H| = 32$ bytes.

c	$\|\pi\|$ (Bytes)	$\|U\|$ (kBytes)	# Group Operations	Time (s)
2	1200	794.9	24,576	0.064
4	628	397.4	18,432	0.048
16	336	198.7	27,648	0.072
64	240	132.5	67,584	0.18
256	192	99.4	198,144	0.52

▶ **Definition 4.** *A VC scheme is said to be* proof-binding *if the following probability is negligible in λ for all PPT adversaries $\mathcal{A}$:*

$$\Pr\left[\begin{array}{r} \textsc{Verify}_{pp}(C,m_{i^*},i^*,\pi)=1 \\ \wedge\textsc{Verify}_{pp}(C',m_{i^*},i^*,\pi)=1 \end{array} : \begin{array}{r} pp\leftarrow\textsc{KeyGen}(1^\lambda,N); \\ \pi,m_{i^*},(m_0,..,m_{i^*-1},m_{i^*+1},..,m_{N-1}), \\ (m'_0,..,m'_{i^*-1},m'_{i^*+1},..,m'_{N-1})\leftarrow\mathcal{A}(pp); \\ (m_0,..,m_{i^*-1},m_{i^*+1},..,m_{N-1}) \\ \neq(m'_0,..,m'_{i^*-1},m'_{i^*+1},..,m'_{N-1}); \\ \textsc{Commit}_{pp}(m_0,..,m_{i^*-1},m_{i^*},m_{i^*+1},..,m_{N-1})=C; \\ \textsc{Commit}_{pp}(m'_0,..,m'_{i^*-1},m_{i^*},m'_{i^*+1},..,m'_{N-1})=C' \end{array}\right]$$

In the definition above, the opening proof while committing to the vector of remaining messages, need not be of the same format as the original VC. For instance, for RSA accummulators, the opening proof is itself an RSA accummulator, whereas for Merkle trees, the opening proof is not a Merkle tree root, but contains a sequence of inner nodes and the index of the opened message. Nevertheless, the proof and the message together act as a commitment to the vector of remaining messages.

The proof-binding property is distinct from the position-binding (security) property of VCs: whereaas position-binding states the difficulty of opening the VC to two different messages at the same index, proof-binding implies the difficulty of creating two VCs with different messages that open to the *same* message at some index i with the exact *same* proof.

We next state the main lower bound for proof-binding VCs.

▶ **Theorem 5.** *Consider a dynamic and proof-binding VC such that for every PPT adversary $\mathcal{A}$, it holds that*

$$\Pr\left[\begin{array}{c} \textsc{Verify}_{pp}(C,m,i,\pi_i)=1\wedge \\ \textsc{Verify}_{pp}(C,m',i,\pi'_i)=1\ \wedge\ m\neq m' \end{array} : \begin{array}{c} pp\leftarrow\textsc{KeyGen}(1^\lambda,N) \\ (C,m,m',\pi_i,\pi'_i)\leftarrow\mathcal{A}(pp) \end{array}\right] \leq e^{-\Omega(\lambda)}.$$

Then, for this VC, if $g_2(k) = O(k^{1-\nu})$, then $g_1 = \Omega(k^\nu)$ for all $\nu \in (0,1)$.

Proof of Theorem 5 is given in the full version of the paper [35, Section 6].

▶ Remark 6. Theorem 5 shows that the update information length scales as $\tilde{\Theta}(k^\nu\lambda)$ when the runtime complexity for proof updates is $\tilde{\Theta}(k^{1-\nu})$ and the error probability for the security of the VC is $e^{-\Omega(\lambda)}$ for a PPT adversary. When the error probability is just stated to be negligible in λ, then the same proof can be used to show that the update information length must scale as $\Omega(k^\nu \operatorname{polylog}(\lambda))$ for any polynomial function of $\log(\lambda)$.

References

1 benchmarks-bls-libs. URL: `https://github.com/AlexiaChen/benchmarks-bls-libs`.

2 Ethereum average block size chart. URL: `https://etherscan.io/chart/blocksize`.

3 Ethereum cumulative unique addresses. URL: `https://ycharts.com/indicators/ethereum_cumulative_unique_addresses`.

4 Gas and fees. URL: `https://ethereum.org/en/developers/docs/gas/`.

5 State trie. URL: `https://ethereum.github.io/execution-specs/autoapi/ethereum/frontier/trie/index.html`.

6 Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. In *ASIACRYPT (3)*, volume 12493 of *Lecture Notes in Computer Science*, pages 839–869. Springer, 2020.

7 Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *EUROCRYPT*, volume 765 of *Lecture Notes in Computer Science*, pages 274–285. Springer, 1993.

8 Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *CRYPTO (1)*, volume 11692 of *Lecture Notes in Computer Science*, pages 561–586. Springer, 2019.

9 Vitalik Buterin. The stateless client concept, 2017. URL: `https://ethresear.ch/t/the-stateless-client-concept/172`.

10 Vitalik Buterin. A state expiry and statelessness roadmap, 2021. URL: `https://notes.ethereum.org/@vbuterin/verkle_and_state_expiry_proposal`.

11 Vitalik Buterin. Verkle trees, 2021. URL: `https://vitalik.ca/general/2021/06/18/verkle.html`.

12 Philippe Camacho and Alejandro Hevia. On the impossibility of batch update for cryptographic accumulators. In *LATINCRYPT*, volume 6212 of *Lecture Notes in Computer Science*, pages 178–188. Springer, 2010.

13 Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public Key Cryptography*, volume 7778 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2013.

14 Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-knowledge sets with short proofs. In *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 433–450. Springer, 2008.

15 Miranda Christ and Joseph Bonneau. Limits on revocable proof systems, with applications to stateless blockchains. *IACR Cryptol. ePrint Arch.*, page 1478, 2022. Appeared in the International Conference on Financial Cryptography and Data Security 2023.

16 Dankrad Feist. Pcs multiproofs using random evaluation, 2021. URL: `https://dankradfeist.de/ethereum/2021/06/18/pcs-multiproofs.html`.

17 Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. In *CCS*, pages 2007–2023. ACM, 2020.

18 Mahabir Prasad Jhanwar and Reihaneh Safavi-Naini. Compact accumulator using lattices. In *SPACE*, volume 9354 of *Lecture Notes in Computer Science*, pages 347–358. Springer, 2015.

19 Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.

20 John Kuszmaul. Verkle trees, 2018. URL: `https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf`.

21 Russell W. F. Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *CRYPTO (1)*, volume 11692 of *Lecture Notes in Computer Science*, pages 530–560. Springer, 2019.

22 Benoît Libert, San Ling, Khoa Nguyen, and Huaxiong Wang. Zero-knowledge arguments for lattice-based accumulators: Logarithmic-size ring signatures and group signatures without trapdoors. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 2016.

23 Benoît Libert, Somindu C. Ramanna, and Moti Yung. Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In *ICALP*,

volume 55 of *LIPIcs*, pages 30:1–30:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

24 Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 499–517. Springer, 2010.

25 Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

26 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL: `https://bitcoin.org/bitcoin.pdf`.

27 Lan Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2005.

28 Anca Nitulescu. Sok: Vector commitments. URL: `https://www.di.ens.fr/~nitulesc/files/vc-sok.pdf`.

29 Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming authenticated data structures. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 353–370. Springer, 2013.

30 Charalampos Papamanthou and Roberto Tamassia. Cryptography for efficiency: Authenticated data structures based on lattices and parallel online memory checking. *IACR Cryptol. ePrint Arch.*, page 102, 2011.

31 Chris Peikert, Zachary Pepin, and Chad Sharp. Vector and functional commitments from lattices. In *TCC (3)*, volume 13044 of *Lecture Notes in Computer Science*, pages 480–511. Springer, 2021.

32 Yi Qian, Yupeng Zhang, Xi Chen, and Charalampos Papamanthou. Streaming authenticated data structures: Abstraction and implementation. In *CCSW*, pages 129–139. ACM, 2014.

33 Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. In *USENIX Security Symposium*, pages 3001–3018. USENIX Association, 2022.

34 Shravan Srinivasan, Ioanna Karantaidou, Foteini Baldimtsi, and Charalampos Papamanthou. Batching, aggregation, and zero-knowledge proofs in bilinear accumulators. In *CCS*, pages 2719–2733. ACM, 2022.

35 Ertem Nusret Tas and Dan Boneh. Vector commitments with efficient updates. *arXiv:2307.04085*, 2023. URL: `https://arxiv.org/abs/2307.04085`.

36 Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *SCN*, volume 12238 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2020.

37 Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated dictionaries with cross-incremental proof (dis)aggregation. *IACR Cryptol. ePrint Arch.*, page 1239, 2020.

38 Hoeteck Wee and David J. Wu. Succinct vector, polynomial, and functional commitments from lattices. *IACR Cryptol. ePrint Arch.*, page 1515, 2022.

39 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. URL: `https://files.gitter.im/ethereum/yellowpaper/VIyt/Paper.pdf`.

40 Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew K. Miller. hbACSS: How to Robustly Share Many Secrets. In *NDSS*. The Internet Society, 2022.