BalanceProofs: Maintainable Vector Commitments with Fast Aggregation

Weijie Wang
Yale University

Annie Ulichney
Yale University

Charalampos Papamanthou
Yale University

Abstract

We present BalanceProofs, the first vector commitment that is *maintainable* (i.e., supporting sublinear updates) while also enjoying fast proof aggregation and verification. The basic version of BalanceProofs has $O(\sqrt{n}\log n)$ update time and $O(\sqrt{n})$ query time and its constant-size aggregated proofs can be produced and verified in milliseconds. In particular, BalanceProofs improves the aggregation time and aggregation verification time of the only known maintainable and aggregatable vector commitment scheme, Hyperproofs (USENIX SECURITY 2022), by up to $1000\times$ and up to $100\times$ respectively. Fast verification of aggregated proofs is particularly useful for applications such as stateless cryptocurrencies (and was a major bottleneck for Hyperproofs), where an aggregated proof of balances is produced once but must be verified multiple times and by a large number of nodes. As a limitation, the updating time in BalanceProofs compared to Hyperproofs is roughly $6\times$ slower, but always stays in the range from 10 to 18 milliseconds. We finally study useful tradeoffs in BalanceProofs between (aggregate) proof size, update time and (aggregate) proof computation and verification, by introducing a bucketing technique, and present an extensive evaluation as well as a comparison to Hyperproofs.

1 Introduction

Vector commitments (VC) is a cryptographic primitive recently proposed as a powerful alternative to traditional Merkle trees [24], due to their additional attractive properties, such as compact, even constant-size proofs, efficient and homomorphic updates as well as the ability to aggregate proofs into a single object. Catalano and Fiore [12] were the first to formalize the notion of VCs. In a VC scheme, a *prover* computes a succinct commitment C of a vector $\mathbf{m} = [m_0, \ldots, m_{n-1}]$ and proofs $\pi_0, \ldots, \pi_{n-1}$ for each position. A *verifier* who has the commitment C can later verify a proof π_i attesting that m_i is the correct value at position i. As with other commitment schemes, VCs maintain the *binding* property that ensures that an adversary cannot forge a commitment or a proof and convince the *verifier* of false information (e.g., that the value of index i is m'_i, instead of m_i). Inspired by applications of VCs, such as stateless cryptocurrencies and proof-of-space protocols (e.g., [1, 2, 6, 11, 13, 21, 22, 25, 31, 33, 35, 37, 42]), in this paper we are interested in two features of VCs, *maintainability* and *aggregatability*, which were recently explored by Srinivasan et al. in their *Hyperproofs* work [33].

A VC scheme is *maintainable*, if the commitment C and all proofs can be updated efficiently (in sublinear time) after receiving an update to one position of the original vector (Typically the sublinear time is achieved by maintaining a data structure that efficiently stores overlapping parts of the proofs, e.g., [33].) A VC scheme is *aggregatable*, if, given an index set I, the prover can take several individual proofs π_i for $i \in I$ and aggregate them into a single, succinct proof π_I efficiently. There are several VC schemes that are maintainable *but not* aggregatable [24, 28, 30, 36, 38]. For example, Merkle trees [24] or the vector commitment by Tomescu [36] are such schemes: While one can update proofs in $O(\log n)$ time, no algorithms are known for proof aggregation. Similarly, there are VC schemes that are aggregatable *but not* maintainable. For example, the vector commitment scheme by Tomescu et al. [37] (referred to as aSVC for the rest of the paper), based on the KZG polynomial commitment [19] as well as the recently proposed *Pointproofs* [15] support proof aggregation but their updates take linear time.

Naturally, there is a fundamental question as to whether we can build a vector commitment that is both *maintainable* and *aggregatable*. To the best of our knowledge, Hyperproofs [33] is the only work to satisfy both properties. In Hyperproofs, aggregation and verification times both take sublinear time. However, the practical aggregation and verification costs of Hyperproofs are very large (e.g., about $100\times$ to $1000\times$ larger than aggregation using other VCs such as aSVC [37]). This could limit the applicability of Hyperproofs in cryptocurrencies where the aggregated proof computed by the miner that finds the next block must be verified by all the nodes in the distributed blockchain. The main reason for the increased aggregation and verification cost is the almost black-box use of an inner-product argument [8] used to produce the aggregate proof. In this paper we are therefore interested in the following question:

Can we build a vector commitment scheme that is both maintainable and naturally aggregatable?

(Here, by "natural aggregation" we refer to the goal of avoiding the use of any black-box arguments in implementing the aggregation—this can lead to significant practical improvements in the aggregation and verification time.) Our work answers the above question in the affirmative. Our detailed contributions are as follows.

First contribution: Our BalanceProofs compiler. Our first contribution is BalanceProofs (see Section 3), a compiler that takes as input any naturally aggregatable vector commitment that is *not* maintainable, such as aSVC [37] and Pointproofs [15], and produces another naturally-aggregatable *and* maintainable vector commitment—in particular one with $O(\sqrt{n}\log n)$ update-all time (In our evaluation, we instantiate BalanceProofs with the aSVC vector commitment.) For the compilation to work, the input vector commitment must support opening all proofs in $O(n\log n)$ time (as is the case with aSVC [37] and Pointproofs [15]). Of course, this transformation introduces a trade-off: The query time for a single proof of the output vector commitment increases to $O(\sqrt{n})$ (which is $O(1)$ in both aSVC and Pointproofs)—however this is still sub-linear, and as we will see, a cost worth paying to support much faster aggregation.

The main idea of our compiler is simple: Suppose we have an aSVC vector commitment for a vector $\mathbf{m} = [m_0, \ldots, m_{n-1}]$ and that we have computed initial aSVC proofs $\pi_0, \ldots, \pi_{n-1}$ for every position of the vector. Whenever there is an update (i, δ) (change m_i to $m_i + \delta$) to the vector, aSVC would apply (i, δ) to all proofs $\pi_0, \ldots, \pi_{n-1}$, leading to $\Omega(n)$ time. Instead of doing this expensive operation, we *just* store the update (i, δ) in a log. Of course this is problematic. Whenever we want to query a proof π_j for an index j in the future, we need to first apply all updates in the log on proof π_j. However, given that updating a *single* aSVC proof π_j is cheap (constant time), we can fetch the updated proof π_j after t updates in time $O(t)$ by applying all t updates one-by-one on π_j. We make sure that t is kept below $\sqrt{n}$, by recomputing all proofs from scratch after $\sqrt{n}$ updates. Clearly, since recomputing all aSVC proofs from scratch takes time $O(n\log n)$ (which is a requirement for our compiler), the *amortized* time for our update algorithm is $O(\sqrt{n}\log n)$. We finally show how to deamortize this algorithm in practice, leading to $O(\sqrt{n}\log n)$ worst-case update time.

Second contribution: Bucketing BalanceProofs. Unfortunately, the $O(\sqrt{n}\log n)$ update operation of the above basic version of BalanceProofs is quite slow in practice. For example, we found it takes around 130 seconds to perform a single update for a vector of 2^{30} positions—this is approximately $1000\times$ slower than Hyperproofs, the only maintainable and aggregatable vector commitment and hence our baseline for comparison. To address this problem, we propose a bucketing technique: The main idea is to split the vector in p buckets $P_0, \ldots, P_{p-1}$ of n/p indices each. Then we apply aSVC *over* the buckets $P_0, \ldots, P_{p-1}$ (namely over *sets* of indices instead of single indices) and our BalanceProofs compiler *within* each bucket P_i. While this might sound like a trivial approach, it is not: For efficiency reasons, we have to use a 2-variate polynomial for the commitment (see "space-efficient" bucketing in Section 4.2) so that the size of public parameters stays linear. Our bucketing data structure maintains two components, *bucket proofs* and *individual proofs*.

A bucket proof Π_i is a batch proof over the indices of P_i, with respect to commitment C of the whole vector. Proofs Π_i are always updated immediately during an update, leading to $O(p)$ update time. An individual evaluation proof $\pi_{i,j}$ is a proof for the value of index j with respect to commitment C_i of bucket P_i. Since BalanceProofs is used within each bucket, updating these proofs takes $O(\sqrt{n/p}\log(n/p))$ time. Therefore for $p = n^{1/3}$, our update time becomes $n^{1/3} + n^{1/3}\log n$.

The above bucketing technique increases the size of individual proofs by one group element. However, the size of the aggregate proof is *not* constant anymore: To support aggregation of an arbitrary set of indices I, one might need to touch more than one buckets, for example, up to $p = n^{1/3}$ buckets. Therefore the aggregated proof size becomes $n^{1/3}$. However, as we will see in the experimental section, this compares very favorably in practice to Hyperproofs (Recall Hyperproofs is using a black-box argument system [8] to aggregate proofs and this leads to increased aggregate proof size.)

To further decrease update time, we also propose to split each bucket into smaller buckets (see "two-layer bucketing" in Section 4.3)—technically this is done by using a 3-variate polynomial for the commitment. In particular, we split the vector into p big buckets, and then each big bucket into t small buckets, leading to $p \cdot t$ small buckets. Using our compiler within a small bucket, updating individual proofs takes $O(\sqrt{n/pt}\log(n/pt))$ time. If we pick $p = t = n^{1/4}$, our update time becomes $n^{1/4} + n^{1/4} + n^{1/4}\log n$. Similarly, individual proofs are now three group elements and aggregate proof size will increase to at most $O(n^{1/4} \cdot n^{1/4}) = O(n^{1/2})$.

Limitations. When we are using two-layer bucketing, the aggregate proof size increases to $\sqrt{n}$. While not an issue in practice, it is an open problem to construct a maintainable and naturally-aggregatable vector commitment that has constant-size aggregate proof, yet $O(n^{1/4})$ update time. See Table 1 for an asymptotic comparison with Hyperproofs.

Evaluation. Our evaluation (Sec. 5) has three components.

Microbenchmarks. We observe that the basic BalanceProofs version has aggregation and aggregate verification that is in the order of milliseconds (for aggregating 1024 individual

Scheme	$\lvert\pi_i\rvert$	$\lvert\pi_I\rvert$	Aggregate	UpdAllProofs	Query π_i	Verify π_i	Verify π_I	Gen	$\lvert$pp$\rvert$
Hyperproofs [33]	$\log n$	$\log(k \log n)$	$k \log n$	$\log n$	$\log n$	$\log n$	$k \log n$	n	n
BalanceProofs (Sec. 3)	1	1	$k \log^2 k$	$\sqrt{n} \log n$	$\sqrt{n}$	1	$k \log^2 k$	$n \log n$	n
Two-layer bucketing (Sec. 4.3)	1	$\min\{k, \sqrt{n}\}$	$k \log^2 k$	$n^{1/4} \log n$	$n^{1/4}$	1	$k \log^2 k$	$n \log n$	n

Table 1: Asymptotic comparison with Hyperproofs. Proof sizes are in terms of group elements. n denotes the vector size, π_i is the individual proof for position i, π_I is the aggregated proof for an index set I, pp represents public parameters and $k = |I|$.

proofs), but has costly updates (more than 100 seconds for performing a single update on a vector of 2^{30} entries). We show that our two-layer bucketing can improve the update time to approximately 18 milliseconds, by increasing the aggregate proof size from 48 bytes to 51KB. We believe this manifests the value of bucketing, and it is a reasonable proof size that is worth having to enjoy much smaller update time.

Comparison with Hyperproofs. The main competitor of BalanceProofs is Hyperproofs [33], the only vector commitment that is both maintainable and aggregatable. Our main findings from comparing with Hyperproofs are as follows.

First, both aggregation of individual proofs and verification of aggregate proofs using two-layer bucketing outperform Hyperproofs by around 1000× and 100× respectively. Again, this is because BalanceProofs is naturally aggregatable, as opposed to Hyperproofs that uses the IPA [8] argument system as a black box. We consider this to be our central contribution.

Second, in terms of proof size, Hyperproofs and two-layer bucketing have approximately the same performance. Furthermore, we expect that Hyperproofs aggregate proof size will be larger as k grows since it depends on k, as opposed to BalanceProofs proof size that is at most $\sqrt{n}$. See Section 5 and Figure 3 for more comparison details.

Finally, Hyperproofs has an advantage over BalanceProofs in update time. Still BalanceProofs update time is practical: While update time for Hyperproofs ranges from 2 to 3 ms, update time for BalanceProofs is at most 18 ms—and could be improved with a multi-threaded implementation.

Macrobenchmarks. We study the application of BalanceProofs in stateless cryptocurrencies. We show that the time to reach consensus on a new block (that includes block proposal, block validation and proof maintenance) using BalanceProofs is almost 10× faster than the time required when using Hyperproofs—this is because aggregating transactions and verifying aggregate proofs is a dominant operation in such applications.

Related work. Catalano and Fiore [12] were the first to formalize the notion of VCs, together with two realizations of VCs based on the RSA and CDH assumptions. Other constructions based on these assumptions followed [11, 22, 37], but, just like [12], required $O(n)$ time to maintain all proofs. Therefore all these constructions are not maintainable. Gorbunov et al. [15] recently introduced Pointproofs, a VC scheme that can aggregate proofs across *different* commitments. However, Pointproofs are also not maintainable, and can be used as input to the BalanceProofs compiler. Pointproofs also provide a design overview on how to use cross-commitment aggregation to reduce storage requirements for smart contracts.

Boneh et al. [5] propose accumulator proof aggregation in groups of unknown order. They provide a constant-sized batch non-membership proof for a large number of elements. These proofs can be used to build the first positional vector commitment (VC) with constant-sized openings and constant-sized public parameters. But again, the resulting vector commitment is not maintainable.

There are maintainable vector commitments, such as simple Merkle trees [24] (transparent but non-homomorphic), lattice-based vector commitments [28, 30] (transparent and homomorphic), and AMT [36] (non-transparent and homomorphic). All these use a tree structure which seems to be the reason for their not supporting efficient proof aggregation.

Hyperproofs [33] is the first scheme to support both maintainability and aggregatability. Hyperproofs introduce multilinear trees (MLT) on the PST [27] commitment to update all proofs in $O(\log n)$ time. However, they use the IPA [8] proof system to support aggregatability, leading to large practical overhead in their proof aggregation.

2 Preliminaries

We use λ to denote the security parameter and $\mathsf{negl}(\cdot)$ to denote a negligible function. We also use multiplicative notation for all groups. With ω we denote a primitive n-th root of unity in $\mathbb{Z}_p$ [39]. Vectors are in bold, lower-case symbols, for example $\mathbf{m} = [m_0, \ldots, m_{n-1}]$.

Lagrange polynomials [3, 10]. For $i \in [0, n)$, we denote the i-th Lagrange polynomial, with roots of unity being used as indices, as

$$\mathcal{L}_i(x) = \prod_{j \in [0,n) \setminus i} \frac{x - \omega^j}{\omega^i - \omega^j}\,.$$

The Lagrange interpolation of vector $\mathbf{m} = [m_0, \ldots, m_{n-1}]$ is

$$\phi(x) = \sum_{i \in [0,n)} \mathcal{L}_i(x) \cdot m_i\,.$$

It is easy to see that for any $i \in [0,n)$, $\phi(\omega^i) = m_i$.

Bilinear pairings [18, 23]. We use $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ to denote the parameters associated with pairings. In particular $\mathbb{G}_1$,$\mathbb{G}_2$ and $\mathbb{G}_T$ are groups of prime order p, g_i is a generator of $\mathbb{G}_i$ and pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is such that $\forall u \in \mathbb{G}_1, w \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$, it is $e(u^a, w^b) = e(u, w)^{ab}$. For simplicity, we use the same group $\mathbb{G}$, with generator g, for both $\mathbb{G}_1$ and $\mathbb{G}_2$ when we describe our protocols—our implementation however uses asymmetric pairings.

Vector commitments. We formalize vector commitments (VC) below. We provide a generalized version of the definition that appeared in Hyperproofs [33]. Our generalized definition uses some auxiliary information aux to represent the underlying data structure used to maintain the proofs.

Definition 2.1 (VC scheme)**.** A VC scheme is a set of the following nine PPT algorithms.

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

(2) $\underline{\mathsf{Commit}_{\mathsf{pp}}(\mathbf{m}) \to (\mathsf{C}, \mathsf{aux})}$: Given vector $\mathbf{m}$, it outputs commitment C along with auxiliary information aux.

(3) $\underline{\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{m}, \mathsf{aux}) \to \pi_i}$: Given index i, vector $\mathbf{m}$ and auxiliary information aux, it outputs a proof π_i.

(4) $\underline{\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{m}) \to (\pi_0, \ldots, \pi_{n-1})}$: Given vector $\mathbf{m}$, it outputs all proofs $\pi_0, \ldots, \pi_{n-1}$.

(5) $\underline{\mathsf{Agg}_{\mathsf{pp}}(I, (\pi_i, m_i)_{i \in I}) \to \pi_I}$: Given proof-value pairs $\{(\pi_i, m_i)\}_{i \in I}$, for $I \subseteq [0,n)$, it outputs an aggregate proof π_I.

(6) $\underline{\mathsf{Verify}_{\mathsf{pp}}(\mathsf{C}, I, \{m_i\}_{i \in I}, \pi_I) := \{0,1\}}$: Given commitment C, values $\{m_i\}_{i \in I}$, for $I \subseteq [0,n)$, and (aggregate) proof π_I, it outputs either 0 or 1.

(7) $\underline{\mathsf{UpdCom}_{\mathsf{pp}}(i, \delta, \mathsf{C}) \to \mathsf{C}'}$: Given index i, value δ, commitment C, it outputs C$'$ reflecting position i changing by δ.

(8) $\underline{\mathsf{UpdAllProofs}_{\mathsf{pp}}(i, \delta, \{\pi_j\}, \mathsf{aux}) \to (\{\pi'_j\}, \mathsf{aux}')}$: Given index i, difference δ, proofs $\{\pi_j\}_{j \in [0,n)}$ and auxiliary information aux, it outputs updated proofs $\{\pi'_j\}_{j \in [0,n)}$ and updated auxiliary information aux$'$, to reflect position i changing by δ.

(9) $\underline{\mathsf{UpdProof}_{\mathsf{pp}}(i, \delta, j, \pi_j) \to \pi'_j}$: Given index i, difference δ, index j and proof π_j, it outputs updated proof π'_j to reflect position i changing by δ.

Definition 2.2 (VC correctness)**.** A VC scheme is correct if for all $\lambda \in \mathbb{N}$ and $n = \mathsf{poly}(\lambda)$, for all $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n)$, for all vectors $\mathbf{m}$, for all $i \in [0,n)$, if $(\mathsf{C}, \mathsf{aux}) = \mathsf{Commit}_{\mathsf{pp}}(\mathbf{m})$ and $\pi_i = \mathsf{Open}_{\mathsf{pp}}(i, \mathbf{m}, \mathsf{aux})$ (or π_i is derived from $\mathsf{OpenAll}_{\mathsf{pp}}$), then, for any polynomial number of updates (j, δ) resulting in a new vector $\mathbf{m}'$, if C$'$ and π'_i are obtained via calls to $\mathsf{UpdCom}_{\mathsf{pp}}$ and $\mathsf{UpdProof}_{\mathsf{pp}}$ (or $\mathsf{UpdAllProofs}_{\mathsf{pp}}$ with aux replaced by aux$'$) respectively, then

1. $\Pr[1 \leftarrow \mathsf{Verify}_{\mathsf{pp}}(\mathsf{C}', i, m'_i, \pi'_i)] = 1$;
2. For all $I \subseteq [0,n)$ it is

   $\Pr[1 \leftarrow \mathsf{Verify}_{\mathsf{pp}}(\mathsf{C}', I, (m'_i)_{i \in I}, \mathsf{Agg}_{\mathsf{pp}}(I, (\pi'_i, m'_i)_{i \in I}))] = 1.$

Definition 2.3 (VC soundness)**.** For all PPT adversaries $\mathcal{A}$,

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n), \\ (\mathsf{C}, I, J, (m_i)_{i \in I}, (m'_j)_{j \in J}, \pi_I, \pi_J) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) : \\ 1 \leftarrow \mathsf{Verify}_{\mathsf{pp}}(\mathsf{C}, I, (m_i)_{i \in I}, \pi_I) \wedge \\ 1 \leftarrow \mathsf{Verify}_{\mathsf{pp}}(\mathsf{C}, J, (m'_j)_{j \in J}, \pi'_J) \wedge \\ \exists k \in I \cap J \text{ such that } m_k \neq m'_k \end{array}\right] \leq \mathsf{negl}(\lambda)$$

aSVC vector commitment. Our construction (compiler) will be using the aSVC [37] vector commitment (Although other commitments can be used as input to our compiler, we have chosen aSVC due to its simplicity and efficiency.) aSVC is based on KZG polynomial commitments [19]. With linear-sized public parameters, it can compute all constant-sized individual proofs in quasilinear time and update a proof in constant time. Furthermore, it is *aggregatable* since one can aggregate proofs for many positions into a constant-sized *batch proof* for those positions. Given SDH parameters [4]

$$(g, g^\tau, \ldots, g^{\tau^{n-1}}),$$

aSVC represents a vector $\mathbf{m} = [m_0, \ldots, m_{n-1}]$ as the polynomial $\phi(x) = \sum_{i \in [0,n)} \mathcal{L}_i(x) \cdot m_i$ such that $\phi(\omega^i) = m_i$, where ω^i is the i-th n-th root of unity. The commitment to the vector is then simply the group element $g^{\phi(\tau)}$ that can be computed using the public parameters above. Similar to [9], the public parameters also contain commitments to all Lagrange polynomials $g^{\mathcal{L}_i(\tau)}$, which are used to compute the proofs.

A proof π_i that m_i is the value of vector $\mathbf{m}$ at position i is simply the commitment to the polynomial

$$q_i(x) = \frac{\phi(x) - m_i}{x - \omega^i}. \tag{1}$$

Aggregating aSVC proofs. aSVC [37] shows how to aggregate a set of proofs $\{\pi_i\}_{i \in I}$ for elements m_i of $\mathbf{m}$ into a constant-sized batch proof π_I for an index set I using *partial fraction decomposition* [41] and Drake and Buterin's observation [7]. In particular, π_I is a commitment to

$$q(x) = \frac{\phi(x) - R(x)}{A_I(x)},$$

where $A_I(x) = \prod_{i \in I}(x - \omega^i)$ and $R(x)$ is such that $R(\omega^i) = m_i$, for all $i \in I$. Let $A'_I(x) = \sum_{j \in I} \frac{A_I(x)}{x - \omega^j}$ be the derivative of $A_I(x)$ [40]. They observe that $q(x)$ can also be written as

$$q(x) = \sum_{i \in I} \frac{1}{A'_I(\omega^i)} \cdot q_i(x).$$

Thus we can compute $c_i = 1/A'_I(\omega^i)$ with $O(|I| \log^2 |I|)$ field operations [40] and aggregate $\pi_I = \prod_{i \in I} \pi_i^{c_i}$ with an $O(|I|)$-sized multi-exponentiation. We now describe the aSVC algorithms in detail (Note that in the following algorithms aux is always empty, so we do not include it for convenience.)

(1) $\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}$: Pick $\tau \in \mathbb{Z}_p^*$ uniformly at random. Output public parameters

$$\mathsf{pp} = \left((g^{\tau^i})_{i\in[0,n)}, (l_i)_{i\in[0,n)}, (a_i, u_i)_{i\in[0,n)} \right),$$

where $l_i = g^{\mathcal{L}_i(\tau)}$, $a_i = g^{A(\tau)/(\tau-\omega^i)}$, $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{\tau-\omega^i}}$, where $A(x) = \prod_{i\in[0,n)}(x-\omega^i)$.

(2) $\mathsf{Commit}_{\mathsf{pp}}(\mathbf{m}) \to \mathsf{C}$: Output $\mathsf{C} = \prod_{i\in[0,n)}(l_i)^{m_i}$.

(3) $\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{m}) \to \pi_i$: Output $\pi_i = g^{q_i(\tau)}$, where $q_i(x)$ is defined in Equation 1.

(4) $\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{m}) \to (\pi_0, \dots, \pi_{n-1})$: Output all proofs for $\mathbf{m}$.

(5) $\mathsf{Agg}_{\mathsf{pp}}(I, (\pi_i, m_i)_{i\in I}) \to \pi_I$: Compute $A_I(x) = \prod_{i\in I}(x - \omega^i)$ and $c_i = (A_I'(\omega^i))^{-1}$. Output $\pi_I = \prod_{i\in I}\pi_i^{c_i}$.

(6) $\mathsf{Verify}_{\mathsf{pp}}(\mathsf{C}, I, (m_i)_{i\in I}, \pi_I) := \{0,1\}$: Output 1 iff

$$e(\mathsf{C}/g^{R_I(\tau)}, g) = e(\pi_I, g^{A_I(\tau)}),$$

where $A_I(x) = \prod_{i\in I}(x-\omega^i)$ and $R_I(x)$ such that $R_I(\omega^i) = m_i$ for all $i \in I$.

(7) $\mathsf{UpdCom}_{\mathsf{pp}}(i, \delta, \mathsf{C}) \to \mathsf{C}'$: Output $\mathsf{C}' = \mathsf{C} \cdot (l_i)^\delta$.

(8) $\mathsf{UpdAllProofs}_{\mathsf{pp}}(i, \delta, \pi_0, \dots, \pi_{n-1}) \to (\pi_0', \pi_1', \dots, \pi_{n-1}')$: Call $\mathsf{VC.UpdProof}_{\mathsf{pp}}$ (see next) for every individual proof.

(9) $\mathsf{UpdProof}_{\mathsf{pp}}(i, \delta, j, \pi_j) \to \pi_j'$: If $i = j$, output $\pi_i' = \pi_i \cdot (u_i)^\delta$. If $i \neq j$, compute $w_{i,j} = a_i^{1/(\omega^i-\omega^j)} \cdot a_j^{1/(\omega^j-\omega^i)}$ and $u_{i,j} = w_{i,j}^{1/A'(\omega^i)}$, and return $\pi_j' = \pi_j \cdot (u_{i,j})^\delta$.

Complexities of aSVC. aSVC needs $O(n)$ size public parameters, $O(n \log n)$ time to open all single proofs, $O(n)$ time to update all the individual proofs and $O(|I| \log^2 |I|)$ to aggregate or verify aggregated proof with index set I. Both individual and aggregated proofs in aSVC have constant proof size.

3 Our BalanceProofs compiler

In this section, we introduce BalanceProofs, which can be viewed as a compiler that takes as input a vector commitment VC that is not maintainable and outputs a maintainable vector commitment VC$'$. Let T be the time of OpenAll and P be the time of UpdProof, of the input vector commitment. The input vector commitment VC must satisfy certain requirements for our compilation to produce a maintainable vector commitment VC$'$. We list them here.

- The time complexity T of OpenAll should be $o(n\sqrt{n})$.
- The time complexity P of UpdProof should be $o(\sqrt{n})$.
- The vector commitment VC must have an efficient aggregation algorithm $\mathsf{Agg}_{\mathsf{pp}}(I, (\pi_i, m_i)_{i\in I})$ (For concrete efficiency, we stress that the aggregation algorithm should be *natural*, i.e., it should not use zk-SNARKs as a blackbox, for example. Therefore vector commitments like Hyperproofs [33] are not good inputs to our compiler.)

Note that both aSVC [37] and Pointproofs [15] can be used as input to our compiler as they satisfy all above properties. However, as we mentioned before, due to its conceptual simplicity, our implementation is using aSVC. Let now VC be the input non-maintainable vector commitment with algorithms

$$\mathsf{VC.Gen}, \mathsf{VC.Commit}, \mathsf{VC.Open}, \mathsf{VC.OpenAll}, \mathsf{VC.Agg}\dots$$

that satisfy the properties above, and let VC$'$ be the output vector commitment with algorithms

$$\mathsf{VC}'.\mathsf{Gen}, \mathsf{VC}'.\mathsf{Commit}, \mathsf{VC}'.\mathsf{Open}, \mathsf{VC}'.\mathsf{OpenAll}, \mathsf{VC}'.\mathsf{Agg}\dots$$

We first note that our compilation does not change the commitment expression and the (aggregate) proof expression. In particular, for the case of aSVC as the input VC, the commitment of VC$'$ will still be $g^{\phi(\tau)}$, where $\phi(x) = \sum_{i\in[0,n)} \mathcal{L}_i(x) \cdot m_i$ and the same holds for the proofs.

The main difference between the input and output vector commitment lies in how the output VC$'$ is handling updates: Whenever an update (i, δ) appears, we do not use the VC.UpdAllProofs algorithm since this would incur $\Omega(n)$ cost (For example, for aSVC, UpdAllProofs iterates through all n proofs one by one.) What we do is *append* the update (i, δ) in a list L of size $\sqrt{n}$, which takes just constant time. The list L serves as the auxiliary information aux. When the list L becomes full, our compiler calls VC.OpenAll to compute fresh proofs $(\pi_0, \dots, \pi_{n-1})$ for all positions. After that, our compiler empties the list L. If a query for an individual proof comes before computing fresh proofs (i.e., before the list reaches $\sqrt{n}$ elements), then all updates are applied to the proof that is requested and an updated fresh individual proof is returned.

Since VC.OpenAll runs $T = o(n\sqrt{n})$ time, our compiler needs amortized

$$O\left(\frac{\sum_{i=1}^{\sqrt{n}-1} 1 + T}{\sqrt{n}}\right) = O(T/\sqrt{n}) = o(n)$$

time to update the proofs, as required. Also note that since the maximum size of the list L is $\sqrt{n}$ and algorithm VC.UpdProof runs in $o(\sqrt{n})$, returning a fresh proof takes at most $P\sqrt{n} = o(n)$ time. It is easy to see that for the case of aSVC, the above complexities become $O(\sqrt{n}\log n)$ and $O(\sqrt{n})$ respectively.

3.1 Compiling VC into VC$'$

We now provide the detailed algorithms for VC$'$:

(1) $\mathsf{VC}'.\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}$:

Return $\mathsf{VC.Gen}(1^\lambda, n)$.

(2) $\mathsf{VC}'.\mathsf{Commit}_{\mathsf{pp}}(\mathbf{m}) \to (\mathsf{C}, \mathsf{aux})$:

Let $\mathsf{VC.Commit}_{\mathsf{pp}}(\mathbf{m}) \to (\mathsf{C}, \mathsf{aux}_0)$. Let $\pi_0, \dots, \pi_{n-1}$ output by $\mathsf{VC.OpenAll}_{\mathsf{pp}}(\mathbf{m})$. Initialize empty list L of size $\sqrt{n}$. Return $(\mathsf{C}, [L; \pi_0, \dots, \pi_{n-1}])$.

(3) $\mathsf{VC}'.\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{m}, \mathsf{aux}) \to \pi_i$:

Let $\mathsf{aux} = [L; \pi_0, \ldots, \pi_{n-1}]$. If $L = \emptyset$, output π_i. Otherwise call $\mathsf{VC.UpdProof_{pp}}(j, \delta_j, i, \pi_i)$ for each update (j, δ_j) in L and return the latest proof π'_i.

(4) $\mathsf{VC'.OpenAll_{pp}}(\mathbf{m}) \to (\pi_0, \pi_1, \ldots, \pi_{n-1})$:

Return $\mathsf{VC.OpenAll_{pp}}(\mathbf{m}) \to (\pi_0, \pi_1, \ldots, \pi_{n-1})$.

(5) $\mathsf{VC'.Agg_{pp}}(I, (\pi_i, m_i)_{i \in I}) \to \pi_I$:

Return $\mathsf{VC.Agg_{pp}}(I, (\pi_i, m_i)_{i \in I}) \to \pi_I$.

(6) $\mathsf{VC'.Verify_{pp}}(\mathsf{C}, I, (m_i)_{i \in I}, \pi_I) := \{0, 1\}$:

Return $\mathsf{VC.Verify_{pp}}(\mathsf{C}, I, (m_i)_{i \in I}, \pi_I) \to b$.

(7) $\mathsf{VC'.UpdCom_{pp}}(i, \delta, \mathsf{C}) \to \mathsf{C}'$:

Return $\mathsf{VC.UpdCom_{pp}}(i, \delta, \mathsf{C}) \to \mathsf{C}'$.

(8) $\mathsf{VC'.UpdAllProofs_{pp}}(i, \delta, \mathsf{aux}) \to \mathsf{aux}'$:

Parse aux as $[L; \pi_0, \ldots, \pi_{n-1}]$. If $|L| < \sqrt{n}$, append (i, δ) to L and return $[L \cup (i, \delta); \pi_0, \ldots, \pi_{n-1}]$; Otherwise call $\mathsf{VC.OpenAll_{pp}}(\mathbf{m})$ to compute $\pi'_0, \ldots, \pi'_{n-1}$ and return $[\emptyset; \pi'_0, \ldots, \pi'_{n-1}]$.

(9) $\mathsf{VC'.UpdProof_{pp}}(i, \delta, j, \pi_j) \to \pi'_j$:

Call $\mathsf{VC.UpdProof_{pp}}(i, \delta, j, \pi_j) \to \pi'_j$ and return π'_j.

3.2 Deamortizing UpdAllProofs

Note that the output algorithm VC′.UpdAllProofs must call VC.OpenAll every $\sqrt{n}$ updates—this leads to large worst-case update time $\Omega(n)$. Here we show how to deamortize VC′.UpdAllProofs and achieve sublinear worst-case update time. The crucial observation for the deamortization is the fact that VC.OpenAll (whose time complexity is T), just like any sequential algorithm, can be written down as $\sqrt{n}$ sequential procedures/code blocks $T_1, \ldots, T_{\sqrt{n}}$ each one running in $T/\sqrt{n}$ time. Then the deamortized VC′.UpdAllProofs works as follows.

1. First, it maintains a list L of size $2\sqrt{n}$, not $\sqrt{n}$.
2. For the first $\sqrt{n}$ updates $u_1 \ldots, u_{\sqrt{n}}$, it behaves exactly the same way as the amortized VC.UpdAllProofs, i.e., it just appends update u_i in L and answers queries by processing all the updates so far. Let $\mathbf{m}$ be the vector with the first $\sqrt{n}$ updates applied to it. Note that at this point there are no fresh proofs for m (These will be computed gradually in the next step.)
3. Every update u_j, $j = \sqrt{n} + 1 \ldots, 2\sqrt{n}$, will first be appended in L. Then the procedure $T_{j-\sqrt{n}}$ is executed on vector $\mathbf{m}$. By the end of update $u_{2\sqrt{n}}$, all proofs for $\mathbf{m}$ have been computed and the first $\sqrt{n}$ entries of L are discarded. Then the algorithm returns to the previous step and repeats the same process.

Clearly every step of the above algorithm takes worst-case time $T/\sqrt{n} = o(n)$, as required. And again, for the case of aSVC, this technique provides an algorithm with $O(\sqrt{n} \log n)$ update time in the worst case. We now have the following theorem.

Theorem 3.1. (Compiler) Let VC be a vector commitment scheme that is correct (per Definition 2.2) and sound (per Definition 2.3). Then (1) the output VC′ is also correct and sound; (2) VC′.UpdAllProofs takes at most $T/\sqrt{n} = o(n)$ time (T is the time of VC.OpenAll) and VC′.Open takes at most $P\sqrt{n} = o(n)$ time (P is the time of VC.UpdProof); (3) other complexities of VC′ are the same as VC.

Proof. The proof of correctness follows by inspection. For soundness, note that the commitment, final proofs, and verification algorithm are all the same in both VC and VC′. If an adversary finds commitment and proofs that break the soundness of VC′, then an adversary can use the same objects to break the soundness of VC. Complexities of VC′.UpdAllProofs and VC′.Open were analyzed previously in this section. □

4 Bucketing BalanceProofs

In the previous section we showed how one can compile a vector commitment that is not maintainable to one that is. The output vector commitment has a trade-off between updating all proofs and querying single proofs. In this section we will be using our compiler to explore a different trade-off, that of update complexities and proof size: We will be aiming for an $n^{1/k}$ update time, for some $k > 2$, and a sublinear-size proof.

The basic version uses bucketing to conceptually separate the original vector into p parts (Yet the commitment expression is just a single group element, as before, and not p group elements.) Then we can perform updates and aggregation inside each part and therefore updates are cheaper but batch proofs can span multiple buckets and therefore their size is $O(p)$. In practice we can choose $p = n^{1/3}$ or $p = n^{1/4}$ to get best performance. It is not useful to choose too small p, e.g., $p = \log n$, since that would make little change on the complexities of updating and querying, or too large p, e.g., $p = \frac{n}{\log n}$, since the resulting batch proof size will be too large. In the following we present our main bucketing idea and then continue with a more space-efficient bucketing scheme. We conclude with our most performant two-layer bucketing scheme—the one that we will be evaluating.

4.1 Basic bucketing

We present our bucketing technique using the aSVC [37] vector commitment as our basis. The public parameters of our bucketing scheme are therefore the same with aSVC, i.e.,

$$g, g^{\tau}, \ldots, g^{\tau^{n-1}}.$$

The vector commitment expression is the same too: If $\mathbf{m} = [m_0, \ldots, m_{n-1}]$ is a vector and $\phi(x) = \sum_{i \in [0,n)} m_i \mathcal{L}_i(x)$ is its Lagrange interpolation, the commitment for the whole vector is the KZG commitment $C = g^{\phi(\tau)}$.

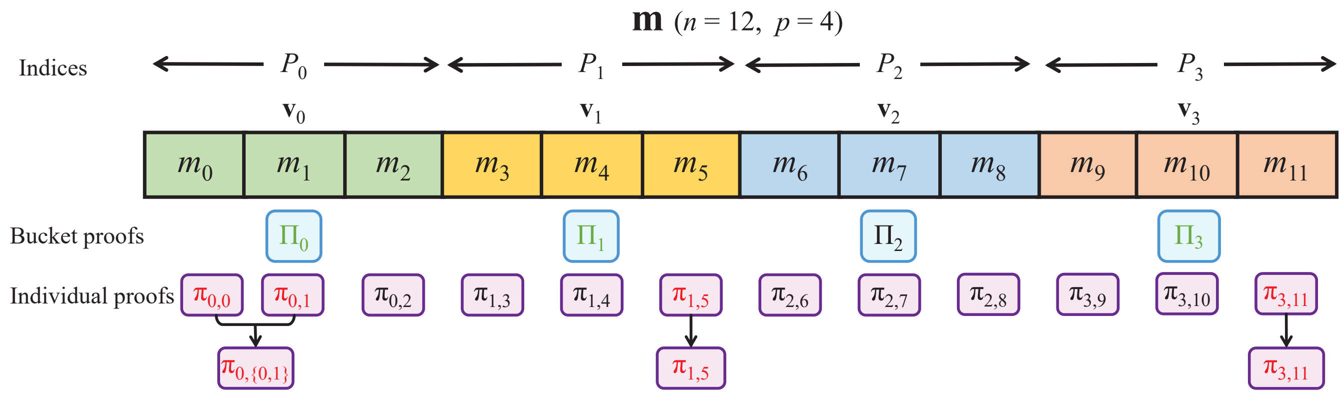


Figure 1: Aggregate proofs in bucketing. The aggregate proof for index set $I = \{0,1,5,11\}$ is $(\Pi_0, \pi_{0,\{0,1\}})$, $(\Pi_1, \pi_{1,5})$, $(\Pi_3, \pi_{3,11})$.

Partitioning, individual proofs and bucket proofs. We naturally partition the index set $[0,n)$ into p parts

$$P_i = \left[i \cdot \frac{n}{p}, (i+1) \cdot \frac{n}{p}\right), \ \forall i \in [0,p).$$

Then we can view $\mathbf{m}$ as p subvectors $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{p-1}$ where each $\mathbf{v}_i$ contains indices in P_i. Note $|\mathbf{v}_i| = \frac{n}{p}$. Similarly, we can write Lagrange interpolations $\phi_i(x)$ for each $\mathbf{v}_i$ as

$$\phi_i(x) = \sum_{j \in P_i} \mathcal{L}_{i,j}(x) \cdot m_j, \text{ where } \mathcal{L}_{i,j}(x) = \prod_{k \in P_i \setminus j} \frac{x - \omega^k}{\omega^j - \omega^k}.$$

Now, if we divide $\phi(x)$ with $\prod_{j \in P_i}(x - \omega^j)$, we can write

$$\phi(x) = \phi_i(x) + q_i(x) \prod_{j \in P_i}(x - \omega^j)$$

for some polynomial $q_i(x)$. In particular, $\Pi_i = g^{q_i(\tau)}$ is a KZG batch proof for the index set P_i of vector $\mathbf{m}$. We call Π_i a *bucket proof*. We can still provide individual proofs for an index $j \in P_i$ inside subvector $\mathbf{v}_i$ based on the KZG equation

$$\phi_i(x) = q_{i,j}(x)(x - \omega^j) + \phi_i(\omega^j) = q_{i,j}(x)(x - \omega^j) + \phi(\omega^j).$$

We call $\pi_{i,j} = g^{q_{i,j}(\tau)}$ an *individual proof* for position j of $\mathbf{v}_i$.

Evaluation proofs. When we need to compute an evaluation proof for one position j, we should first find i such that $j \in P_i$, then we provide both the bucket proof Π_i for index set P_i inside vector $\mathbf{m}$ and the individual proof $\pi_{i,j}$ for position j inside vector $\mathbf{v}_i$. Therefore, the resulting *evaluation proof* for position j is $(\Pi_i, \pi_{i,j})$. A verifier who has the commitment C, the claimed evaluation $(j, \phi(\omega^j) = z)$ and verification key $(g^\tau, (g^{a_k(\tau)})_{k \in [0,p)})$, where $a_k(x) = \prod_{j \in P_k}(x - \omega^j)$, can verify one evaluation proof $(\Pi_i, \pi_{i,j})$ by checking the following equation (say $j \in P_i$):

$$e(\mathsf{C}/g^z, g) = e\left(\Pi_i, g^{a_i(\tau)}\right) \cdot e\left(\pi_{i,j}, g^\tau / g^{\omega^j}\right).$$

Although the size of the verification key is $O(p)$, we show how to reduce it to constant size in the next section.

Batch proofs. By following standard KZG tricks, we can naturally compute a batch proof for an index set J. We distinguish two cases. If J falls within a single P_i, it is enough to provide an *evaluation-batch proof* $(\Pi_i, \pi_{i,J})$ where Π_i is the same as before and $\pi_{i,J} = g^{q_{i,J}}$ where

$$\phi_i(x) = q_{i,J}(x) \prod_{j \in J}(x - \omega^j) + c_J(x),$$

and $c_J(x)$ is the Lagrange interpolation over J of the claimed evaluations $\phi(\omega^j)$ for $j \in J$. A verifier who has the commitment C and the claimed evaluations $\{(j, \phi(\omega^j)\}_{j \in J}$ can compute $c_J(x)$ and verify one evaluation-batch proof $(\Pi_i, \pi_{i,J})$ by checking the following equation

$$e(\mathsf{C}/g^{c_J(\tau)}, g) = e\left(\Pi_i, g^{a_i(\tau)}\right) \cdot e\left(\pi_{i,J}, g^{\prod_{j \in J}(\tau - \omega^j)}\right).$$

If J spans multiple partitions P_i (say k), we provide the respective k evaluation-batch proofs.

Proof aggregation. We can aggregate multiple evaluation proofs into one batch proof naturally. See the example in Figure 1. The index set is $I = \{0,1,5,11\}$ and we are given four evaluation proofs $(\Pi_0, \pi_{0,0})$, $(\Pi_0, \pi_{0,1})$, $(\Pi_1, \pi_{1,5})$, $(\Pi_3, \pi_{3,11})$. Here we can only aggregate $\pi_{0,0}$ and $\pi_{0,1}$ to one batch proof inside $\mathbf{v}_0$. We cannot aggregate other proofs further. In the general case, the savings due to aggregation depends on whether the indices to be aggregated span multiple partitions or not.

We now continue with describing how to process updates: Whenever an update request (j, δ) is received, *both bucket proofs and individual proofs* must be updated.

Updating bucket proofs. Note all p bucket proofs can be updated in $O(p)$ time after receiving an update request (j, δ). To do that, we can update each bucket proof Π_i in $O(1)$ time. We now explain how to do that. We distinguish two cases.

The $j \in P_i$ case. Both polynomials $\phi(x)$ and $\phi_i(x)$ must be updated. In particular, $\phi_i(x)$ is updated as $\phi_i'(x) = \phi_i(x) +$

$\delta \cdot \mathcal{L}_{i,j}(x)$. Consider now the quotient polynomial $q_i(x)$ as defined before. This can be written as

$$q_i'(x) = \frac{\phi'(x) - \phi_i'(x)}{\prod_{k \in P_i}(x - \omega^k)} = q_i(x) + \frac{\delta \cdot (\mathcal{L}_j(x) - \mathcal{L}_{i,j}(x))}{\prod_{k \in P_i}(x - \omega^k)}\,.$$

Thus we can precompute

$$r_j(x) = \frac{\mathcal{L}_j(x) - \mathcal{L}_{i,j}(x)}{\prod_{k \in P_i}(x - \omega^k)}$$

and save update parameters

$$\{g^{r_j(\tau)}\}_{j \in [0,n)}$$

during the generation algorithm. Then proof Π_i can be updated as $\Pi_i' = \Pi_i \cdot (g^{r_j(\tau)})^{\delta}$.

We note here that $r_j(x)$ is indeed a polynomial and can be computed using the initial public parameters (Therefore publishing the update parameters does not affect security.) To see that, we can prove that $\prod_{k \in P_i}(x - \omega^k) | \mathcal{L}_j(x) - \mathcal{L}_{i,j}(x)$. For that, we only need to show that $\forall k \in P_i$, $x - \omega^k | \mathcal{L}_j(x) - \mathcal{L}_{i,j}(x)$. If $k \in P_i$ and $k \neq j$, then then this is trivial from the definition. For $k = j$, $\mathcal{L}_j(\omega^j) - \mathcal{L}_{i,j}(\omega^j) = 1 - 1 = 0$, so $\mathcal{L}_j(x) - \mathcal{L}_{i,j}(x)$ has a factor $x - \omega^j$ and then $x - \omega^j | \mathcal{L}_j(x) - \mathcal{L}_{i,j}(x)$.

The $j \notin P_i$ case. Since $j \notin P_i$, there is no change of the polynomial $\phi_i(x)$. Consider the quotient polynomial $q_i(x)$, which can be written as

$$\begin{aligned} q_i'(x) &= \frac{\phi'(x) - \phi_i(x)}{\prod_{k \in P_i}(x - \omega^k)} = \frac{(\phi(x) - \phi_i(x)) + \delta \cdot \mathcal{L}_j(x)}{\prod_{k \in P_i}(x - \omega^k)} \\ &= q_i(x) + \frac{\delta \cdot \mathcal{L}_j(x)}{\prod_{k \in P_i}(x - \omega^k)}\,. \end{aligned}$$

Similarly, we can just precompute $r_{i,j}(x) = \mathcal{L}_j(x) / \prod_{k \in P_i}(x - \omega^k)$ and save update parameters

$$\{g^{r_{i,j}(\tau)}\}_{i \in [0,p), j \in [0,n)}$$

in the generation algorithm. The proof Π_i can be updated as $\Pi_i' = \Pi_i \cdot (g^{r_{i,j}(\tau)})^{\delta}$. Note, that as opposed to the $j \in P_i$ case, the number of public parameters for this case is $n \cdot p$, which is one of the major drawbacks of this approach, and which we will address in the next section.

Updating individual proofs. After receiving an update (j, δ), assuming $j \in P_i$, we only need to update individual proofs inside $\mathbf{v}_i$ since for any $k \neq i$, $\phi_k(x)$ does not change at all. Using our compiler technique from Section 3 (which involves keeping record lists $(L_i)_{i \in [0,p)}$), the update time is $O(\sqrt{n/p}\log(n/p))$ (Note that in order to balance the update time, the optimal way to pick p is to make $p \approx \sqrt{n/p}\log(n/p)$, which shows that $p \approx n^{1/3}$.)

4.2 Space-efficient bucketing

In the previous subsection we presented a method to update bucket proofs. As we saw, its main limitation is that it requires public update keys of $O(np)$ size (This is because of the case $j \notin P_i$.) To address this issue, we propose using the same index set inside each subvector. For the rest of this section, we set $\varphi = \omega^{\frac{n}{p}}$ and $\theta = \omega^p$, where ω is an n-th root of unity.

Partitioning, individual proofs and bucket proofs. Same as before, we can write Lagrange interpolations for each subvector $\mathbf{v}_i$, but at this time we use a different variable y so that we have the same set of indices inside each $\mathbf{v}_i$. In particular, we view the initial vector as a collection of p vectors $\mathbf{v}_i$, and we refer to the j-th element of vector $\mathbf{v}_i$ as $v_{i,j} = m_{j+i\cdot\frac{n}{p}}$. Note that j runs from 0 to $n/p - 1$ for all vectors $\mathbf{v}_i$. Therefore

$$\phi_i(y) = \sum_{j \in [0,n/p)} \mathcal{L}_j'(y) \cdot v_{i,j}, \text{ where } \mathcal{L}_j'(y) = \prod_{k \in [0,\frac{n}{p}) \setminus j} \frac{y - \theta^k}{\theta^j - \theta^k}.$$

Then the two-variable polynomial for the whole vector is a Lagrange interpolation over all $\phi_i(y)$, i.e.,

$$\phi(x,y) = \sum_{i \in [0,p)} \mathcal{L}_i(x)\phi_i(y), \text{ where } \mathcal{L}_i(x) = \prod_{k \in [0,p) \setminus i} \frac{x - \varphi^k}{\varphi^i - \varphi^k}.$$

Note that $\phi(\varphi^i, \theta^j) = v_{i,j}$ for all $i \in [0,p)$, $j \in [0, \frac{n}{p})$. Now can similarly define bucket proofs using the polynomial $q_i(x,y)$ derived from the division

$$\phi(x,y) = \phi_i(y) + q_i(x,y)(x - \varphi^i)\,.$$

as well as individual proofs for element v_{ij} by using the polynomial $q_{i,j}(y)$ derived by the division

$$\phi_i(y) = q_{i,j}(y)(y - \theta^j) + \phi_i(\theta^j) = q_{i,j}(y)(y - \theta^j) + \phi(\varphi^i, \theta^j)\,.$$

Commitments and evaluation proofs. From the equations above, we can see that in $\phi(x,y)$, x has degree at most p and y has degree at most n/p, which suggests that the public parameters for our new vector commitment are

$$(g^{\alpha^i \beta^j})_{i \in [0,p), j \in [0, \frac{n}{p})}\,, \tag{2}$$

where α, β are secret and uniform. Therefore the size of the public parameters is $O(p \cdot \frac{n}{p}) = O(n)$. We can also write our new vector commitment as $C = g^{\phi(\alpha,\beta)}$.

To prove a claimed evaluation $v_{i,j} = z$, the prover should compute the bucket proof $\Pi_i = g^{q_i(\alpha,\beta)}$ as well as the individual proof $\pi_{i,j} = g^{q_{i,j}(\beta)}$, and then provide the evaluation proof as $(\Pi_i, \pi_{i,j})$. The verifier can verify it by checking the following is true:

$$e(\mathsf{C}/g^z, g) = e(\Pi_i, g^{\alpha - \varphi^i}) \cdot e(\pi_{i,j}, g^{\beta - \theta^j})\,.$$

To aggregate evaluation proofs, we use the same idea as in Section 4.1: Just aggregate inside subvectors and combine the resulting batch proofs.

Updating bucket proofs. Similarly as before, all p bucket proofs can be updated in $O(p)$ time. Unlike before however, the size of the public parameters required for this update is $O(n)$ as we analyze in the following. We now explain how to update bucket proof Π_i: After receiving an update request $((k,j),\delta)$, we have $\phi'(x,y) = \phi(x,y) + \delta \cdot \mathcal{L}_k(x)\mathcal{L}'_j(y)$, $\phi'_k(y) = \phi_k(y) + \delta \cdot \mathcal{L}'_j(y)$.

The $k = i$ case. When the bucket proof Π_i we wish to update corresponds to the bucket $k = i$ where the actual update is happening, the new quotient polynomial $q'_i(x,y)$ can be written as

$$q'_i(x,y) = \frac{\phi'(x,y) - \phi'_i(y)}{x - \varphi^i} = q_i(x,y) + \frac{\delta \cdot \mathcal{L}'_j(y)(\mathcal{L}_i(x) - 1)}{x - \varphi^i}\,.$$

Thus we can precompute $r_{i,j}(x,y) = \mathcal{L}'_j(y)(\mathcal{L}_i(x) - 1)/(x - \varphi^i)$ and save update parameters

$$(g^{r_{i,j}(\alpha,\beta)})_{i \in [0,p), j \in [0,\frac{n}{p})}$$

in the generation algorithm. The proof Π_i can be updated to $\Pi'_i = \Pi_i \cdot (g^{r_{i,j}(\alpha,\beta)})^{\delta}$. Note that the update parameters size is only $O(n)$ in this case.

The $k \neq i$ case. In this case the new quotient polynomial $q'_i(x,y)$ can be written as

$$\begin{aligned} q'_i(x,y) &= \frac{\phi'(x,y) - \phi_i(y)}{x - \varphi^i} = q_i(x,y) + \frac{\delta \cdot \mathcal{L}_k(x)\mathcal{L}'_j(y)}{x - \varphi^i} \\ &= q_i(x,y) + \frac{\delta \cdot \prod_{t \in [0,p)}(x - \varphi^t)\mathcal{L}'_j(y)}{\prod_{t \in [0,p)\setminus k}(\varphi^k - \varphi^t)(x - \varphi^k)(x - \varphi^i)} \\ = q_i(x,y) &+ \frac{\delta \cdot \prod_{t \in [0,p)}(x - \varphi^t)\mathcal{L}'_j(y)}{c_k(\varphi^k - \varphi^i)}\left(\frac{1}{x - \varphi^k} - \frac{1}{x - \varphi^i}\right) \\ &= q_i(x,y) + \frac{\delta}{c_k(\varphi^k - \varphi^i)}(s_{k,j}(x,y) - s_{i,j}(x,y))\,, \end{aligned}$$

where $c_k = \prod_{t \in [0,p)\setminus k}(\varphi^k - \varphi^t) = p \cdot \varphi^{-k}$ and

$$s_{i,j}(x,y) = \mathcal{L}'_j(y) \prod_{k \in [0,p)\setminus i}(x - \varphi^k)\,.$$

Thus we can just pre-compute and save update parameters

$$\{g^{s_{i,j}(\alpha,\beta)}\}_{i \in [0,p), j \in [0,\frac{n}{p})}$$

in the generation algorithm. The proof Π_i can be updated as

$$\Pi'_i = \Pi_i \cdot (g^{s_{k,j}(\alpha,\beta)\cdot})^{u_{k,i}} \cdot (g^{s_{i,j}(\alpha,\beta)\cdot})^{-u_{k,i}}\,,$$

where $u_{k,i} = \delta/(c_k(\varphi^k - \varphi^i))$.

Updating individual proofs. After receiving an update $((i,j),\delta)$, we need to update individual proofs only inside $\mathbf{v}_i$ since any $\phi_k(y)$ for $k \neq i$ does not change. Again, using our compiler technique, the update time is $O(\sqrt{n/p}\log(n/p))$.

Halving batch proof size technique. Recall that the batch proof size for a set of indices I is $O(p)$—see Section 4.1. Concretely, it is $2 \cdot f$ group elements (f bucket proofs and f individual evaluation proofs), where $f \leq p$ is the number of buckets that index set I spans. For example, in Figure 1, we have $f = 3$. In this section we propose a simple optimization that reduces the size of the batch proof from $2f$ group elements to $f + 1$ group elements. The idea is very natural: We extend previous techniques developed in [37] to define a single batch bucket proof. Therefore we do not have to include every bucket proof.

To understand this better, consider two evaluation proofs $(\Pi_i = g^{q_i(\alpha,\beta)}, \pi_{i,2} = g^{q_{i,2}(\beta)})$ and $(\Pi_j = g^{q_j(\alpha,\beta)}, \pi_{j,3} = g^{q_{j,3}(\beta)})$. The first is for index 2 with value z in bucket i (i.e., $v_{i,2} = z$) and the second is for index 3 with value w in bucket j (i.e., $v_{j,3} = w$). Recall from before that the polynomials $q_i(x,y)$ and $q_j(x,y)$ satisfy the following equations

$$\phi(x,y) = q_i(x,y)(x - \varphi^i) + \phi_i(y)$$

and

$$\phi(x,y) = q_j(x,y)(x - \varphi^j) + \phi_j(y)$$

and therefore, by [37], we can easily define the batch bucket proof for buckets i and j as $\Pi_{\{i,j\}} = g^{q_{\{i,j\}}(\alpha,\beta)}$ where $q_{\{i,j\}}(x,y)$ satisfies

$$\phi(x,y) = r(x,y) + q_{\{i,j\}}(x,y)(x - \varphi^i)(x - \varphi^j)\,. \tag{3}$$

In the above, $r(x,y)$ is such that $r(\varphi^i, y) = \phi_i(y)$ and $r(\varphi^j, y) = \phi_j(y)$, as in [37]. We define our new, optimized batch proof to simply be $(\Pi_{\{i,j\}}, \pi_{i,2}, \pi_{j,3})$, down to 3 group elements from 4 group elements. (In general case the reduction is from $2f$ to $f + 1$ since we can batch f bucket proofs to a single one.)

To verify our batch proof $(\Pi_{\{i,j\}}, \pi_{i,2}, \pi_{j,3})$ for respective values $v_{i,2} = z$ and $v_{j,3} = w$, we observe that we can write $r(x,y)$ as

$$\begin{aligned} r(x,y) &= \ell_{i,j}(x) \cdot \phi_i(y) + \ell_{j,i} \cdot \phi_j(y) \\ &= z \cdot \ell_{i,j}(x) + q_{i,2}(y) \cdot (y - \theta^2) \cdot \ell_{i,j}(x) \\ &\quad + w \cdot \ell_{j,i}(x) + q_{j,3}(y) \cdot (y - \theta^3) \cdot \ell_{j,i}(x)\,, \end{aligned} \tag{4}$$

where $\ell_{i,j}(x) = (x - \varphi^j)/(\varphi^i - \varphi^j)$ and since, by KZG, $\phi_i(y) = z + q_{i,2}(y)(y - \theta^2)$ and $\phi_j(y) = w + q_{j,3}(y)(y - \theta^3)$. Then, on input $(\Pi_{\{i,j\}}, \pi_{i,2}, \pi_{j,3}, z, w)$ (and by combining Equations 3 and 4), the verification proceeds in two steps. The verifier first computes R as

$$e(g^{z\ell_{i,j}(\alpha) + w\ell_{j,i}(\alpha)}, g)e(\pi_{i,2}, g^{(\beta - \theta^2)\ell_{i,j}(\alpha)})e(\pi_{j,3}, g^{(\beta - \theta^3)\ell_{j,i}(\alpha)})$$

and then checks to see if

$$e(g^{\phi(\alpha,\beta)}, g) = R \cdot e(\Pi_{\{i,j\}}, g^{(\alpha - \varphi^i)(\alpha - \varphi^j)})\,.$$

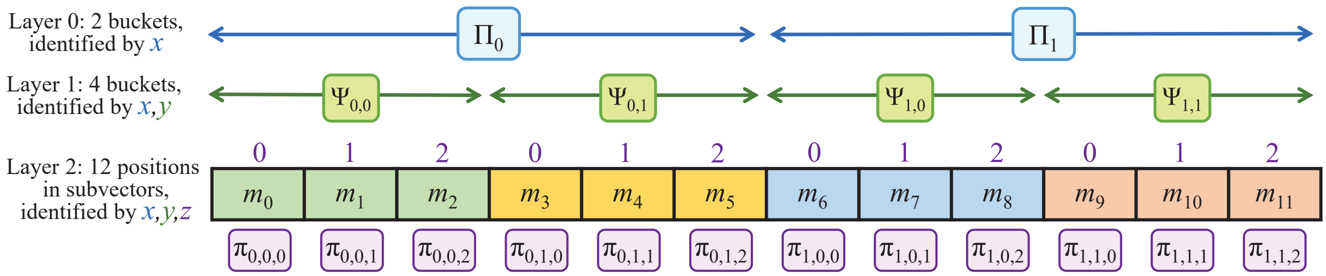


Figure 2: 2-layer bucketing. In this example, we set $n = 12$, $p = t = 2$.

It is easy to see that the verifier can compute everything needed for verification by using the public parameters as defined in Equation 2. We finally note that by [37], $\Pi_{\{i,j\}}$ can be produced from Π_i and Π_j as

$$\Pi_i^{1/h(\varphi^i)} \cdot \Pi_j^{1/h(\varphi^j)},$$

where $h(x) = x - \varphi^i + x - \varphi^j$.

The above halving approach can be easily generalized for an arbitrary set of indices I.

4.3 Two-layer bucketing

Our space-efficient construction can be easily extended to three variables to further reduce update time, at the expense of increasing proof size by one group element. In particular, we can introduce an additional t-partition of the n/p-sized subvectors, leading to $p \cdot t$ subvectors of $n/(p \cdot t)$ size each.

In this new two-layer scheme, when receiving an update request, we need $O(p)$ time to update all bucket proofs in the first layer (as before) and $O(t)$ time to update all bucket proofs in the second layer. As before, we will use our compiler for each final subvector, meaning we will have to maintain $p \cdot t$ update lists of size at most $\sqrt{n/(p \cdot t)}$ each to handle the updates within each subvector. Based on our compiler complexities, we can update individual proofs in the third layer in $O(\sqrt{n/(p \cdot t)} \log(n/(p \cdot t)))$ time. See Figure 2.

For optimal performance in practice, we can pick $p = t = n^{1/4}$, so that the resulting update time is $O(n^{1/4} \log n)$ and query time for each proof is $O(n^{1/4})$. As for proof size, the individual proof size is three group elements (still $O(1)$) and the aggregated proof size is $O(\sqrt{n})$ since there are at most $p \cdot t = \sqrt{n}$ subvectors. Note that the two-layer scheme is what we evaluate in Section 5 since it is the most performant one.

Obviously, we can add more layers in a similar manner: For $k > 2$ layers we achieve $O(n^{1/(k+2)} \log n)$ update time and $O(n^{k/(k+2)})$ aggregate proof size. To keep proof size small, we use exactly two layers.

Theorem 4.1. (Two-layer bucketing VC) Our VC based on two-layer bucketing with $p = t = n^{1/4}$ is correct (per Definition 2.2) and sound (per Definition 2.3). It also has the following complexities:

1. $O(n)$ public parameters size;
2. $O(1)$ commitment size;
3. $O(p+t+\sqrt{n/(p \cdot t)} \log(n/(p \cdot t))) = O(n^{1/4} \log n)$ time to update all proofs;
4. $O(\sqrt{n/(p \cdot t)}) = O(n^{1/4})$ time to query a single proof;
5. $O(1)$ individual proof size (consisting of three group elements) and $O(p \cdot t) = O(\sqrt{n})$ batch proof size;
6. $O(|I| \log^2 |I|)$ to aggregate proofs corresponding to an index set I;
7. $O(|I| \log^2 |I|)$ to verify a batch proof corresponding to an index set I.

The two-layer bucketing detailed construction and the proof of the above theorem can be found in Appendix A and C.

5 Evaluation

In this section we measure the performance of BalanceProofs. We fully implemented two versions of our compiler using aSVC as the input VC scheme: basic BalanceProofs (Section 3), and two-layer bucketing (Section 4.3).

Our implementation is in Golang and available online[1]. We use go-kzg [29] as a reference to implement KZG proofs. We chose BLS12-381 [20], a pairing-friendly elliptic curve, which is also the elliptic curve used in Hyperproofs and offers 128 bits of security. We run each experiment several times and report the average.

Hardware. Experiments are executed on an AWS EC2 m5d.4xlarge instance with Intel(R) Xeon(R) Platinum 8259CL CPU with 2.50GHz, 8 cores and 64GB of RAM. We only utilize a single CPU core in our experiments, but all of our algorithms are parallelizable.

Deamortizing updates. Our implementation uses a deamortized version of the update algorithm. Here we give some more details about the implementation of the deamortization. Recall that in order to deamortize updates, we must separate

[1] https://github.com/wangnick2017/balanceproofs-go

the computation in the $O(n \log n)$-time algorithm VC.OpenAll into $\sqrt{n}$ $O(\sqrt{n} \log n)$-time sub-steps. We examined VCs that can serve as input to our compiler, such as [15, 37], and found that their VC.OpenAll can indeed be separated.

We implement this separation as follows. Take aSVC [37] as an example. The VC.OpenAll algorithm of aSVC runs in $O(n \log n)$ time—it is the technique from FK20 [14]. It contains $k = O(1)$ single loops, each needing at most $O(n \log n)$ operations. We can then separate each loop into $\sqrt{n}$ small loops, each with $O(\sqrt{n} \log n)$ operations. An alternative approach is to focus on the operations with the highest cost. This type of operation could be, for instance, group operations on elliptic curves. We can use a counter to count how many operations we have done so far. As soon as the counter reaches some threshold, we save the current configuration, and exit this part temporarily. In the next round, we can restart from where we left off. Combining the two methods above, we can finish the whole algorithm in $O(\sqrt{n})$ rounds where each round requires almost equal time to complete.

Constant adjustments for bucket sizes. Recall that from Section 4.3, the time to update bucket proofs is $O(n^{1/4})$ and the time to update individual proofs inside subvectors is $O(n^{1/4} \log n)$. While asymptotically they are close, in practice the time to update individual proofs might be 100× slower than the time to update bucket proofs.

In order to balance them and decrease the update time overall, in our implementation we apply some constant c to the number of buckets in each layer, so that $p = t = c \cdot n^{1/4}$ and each subvector has size $\sqrt{n}/c^2$. Then the resulting update times are $O(cn^{1/4})$ for bucket proofs and $(n^{1/4}/c) \log n$ for individual proofs. Note that with this constant c, the aggregate proof size may increase to at most $O(c^2\sqrt{n})$.

Table 2: Single-thread execution for basic BalanceProofs with aSVC as input. Times with an asterisk (*) are too long (more than 5 hours).

$L = \log_2 n$	20	22	24	26	28	30
Commit (min)	0.47	1.69	6.99	28.4	114.8	*
OpenAll (hrs)	0.86	3.74	*	*	*	*
UpdAllProofs (s)	3.03	6.65	14.3	30.5	64.4	135.7
Query Indiv. (s)	0.02	0.05	0.09	0.18	0.38	0.77
Indiv. Verify (ms)	1.18	1.20	1.19	1.21	1.20	1.21
Aggregate (s)	0.38	0.41	0.35	0.43	0.39	0.41
Agg. Verify (s)	0.43	0.42	0.44	0.42	0.43	0.42
Indiv. proof size	48 bytes					
Agg. proof size	48 bytes					

Table 3: Single-thread execution for two-layer bucketing. Times with an asterisk (*) are too long (more than 5 hours). Proof sizes with halving technique are marked with †.

$L = \log_2 n$	20	22	24	26	28	30
Commit (min)	0.47	1.69	6.99	28.5	114.8	*
OpenAll (hrs)	0.58	2.49	*	*	*	*
UpdAllProofs (ms)	4.59	6.08	8.84	10.67	15.51	18.96
Query Indiv. (ms)	3.25	4.06	7.81	9.90	17.89	19.41
Indiv. Verify (ms)	1.56	1.57	1.59	1.55	1.57	1.59
Aggregate (ms)	109	69.1	36.9	23.1	12.6	8.41
Agg. Verify (ms)	209	163	132	117	114	106
Indiv. proof size	144 bytes					
Agg. proof size	61KB	62KB	88KB	90KB	98KB	99KB
Agg. proof size†	31KB	32KB	45KB	46KB	50KB	51KB

5.1 Microbenchmarks

We benchmark the performance of basic BalanceProofs in Table 2 and two-layer bucketing in Table 3.

Committing. We commit to vectors of size $n = 2^L$ where L ranges from 20 to 30. For $L = 28$ it takes roughly 114 minutes to compute the commitment. This is typically a one-time operation in our applications.

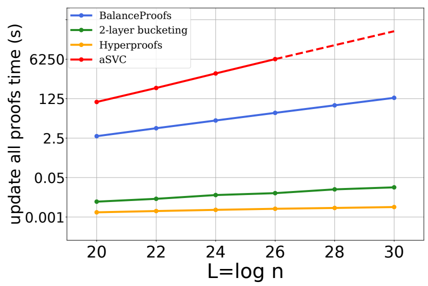
Opening all proofs. For BalanceProofs, it takes hour-level time to open all proofs so we were able to run experiments only for $L < 24$. For two-layer bucketing, the time is slightly smaller, since computing bucketing proofs and then computing individual proofs has smaller constants.

Updating all proofs. We measure the average time for performing 1024 updates chosen at random. Although basic BalanceProofs requires about 135 seconds to update all proofs for $L = 30$, our two-layer bucketing reduces this to millisecond-level (10 to 20 ms).

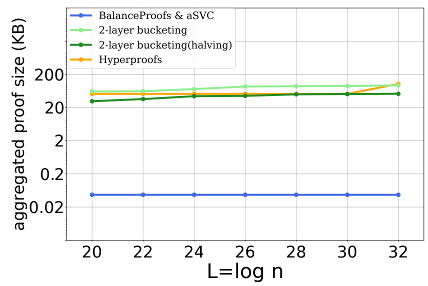
Querying individual proofs. The size of the list storing the updates could be from 0 to $\sqrt{n}$. On average, this is $\sqrt{n}/2$. This is what we measure, i.e., a query on a list of size $O(\sqrt{n}/2)$ (with constant adjustments). For this list size, querying an individual proof requires about 0.8 seconds for $L = 30$ in basic BalanceProofs, and 20 ms for two-layer bucketing.

Proof size and verification time. BalanceProofs has proofs that contain one $\mathbb{G}_1$ element and can be verified with two pairings. Our two-layer bucketing scheme has proofs with three $\mathbb{G}_1$ elements, which can be verified with three pairings.

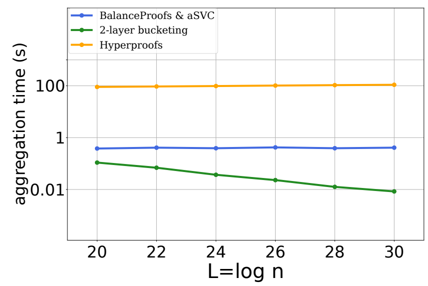
Aggregation. We aggregate 1024 individual proofs in our experiments, since 1024 is a common average number of transactions in one block of cryptocurrencies [17, 26]. The time of aggregation in Table 2 and Table 3 remains almost unchanged when L ranges from 20 to 30 because the time



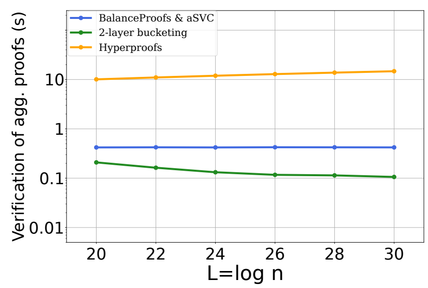
(a) Time to update all proofs



(b) Batch proof size



(c) Aggregation time



(d) Verification time for aggregated proofs

Figure 3: Comparison between BalanceProofs , aSVC and Hyperproofs. Including time to update all proofs (average time of 1024 updates), aggregate proof size (for 1024 individual proofs), time to aggregate 1024 proofs and to verify aggregated proofs.

complexity of aggregation is $O(|I|\log^2|I|)$. In the next subsection, we will show that our aggregation can be about $100\times$ to $1000\times$ faster than Hyperproofs.

Size and verification time of batch proofs. The batch proof size is one $\mathbb{G}_1$ element for basic BalanceProofs, and $O(\min\{n^{1/4},|I|\}+2\min\{n^{1/2},|I|\})$ $\mathbb{G}_1$ elements for two-layer bucketing. The verifier in BalanceProofs requires $O(|I|\log^2|I|)$ time — while the verifier in two-layer bucketing also needs to verify multiple equations.

Comparison with non-maintainable schemes. The main difference of BalanceProofs with vector commitments that are non-maintainable, such as aSVC [37], is in proof update and in querying individual proofs.

In particular, the time to update all proofs in aSVC is $O(n)$, which is about 90s when $L=20$ and over 90000s when $L=30$. Even our basic BalanceProofs scheme is $30\times$ to $700\times$ faster—see Figure 3(a) for update times comparison. However, query time of aSVC is $O(1)$ (just a lookup), while BalanceProofs needs sublinear time ranging from 3ms to 20ms.

5.2 Comparison with Hyperproofs

In Figure 3, we compare BalanceProofs with Hyperproofs on the same machine. Both implementations are in Golang and use the BLS12-381 [20] elliptic curve. The code of Hyperproofs we used is cloned from GitHub [32].

Opening all proofs. Hyperproofs compute a multilinear tree (MLT) to open all proofs, which needs $O(n\log n)$ time asymptotically and about 2.5 hours in practice when $L=24$. While our schemes may require 10+ hours to open all proofs when $L=24$, in practice this is not executed frequently.

Updating all proofs. The time required by Hyperproofs to update all proofs is relatively small (up to 3ms) since their proofs are in a tree structure required $O(L)$ group operations to be updated. Although the time to update all proofs in our schemes requires more time, the numbers are all reasonable in practice (up to 18ms for $L=30$) for two-layer bucketing.

Querying individual proofs. The query time of Hyperproofs is $O(\log n)$ and ours is $O(\sqrt{n})$. In practice, query time is less than 1ms for Hyperproofs and 3ms to 20ms for ours.

Aggregation. In the experiments, we show the results for aggregating 1024 proofs. Due to the black-box use of IPA argument [8], Hyperproofs needs $90\sim110$s to aggregate 1024 proofs and $13\sim17$s to verify the aggregated proofs, when $20\leq L\leq30$. This large cost limits the applicability of Hyperproofs in cryptocurrencies where the proof must be computed once and the verification has to be performed by multiple parties. As a comparison, aggregation in our schemes takes at most 0.43s and verification is millisecond-level.

Basic BalanceProofs has $1000\times$ smaller batch proof than Hyperproofs, while two-layer bucketing has almost same-level batch proof size with Hyperproofs. However, the size of batch proofs in Hyperproofs depends on the smallest power of two $\geq\log(|I|\log n)=\log|I|+\log\log n$, which remains the

Scheme	Two-layer	Hyperproofs	Merkle
Block proposal (P)	1.82 sec	2.23 min	81 min
Block validation (V)	0.19 sec	17.5 sec	0.18 sec
Proof maintenance(M)	38.72 sec	5.14 sec	4.7 sec
Total ($P+hV+M$)	45 sec	8 min	81 min

Table 4: Stateless cryptocurrency macrobenchmarks.

same when $|I| = 1024$ and L ranges from 20 to 30. When $L = 32$, this power of two will be doubled and the batch proof size will also be doubled, i.e., 103 KB, while batch proof size in our two-layer bucketing will almost remain the same (when using halving technique, around 50 KB for $L = 32$). We simulate the case for $L = 32$ for both schemes in Figure 3.

Parameterization. We stress that BalanceProofs are more flexible compared to Hyperproofs: Hyperproofs provide just one option where you can update all proofs quickly but aggregation is costly whereas BalanceProofs offers multiple tradeoffs between update time, proof size and aggregation time.

In particular, the BalanceProofs two-layer bucketing technique should be used for applications where having low computation is more critical than having lower bandwidth, such as maintaining a stateless blockchain with light clients (as in proof-of-stake systems). For nodes that can afford more computation (as in proof-of-work), saving on bandwidth might be more critical. For a more detailed discussion of the concrete impact of BalanceProofs in a stateless cryptocurrency application, in our macrobenchmarks in Section 5.3.

5.3 Macrobenchmarks

In this subsection, we discuss the application of BalanceProofs in stateless blockchains. In particular, we will measure the VC-induced overhead of statelessly reaching consensus on a new block—we will follow the same framework with Hyperproofs [33] (see Section 5.3 in Hyperproofs).

Problem background: stateless validation. In an account-based cryptocurrency, the miners store all the balances of user accounts, which can be represented as a long vector. However this state can be too large. In *stateless* cryptocurrencies, miners just store a constant-size vector commitment of this state and then access the committed data via vector commitment proofs: In particular, to propose a new block, a miner verifies a fixed number of transactions (balances and their proofs), aggregate those proofs and updates the commitment with respect to the previous transactions. When receiving a block from others, the block validation requires the miner to verify the batch proof in the incoming block. Users do not maintain their balance proofs locally—instead they contact incentivized proof-serving nodes (PSNs) to have their proofs served. PSNs are responsible for updating all individual proofs by replaying all new transactions, so that fresh proofs can be fetched by users efficiently.

Experimental setting. Assuming $L = 30$ and blocks of 1024 transactions, we define three measures useful for our macrobenchmarks:

- Block proposal time (P): time of a miner to propose a new block with 1024 transactions, where the miner needs to verify 1024 proofs, aggregate them and update the commitment.
- Block validation time (V): time to verify a block with a batch proof and its commitment.
- Proof maintenance time (M): time of a proof-serving node (PSN) to update all proofs from a new block.

Also we denote $h = 20$ to be an estimate of the network diameter and estimate the VC overhead as $P+hV+M$ since h sequential verification must be performed until the block reaches all nodes in the network.

Findings. Our comparison results are in Table 4: Compared to Hyperproofs and Merkle trees with SNARKs, for block proposal (P), our scheme is $60\times$ faster than Hyperproofs and $2000\times$ faster than Merkle trees. For block validation (V), our scheme is $90\times$ faster than Hyperproofs and performs similarly to Merkle trees. For proof maintenance (M), our scheme is $7\times$ slower than Hyperproofs and $9\times$ slower than Merkle trees. For the total overhead ($P+hV+M$), BalanceProofs is $10\times$ faster than Hyperproofs and $100\times$ faster than Merkle trees.

Trade-offs. We note here that two-layer bucketing appears to be the best point in the design space for this application. For example, if we use the (1-layer) space-efficient bucketing technique, the batch proof size is almost halved at the expense of much worse total time of 73 minutes—this is much worse than Hyperproofs and a little better than Merkle trees.

6 Conclusion

We presented BalanceProofs, a compiler that produces efficiently *maintainable* and *aggregatable* VC schemes. We also presented bucketing variants of BalanceProofs which have a tradeoff between update time, aggregation complexity and proof size. We showed that two-layer bucketing BalanceProofs has practical update time and around $1000\times$ better aggregation performance than Hyperproofs.

Future work. In our experiments, we picked aSVC [37] as the input VC to our compiler. It would be very interesting to try other VC schemes, such as Pointproofs [15], BBF [5] and achieve possible improvements. Also, we can try using multilinear trees and PST commitments [34,43,44]in the bucketing technique to explore other improvements. Lastly, the idea of bookkeeping to balance the time to update and query may be applicable in other cryptographic building blocks.

Acknowledgements

This work was supported primarily by Protocol Labs and the Ethereum Foundation as well as by the NSF, the Algorand Foundation through the ACE program and VMware.

References

[1] Shashank Agrawal and Srinivasan Raghuraman. KVaC: Key-Value Commitments for Blockchains and Beyond. Cryptology ePrint Archive, Report 2020/1161, 2020. https://eprint.iacr.org/2020/1161.

[2] Josh Benaloh and Michael de Mare. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In Tor Helleseth, editor, *EUROCRYPT '93*, pages 274–285, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[3] J. Berrut and L. Trefethen. Barycentric Lagrange Interpolation. *SIAM Review*, 46(3):501–517, 2004.

[4] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 56–73. Springer, 2004.

[5] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In *CRYPTO'19*, 2019.

[6] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized Cryptocurrency at Scale, 2020. https://eprint.iacr.org/2020/352.

[7] Vitalik Buterin. Using polynomial commitments to replace state roots. https://ethresear.ch/t/using-polynomial-commitments-to-replace-state-roots/7095, 2020.

[8] Benedikt Bünz, Mary Maller, Pratyush Mishra, and Noah Vesely. Proofs for inner pairing products and applications. Cryptology ePrint Archive, Report 2019/1177, 2019. https://eprint.iacr.org/2019/1177.

[9] Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and Modular Anonymous Credentials: Definitions and Practical Constructions. In *ASIACRYPT'15*, 2015.

[10] Jan Camenisch and Markus Stadler. Proof Systems for General Statements about Discrete Logarithms. Technical report, ETH Zurich, 1997.

[11] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 3–35, Cham, 2020. Springer International Publishing.

[12] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*, pages 55–72, 2013.

[13] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set, 2019. https://eprint.iacr.org/2019/611.

[14] Dankrad Feist and Dmitry Khovratovich. Fast amortized Kate proofs, 2020. https://github.com/khovratovich/Kate.

[15] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 2007–2023. ACM, 2020.

[16] Vipul Goyal. Reducing Trust in the PKG in Identity Based Cryptosystems. In *CRYPTO'07*, 2007.

[17] J. Göbel and A.E. Krzesinski. Increased block size and bitcoin blockchain dynamics. In *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–6, 2017.

[18] Antoine Joux. A One Round Protocol for Tripartite Diffie–Hellman. In *Algorithmic Number Theory*, 2000.

[19] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT'10*, 2010.

[20] kilic. High Speed BLS12-381 Implementation in Go. https://github.com/kilic/bls12-381, 2020. Accessed: 2022-04.

[21] Protocol Labs. Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf, 2017.

[22] Russell W. F. Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I*, pages 530–560, 2019.

[23] Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. In *ACM STOC*, 1991.

[24] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In Carl Pomerance, editor, *CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[25] Andrew Miller. Storing UTXOs in a Balanced Merkle Tree (zero-trust nodes with O(1)-storage), 2012. https://bitcointalk.org/index.php?topic=101734.msg1117428.

[26] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.

[27] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of Correct Computation. In *TCC'13*, 2013.

[28] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming Authenticated Data Structures. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, pages 353–370, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[29] protolambda. Implementation KZG in Go. https://github.com/protolambda/go-kzg, 2021. Accessed: 2022-04.

[30] Yi Qian, Yupeng Zhang, Xi Chen, and Charalampos Papamanthou. Streaming authenticated data structures: Abstraction and implementation. In Gail-Joon Ahn, Alina Oprea, and Reihaneh Safavi-Naini, editors, *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*, pages 129–139. ACM, 2014.

[31] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies. In *FC'17*, 2017.

[32] Shravan Srinivasan. Implementation of Hyperproofs in Go. https://github.com/hyperproofs/hyperproofs-go, 2021. Accessed: 2022-04.

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

[34] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf, 2020.

[35] Peter Todd. Making UTXO Set Growth Irrelevant With Low-Latency Delayed TXO Commitments, 2016. https://petertodd.org/2016/delayed-txo-commitments.

[36] Alin Tomescu. *How to Keep a Secret and Share a Public Key (Using Polynomial Commitments)*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2020.

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

[38] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards Scalable Threshold Cryptosystems. In *IEEE S&P'20*, May 2020.

[39] Joachim von zur Gathen and Jurgen Gerhard. Fast Multiplication. In *Modern Computer Algebra*, chapter 8, pages 221–254. Cambridge University Press, 3rd edition, 2013.

[40] Joachim von zur Gathen and Jurgen Gerhard. Fast polynomial evaluation and interpolation. In *Modern Computer Algebra*, chapter 10, pages 295–310. Cambridge University Press, 3rd edition, 2013.

[41] Wikipedia contributors. Partial fraction decomposition - Wikipedia, the free encyclopedia, 2022. https://en.wikipedia.org/wiki/Partial_fraction_decomposition, 2022. Accessed: 2022-04.

[42] Joachim Zahnentferner. Chimeric Ledgers: Translating and Unifying UTXO-based and Account-based Cryptocurrencies. Cryptology ePrint Archive, Report 2018/262, 2018. https://eprint.iacr.org/2018/262.

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

[44] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 203–220, 2018.

A Detailed description of two-layer bucketing BalanceProofs

We present the detailed algorithms for two-layer bucketing BalanceProofs by taking aSVC [37] as the input

VC scheme. For notation simplicity, we also view $\mathbf{m} = [\mathbf{v}_{0,0}, \mathbf{v}_{0,1}, \ldots, \mathbf{v}_{p-1,t-1}]$ where p,t are the bucket sizes for the two layers and $\mathbf{v}_{i,j} = [m_{i\cdot\frac{n}{p}+j\cdot\frac{n}{pt}}, \ldots, m_{i\cdot\frac{n}{p}+(j+1)\cdot\frac{n}{pt}-1}]$. We set $\varphi = \omega^{\frac{n}{p}}$, $\theta = \omega^{\frac{n}{t}}$ and $\eta = \omega^{pt}$, where ω is an n-th root of unity.

(1) $\mathsf{VC}'.\mathsf{Gen}(1^\lambda, n, p, t) \to \mathsf{pp}$:

Pick $\alpha, \beta, \gamma \in \mathbb{Z}_p^*$ uniformly at random. Set

$$\mathsf{pp} = \left((g^{\gamma^k})_{k\in[0,\frac{n}{pt})}, (l_k = g^{\mathcal{L}_k(\gamma)})_{k\in[0,\frac{n}{pt})} \right).$$

Return

$$\mathsf{pp}' = \Big(\mathsf{pp}, (g^{r'_{j,k}(\beta,\gamma)}, g^{s'_{j,k}(\beta,\gamma)})_{j\in[0,t),k\in[0,\frac{n}{pt})}, \\ (g^{\alpha^i\beta^j\gamma^k}, l_{i,j,k}, g^{r_{i,j,k}(\alpha,\beta,\gamma)}, g^{s_{i,j,k}(\alpha,\beta,\gamma)})_{i\in[0,p),j\in[0,t),k\in[0,\frac{n}{pt})} \Big)$$

where $\mathcal{L}_i(x) = \prod_{l\in[0,p)\setminus i} \frac{x-\varphi^l}{\varphi^i-\varphi^l}$, $\mathcal{L}'_j(y) = \prod_{l\in[0,t)\setminus j} \frac{y-\theta^l}{\theta^j-\theta^l}$, $\mathcal{L}''_k(z) = \prod_{l\in[0,\frac{n}{pt})\setminus k} \frac{z-\eta^l}{\eta^k-\eta^l}$, $l_{i,j,k} = g^{\mathcal{L}_i(\alpha)\mathcal{L}'_j(\beta)\mathcal{L}''_k(\gamma)}$, and

$$r_{i,j,k}(x,y,z) = \frac{(\mathcal{L}_i(x)-1)\mathcal{L}'_j(y)\mathcal{L}''_k(z)}{x-\varphi^i},$$

$$s_{i,j,k}(x,y,z) = \mathcal{L}'_j(y)\mathcal{L}''_k(z) \prod_{l\in[0,p)\setminus i} (x-\varphi^l),$$

$$r'_{j,k}(y,z) = \frac{(\mathcal{L}'_j(y)-1)\mathcal{L}''_k(z)}{y-\theta^j}, s'_{j,k}(y,z) = \mathcal{L}''_k(z) \prod_{l\in[0,t)\setminus j} y-\theta^l.$$

(2) $\mathsf{VC}'.\mathsf{Commit}_{\mathsf{pp}'}(\mathbf{m}) \to (\mathsf{C}, \mathsf{aux})$:

Let

$$\mathsf{C} = \prod_{i\in[0,p),j\in[0,t),k\in[0,\frac{n}{pt})} (l_{i,j,k})^{v_{i,j,k}}.$$

Let $\Pi = (\Pi_i, (\Psi_{i,j}, (\pi_{i,j,k})_{k\in[0,\frac{n}{pt})})_{j\in[0,t)})_{i\in[0,p)}$ output by $\mathsf{VC}'.\mathsf{OpenAll}_{\mathsf{pp}'}(\mathbf{m})$. Initialize empty lists $(L_{i,j})_{i\in[0,p),j\in[0,t)}$. Return

$$(\mathsf{C}, [(L_{i,j})_{i\in[0,p),j\in[0,t)}; \Pi]).$$

(3) $\mathsf{VC}'.\mathsf{Open}_{\mathsf{pp}'}((i,j,k), \mathbf{m}, \mathsf{aux}) \to (\Pi_i, \Psi_{i,j}, \pi_{i,j,k})$:

Parse $\mathsf{aux} = [(L_{i,j})_{i\in[0,p),j\in[0,t)}; \Pi]$. If $L_{i,j} = \emptyset$, output $(\Pi_i, \Psi_{i,j}, \pi_{i,j,k})$ in Π. Otherwise for each update request (l, δ_l) in $L_{i,j}$ call $\mathsf{VC}.\mathsf{UpdProof}_{\mathsf{pp}}(l, \delta_l, k, \pi_{i,j,k})$ in turn and finally return the correct latest proof $(\Pi_i, \Psi_{i,j}, \pi_{i,j,k})$.

(4) $\mathsf{VC}'.\mathsf{OpenAll}_{\mathsf{pp}'}(\mathbf{m}) \to \Pi$:

Compute batch proofs $(\Pi_i, (\Psi_{i,j})_{j\in[0,t)})_{i\in[0,p)}$ from $\mathbf{m}$. Call $\mathsf{VC}.\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{v}_{i,j}) \to (\pi_{i,j,k})_{k\in[0,\frac{n}{pt})}$ for all $i \in [0,p), j \in [0,t)$ and return

$$\Pi = (\Pi_i, (\Psi_{i,j}, (\pi_{i,j,k})_{k\in[0,\frac{n}{pt})})_{j\in[0,t)})_{i\in[0,p)}.$$

(5) $\mathsf{VC}'.\mathsf{Agg}_{\mathsf{pp}'}(I, (v_{ijk}, (\Pi_i, \Psi_{i,j}, \pi_{i,j,k}))_{(i,j,k)\in I}) \to \pi_I$:

Denote sets

$$S = \{i | \exists j,k, \text{ s.t., } (i,j,k) \in I\},$$

$$T = \{(i,j) | \exists k, \text{ s.t., } (i,j,k) \in I\},\ T_i = \{j | \exists k, \text{ s.t., } (i,j,k) \in I\}.$$

Partition $I = \bigcup_{(i,j)\in T} K_{i,j}$. For each $(i,j) \in T$, call

$$\mathsf{VC}.\mathsf{Agg}_{\mathsf{pp}}(K_{i,j}, (v_{i,j,k}, \pi_{i,j,k})_{(i,j,k)\in K_{i,j}}) \to \pi_{K_{i,j}}.$$

Return

$$\pi_I := (\Pi_i, \Psi_{i,j}, \pi_{K_{i,j}})_{(i,j)\in T}.$$

(6) $\mathsf{VC}'.\mathsf{Verify}_{\mathsf{pp}'}(\mathsf{C}, I, (v_{i,j,k})_{(i,j,k)\in I}, \pi_I) := \{0,1\}$:

If $|I| = 1$, then parse π_I as $(\Pi_i, \Psi_{i,j}, \pi_{i,j,k})$ and then check if the following holds:

$$e(\mathsf{C}/g^{v_{i,j,k}}, g) = e(\Pi_i, g^{\alpha-\varphi^i}) \cdot e(\Psi_{i,j}, g^{\beta-\theta^j}) \cdot e(\pi_{i,j,k}, g^{\gamma-\eta^k}).$$

If $|I| > 1$, then parse π_I as form $(\Pi_i, \Psi_{i,j}, \pi_{K_{i,j}})_{(i,j)\in T}$ where $K_{i,j}$ defined on I is the same as in (5). Check the following (where $c_{i,j}(z)$ is interpolation over $(v_{i,j,k})_{(i,j,k)\in K_{i,j}}$):

$$e(\mathsf{C}/g^{c_{i,j}(\gamma)}, g) = e(\Pi_i, g^{\alpha-\varphi^i}) \cdot e(\Psi_{i,j}, g^{\beta-\theta^j}) \cdot \\ e(\pi_{K_{i,j}}, g^{\prod_{(i,j,k)\in K_{i,j}}(\gamma-\eta^k)}).$$

(7) $\mathsf{VC}'.\mathsf{UpdCom}_{\mathsf{pp}'}((i,j,k), \delta, \mathsf{C}) \to \mathsf{C}'$:

Return $\mathsf{C}' = \mathsf{C} \cdot (l_{i,j,k})^\delta$.

(8) $\mathsf{VC}'.\mathsf{UpdAllProofs}_{\mathsf{pp}'}((i,j,k), \delta, \mathsf{aux}) \to \mathsf{aux}'$:

Parse $\mathsf{aux} = [(L_{i,j})_{i\in[0,p),j\in[0,t)}; \Pi]$. Then update $\Pi_{i'}$ and $\Psi_{i',j'}$ for any $i' \in [0,p)$, $j' \in [0,t)$ as follows:

If $i' = i$, then $\Pi'_{i'} = \Pi_{i'} \cdot (g^{r_{i,j,k}(\alpha,\beta,\gamma)})^\delta$; otherwise, $\Pi'_{i'} = \Pi_{i'} \cdot (g^{s_{i,j,k}(\alpha,\beta,\gamma)})^u \cdot (g^{s_{i',j,k}(\alpha,\beta,\gamma)})^{-u}$ where $u = \delta/(p(1-\varphi^{i'-i}))$.

If $i' = i$ and $j' = j$, then $\Psi'_{i',j'} = \Psi_{i',j'} \cdot (g^{r'_{j,k}(\beta,\gamma)})^\delta$; else if $i' = i, j' \neq j$, then $\Psi'_{i',j'} = \Psi_{i',j'} \cdot (g^{s'_{j,k}(\beta,\gamma)})^{u'} \cdot (g^{s'_{j',k}(\beta,\gamma)})^{-u'}$ where $u' = \delta/(t(1-\theta^{j'-j}))$; otherwise, do nothing to $\Psi_{i',j'}$.

Parse $\mathbf{m}' = (\mathbf{v}'_{i,j})_{i\in[0,p),j\in[0,t)}$. Append (k,δ) to $L_{i,j}$. If $|L_{i,j}| \geq \sqrt{\frac{n}{pt}}$, then call $\mathsf{VC}.\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{v}'_{i,j})$ to get all new individual proofs inside $\mathbf{v}'_{i,j}$: $(\pi'_{i,j,l})_{l\in[0,\frac{n}{pt})}$ and empty $L_{i,j}$; otherwise set $(\pi'_{i,j,l})_{l\in[0,\frac{n}{pt})} = (\pi_{i,j,l})_{l\in[0,\frac{n}{pt})}$.

Let aux' collect all the new lists and proofs. Return aux'.

(9) $\mathsf{VC}'.\mathsf{UpdProof}_{\mathsf{pp}'}((i,j,k), \delta, (i',j',k'), (\Pi_{i'}, \Psi_{i',j'}, \pi_{i',j',k'})) \to (\Pi'_{i'}, \Psi'_{i',j'}, \pi'_{i',j',k'})$:

Use $g^{r_{i,j,k}(\alpha,\beta,\gamma)}, g^{s_{i,j,k}(\alpha,\beta,\gamma)}, g^{r'_{j,k}(\beta,\gamma)}, g^{s'_{j,k}(\beta,\gamma)}$ to update $(\Pi_{i'}, \Psi_{i',j'})$ to $(\Pi'_{i'}, \Psi'_{i',j'})$. If $i = i'$ and $j = j'$, then call $\mathsf{VC}.\mathsf{UpdProof}_{\mathsf{pp}}$ $(k, \delta, k', \pi_{i',j',k'}) \to \pi'_{i',j',k'}$ and return $(\Pi'_{i'}, \Psi'_{i',j'}, \pi'_{i',j',k'})$; otherwise, return $(\Pi'_{i'}, \Psi'_{i',j'}, \pi_{i',j',k'})$.

B Assumptions

We first present q-SDH assumption [4].

Assumption B.1 (q-Strong Diffie-Hellman (q-SDH))**.** Let $\tau \in_R \mathbb{Z}_p^*$. Given as input a $(q+1)$-tuple $(g, g^{\tau}, \ldots, g^{\tau^q}) \in \mathbb{G}^{q+1}$, for any adversary $\mathcal{A}_{q\text{-SDH}}$, we have the following for any $a \in \mathbb{Z}_p \setminus \{-\tau\}$:

$$\Pr[\mathcal{A}_{q\text{-SDH}}(g, g^{\tau}, \ldots, g^{\tau^q}) = (a, g^{\frac{1}{\tau+a}})] \leq \mathsf{negl}(\lambda)$$

Next, we show the q-SBDH assumption [16] which will be used to give soundness proofs for our VC schemes. q-SBDH assumption is a variant of q-SDH assumption.

Assumption B.2 (q-Strong Bilinear Diffie-Hellman (q-SBDH))**.** Let $\tau \in_R \mathbb{Z}_p^*$. Given as input a $(q+1)$-tuple $(g, g^{\tau}, \ldots, g^{\tau^q}) \in \mathbb{G}^{q+1}$, for any adversary $\mathcal{A}_{q\text{-SBDH}}$, we have the following for any $a \in \mathbb{Z}_p \setminus \{-\tau\}$:

$$\Pr[\mathcal{A}_{q\text{-SBDH}}(g, g^{\tau}, \ldots, g^{\tau^q}) = (a, e(g,g)^{\frac{1}{\tau+a}})] \leq \mathsf{negl}(\lambda).$$

C Security Proofs

In this section, we show the soundness proof of Theorem 4.1 through the following lemmas.

Lemma C.1. Our 2-layer bucketing VC presented in Appendix A has the complexities mentioned in Theorem 4.1.

Proof.

1. The public parameter size is $O(p \cdot t \cdot \frac{n}{pt}) = O(n)$.
2. The commitment needs only one group element.
3. Updating the bucket proofs requires $O(p+t)$ time. For each subvector, it has size $n/(pt)$ and requires $O(\sqrt{n/(pt)}\log(n/(pt)))$ time to update all proofs.
4. Each subvector has list size at most $O(\sqrt{n/(pt)})$.
5. For batch proof size, there are at most pt buckets and thus $O(pt)$ proof size.
6. Aggregation and its verification time depend on the polynomial calculations (interpolations) over the index set.

□

Lemma C.2. Our two-layer individual evaluation proofs ($|I| = 1$) from Appendix A are sound as per Definition 2.3 under q-SBDH assumption.

Proof. Suppose there exists some adversary $\mathcal{A}$ that breaks Definition 2.3 where $I = J$ and $|I| = 1$. We show how to break $(n-1)$-SBDH assumption by constructing an adversary $\mathcal{B}$.

Suppose $\mathcal{B}$ is given $(n-1)$-SBDH parameters $(g^{\alpha^i})_{i\in[0,n)}$. $\mathcal{B}$ first guesses the index (i,j,k) that $\mathcal{A}$ forged, which he can do with probability $\frac{1}{\mathsf{poly}(\lambda)}$. Second, $\mathcal{B}$ "tweaks" the SDH public parameters into the protocol public parameters, i.e., sets $\beta - \theta^j = r_0(\alpha - \varphi^i)$ and $\gamma - \eta^k = r_1(\alpha - \varphi^i)$, where r_0, r_1 are randomly chosen. Third, $\mathcal{B}$ calls $\mathcal{A}$ with the "tweaked" public parameters as input.

$\mathcal{A}$ should output the forged index (i,j,k) together with $\mathsf{C}, w_0, w_0', w_1, w_1', w_2, w_2', z, z'$ such that we have the following:

$$e(\mathsf{C}/g^z, g) = e\left(w_0, g^{\alpha-\varphi^i}\right) \cdot e\left(w_1, g^{\beta-\theta^j}\right) \cdot e\left(w_2, g^{\gamma-\eta^k}\right)$$

$$e(\mathsf{C}/g^{z'}, g) = e\left(w_0', g^{\alpha-\varphi^i}\right) \cdot e\left(w_1', g^{\beta-\theta^j}\right) \cdot e\left(w_2', g^{\gamma-\eta^k}\right)$$

Divide the two equations:

$$e(g^{z'-z}, g) = e\left(\frac{w_0}{w_0'}, g^{\alpha-\varphi^i}\right) \cdot e\left(\frac{w_1}{w_1'}, g^{\beta-\theta^j}\right) \cdot e\left(\frac{w_2}{w_2'}, g^{\gamma-\eta^k}\right)$$

Note that $\beta - \theta^j = r_0(\alpha - \varphi^i)$ and $\gamma - \eta^k = r_1(\alpha - \varphi^i)$, then we have

$$e(g,g)^{z'-z} = \left(e\left(\frac{w_0}{w_0'}, g\right) e\left(\frac{w_1}{w_1'}, g^{r_0}\right) e\left(\frac{w_2}{w_2'}, g^{r_1}\right)\right)^{\alpha-\varphi^i}$$

Finally we have

$$e(g,g)^{\frac{1}{\alpha-\varphi^i}} = \left(e\left(\frac{w_0}{w_0'}, g\right) e\left(\frac{w_1}{w_1'}, g^{r_0}\right) e\left(\frac{w_2}{w_2'}, g^{r_1}\right)\right)^{\frac{1}{z'-z}},$$

which breaks the $(n-1)$-SBDH assumption. □

Then we present the soundness lemma for batch proofs (we can even provide the lemma for the version with the halving technique).

Lemma C.3. Our two-layer batch evaluation proofs ($|I| > 1$) from Appendix A with the halving technique are sound as per Definition 2.3 under q-SBDH assumption.

Proof. First note that with halving technique, the batch bucket proofs should be (sets like S, T are defined in Appendix A):

$$\Pi_I := \prod_{i\in S} \Pi_i^{1/h(\varphi^i)}, \ \Psi_{i,J} := \prod_{j\in T_i} \Psi_{i,j}^{1/h_i(\theta^j)},$$

where $h(x) = \sum_{i\in S}\prod_{i'\in S\setminus i}(x - \varphi^{i'})$ and $\forall i \in S$, $h_i(y) = \sum_{j\in T_i}\prod_{j'\in T_i\setminus j}(y - \theta^{j'})$. To verify these, we check if the following holds:

$$e(\mathsf{C}, g) = e(\Pi_I, g^{A(\alpha)}) \cdot \prod_{i\in S}\Bigg(e\left(\Psi_{i,J}, g^{A_i(\alpha,\beta)}\right) \cdot \prod_{j\in T_i}\left(e\left(\pi_{K_{i,j}}, g^{A_{i,j}(\alpha,\beta,\gamma)}\right) \cdot e\left(g^{c_{i,j}(\gamma)}, g^{\mathcal{L}'_{i,j}(\alpha,\beta)}\right)\right)\Bigg)$$

where $c_{i,j}(z)$ is interpolation over $(v_{i,j,k})_{(i,j,k)\in K_{i,j}}$, and

$$\mathcal{L}_i(x) = \prod_{i'\in S\setminus i}\frac{x-\varphi^{i'}}{\varphi^i - \varphi^{i'}}, \quad \mathcal{L}'_{i,j}(x,y) = \mathcal{L}_i(x)\prod_{j'\in T_i\setminus j}\frac{y-\theta^{j'}}{\theta^j - \theta^{j'}},$$

$$A(x) = \prod_{i\in S}(x-\varphi^i), \quad A_i(x,y) = \mathcal{L}_i(x)\prod_{j\in T_i}(y-\theta^j),$$

$$A_{i,j}(x,y,z) = \mathcal{L}'_{i,j}(x,y) \prod_{(i,j,k)\in K_{i,j}} (z-\eta^k).$$

Now suppose there exists some $\mathcal{A}$ that breaks Definition 2.3. We show how to break $(n-1)$-SBDH assumption by constructing an adversary $\mathcal{B}$. Suppose $\mathcal{B}$ is given $(n-1)$-SBDH parameters $(g^{\alpha^i})_{i\in[0,n)}$. $\mathcal{B}$ first guesses the index $(\mathbf{i},\mathbf{j},\mathbf{k})$ that $\mathcal{A}$ forged, which he can do with probability $\frac{1}{\mathsf{poly}(\lambda)}$. Second, $\mathcal{B}$ "tweaks" the SDH public parameters into protocol public parameters, i.e., sets $\beta-\theta^{\mathbf{j}} = r_0(\alpha-\varphi^{\mathbf{i}})$, $\gamma-\eta^{\mathbf{k}} = r_1(\alpha-\varphi^{\mathbf{i}})$. Third, $\mathcal{B}$ calls $\mathcal{A}$ with the "tweaked" public parameters.

$\mathcal{A}$ should output C and some I,I', where $(\mathbf{i},\mathbf{j},\mathbf{k}) \in I\cap I'$ is the position $\mathcal{A}$ will forge, together with proof $(\Pi_I,(\Psi_{i,J})_{i\in S},(\pi_{K_{i,j}})_{(i,j)\in T})$ for $(v_{i,j,k})_{(i,j,k)\in I}$ and proof $(\Pi_{I'},(\Psi'_{i,J'})_{i\in S'},(\pi'_{K'_{i,j}})_{(i,j)\in T'})$ for $(v'_{i,j,k})_{(i,j,k)\in I'}$. These satisfy the following equations (polynomials are derived in the same way for I,I', e.g., $A(x)$ for I and $A'(x)$ for I'):

$$e(\mathsf{C},g) = e(\Pi_I,g^{A(\alpha)})\cdot\prod_{i\in S}\Bigg(e\left(\Psi_{i,J},g^{A_i(\alpha,\beta)}\right)\cdot \prod_{j\in T_i}\left(e\left(\pi_{K_{i,j}},g^{A_{i,j}(\alpha,\beta,\gamma)}\right)\cdot e\left(g^{c_{i,j}(\gamma)},g^{\mathcal{L}'_{i,j}(\alpha,\beta)}\right)\right)\Bigg)$$

$$e(\mathsf{C},g) = e(\Pi'_{I'},g^{A'(\alpha)})\cdot\prod_{i\in S'}\Bigg(e\left(\Psi'_{i,J'},g^{A'_i(\alpha,\beta)}\right)\cdot \prod_{j\in T'_i}\left(e\left(\pi'_{K'_{i,j}},g^{A'_{i,j}(\alpha,\beta,\gamma)}\right)\cdot e\left(g^{c'_{i,j}(\gamma)},g^{\mathcal{L}''_{i,j}(\alpha,\beta)}\right)\right)\Bigg)$$

$c_{i,j}(z)$ is the interpolation of claimed values in $K_{i,j}$. Let $c_0 = c_{\mathbf{i},\mathbf{j}}(\eta^{\mathbf{k}}) = v_{\mathbf{i},\mathbf{j},\mathbf{k}}$, we can write $c_{\mathbf{i},\mathbf{j}}(z) = d(z)(z-\eta^{\mathbf{k}})+c_0$ so that $c_{\mathbf{i},\mathbf{j}}(\gamma) = r_1 d(\gamma)(\alpha-\varphi^{\mathbf{i}})+c_0$.

Write $A(x) = a(x)(x-\varphi^{\mathbf{i}})$ so that $A(\alpha) = a(\alpha)(\alpha-\varphi^{\mathbf{i}})$.

Also, if $i=\mathbf{i}$, write $A_{\mathbf{i}}(x,y) = b(x,y)(y-\theta^{\mathbf{j}})$ so that $A_{\mathbf{i}}(\alpha,\beta) = r_0 b(\alpha,\beta)(\alpha-\varphi^{\mathbf{i}})$. If $i\neq\mathbf{i}$, write $A_i(x,y) = a_i(x,y)(x-\varphi^{\mathbf{i}})$ so that $A_i(\alpha,\beta) = a_i(\alpha,\beta)(\alpha-\varphi^{\mathbf{i}})$.

If $i\neq\mathbf{i}$, write $A_{i,j}(x,y,z) = a_{i,j}(x,y,z)(x-\varphi^{\mathbf{i}})$ so that $A_{i,j}(\alpha,\beta,\gamma) = a_{i,j}(\alpha,\beta,\gamma)(\alpha-\varphi^{\mathbf{i}})$. If $i=\mathbf{i}, j\neq\mathbf{j}$, write $A_{\mathbf{i},j}(x,y,z) = b_j(x,y,z)(y-\theta^{\mathbf{j}})$ so that $A_{\mathbf{i},j}(\alpha,\beta,\gamma) = r_0 b_j(\alpha,\beta,\gamma)(\alpha-\varphi^{\mathbf{i}})$. If $i=\mathbf{i}, j=\mathbf{j}$, write $A_{\mathbf{i},\mathbf{j}}(x,y,z) = f(x,y,z)(z-\eta^{\mathbf{k}})$ so that $A_{\mathbf{i},\mathbf{j}}(\alpha,\beta,\gamma) = r_1 f(\alpha,\beta,\gamma)(\alpha-\varphi^{\mathbf{i}})$.

If $i\neq\mathbf{i}$ or $j\neq\mathbf{j}$, write $\mathcal{L}'_{i,j}(x,y) = l_1(x,y)(x-\varphi^{\mathbf{i}})$ or $\mathcal{L}'_{i,j}(x,y) = l_2(x,y)(y-\theta^{\mathbf{j}})$ so that $\mathcal{L}'_{i,j}(\alpha,\beta) = l_1(\alpha,\beta)(\alpha-\varphi^{\mathbf{i}})$ or $\mathcal{L}'_{i,j}(\alpha,\beta) = r_0 l_2(\alpha,\beta)(\alpha-\varphi^{\mathbf{i}})$.

Above all, our conclusion here is that we will have the following form for the first equation above, where "…" stands for some polynomial evaluated on (α,β,γ):

$$e(\mathsf{C},g) = e\left(g^{r_1 d(\gamma)},g^{\mathcal{L}'_{\mathbf{i},\mathbf{j}}(\alpha,\beta)}\right)^{\alpha-\varphi^{\mathbf{i}}}\cdot e\left(\Pi_I,g^{\cdots}\right)^{\alpha-\varphi^{\mathbf{i}}}\cdot \prod_{i\in S}\Bigg(e\left(\Psi_{i,J},g^{\cdots}\right)^{\alpha-\varphi^{\mathbf{i}}}\cdot\left(\prod_{j\in T_i} e\left(\pi_{K_{i,j}},g^{\cdots}\right)^{\alpha-\varphi^{\mathbf{i}}}\right)\cdot \left(\prod_{j\in T_i,(i,j)\neq(\mathbf{i},\mathbf{j})} e\left(g^{c_{i,j}(\gamma)},g^{\cdots}\right)^{\alpha-\varphi^{\mathbf{i}}}\right)\Bigg)\cdot e\left(g^{c_0},g^{\mathcal{L}'_{\mathbf{i},\mathbf{j}}(\alpha,\beta)}\right)$$

Similarly, we do the same things to I' and its polynomials $A'(x), A'_i(x,y), A'_{i,j}(x,y,z), \mathcal{L}''_{i,j}(x,y)$ as above and write a similar equation. Divide these two equations we have

$$e(g,g)^{c_0\mathcal{L}'_{\mathbf{i},\mathbf{j}}(\alpha,\beta)-c'_0\mathcal{L}''_{\mathbf{i},\mathbf{j}}(\alpha,\beta)} = \left(\frac{\triangle'}{\triangle}\right)^{\alpha-\varphi^{\mathbf{i}}},$$

where $\triangle$ is as follows and $\triangle'$ is computed similarly:

$$\triangle = e\left(\Pi_I,g^{a(\alpha)}\right)\cdot\prod_{i\in S\setminus\mathbf{i}}\Bigg(e\left(\Psi_{i,J},g^{a_i(\alpha,\beta)}\right)\cdot \prod_{j\in T_i}\left(e\left(\pi_{K_{i,j}},g^{a_{i,j}(\alpha,\beta,\gamma)}\right)\cdot e\left(g^{c_{i,j}(\gamma)},g^{l_1(\alpha,\beta)}\right)\right)\Bigg)\cdot e\left(\Psi_{\mathbf{i},J},g^{r_0 b(\alpha,\beta)}\right)\cdot\prod_{j\in T_{\mathbf{i}}\setminus\mathbf{j}}\left(e\left(\pi_{K_{\mathbf{i},j}},g^{r_0 b_j(\alpha,\beta,\gamma)}\right)\cdot e\left(g^{c_{\mathbf{i},j}(\gamma)},g^{r_0 l_2(\alpha,\beta)}\right)\right)\cdot e\left(\pi_{K_{\mathbf{i},\mathbf{j}}},g^{r_1 f(\alpha,\beta,\gamma)}\right)\cdot e\left(g^{r_1 d(\gamma)},g^{\mathcal{L}'_{\mathbf{i},\mathbf{j}}(\alpha,\beta)}\right).$$

Equivalently, we have

$$e(g,g)^{\frac{c_0\mathcal{L}'_{\mathbf{i},\mathbf{j}}(\alpha,\beta)-c'_0\mathcal{L}''_{\mathbf{i},\mathbf{j}}(\alpha,\beta)}{\alpha-\varphi^{\mathbf{i}}}} = \frac{\triangle'}{\triangle}.$$

Recall that $\beta = r_0(\alpha-\varphi^{\mathbf{i}})+\theta^{\mathbf{j}}$. Denote

$$M(x) = \mathcal{L}'_{\mathbf{i},\mathbf{j}}(x, r_0(x-\varphi^{\mathbf{i}})+\theta^{\mathbf{j}}),$$

$$M'(x) = \mathcal{L}''_{\mathbf{i},\mathbf{j}}(x, r_0(x-\varphi^{\mathbf{i}})+\theta^{\mathbf{j}}),$$

here we have $M(\alpha) = \mathcal{L}'_{\mathbf{i},\mathbf{j}}(\alpha,\beta)$ and $M'(\alpha) = \mathcal{L}''_{\mathbf{i},\mathbf{j}}(\alpha,\beta)$.

Note that $M(\varphi^{\mathbf{i}}) = M'(\varphi^{\mathbf{i}}) = 1$, then

$$c_0 M(\varphi^{\mathbf{i}}) - c'_0 M'(\varphi^{\mathbf{i}}) = c_0 - c'_0 \neq 0,$$

thus we know $c_0M(x)-c'_0M'(x)$ is not divisible by $x-\varphi^{\mathbf{i}}$. Then we can compute through polynomial division that

$$c_0M(x)-c'_0M'(x) = q(x)(x-\varphi^{\mathbf{i}})+r,\ r\neq 0,$$

and evaluate this equation on $x=\alpha$:

$$c_0\mathcal{L}'_{\mathbf{i},\mathbf{j}}(\alpha,\beta)-c'_0\mathcal{L}''_{\mathbf{i},\mathbf{j}}(\alpha,\beta) = q(\alpha)(\alpha-\varphi^{\mathbf{i}})+r.$$

Finally, we have

$$e(g,g)^{q(\alpha)+\frac{r}{\alpha-\varphi^{\mathbf{i}}}} = \frac{\triangle'}{\triangle},$$

so we can compute

$$e(g,g)^{\frac{1}{\alpha-\varphi^{\mathbf{i}}}} = \left(\frac{\triangle'/\triangle}{e(g,g^{q(\alpha)})}\right)^{1/r},$$

which breaks the $(n-1)$-SBDH assumption. $\square$