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× and up to 100× 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× 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.

**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× and up to 100× 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× 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 $m = [m_0, \ldots, m_{n-1}]$ and proofs $\pi_0, \ldots, \pi_{n-1}$ for each position. A verifier who has the commitment $C$ can later verify a proof $\pi_i$ 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 $\pi_i$ for $i \in I$ and aggregate them into a single, succinct proof $\pi$ 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 \cite{8} used to produce the aggregate proof. In this paper we are therefore interested in the following question:

\textit{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 \textit{not} maintainable, such as aSVC \cite{37} and Pointproofs \cite{15}, and produces another naturally-aggregatable \textit{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 work, the input vector commitment must support opening all proofs in $O(n \log n)$ time (as is the case with aSVC \cite{37} and Pointproofs \cite{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 $\sigma_0, \ldots, \sigma_{n-1}$ for every position of the vector. Whenever there is an update $(i, \delta)$ (change $m_i$ to $m_i + \delta$) to the vector, aSVC would apply $(i, \delta)$ to all proofs $\sigma_0, \ldots, \sigma_{n-1}$, leading to $\Omega(n)$ time. Instead of doing this expensive operation, we just store the update $(i, \delta)$ in a log. Of course this is problematic. When we want to query a proof $\sigma_j$ for an index $j$ in the future, we need to first apply all updates in the log on proof $\sigma_j$. However, given that updating a single aSVC proof $\sigma_j$ is cheap (constant time), we can fetch the updated proof $\sigma_j$ after $t$ updates in time $O(t)$ by applying all $t$ updates one-by-one on $\sigma_j$. 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 \textit{amortized} time for our update algorithm is $O(\sqrt{n} \log n)$. We finally show how to deanmortize 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× 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 \textit{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 $\Pi_i$ is a batch proof over the indices of $P_i$, with respect to commitment $C$ of the whole vector. Proofs $\Pi_i$ are always updated immediately during an update, leading to $O(p)$ update time. An individual evaluation proof $\sigma_{i,j}$ 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 \textit{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 \cite{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 $O(\sqrt{n/p} \log(n/p))$ 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
While update time for Hyperproofs ranges from 2 to 3 ms, Macrobenchmarks 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.

| Scheme               | $|\pi_i|$ | $|\pi_I|$ | Aggregate | UpdAllProofs | Query $\pi_i$ | Verify $\pi_i$ | Verify $\pi_I$ | Gen | $\text{pp}$ |
|----------------------|----------|----------|-----------|-------------|---------------|----------------|----------------|-----|----------|
| 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, $\pi_i$ is the individual proof for position $i$, $\pi_I$ is the aggregated proof for an index set $I$, $\text{pp}$ represents public parameters and $k = |I|$.
It is easy to see that for any \( i \in [0, n] \), \( \phi(\omega') = m_i \).

**Bilinear pairings** [18, 23]. We use \((p, G_1, G_2, G_T, e, g_1, g_2)\) to denote the parameters associated with pairings. In particular, \( G_1, G_2 \) and \( G_T \) are groups of prime order \( p \), \( g_i \) is a generator of \( G_i \) and pairing function \( e : G_1 \times G_2 \rightarrow G_T \) is such that \( \forall u \in G_1, w \in G_2 \) and \( a, b \in \mathbb{Z}_p \), it is \( e(u, w)^{ab} = e(u, w)^{ab} \). For simplicity, we use the same group \( G \), with generator \( g \), for both \( G_1 \) and \( G_2 \) 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. \( \text{Gen}(1^\lambda, n) \rightarrow \text{pp} \): Given security parameter \( \lambda \) and vector size \( n \), it outputs public parameters \( \text{pp} \).

2. \( \text{Commit}_{\text{pp}}(m) \rightarrow (C, aux) \): Given vector \( m \), it outputs commitment \( C \) along with auxiliary information \( aux \).

3. \( \text{Open}_{\text{pp}}(i, m, aux) \rightarrow \pi_i \): Given index \( i \), vector \( m \) and auxiliary information \( aux \), it outputs a proof \( \pi_i \).

4. \( \text{OpenAll}_{\text{pp}}(m) \rightarrow (\pi_0, \ldots, \pi_{n-1}) \): Given vector \( m \), it outputs all proofs \( \pi_0, \ldots, \pi_{n-1} \).

5. \( \text{Agg}_{\text{pp}}(f, \{\pi_i, m_i\}_{i\in I}) \rightarrow \pi_I \): Given proof-value pairs \( \{(\pi_i, m_i)\}_{i \in I} \), for \( I \subseteq [0, n] \), it outputs an aggregate proof \( \pi_I \).

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

7. \( \text{UpdCom}_{\text{pp}}(i, \delta, C) \rightarrow C' \): Given index \( i \), value \( \delta \), commitment \( C \), it outputs \( C' \) reflecting position \( i \) changing by \( \delta \).

8. \( \text{UpdAllProofs}_{\text{pp}}(i, \delta, \{\pi_i, aux\}) \rightarrow (\{\pi'_i\}, aux') \): Given index \( i \), difference \( \delta \), proofs \( \{\pi_i\}_{j \in [0,n]} \) and auxiliary information \( aux \), it outputs updated proofs \( \{\pi'_i\}_{j \in [0,n]} \) and updated auxiliary information \( aux' \), to reflect position \( i \) changing by \( \delta \).

9. \( \text{UpdProof}_{\text{pp}}(i, \delta, j, \pi_j) \rightarrow \pi_{j'} \): Given index \( i, \delta, \pi_j \), it outputs updated proof \( \pi_{j'} \) to reflect position \( i \) changing by \( \delta \).

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

1. \( \Pr[1 \leftarrow \text{Verify}_{\text{pp}}(C', i, m'_i, \pi'_i)] = 1 \);

2. For all \( I \subseteq [0, n] \) it is

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

**Definition 2.3** (VC soundness). For all PPT adversaries \( A \),

\[ \Pr \begin{bmatrix} \text{pp} \leftarrow \text{Gen}(1^\lambda, n), \\ (C, I, (m_i)_{i \in I}, (\pi_i, \pi_j) \leftarrow A(1^\lambda, \text{pp}) : \\ 1 \leftarrow \text{Verify}_{\text{pp}}(C, I, (m_i)_{i \in I}, \pi_I) \land \\ 1 \leftarrow \text{Verify}_{\text{pp}}(C, J, (m'_j)_{j \in J}, \pi'_J) \land \\ \exists k \in I \cap J \text{ such that } m_k \neq m'_k \end{bmatrix} \leq \text{neg}(\lambda) \]

**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 linearized 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]

\[ (g, g^\alpha, \ldots, g^{\alpha^{n-1}}) \]
aSVC represents a vector \( m = [m_0, \ldots, m_{n-1}] \) as the polynomial \( \phi(x) = \sum_{i \in [0,n]} L_i(x) \cdot m_i \) such that \( \phi(\omega') = m_i \), where \( \omega' \) is the \( i \)-th \( n \)-th root of unity. The commitment to the vector is then simply the group element \( g^{\phi(t)} \) that can be computed using the public parameters above. Similar to [9], the public parameters also contain commitments to all Lagrange polynomials \( g^{e_L(t)} \), which are used to compute the proofs.

A proof \( \pi_i \) that \( m_i \) is the value of vector \( m \) at position \( i \) is simply the commitment to the polynomial

\[ q_i(x) = \frac{\phi(x) - m_i}{x - \omega'} \]  

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

\[ q(x) = \frac{\phi(x) - R(x)}{A_I(x)} \]

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

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

Thus we can compute \( c_i = 1/A'_I(\omega') \) with \( O(|I| \log^2 |I|) \) field operations [40] and aggregate \( \pi_I = \prod_{i \in I} \pi_i' \) 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) \(\text{Gen}(1^k, n) \rightarrow \text{pp}:\) Pick \(\tau \in \mathbb{Z}_p^\ast\) uniformly at random. Output public parameters

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

where \(l_i = g^{c_i(\tau)}, a_i = g^{A_i(\tau)/\tau - \omega'}, u_i = g^{e - \omega'}\), where \(A_i(x) = \prod_{i \in [0,n]} (x - \omega').\)

(2) \(\text{Commit}_\text{pp}(m) \rightarrow C:\) Output \(C = \prod_{i \in [0,n]} (l_i)^{m_i} \).

(3) \(\text{Open}_\text{pp}(i, m) \rightarrow \pi_i:\) Output \(\pi_i = g^{q_i(x)},\) where \(q_i(x)\) is defined in Equation 1.

(4) \(\text{OpenAll}_\text{pp}(m) \rightarrow (\pi_0, \ldots, \pi_{n-1}):\) Output all proofs for \(m.\)

(5) \(\text{Agg}_\text{pp}(I, (\pi_i, m_i)_{i \in I}) \rightarrow \pi'_I:\) Compute \(A_I(x) = \prod_{i \in I} (x - \omega')\) and \(e_i = (A_i'(\omega'))^{-1}.\) Output \(\pi'_I = \prod_{i \in I} \pi'_i.\)

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

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

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

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

(8) \(\text{UpdAllProofs}_\text{pp}(i, \delta, \pi_0, \ldots, \pi_{n-1}) \rightarrow (\pi'_0, \pi'_1, \ldots, \pi'_{n-1}):\) Call \(\text{VC.}\text{UpdProof}_\text{pp}\) (see next) for every individual proof.

(9) \(\text{UpdProofs}_\text{pp}(i, \delta, j, \pi_j) \rightarrow \pi'_j:\) If \(i = j,\) output \(\pi'_j = \pi_j \cdot (u_i)^{\delta};\) if \(i \neq j,\) compute \(w_{i,j} = a_j^{1/(\omega'' - \omega')}, a_j^{1/(\omega'' - \omega')}\) and \(u_{i,j} = a_j^{1/(\omega'' - \omega')}\) and \(u_{i,j} = w_i^{1/(\omega'' - \omega')},\) and return \(\pi'_j = \pi_j \cdot (u_i)^{\delta}.\)

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 \(\text{VC}'\). 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 \(\text{VC}'\). 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\(\text{Agg}_\text{pp}(I, (\pi_i, m_i)_{i \in I})\). (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

\(\text{VC.Gen, VC.Commit, VC.Open, VC.OpenAll, VC.Aggregate...}\)

that satisfy the properties above, and let \(\text{VC}'\) be the output vector commitment with algorithms

\(\text{VC'.Gen, VC'.Commit, VC'.Open, VC'.OpenAll, VC'.Aggregate...}\)

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 \(\text{VC}'\) will still be \(g^{q_i(x)},\) where \(q(x) = \sum_{j \in [0,n]} L_j(x) \cdot m_j\) 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 \((i, \delta)\) appears, we do not use the VC.\text{UpdAllProofs} algorithm since this would incur \(\Omega(n)\) cost (For example, for aSVC, \text{UpdAllProofs} iterates through all \(n\) proofs one by one.) What we do is \text{append} the update \((i, \delta)\) in a list \(L\) of size \(\sqrt{n}\), which takes just constant time. The list \(L\) serves as the auxiliary information \text{aux}. When the list \(L\) becomes full, our compiler calls VC.\text{OpenAll} to compute fresh proofs \((\pi_0, \ldots, \pi_{n-1})\) 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.\text{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.\text{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 \(\text{VC}\) into \(\text{VC}'\)

We now provide the detailed algorithms for \(\text{VC}'\):

(1) \(\text{VC}'.\text{Gen}(1^k, n) \rightarrow \text{pp}:\) \n
\[
\text{Return VC.Gen}(1^k, n).
\]

(2) \(\text{VC}'.\text{Commit}_\text{pp}(m) \rightarrow (C, \text{aux}):\) \n
\[
\text{Let VC.Commit}_\text{pp}(m) \rightarrow (C, \text{aux}_0). \text{ Let } \pi_0, \ldots, \pi_{n-1} \text{ output by VC.\text{OpenAll}_\text{pp}(m). Initialize empty list } L \text{ of size } \sqrt{n}. \text{ Return } (C, [L, \pi_0, \ldots, \pi_{n-1}]).
\]

(3) \(\text{VC}'.\text{Open}_\text{pp}(i, m, \text{aux}) \rightarrow \pi_i:\)
Let \( \text{aux} = [L; \pi_0, \ldots, \pi_{n-1}] \). If \( L = \emptyset \), output \( \pi_t \). Otherwise call \( \text{VC} . \text{UpdProof}_{pp}(j, \delta, i, \pi_t) \) for each update \((j, \delta)\) in \( L \) and return the latest proof \( \pi'_j \).

(4) \( \text{VC}' . \text{OpenAll}_{pp}(\mathbf{m}) \rightarrow (\pi_0, \pi_1, \ldots, \pi_{n-1}) \):

Return \( \text{VC} . \text{OpenAll}_{pp}(\mathbf{m}) \rightarrow (\pi_0, \pi_1, \ldots, \pi_{n-1}) \).

(5) \( \text{VC}' . \text{Agg}_{pp}(I, (\pi_i, m_i)_{i \in I}) \rightarrow \pi'_I \):

Return \( \text{VC} . \text{Agg}_{pp}(I, (\pi_i, m_i)_{i \in I}) \rightarrow \pi'_I \).

(6) \( \text{VC}' . \text{Verify}_{pp}(C, I, (m_i)_{i \in I}, \pi_I) \rightarrow \{0, 1\} \):

Return \( \text{VC} . \text{Verify}_{pp}(C, I, (m_i)_{i \in I}, \pi_I) \rightarrow \{0, 1\} \).

(7) \( \text{VC}' . \text{UpdCom}_{pp}(i, \delta, C) \rightarrow C' \):

Return \( \text{VC} . \text{UpdCom}_{pp}(i, \delta, C) \rightarrow C' \).

(8) \( \text{VC}' . \text{UpdAllProofs}_{pp}(i, \delta, \text{aux}) \rightarrow \text{aux}' \):

Parse \( \text{aux} \) as \([L; \pi_0, \ldots, \pi_{n-1}]\). If \( |L| < \sqrt{n} \), append \((i, \delta)\) to \( L \) and return \([L \cup (i, \delta); \pi_0, \ldots, \pi_{n-1}]\); Otherwise call \( \text{VC} . \text{OpenAll}_{pp}(\mathbf{m}) \) to compute \( \pi'_0, \ldots, \pi'_{n-1} \) and return \([0; \pi'_0, \ldots, \pi'_{n-1}]\).

(9) \( \text{VC}' . \text{UpdProof}_{pp}(i, \delta, j, \pi_j) \rightarrow \pi'_j \):

Call \( \text{VC} . \text{UpdProof}_{pp}(i, \delta, j, \pi_j) \rightarrow \pi'_j \) and return \( \pi'_j \).

### 3.2 Deamortizing UpdAllProofs

Note that the output algorithm \( \text{VC}' . \text{UpdAllProofs} \) must call \( \text{VC} . \text{OpenAll} \) every \( \sqrt{n} \) updates—this leads to large worst-case update time \( \Omega(n) \). Here we show how to de-amortize \( \text{VC}' . \text{UpdAllProofs} \) and achieve sublinear worst-case update time. The crucial observation for the deamortization is the fact that \( \text{VC} . \text{OpenAll} \) (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 de-amortized \( \text{VC}' . \text{UpdAllProofs} \) 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 \( \text{VC} . \text{UpdAllProofs} \), 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 \( T_{\sqrt{n}} \) 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 \( \text{aSVC} \), 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 \( \text{VC} \) be a vector commitment scheme that is correct (per Definition 2.2) and sound (per Definition 2.3). Then (1) the output \( \text{VC}' \) is also correct and sound; (2) \( \text{VC}' . \text{UpdAllProofs} \) takes at most \( T / \sqrt{n} = o(n) \) time \((T \text{ is the time of } \text{VC} . \text{OpenAll}) \) and \( \text{VC}' \) takes at most \( P / \sqrt{n} = o(n) \) time \((P \text{ is the time of } \text{VC} . \text{UpdProof}) \); (3) other complexities of \( \text{VC}' \) are the same as \( \text{VC} \).

**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 \( \text{VC} \) and \( \text{VC}' \). If an adversary finds commitment and proofs that break the soundness of \( \text{VC}' \), then an adversary can use the same objects to break the soundness of \( \text{VC} \). Complexities of \( \text{VC}' . \text{UpdAllProofs} \) and \( \text{VC}' . \text{Open} \) were analyzed previously in this section. \( \square \)

### 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.,

\[
g, g^x, \ldots, g^{(x)^{p-1}}
\]

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

Bucket proofs

m

π

P

π

P

π

m (n = 12, p = 4)

Then we can view m as p subvectors v₀, v₁, ..., vₚ₋₁ where each vᵢ contains indices in Pᵢ. Note |vᵢ| = n/p. Similarly, we can write Lagrange interpolations \( \phi_i(x) \) for each vᵢ as

\[
\phi_i(x) = \sum_{j \in P_i} L_{i,j}(x) \cdot m_j,
\]

where \( L_{i,j}(x) = \prod_{k \in P_i \setminus j} \frac{x - \omega^k}{\omega^j - \omega^k} \).

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, \( \Pi_i = g^{q_i(x)} \) is a KZG batch proof for the index set \( P_i \) of vector m. We call \( \Pi_i \) a bucket proof. We can still provide individual proofs for an index \( j \in P_i \) inside subvector vᵢ based on the KZG equation

\[
\phi_i(x) = q_i(x)(x - \omega^j) + \phi_i(\omega^j) = q_i(x)(x - \omega^j) + \phi(\omega^j).
\]

We call \( \pi_{i,j} = g^{q_i(x)} \) an individual proof for position \( j \) of vᵢ.

**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 \( \Pi_i \) for index set \( P_i \) inside vector m and the individual proof \( \pi_{i,j} \) for position \( j \) inside vector vᵢ. Therefore, the resulting evaluation proof for position \( j \) is \( (\Pi_i, \pi_{i,j}) \). A verifier who has the commitment C and the claimed evaluations \( \Phi(\omega^j) \) for \( j \in J \), can verify one evaluation proof \( (\Pi_i, \pi_{i,j}) \) by checking the following equation (say \( j \in P_i)\):

\[
e(C / g^\zeta, g) = e(\Pi_i, g^{q_i(x)}) \cdot e(\pi_{i,j}, g^{q_i(x)}/g^{\omega^j}).
\]

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 \( \Pi_i \) is the same as before and \( \pi_{i,J} = g^{q_i,J} \)

\[
\phi_i(x) = q_i,J(x) \prod_{j \in J} (x - \omega^j) + c_J(x),
\]

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

\[
e(C / g^{\zeta'(\tau)}, g) = e(\Pi_i, g^{q_i,J(\tau)}) \cdot e(\pi_{i,J}, g^{\prod_{j \in J}(x - \omega^j)}). \]

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 give 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 v₀. 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 \((j, \delta)\) 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 \((j, \delta)\). To do that, we can update each bucket proof \( \Pi_i \) in \( O(1) \) time. We now explain how to do that. We distinguish two cases.

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

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

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

Figure 1: Aggregate proofs in bucketing. The aggregate proof for index set \( I = \{0,1,5,11\} \) is \((\Pi_0, \pi_{0,\{0,1\}}), (\Pi_1, \pi_{1,5}), (\Pi_3, \pi_{3,11})\).
δ \cdot L_{i,j}(x). Consider now the quotient polynomial \( q_i(x) \) as defined before. This can be written as

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

Thus we can precompute

$$r_{j,i}(x) = \frac{L_j(x) - L_{i,j}(x)}{\prod_{k \in P_i}(x - \omega^k)}$$

and save update parameters

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

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

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 \( \prod_{k \in P_i}(x - \omega^k) | L_j(x) - L_{i,j}(x) \). For that, we only need to show that \( \forall k \in P_i, x - \omega^k | L_j(x) - L_{i,j}(x) \). If \( k \in P_i \) and \( k \neq j \), then this is trivial from the definition. For \( k = j \), \( L_j(\omega^k) - L_{i,j}(\omega^k) = 1 - 1 = 0 \), so \( L_j(x) - L_{i,j}(x) \) has a factor \( x - \omega^j \) and then \( x - \omega^j | L_j(x) - L_{i,j}(x) \).

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

$$q_i'(x) = \frac{\phi'(x) - \phi_i(x)}{\prod_{k \in P_i}(x - \omega^k)} = \frac{(\phi(x) - \phi_i(x)) + \delta \cdot L_j(x)}{\prod_{k \in P_i}(x - \omega^k)} = q_i(x) + \frac{\delta \cdot L_j(x)}{\prod_{k \in P_i}(x - \omega^k)}.$$

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

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

in the generation algorithm. The proof \( \Pi_i \) can be updated as \( \Pi'_i = \Pi_i \cdot (g'^{(i)}(\tau))^{\delta} \). 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 \( (j, \delta) \), assuming \( j \notin P_i \), we only need to update individual proofs inside \( \Pi_i \) since for any \( k \neq i \), \( \phi_i(x) \) does not change at all. Using our compiler technique from Section 3 (which involves keeping record lists \( \{\hat{L}_i\}_{i \in [0,p]} \)), 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 \( \phi = \omega^\alpha \) and \( \theta = \omega^\beta \), where \( \omega \) is an \( n \)-th root of unity.

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

| \( j \in [0,n/p] \) | \( \sum_{j \in [0,n/p]} L_j'(y) \cdot v_{i,j} \) | \( \prod_{k \in [0,n/p]} x - \omega^k \), where \( L_j(y) = \prod_{k \in [0,n/p]} x - \omega^k \).

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

$$\phi(x,y) = \sum_{i \in [0,p]} L_i(x) \phi_i(y), \text{ where } L_i(x) = \prod_{k \in [0,n/p], k \neq i} x - \omega^k.$$

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

$$\phi(x,y) = \phi_i(y) + q_i(x,y)(x - \omega^\beta),$$

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

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

**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

$$\left( g'^{(i)}(\tau) \right)_{i \in [0,p], j \in [0,n/p]}.$$

where \( \alpha, \beta \) 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 \( \Pi_i = g^{\phi(\alpha,\beta)} \) as well as the individual proof \( \pi_{i,j} = g^{\phi_{i,j}(\beta)} \) and then provide the evaluation proof as \( (\Pi, \pi_{i,j}) \). The verifier can verify it by checking the following is true:

$$e(C/g^{\beta}, g) = e(\Pi_i, g^{\alpha - \beta}) \cdot e(\pi_{i,j}, g^{\beta - \theta^j}).$$
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 $\Pi_i$. After receiving an update request $((k, j), \delta)$, we have $\phi_i'(x, y) = \phi_i(x, y) + \delta \cdot L_k(x) L_j(y)$, $\Phi_i(y) = \Phi_i(y) + \delta \cdot L_j(y)$.

The $k = i$ case. When the bucket proof $\Pi_i$ 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

$$q_i'(x, y) = \frac{\phi_i'(x, y) - \phi_i(y)}{x - \phi^i} = q_i(x, y) + \frac{\delta \cdot L_k(x) L_j(y)}{x - \phi^i}.$$ 

Thus we can precompute $r_i,j(x, y) = L_j(y) (L_i(x) - 1) / (x - \phi^i)$ and save update parameters

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

in the generation algorithm. The proof $\Pi_i$ can be updated to $\Pi_i' = \Pi_i \cdot (g^{\phi_i, (\alpha, \beta)} \delta)$. Note that the update parameter 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

$$q_i'(x, y) = \frac{\phi_i'(x, y) - \phi_i(y)}{x - \phi^i} = q_i(x, y) + \frac{\delta \cdot L_k(x) L_j(y)}{x - \phi^i}$$

$$= q_i(x, y) + \prod_{k \in [0, p] \setminus \{i\}} \frac{\delta \cdot L_k(x) L_j(y)}{x - \phi^i}$$

$$= q_i(x, y) + \frac{\delta \cdot \prod_{k \in [0, p] \setminus \{i\}} (\phi^k - \phi^i) (x - \phi^i) (x - \phi^k)}{c_k (x - \phi^k - \phi^i)}$$

$$= q_i(x, y) + \frac{\delta}{c_k (x - \phi^k - \phi^i)} (s_{i,j}(x, y) - s_{i,j}(x, y)),$$

where $c_k = \prod_{k \in [0, p] \setminus \{i\}} (\phi^k - \phi^i) = p \cdot \phi^{k-1}$ and

$$s_{i,j}(x, y) = L_j(y) \prod_{k \in [0, p] \setminus \{i\} \cup \{j\}} (x - \phi^k).$$

Thus we can just pre-compute and save update parameters

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

in the generation algorithm. The proof $\Pi_i$ can be updated as

$$\Pi_i' = \Pi_i \cdot (g^{\phi_i, (\alpha, \beta)} u_{i,j} \cdot g^{\phi_i, (\alpha, \beta)} - u_{i,j},$$

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

**Updating individual proofs.** After receiving an update $((i, j), \delta)$, we need to update individual proofs only inside $v_i$, since any $\phi_i(y)$ 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 $(\Pi_i = g^{\phi_i, (\alpha, \beta)}, \pi_{i,2} = g^{\pi_i, (\alpha, \beta)})$ and $(\Pi_j = g^{\phi_j, (\alpha, \beta)}, \pi_{j,3} = g^{\pi_j, (\alpha, \beta)})$. 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

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

and

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

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

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

(3)

In the above, $r(x, y)$ is such that $r(\phi^i, y) = \phi_i(y)$ and $r(\phi^j, y) = \phi_j(y)$, 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 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

$$r(x, y) = \ell_{i,j}(x) \cdot \phi_i(y) + \ell_{i,j}(y) \cdot \phi_j(y)$$

$$= z \cdot \ell_{i,j}(x) + q_{i,2}(y) \cdot (y - \theta^2) \cdot \ell_{i,j}(x)$$

$$+ w \cdot \ell_{i,j}(x) + q_{j,3}(y) \cdot (y - \theta^3) \cdot \ell_{i,j}(x)$$

(4)

where $\ell_{i,j}(x) = (x - \phi^i)/(\phi^i - \phi^j)$ 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

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

and then checks to see if

$$e(g^{\phi(\alpha, \beta)}, g) = R \cdot e(\Pi_{i,j}, g^{(\alpha - \phi^i)(\alpha - \phi^j)}).$$
Layer 0: 2 buckets, identified by $x$
Layer 1: 4 buckets, identified by $x,y$
Layer 2: 12 positions in subvectors, identified by $x,y,z$

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 $\Pi_i$ and $\Pi_j$ as

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

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

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 $r$-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 $O(n^{1/(k+2)})$ 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. $O(|l| \log^2 |l|)$ to aggregate proofs corresponding to an index set $I$;
7. $O(|l| \log^2 |l|)$ 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.

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

---

the computation in the $O(n \log n)$-time algorithm VC.OpenAll into $\sqrt{n} \cdot O(\sqrt{n} \log n)$-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 $100 \times$ 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>
<thead>
<tr>
<th>$L = \log_2 n$</th>
<th>20</th>
<th>22</th>
<th>24</th>
<th>26</th>
<th>28</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit (min)</td>
<td>0.47</td>
<td>1.69</td>
<td>6.99</td>
<td>28.4</td>
<td>114.8 *</td>
<td></td>
</tr>
<tr>
<td>OpenAll (hrs)</td>
<td>0.86</td>
<td>3.74 *</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>UpdAllProofs (s)</td>
<td>3.03</td>
<td>6.65</td>
<td>14.3</td>
<td>30.5</td>
<td>64.4</td>
<td>135.7</td>
</tr>
<tr>
<td>Query Indiv. (s)</td>
<td>0.02</td>
<td>0.05</td>
<td>0.09</td>
<td>0.18</td>
<td>0.38</td>
<td>0.77</td>
</tr>
<tr>
<td>Indiv. Verify (ms)</td>
<td>1.18</td>
<td>1.20</td>
<td>1.19</td>
<td>1.21</td>
<td>1.20</td>
<td>1.21</td>
</tr>
<tr>
<td>Aggregate (s)</td>
<td>0.38</td>
<td>0.41</td>
<td>0.35</td>
<td>0.43</td>
<td>0.39</td>
<td>0.41</td>
</tr>
<tr>
<td>Agg. Verify (s)</td>
<td>0.43</td>
<td>0.42</td>
<td>0.44</td>
<td>0.42</td>
<td>0.43</td>
<td>0.42</td>
</tr>
<tr>
<td>Indiv. proof size</td>
<td>48 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agg. proof size</td>
<td>48 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Single-thread execution for two-layer bucketing.

### 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 $G_1$ element and can be verified with two pairings. Our two-layer bucketing scheme has proofs with three $G_1$ 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...
Batch proof size

Aggregation time

Verification time for aggregated proofs

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 ~ 110s to aggregate 1024 proofs and 13 ~ 17s to verify the aggregated proofs, when $20 \leq L \leq 30$. 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× 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
We simulate the case for $L$ (when using halving technique, around 50 KB for $I$) are responsible for updating all individual proofs by replaying proof-serving nodes (PSNs) to have their proofs served. PSNs their balance proofs locally—instead they contact incentivized the batch proof in the incoming block. Users do not maintain spectator to the previous transactions. When receiving a block aggregate those proofs and updates the commitment with re-computation (as in proof-of-work), saving on bandwidth might be more critical. For a more detailed discussion of the cost 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).

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Two-layer</th>
<th>Hyperproofs</th>
<th>Merkle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block proposal ($P$)</td>
<td>1.82 sec</td>
<td>2.23 min</td>
<td>81 min</td>
</tr>
<tr>
<td>Block validation ($V$)</td>
<td>0.19 sec</td>
<td>17.5 sec</td>
<td>0.18 sec</td>
</tr>
<tr>
<td>Proof maintenance ($M$)</td>
<td>38.72 sec</td>
<td>5.14 sec</td>
<td>4.7 sec</td>
</tr>
<tr>
<td>Total ($P + hV + M$)</td>
<td>45 sec</td>
<td>8 min</td>
<td>81 min</td>
</tr>
</tbody>
</table>

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.

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 multi-linear 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


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 $m = [v_0, v_1, \ldots, v_{p-1}]$ where $p$ are the bucket sizes for the two layers and $v_{i,j} = [m_{i,j}^0, m_{i,j}^1, \ldots, m_{i,j}^{p-1}]$. We set $\varphi = \omega^a\theta^\tau$ and $\eta = \omega^a\theta^\tau$, where $\omega$ is an $n$-th root of unity.

(1) VC'.Gen($1^k, n, p, t$) → pp:
- Pick $\alpha, \beta, \gamma \in \mathbb{Z}_p$ uniformly at random. Set
  $$pp = \left( (g^{\varphi l}_k)_{k \in [0, p]}, (l_k = g^{\varphi l_k}_k)_{k \in [0, p]} \right).$$
- Return

$$pp' = (pp, (g^{\alpha j}_k, l'_{i,j,k}, g^{\alpha j}_k, l'_{i,j,k}, 0)), (g^{\alpha j}_k, l'_{i,j,k}, g^{\alpha j}_k, l'_{i,j,k}, 0)_{i \in [0, p], j \in [0, t]}$$

where $L(x) = \prod_{i \in [0, p]} \frac{x - \theta^i}{x - \varphi^i}$, $L'_j(y) = \prod_{i \in [0, p]} \frac{y - \theta^i}{y - \varphi^i}$, $L''_j(z) = \prod_{i \in [0, p]} \frac{z - \theta^i}{z - \varphi^i}$, $L'_{i,j}(y) = \prod_{i \in [0, p], j \in [0, t]} (x - \varphi^j)$, $L''_j(z) = \prod_{i \in [0, p], j \in [0, t]} (x - \varphi^j)$, $r_{i,j}(x,y,z) = \frac{L'_j(y) - 1}{y - \theta^i} L''_j(z) \prod_{i \in [0, p]} (x - \varphi^j)$, $s_{i,j}(x,y,z) = \frac{L'_j(y) - 1}{y - \theta^i} L''_j(z) \prod_{i \in [0, p]} (x - \varphi^j)$.

(2) VC'.Commit$_{pp'}$(m) → (C, aux):
- Let
  $$C = \prod_{i \in [0, p], j \in [0, t]} (l_{i,j,k})^{n_{i,j,k}}.$$ Let $\Pi = (\Pi_{i,j,k}, n_{i,j,k})_{i \in [0, p], j \in [0, t]}$ output by VC.OpenAll$_{pp'}$(m). Initialize empty lists $(L_{i,j,k})_{i \in [0, p], j \in [0, t]}$. Return
  $$\Pi = (\Pi_{i,j,k}, n_{i,j,k})_{i \in [0, p], j \in [0, t]} : \Pi_i.$$ (3) VC'.OpenAll$_{pp'}((i, j, k), m, aux)$ → (Pi, $\Psi_{i,j,k}$):
- Parse aux = $[(L_{i,j,k})_{i \in [0, p], j \in [0, t]} : \Pi_i]$. If $L_{i,j,k} = \emptyset$, output $(\Pi_i, \Psi_{i,j,k}, n_{i,j,k})$ in $\Pi$. Otherwise for each update request $(l, \delta_{i,j,k})$ in $L_{i,j,k}$ call VC.UpdCom$_{pp'}(l, \delta_{i,j,k}, \Psi_{i,j,k})$ in turn and finally return the correct latest proof $(\Pi_i, \Psi_{i,j,k}, n_{i,j,k})$.

(4) VC'.UpdCom$_{pp'}((i, j, k), (i', j', k'), (\Pi_i, \Psi_{i,j,k}), \pi_{i,j,k})$ → (\Pi', \Psi', \pi')
- Compute batch proofs $(\Pi_i', \Psi_{i,j,k})_{i \in [0, p]}$ from m. Call VC.OpenAll$_{pp'}(m, \Psi_{i,j,k})$ → ($\pi_{i,j,k}$) for all $i \in [0, p], j \in [0, t]$ and return
  $$\Pi = (\Pi_i, \Psi_{i,j,k}, n_{i,j,k})_{i \in [0, p], j \in [0, t]} : \Pi_i.$$ (5) VC'.Agg$_{pp'}((i, j, k), (\Pi_i, \Psi_{i,j,k}, \pi_{i,j,k}))_{(i, j, k) \in I}$ → $\pi_I$:
- Denote sets
  $$S = \{ i | \exists j, k, s.t., (i, j, k) \in I \},$$
  $$T = \{ (i, j) | \exists k, s.t., (i, j, k) \in I \},$$
  $$T_i = \{ j k, s.t., (i, j, k) \in I \}.$$ Partition $I = \bigcup_{(i, j) \in T} K_{i,j}$. For each $(i, j) \in T$, call
  $$VC.Agg_{pp'}(K_{i,j}, \pi_{i,j,k})_{(i,j,k) \in K_{i,j}} \rightarrow \pi_{K_{i,j}}.$$ Return
  $$\pi_I := (\Pi_i, \Psi_{i,j,k}, \pi_{i,j,k})_{(i,j,k) \in T}.$$ (6) VC'.Verify$_{pp'}(C, I, (\pi_{i,j,k})_{(i,j,k) \in I}, \pi_I) := \{0, 1\}$:
- If $|I| = 1$, then parse $\pi_I$ as $(\Pi_i, \Psi_{i,j,k})$ and then check if the following holds:
  $$e(C / g_i^{\varphi j} \cdot g_i^\varphi, g_i^\varphi) \cdot e(\Psi_{i,j,k}, g_i^\varphi, g_i^\varphi) \cdot e(\pi_{i,j,k}, g_i^\varphi, g_i^\varphi).$$
- If $|I| > 1$, then parse $\pi_I$ as $(\Pi_i, \Psi_{i,j,k}, \pi_{i,j,k})_{(i,j,k) \in K_{i,j}}$ where $K_{i,j}$ defined on $I$ is the same as in (5). Check the following (where $c_{i,j}(z)$ is interpolation over $(\pi_{i,j,k})_{(i,j,k) \in K_{i,j}}$):
  $$e(C / g_i^{\varphi j} \cdot g_i^\varphi, g_i^\varphi) \cdot e(\Psi_{i,j,k}, g_i^\varphi, g_i^\varphi, g_i^\varphi) \cdot e(c_{i,j}(z), g_i^\varphi, g_i^\varphi) \cdot e(c_{i,j}(z), g_i^\varphi, g_i^\varphi).$$

(7) VC'.UpdCom$_{pp'}((i, j, k), \delta, C) \rightarrow C'$:
- Return $C' = C - (l_{i,j,k})^\delta$.

(8) VC'.UpdAllProofs$_{pp'}((i, j, k), \delta, aux) \rightarrow aux'$:
- Parse aux = $[(L_{i,j,k})_{i \in [0, p], j \in [0, t]} : \Pi_i]$. Then update $\Pi_i$ and $\Psi_{i,j,k}$ for any $i \in [0, p], j \in [0, t]$ as follows:
  If $i = i'$, then $\Pi_i' = \Pi_i + (g^{\varphi j}_{i,j,k} (\delta_j) \cdot \varphi^j)$; otherwise, $\Pi_i' = \Pi_i + (g^{\varphi j}_{i,j,k} (\delta_j) \cdot \varphi^j)$.
  If $j = j'$, then $\Psi_{i,j,k}' = \Psi_{i,j,k} + (g^{\varphi j}_{i,j,k} (\delta_j) \cdot \varphi^j)$, $\Psi_{i,j,k}' = \Psi_{i,j,k} + (g^{\varphi j}_{i,j,k} (\delta_j) \cdot \varphi^j)$.
  Otherwise, do nothing to $\Psi_{i,j,k}'$.
- Parse $aux' = (v'_{i,j,k})_{i \in [0, p], j \in [0, t]}$. Append $(\delta, k)$ to $L_{i,j,k}$. If $L_{i,j,k} \leq \sqrt{\frac{p}{pt}}$, then call VC.OpenAll$_{pp'}(v'_{i,j,k})$ to get all new individual proofs inside $v'_{i,j,k}$; $(\pi'_{i,j,k})_{i \in [0, p]}$, and empty $L_{i,j,k}$; otherwise set $(\pi'_{i,j,k})_{i \in [0, p]} = (\pi_{i,j,k})_{i \in [0, p]}$.

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

(9) VC'.UpdAllProofs$_{pp'}((i, j, k), \delta, (i', j', k'), (\Pi_{i,j,k}, \Psi_{i,j,k}, \pi_{i,j,k})) \rightarrow (\Pi_{i,j,k}', \Psi_{i,j,k}', \pi_{i,j,k}')$:
- Use $g^{\varphi j}_{i,j,k} (\delta_j) \cdot g^{\varphi j}_{i,j,k} (\delta_j) \cdot g^{\varphi j}_{i,j,k} (\delta_j) \cdot g^{\varphi j}_{i,j,k} (\delta_j)$ to update $(\Pi_i, \Psi_{i,j,k})$ to $(\Pi_i', \Psi_{i,j,k}')$. If $i = i'$ and $j = j'$, then call VC.UpdProof$_{pp'}(k, \delta, (\pi_{i,j,k}')) \rightarrow \pi_{i,j,k}'$ and return $(\Pi_i', \Psi_{i,j,k}', \pi_{i,j,k}')$; otherwise, return $(\Pi_i', \Psi_{i,j,k}', \pi_{i,j,k}')$.

B Assumptions

We first present $q$-SDH assumption [4].
Assumption B.1 ($q$-Strong Diffie-Hellman ($q$-SDH)). Let $\tau \in \mathbb{Z}_p^*$. Given as input a $(q+1)$-tuple $(g, g^q, \ldots, g^{q^r}) \in \mathbb{G}_q^{q+1}$, for any adversary $\mathcal{A}_{q}$-SDH, we have the following for any $a \in \mathbb{Z}_p \setminus \{-\tau\}$:

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

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

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

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

### 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 $O(p \cdot \log \frac{n}{p}) = O(n)$.
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 $O(\sqrt{n/(pt)})$ 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 $(g^x_i)_{i \in [0,n)}$. $\mathcal{B}$ first guesses the index $(i, j, k)$ that $\mathcal{A}$ forged, which he can do with probability $1/\text{poly}(\lambda)$. Second, $\mathcal{B}$ "tweak"s the SDH public parameters into the protocol public parameters, i.e., sets $\beta - \theta' = r_0(\alpha - \varphi')$ and $\gamma - \eta^k = r_1(\alpha - \varphi')$, 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 $C, v_0, w_0, w_1, w_2, w_3, z, \tilde{z}$ such that we have the following:

$$e(C/\gamma, g) = e(w_0, g^{\alpha \varphi'}) \cdot e(w_1, g^{\beta - \theta'}) \cdot e(w_2, g^{\tau \gamma})$$

$$e(C/\gamma', g) = e(w_0', g^{\alpha \varphi'}) \cdot e(w_1', g^{\beta - \theta'}) \cdot e(w_2', g^{\tau \gamma})$$

Divide the two equations:

$$e(g^{\gamma' - \gamma}, g) = e(w_0, w_0') \cdot e(w_1, w_1') \cdot e(w_2, w_2')$$

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

$$e(g, g)^{\gamma' - \gamma} = \left(e(w_0, w_0') \cdot e(w_1, w_1') \cdot e(w_2, w_2')\right)^{\frac{1}{\tau \gamma}}$$

Finally we have

$$e(g, g)^{\alpha \varphi'} = e(w_0, w_0') \cdot e(w_1, w_1') \cdot e(w_2, w_2')$$

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

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 $q$-SDBH assumption.

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

$$\Pi_I := \prod_{i \in S} \Psi_{I, j} := \prod_{i \in T_j} \Psi_{I, j}^{1/h(\theta')},$$

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

$$e(C, g) = e(\Pi_I, g^A(\alpha)) \cdot \prod_{i \in S} \left(e(\Psi_{I, j}^{h(\alpha \theta')}, g^A(\alpha \theta'))\right)$$

$$\prod_{j \in T_j} \left(\prod_{i \in S} \left(\prod_{j \in T_i} \left(\prod_{i \in S} (x - \varphi^i) \cdot \Psi_{I, j}^{1/h(\theta')}, e(g^A(\alpha \theta'), g^A(\alpha \theta'))\right)\right)\right)$$

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

$$\Lambda(x) := \prod_{i \in S} (x - \varphi^i), \quad \Lambda_{i, j}(x, y) = \Lambda_j(x) \prod_{j \in T_j} (y - \theta^j),$$

$$A(\Lambda) := \prod_{i \in S} (x - \varphi^i), \quad A_{i, j}(x, y) = \Lambda_j(x) \prod_{j \in T_j} (y - \theta^j)$$
\[ A_{i,j}(x,y,z) = L'_{i,j}(x,y) \prod_{(i,j,k) \in K_i} (z - \eta^k). \]

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 \((g^d_{i,j})_{i \in \mathcal{O}_n}\). \( \mathcal{B} \) first guesses the index \((i,j,k)\) that \( \mathcal{A} \) forged, which he can do with probability \( \frac{1}{\text{poly}(\gamma)} \). Second, \( \mathcal{B} \) “tweaks” the SDH public parameters into protocol public parameters, i.e., sets \( \beta - \theta l = r_0 (\alpha - \varphi^l), \gamma - \eta^k = r_1 (\alpha - \varphi^l) \).

Third, \( \mathcal{B} \) calls \( \mathcal{A} \) with the “tweaked” public parameters.

\( \mathcal{A} \) should output \( C \) and some \( I', I' \), where \((i,j,k) \in I \cap I' \) is the position \( \mathcal{A} \) will forge, together with proof \((\Pi_I, (\Psi_{i,j})_{i \in \mathcal{S}}(\pi_{K_i}(i,j) \in \mathcal{T}))\) for \((v_{i,j,k}(i,j,k) \in I) \) and proof \((\Pi'_{I'}, (\Psi'_{i,j})_{i' \in \mathcal{S'}}(\pi_{K_{i'}}(i,j) \in \mathcal{T'})\) 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 \( I' \)):

\[
e(C, g) = \left( \prod_{j \in I} \left( e\left( \pi_{K_i,j}, g^{\alpha_i(j,\alpha,\beta)} \right) \right) \prod_{j \in I'} \left( e\left( \pi'_{K'_i,j}, g^{\alpha'_i(j,\alpha,\beta)} \right) \right) \right)
\]

\[
e(C, g) = \left( \prod_{j \in I} \left( e\left( \pi_{K_i,j}, g^{\alpha_i(j,\alpha,\beta)} \right) \right) \prod_{j \in I'} \left( e\left( \pi'_{K'_i,j}, g^{\alpha'_i(j,\alpha,\beta)} \right) \right) \right)
\]

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

Write \( A(x) = a(x)(x - \varphi^l) \) so that \( A(\alpha) = a(\alpha)(\alpha - \varphi^l) \).

Also, if \( i = 1 \), write \( A_1(x,y) = b(x,y)(y - \theta_0) \) so that \( A_1(\alpha, \beta) = r_0 b(\alpha, \beta)(\alpha - \varphi^l) \). If \( i \neq 1 \), write \( A_{i,1}(x,y) = a_i(x)(x - \varphi^l) \) so that \( A_i(\alpha, \beta) = a_i(\alpha)(\alpha - \varphi^l) \).

If \( i = 1 \) or \( i \neq 1 \), write \( A_{i,1}(x,y) = a_i(x)(x - \varphi^l) \) so that \( A_i(\alpha, \beta, \gamma) = a_i(\alpha, \beta, \gamma)(\alpha - \varphi^l) \).

Equivalently, we have

\[
e(g,g)^{\varphi l} = \frac{\varphi l}{\varphi l}.
\]

Recall that \( \varphi = r_0 (\alpha - \varphi^l) + \theta l \). Denote

\[
M(x) = L'_{i,j}(x, r_0 (x - \varphi^l) + \theta l),
M'(x) = L''_{i,j}(x, r_0 (x - \varphi^l) + \theta l),
\]

here we have \( M(\alpha) = L'_{i,j}(\alpha, \beta) \) and \( M'(\alpha) = L''_{i,j}(\alpha, \beta) \). Note that \( M(\varphi^l) = M'(\varphi^l) = 1 \), then

\[
c_0 M(\varphi^l) - c_0 M'(\varphi^l) = c_0 - c_0' \neq 0,
\]

thus we know \( c_0 M(x) - c_0' M'(x) \) is not divisible by \( x - \varphi^l \).

Then we can compute through polynomial division that

\[
c_0 M(x) - c_0' M'(x) = q(x)(x - \varphi^l) + r, r \neq 0,
\]

and evaluate this equation on \( x = \alpha \):

\[
c_0 L'_{i,j}(\alpha, \beta) - c_0' L''_{i,j}(\alpha, \beta) = q(\alpha)(\alpha - \varphi^l) + r.
\]

Finally, we have

\[
e(g,g)^{\varphi l} + \frac{\varphi l}{\varphi l} = \frac{\Delta'}{\Delta},
\]

so we can compute

\[
e(g,g)^{\alpha - \varphi} = \left( \frac{\Delta'/\Delta}{\varphi l} \right)^{1/r},
\]

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