BalanceProofs: Maintainable Vector Commitments with Fast Aggregation

Weijie Wang* Annie Ulichney* Charalampos Papamanthou*

July 1, 2022

Abstract

We present BalanceProofs, the first vector commitment scheme that is *maintainable* (i.e., supporting sublinear updates) while also supporting 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×. 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, BalanceProofs' update time compared to Hyperproofs roughly doubles, but always stays in the range from 2 to 3 seconds. BalanceProofs can be viewed as a compiler that transforms any non-maintainable vector commitment with fast aggregation to a maintainable one with the aforementioned complexities. 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, including a comparison to Hyperproofs as well as applications of BalanceProofs to Verkle trees.

1 Introduction

Vector commitments (VC) is a cryptographic primitive recently proposed as a powerful alternative to traditional Merkle trees [29], 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 proof that might 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., [38, 42, 2, 15, 36, 48, 40, 6, 26, 11, 30, 1, 25]), 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 [38].

*Yale University. **Email:** {weijie.wang,annie.ulichney,charalampos.papamanthou}@yale.edu.

Scheme	$\lvert\pi_i\rvert$	$\lvert\pi_I\rvert$	Aggregate	UpdateAllProofs	Query π_i	Verify π_i	Verify π_I	Gen	$\lvert$pp$\rvert$
Hyperproofs[38]	$\log n$	$\log(k \log n)$	$k \log n$	$\log n$	$\log n$	$\log n$	$k \log n$	n	n
BalanceProofs	1	1	$k \log^2 k$	$\sqrt{n} \log n$	$\sqrt{n}$	1	$k \log^2 k$	$n \log n$	n
aSVC-bucketing	1	$\min\{k, n^{1/3}\}$	$k \log^2 k$	$n^{1/3} \log n$	$n^{1/3}$	1	$k \log^2 k$	$n^{4/3} \log n$	$n^{4/3}$
AMT-bucketing	$\log n$	$\min\{k \log n, n^{1/3}\}$	$k \log^2 k$	$n^{1/3} \log n$	$n^{1/3}$	$\log n$	$k \log^2 k$	$n \log^2 n$	$n \log n$

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

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., [38].) 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 [29, 43, 32, 35, 41]. For example, Merkle trees [29] or the vector commitment by Tomescu [41] (referred to as AMT for the rest of the paper) 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. [42] (referred to as aSVC for the rest of the paper), based on the KZG polynomial commitment [22] as well as the recently proposed *Pointproofs* [18] 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 [38] 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× to 1000× larger than aggregation using other VCs such as aSVC [42]). 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, a compiler that takes as input any naturally aggregatable vector commitment that is *not* maintainable, such as aSVC [42] and Pointproofs [18], and produces another naturally-aggregatable *and* maintainable vector commitment—in particular one with $O(\sqrt{n} \log n)$ update-all time (In our evalution, 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 [42] and Pointproofs [18]). Of course, this transformation introduces a tradeoff: 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 sublinear, 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 a non-maintainable vector commitment for a vector $\mathbf{m} = [m_0, \ldots, m_{n-1}]$ and suppose we have computed initial 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, a non-maintainable vector commitment would typically apply update (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—this is our data structure to maintain all proofs! 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 the proof π_j. However, given that updating a *single* proof π_j is cheap (in particular for aSVC and Pointproofs it is $O(1)$), 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 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× 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, which is a hybrid that uses *another* "base" vector commitment (e.g., aSVC and AMT) along with BalanceProofs.

The main idea is to spit the vector in p buckets $P_0, \ldots, P_{p-1}$ of n/p indices each (Note that for efficiency reasons, we have to carefully pick the split of indices into the buckets—we analyze this in the main body of the paper.) Then we apply the base vector commitment (such as 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. Our data structure now maintains the following components.

1. Proofs Π_i, with respect to the commitment C of the whole vector, are maintained for the commitment C_i of each bucket P_i $(i = 0, \ldots, p-1)$. Note that each proof Π_i is really an aggregate proof because each P_i contains n/p indices. Proofs Π_i are always updated immediately when there is an update, leading to $O(u(p))$ update time, where $u(p)$ is the update time of the base vector commitment.

2. Individual proofs $\pi_{i,j}$, with respect to the vector commitment C_i that corresponds to bucket P_i, for every index $j \in P_i$ are also maintained. Since we are using the BalanceProofs technique within each bucket, updating those proofs take $O(\sqrt{n/p} \log(n/p))$ time.

It is easy to see that for $p = n^{1/3}$, our update time becomes

$$u(n^{1/3}) + n^{1/3} \log n \,.$$

In our experiments, we instantiate the bucketing approach using two base commitments, aSVC and AMT, leading to different tradeoffs (For AMT $u(n) = \log n$ while for aSVC $u(n) = n$ and therefore, while asymptotically the same, bucketing with AMT has slightly better update time.) Interestingly enough, the bucketing technique increases the size of individual proofs by just a single group element (e.g., when using aSVC). This is because, for proving a single vector index

i, one needs to prove the correctness of i within P_j as well as the correctness of P_j within the original vector. However, the size of the aggregate proof is *not* constant anymore: This is because to support aggregation of an arbitrary set of indices I, one might need to touch more than one buckets, and in particular, 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.)

Evaluation. We perform an extensive evaluation of BalanceProofs. Our evaluation has three main components.

Benchmarking BalanceProofs (with bucketing). We observe that the basic BalanceProofs version (without bucketing and with aSVC as input to the compiler) has aggregation and aggregate verification that is in the order of milliseconds (for aggregating 1024 individual proofs), but has costly updates (more that 100 seconds for performing a single update on a vector of 2^{30} entries). We show that our bucketing approach can improve the update time to approximately 2.5 seconds, by increasing the aggregate proof size to 96KB. We believe this manifests that value of our bucketing approach—96KB is a reasonable proof size that is worth having to enjoy much smaller update time.

Comparison with Hyperproofs. BalanceProofs' main competitor is Hyperproofs [38], the only vector commitment that is both maintainable and aggregatable. Our main findings from comparing with Hyperproofs are as follows.

1. Both verification of aggregate proofs and aggregation of individual proofs using our bucketing technique outperform Hyperproofs by around 1000×. Again, this is because BalanceProofs is naturally aggregatable, as opposed to Hyperproofs that uses an argument system like IPA [8] as a black box. We consider this to be our central contribution.

2. In terms of proof size, Hyperproofs and BalanceProofs with bucketing have approximately the same performance (i.e., within a factor of 2 from each other for aggregating $k = 1024$ individual proofs). However, 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 does not.

3. Hyperproofs outperforms BalanceProofs in update time. Still BalanceProofs' update time is practical: When AMT bucketing is used, update time is at most 2 seconds—and could be improved with a multi-threaded implementation.

Applications to Verkle trees. We finally explore using our aSVC-bucketing technique in Verkle trees [24] for producing maintainable vector commitments with small proof sizes. Verkle trees is a bandwidth-efficient alternative to Merkle trees [29]. A Verkle tree uses the same hierarchical technique as Merkle trees, with the difference that a parent node is the vector commitment of its children. A Verkle tree with branching factor b has $O(\log_b n)$ proof size. However updating a Verkle tree requires $O(u(b) \cdot \log_b n)$ time, where $u(b)$ is the update time of the underlying vector commitment, which is $O(b^{1/3} \cdot \log_b n)$ when our bucketing technique is used. Additionally, with the help of Verkle trees, we can achieve small size public parameters. We provide a comparison of Verkle trees using BalanceProofs with bucketing with Hyperproofs, showing a decrease in proof size at the expense of a slighly worse update time. Finally, we note here that supporting efficient aggregation in Verkle trees is challenging—we leave this as an open problem.

Limitations. There are two main limitations in BalanceProofs. First, note that when we are using the bucketing technique, the aggregate proof size increases to $n^{1/3}$. As we saw this is not an issue in practice, but it is an open problem to construct a maintainable and naturally-aggregatable vector commitment that has constant-size aggregate proof, yet with $n^{1/3}$ update

time. The second limitation is the fact the the bucketing technique requires quasilinear-size public parameters. See Table 1 for an asympotic comparison with Hyperproofs.

1.1 Paper outline

The paper outline is as follows. In Section 2 we give preliminary definition and notations. In Section 3 we present our basic BalanceProofs compiler. In Section 4 we introduce the bucketing technique used with BalanceProofs, including aSVC-bucketing and AMT-bucketing. In Section 5 we present our evalution.

2 Preliminaries

We use λ to denote the security parameter and $\mathsf{negl}(\cdot)$ to denote any negligible function. We also use multiplicative notation for all groups. With ω we denote a primitive n-th root of unity in $\mathbb{Z}_p$ [44]. Also, let $[i,j) = \{i, i+1, \ldots, j-2, j-1\}$ and $[i,j] = \{i, i+1, \ldots, j-1, j\}$. Vectors are in bold, lower-case symbols, for example $\mathbf{m} = [m_0, \ldots, m_{n-1}]$.

Lagrange polynomials [10, 3]. For $i = 0, \ldots, n-1$, we denote the i-th Lagrange polynomial, with roots of unity as an index, as

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

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

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

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

Bilinear pairings [28, 21]. 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 $e(u^a, w^b) = e(u, w)^{ab}$. In this paper, we use symmetric pairings notation where $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}$ for simplicity.

2.1 Vector commitments (VCs)

We formalize vector commitments below, similar to Hyperproofs [38]. We provide a generalized version that uses some auxiliary information aux to represent the underlying data structure used to maintain the proofs.

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

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

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

(3) $\mathsf{VC.Open}_{\mathsf{pp}}(i, \mathbf{m}, \mathsf{aux}) \to \pi_i$: It outputs a proof π_i for position i in $\mathbf{m}$, based on the auxiliary information aux.

(4) $\mathsf{VC.OpenAll_{pp}}(\mathbf{m}) \to (\pi_0, \pi_1, \ldots, \pi_{n-1})$: It outputs all single proofs for $\mathbf{m}$.

(5) $\mathsf{VC.Agg_{pp}}(I, (m_i, \pi_i)_{i\in I}) \to \pi_I$: It combines individual proofs π_i for values m_i into an aggregate proof π_I.

(6) $\mathsf{VC.Verify_{pp}}(\mathsf{C}, I, (m_i)_{i\in I}, \pi_I) \to \{0,1\}$: It verifies proof π_I that each position $i \in I$ has value m_i against commitment C.

(7) $\mathsf{VC.UpdateCom_{pp}}(i, \delta, \mathsf{C}) \to \mathsf{C}'$: It updates commitment C to C' to reflect position i changing by $\delta \in \mathbb{Z}_p$.

(8) $\mathsf{VC.UpdateAllProofs_{pp}}(i, \delta, \pi_0, \ldots, \pi_{n-1}, \mathsf{aux}) \to \ (\pi'_0, \pi'_1, \ldots, \pi'_{n-1}, \mathsf{aux}')$: It updates all proofs π_i to π'_i to reflect position i changing by $\delta \in \mathbb{Z}_p$. It also updates the auxiliary information.

(9) $\mathsf{VC.UpdateProof_{pp}}(i, \delta, j, \pi_j) \to \pi'_j$: It updates proof π_j to π'_j to reflect position i changing by $\delta \in \mathbb{Z}_p$.

Definition 2.2 (VC Correctness)**.** A VC scheme is correct if for all $\lambda \in \mathbb{N}$ and $n = \mathsf{poly}(\lambda)$, for all $\mathsf{pp} \leftarrow \mathsf{VC.Gen}(1^\lambda, n)$, for all vectors $\mathbf{m}$, if $(\mathsf{C}, \mathsf{aux}) = \mathsf{VC.Commit_{pp}}(\mathbf{m})$ and $\pi_i = \mathsf{VC.Open_{pp}}(i, \mathbf{m}, \mathsf{aux})$, $\forall i \in [0, n)$ (or π_i from $\mathsf{VC.OpenAll_{pp}}$), then, for any polynomial number of updates (i, δ) resulting in a new vector $\mathbf{m}'$, if C' and π'_i are obtained via calls to $\mathsf{VC.UpdateCom_{pp}}$ and $\mathsf{VC.UpdateProof_{pp}}$ (or $\mathsf{VC.UpdateAllProofs_{pp}}$ with aux replaced by aux') respectively, then

(1) $\Pr[1 \leftarrow \mathsf{VC.Verify_{pp}}(\mathsf{C}', \{i\}, (m'_i), \pi'_i)] = 1$ for all i;

(2) $\forall I \subseteq [n]$,

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

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

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

2.2 aSVC

Our construction (compiler) will be using the aSVC [42] 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 [22]. With linear-sized public parameters, it can compute all constant-sized individual proofs in quasilinear time and update proofs in constant time. Furthermore, it is *aggregatable* since one can aggregate b proofs for individual positions into a single constant-sized *batch proof* for those positions. Given SDH public parameters [4]

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

for a vector $\mathbf{m} = [m_0, \ldots, m_{n-1}]$, aSVC represents $\mathbf{m}$ 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. Similar to [9], the proving keys include commitments to all Lagrange polynomials $g^{\mathcal{L}_i(\tau)}$.

Aggregating KZG proofs. aSVC [42] 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 based on *partial fraction decomposition* [47] and Drake and Buterin's observation [7]. More specifically, in KZG proofs, π_i is actually a commitment to $q_i(x) = \frac{\phi(x) - m_i}{x - \omega^i}$ and 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 $R(\omega^i) = m_i,\ \forall i \in I$. Let $A'_I(x) = \sum_{j\in I} \frac{A_I(x)}{x-\omega^j}$ be the derivative of $A_I(x)$ [45]. They observe that

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

thus we can compute $c_i = 1/A'_I(\omega^i)$ with $O(|I|\log^2|I|)$ field operations [45] 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) $\mathsf{VC.Gen}(1^\lambda, n) \to \mathsf{pp}$: It generates SDH public parameters $(g^{\tau^i})_{i\in[0,n)}$. It then computes $a = g^{A(\tau)}$ where $A(x) = x^n - 1$, $a_i = g^{A(\tau)/(x-\omega^i)}$ and $l_i = g^{\mathcal{L}_i(\tau)}$, for all $i \in [0, n)$. Then it also computes KZG proofs $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{x-\omega^i}}$ for the evaluation $\mathcal{L}_i(\omega^i) = 1$. Finally it sets

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

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

(3) $\mathsf{VC.Open}_{\mathsf{pp}}(i, \mathbf{m}) \to \pi_i$: Divides $\phi(x) = \sum_{i\in[0,n)} m_i\mathcal{L}_i(x)$ by $x-\omega^i$, obtaining a quotient $q(x)$ and a remainder $r(x)$. Output a proof $\pi_i = g^{q(\tau)}$.

(4) $\mathsf{VC.OpenAll}_{\mathsf{pp}}(\mathbf{m}) \to (\pi_0, \pi_1, \ldots, \pi_{n-1})$: Outputs all proofs for $\mathbf{m}$. This can be done in $O(n\log n)$ time.

(5) $\mathsf{VC.Agg}_{\mathsf{pp}}(I, (m_i, \pi_i)_{i\in I}) \to \pi_I$: Computes $A_I(x) = \prod_{i\in I}(x-\omega^i)$, its derivative $A'_i(x)$ and all $c_i = (A'_I(\omega^i))^{-1}$ in $O(|I|\log^2|I|)$ time. Outputs $\pi_I = \prod_{i\in I}\pi_i^{c_i}$.

(6) $\mathsf{VC.Verify}_{\mathsf{pp}}(\mathsf{C}, I, (m_i)_{i\in I}, \pi_I) \to \{0,1\}$: Computes $A_I(x) = \prod_{i\in I}(x-\omega^i)$ in $O(|I|\log^2|I|)$ time and commits to it as $g^{A_I(\tau)}$ in $O(|I|)$ time. Interpolates $R_I(x)$ such that $R_I(\omega^i) = m_i,\ \forall i \in I$ in $O(|I|\log^2|I|)$ time and commits to it as $g^{R_I(\tau)}$ in $O(|I|)$ time. Outputs 1 iff

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

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

(8) $\mathsf{VC.UpdateAllProofs}_{\mathsf{pp}}(i, \delta, \pi_0, \ldots, \pi_{n-1}) \to (\pi'_0, \pi'_1, \ldots, \pi'_{n-1})$: Not available. Call $\mathsf{VC.UpdateProof}_{\mathsf{pp}}$ multiple times instead.

(9) $\mathsf{VC.UpdateProof}_{\mathsf{pp}}(i, \delta, j, \pi_j) \to \pi'_j$: If $i = j$, outputs $\pi'_i = \pi_i\cdot(u_i)^\delta$. If $i \neq j$, computes $w_{i,j} = a_i^{1/(\omega^i-\omega^j)}\cdot a_j^{1/(\omega^j-\omega^i)}$ and $u_{i,j} = w_{i,j}^{1/A'(\omega^i)}$, and returns $\pi'_j = \pi_j\cdot(u_{i,j})^\delta$.

2.3 AMT proofs

For our bucketing technique, we will also be using AMT proofs by Tomescu [41]. AMT proofs extend KZG proofs [22] to proofs that have logarithmic size while at the same time being efficiently updatable (i.e., in logarithmic time). The technique used in AMT is similar to [13, 12]. Here we directly present AMT proofs formally. Given $(n-1)$-SBDH public parameters [19] $(g^{\tau^i})_{i\in[0,n)}$, consider a vector $\mathbf{m} = [m_0, \ldots, m_{n-1}]$ where $n = 2^L$. The commitment for $\mathbf{m}$ is the same as in aSVC, $C = g^{\phi(\tau)}$, where $\phi(x) = \sum_{i\in[0,n)} m_i\cdot\mathcal{L}_i(x)$ is the Lagrange interpolation.

In order to simplify notation we define the following index sets. Set $P_{i,j}$ contains all the indices of one subtree in level i (root is in level 0). See Figure 1 for an example of $n = 8$. We

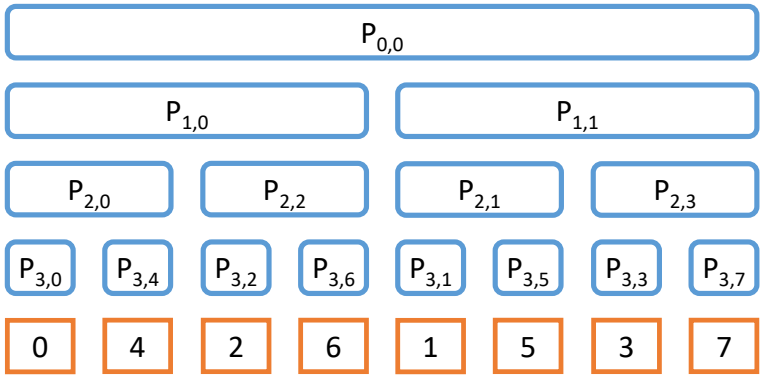


Figure 1: $P_{i,j}$ example when $n = 8$.

define

$$\begin{aligned}
P_{0,0} &= [0, n) \dots \\
P_{i,j} &= \left\{ j + k \cdot 2^i \ : \ k \in \left[0, \frac{n}{2^i}\right) \right\}, \forall j \in [0, 2^i) \\
P_{L,j} &= \{j\}, \forall j \in [0, n)
\end{aligned}$$

From observation, we have $P_{i,j} = P_{i+1,j} \cup P_{i+1,j+2^i}, \ \forall j \in [0, 2^i)$. The intuition we define $P_{i,j}$ in this way is like FFT[14]. In fact, all $P_{L,j}$s in the leaf level are organized in the reversed bit order (see Figure 1). More importantly, if ω is an n-th root of unity, we have the following:

$$\prod_{k \in P_{i,j}} (x - \omega^k) = x^u - (\omega^j)^u, \ u = \frac{n}{2^{i+1}}. \tag{1}$$

$x^u - (\omega^j)^u$ is very helpful to make polynomial multiplication and division easier in AMT.

Similarly, we define $\phi_{i,j}(x)$ as Lagrange interpolation over points in $P_{i,j}$ and thus we have

$$\begin{aligned}
\phi_{i,j}(x) &= \phi_{i+1,j}(x) + q_{i,j}(x) \left(x^u - (\omega^j)^u\right) \\
&= \phi_{i+1,j+2^i}(x) + q_{i,j}(x) \left(x^u - (\omega^{j+2^i})^u\right), \quad u = \frac{n}{2^{i+1}}
\end{aligned} \tag{2}$$

for some polynomial $q_{i,j}(x)$. In this way, we can rewrite $\phi(x)$ in the following way for any $\phi_{L,j}(x) = m_j, \ j \in [0, n)$:

$$\begin{aligned}
\phi(x) = \phi_{0,0}(x) &= \phi_{1,j_1}(x) + q_{0,0}(x)(x^{u_0} - (\omega^{j_1})^{u_0}) \\
&= \phi_{2,j_2}(x) + q_{1,j_1}(x)(x^{u_1} - (\omega^{j_2})^{u_1}) \\
&\qquad + q_{0,0}(x)(x^{u_0} - (\omega^{j_1})^{u_0}) \\
&= \dots \\
&= \phi_{L,j}(x) + \sum_{k \in [0,L)} q_{k,j_k}(x)(x^{u_k} - (\omega^{j_{k+1}})^{u_k}) \\
&= m_j + \sum_{k \in [0,L)} q_{k,j_k}(x)(x^{u_k} - (\omega^{j_{k+1}})^{u_k})
\end{aligned} \tag{3}$$

where $u_k = \frac{n}{2^{k+1}}$, $j_k = j_{k-1}$ or $j_k = j_{k-1} + 2^{k-1}$. In fact,

$$j_k = j \mod 2^k, \quad \forall k \in [0, L].$$

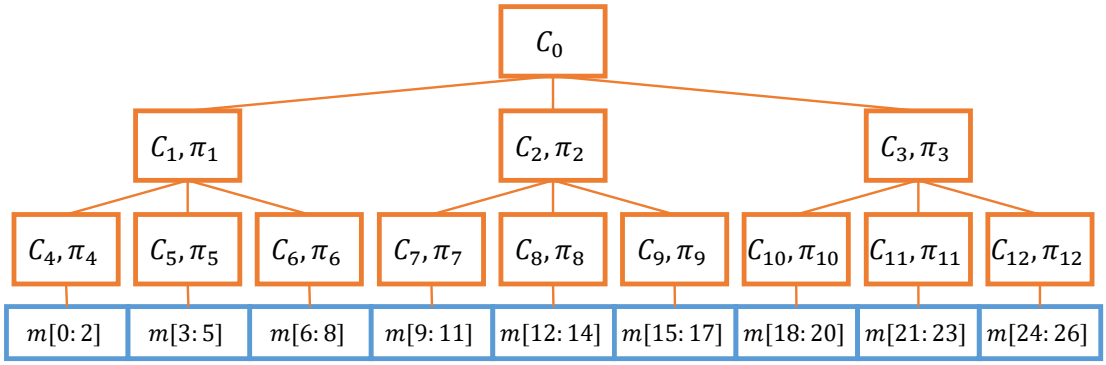


Figure 2: Verkle tree example. $C_4, \ldots, C_{12}$ are vector commitments for subvectors of size 3; C_1 is a vector commitment for (C_4, C_5, C_6); C_2 is a vector commitment for (C_7, C_8, C_9); C_3 is a commitment for (C_{10}, C_{11}, C_{12}); C_0 is the root commitment for (C_1, C_2, C_3).

Thus, the log-size AMT proof for position j is

$$\left(g^{q_{k,j_k}(\tau)}\right)_{k\in[0,L)}.$$

A verifier who has the commitment C, the claimed evaluation $(i, \phi(\omega^i) = z)$ and log-size verification key $\left(s_k = g^{\tau^{2^k}}\right)_{k\in[0,L)}$ can verify the proof $(w_k)_{k\in[0,L)}$ by checking if the following holds:

$$e(C/g^z, g) = \prod_{k\in[0,L)} e\left(w_k, s_{L-1-k}/g^{\omega^{j_{k+1}\cdot u_k}}\right) \tag{4}$$

Updating AMT proofs [41]. After receiving an update request for the vector, we can update all AMT proofs in $O(\log n) = O(L)$ time. For all $i \in [0, n)$, consider a new vector $\mathbf{m}'$ where only $m_i = 1$ and in all other positions $m_j = 0$. In this case we have $\phi(x) = \mathcal{L}_i(x)$ and we can pre-compute in $O(n \log^2 n)$ time all the group elements $(\mathsf{upkTree}_{i,j,k} = g^{q_{j,k}(\tau)})_{i\in[0,n),j\in[0,L),k\in[0,2^j)}$ in the tree (In fact, only those elements in the path from the root to the leaf of position i are non-trivial, i.e., $q_{j,k}(x) \neq 0$ if and only if $i \in P_{j,k}$.) After receiving an update request (i, δ), it is $\phi'(x) = \phi(x) + \delta\mathcal{L}_i(x)$, and therefore we can update the proofs by setting

$$g^{q'_{j,k}(\tau)} = g^{q_{j,k}(\tau)} \cdot (\mathsf{upkTree}_{i,j,k})^{\delta}. \tag{5}$$

2.4 Verkle trees

Verkle trees [24, 27] are similar to Merkle trees [29] except for that fact that hash functions are replaced with vector commitments. In order to compute a Verkle Tree over messages $\mathbf{m} = (m_0, \ldots, m_{n-1})$, we choose b as the branching factor of the tree. Then we group messages into subsets of b messages and compute vector commitments and corresponding proofs over each of those subsets. See Figure 2 for an simple example of Verkle Tree with $n = 27$ and $b = 3$.

One of the main advantages of Verkle trees comparing to Merkle trees is that they can reduce the proof size. For example, for $n = 2^L$ messages, a Merkle tree with branching factor 2 has proof size $O(L)$ while a Verkle tree with branching factor b has proof size $O(\log_b n) = \frac{1}{\log b} O(L)$ if the Verkle tree uses a vector commitment with constant-sized proofs.

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′. The input vector commitment VC must satisfy certain requirements for our compilation to work and produce an improved vector commitment VC′. We list them here.

- $\mathsf{VC.OpenAll}_{\mathsf{pp}}(\mathbf{m})$ should run in $O(n \log n)$ time.
- $\mathsf{VC.UpdateCom}_{\mathsf{pp}}(i, \delta, \mathsf{C})$ should run in $O(1)$ time.
- $\mathsf{VC.UpdateProof}_{\mathsf{pp}}(i, \delta, j, \pi_j)$ should run in $O(1)$ time.
- VC should have an aggregation algorithm $\mathsf{VC.Agg}_{\mathsf{pp}}(I, (m_i, \pi_i)_{i \in I})$.

Remarks on asymptotics. We note here that the first three requirements can be relaxed to some extent. For example, in the first requirement, the time complexity of $\mathsf{VC.OpenAll}_{\mathsf{pp}}(\mathbf{m})$ can be relaxed to any $O(n \cdot \mathsf{poly}\log(n))$ time, resulting in a slight change of the final time complexity analysis. However, we cannot relax the time complexity of $\mathsf{VC.OpenAll}_{\mathsf{pp}}(\mathbf{m})$ to $O(n^2)$ (See details in later analysis.)

3.1 Compiler details

The main idea of our compiler is the following. First, the expression of the commitment of the output vector commitment VC′ is exactly the same as the commitment of the input vector commitment VC. For example, in the case of aSVC being used as input, the commitment of the output vector commitment will be $g^{\phi(\tau)}$, where $\phi(x) = \sum_{i \in [0,n)} \mathcal{L}_i(x) \cdot m_i$. Similarly the individual proofs and aggregate proofs are exactly the same as the ones defined for the input vector commitment.

Whenever an update (i, δ) appears, we do not use the $\mathsf{VC.UpdateProofs}_{\mathsf{pp}}(i, \delta)$ algorithm since this would incur linear cost. What we do is *append* the update in a list L, which takes just constant time. The list L serves as the auxiliary information aux. When the size of L reaches $\sqrt{n}$, our compiler calls $\mathsf{VC.OpenAll}_{\mathsf{pp}}$ 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 the fresh proofs (i.e., before the list reaches $\sqrt{n}$ elements), then all the updates are applied to the proof that is requested and an updated fresh individual proof is returned.

Since $\mathsf{VC.OpenAll}_{\mathsf{pp}}$ runs $O(n \log n)$ time to update all proofs, our compiler needs amortized

$$O\left(\frac{n \log n}{\sqrt{n}}\right) = O(\sqrt{n} \log n)$$

time to update the proofs. Therefore $\mathsf{VC'.UpdateAllProofs}_{\mathsf{pp}}$ runs in $O(\sqrt{n} \log n)$ amortized time. Also note that since the maximum size of the list L is $\sqrt{n}$ and algorithm $\mathsf{VC.UpdateProof}_{\mathsf{pp}}$ to update an individual proof runs in constant time, returning a fresh individual proof takes at most $O(\sqrt{n})$ time.

3.1.1 Algorithms for VC′

We now provide the detailed algorithms for VC′:

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

(2) $\mathsf{VC}'.\mathsf{Commit}_{\mathsf{pp}}(\mathbf{m}) \to (\mathsf{C}, \mathsf{aux})$: Calls $\mathsf{VC}.\mathsf{Commit}_{\mathsf{pp}}(\mathbf{m}) \to (\mathsf{C}, \mathsf{aux}_0)$. For simplicity, we ignore aux_0 later in this section. Calls $\mathsf{VC}.\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{m})$ and returns $\pi_0, \ldots, \pi_{n-1}$. Initializes an empty list L. Returns $(\mathsf{C}, [L; \pi_0, \ldots, \pi_{n-1}])$.

(3) $\mathsf{VC}'.\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{m}, \mathsf{aux}) \to \pi_i$: Parses $\mathsf{aux} = [L; \pi_0, \ldots, \pi_{n-1}]$. If $L = \emptyset$, outputs π_i. Otherwise calls

$$\mathsf{VC}.\mathsf{UpdateProof}_{\mathsf{pp}}(j, \delta_j, i, \pi_i)$$

for each update request (j, δ_j) in L and finally returns the correct latest proof π'_i.

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

(5) $\mathsf{VC}'.\mathsf{Agg}_{\mathsf{pp}}(I, (m_i, \pi_i)_{i \in I}) \to \pi_I$: Calls $\mathsf{VC}.\mathsf{Agg}_{\mathsf{pp}}(I, (m_i, \pi_i)_{i \in I}) \to \pi_I$ and returns π_I.

(6) $\mathsf{VC}'.\mathsf{Verify}_{\mathsf{pp}}(\mathsf{C}, I, (m_i)_{i \in I}, \pi_I) \to \{0, 1\}$: Calls $\mathsf{VC}.\mathsf{Verify}_{\mathsf{pp}}(\mathsf{C}, I, (m_i)_{i \in I}, \pi_I) \to b$ and returns b.

(7) $\mathsf{VC}'.\mathsf{UpdateCom}_{\mathsf{pp}}(i, \delta, \mathsf{C}) \to \mathsf{C}'$: Calls $\mathsf{VC}.\mathsf{UpdateCom}_{\mathsf{pp}}(i, \delta, \mathsf{C}) \to \mathsf{C}'$ and returns C'.

(8) $\mathsf{VC}'.\mathsf{UpdateAllProofs}_{\mathsf{pp}}(i, \delta, \mathsf{aux}) \to \mathsf{aux}'$: Parses $\mathsf{aux} = [L; \pi_0, \ldots, \pi_{n-1}]$. If $|L| < \sqrt{n}$, appends (i, δ) to L and return $[L \cup (i, \delta); \pi_0, \ldots, \pi_{n-1}]$; Otherwise calls

$$\mathsf{VC}.\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{m})$$

to compute new proofs $\pi'_0, \ldots, \pi'_{n-1}$ and returns $[\emptyset; \pi'_0, \ldots, \pi'_{n-1}]$.

(9) $\mathsf{VC}'.\mathsf{UpdateProof}_{\mathsf{pp}}(i, \delta, j, \pi_j) \to \pi'_j$: Calls $\mathsf{VC}.\mathsf{UpdateProof}_{\mathsf{pp}}(i, \delta, j, \pi_j) \to \pi'_j$ and returns π'_j.

3.2 De-amortization for UpdateAllProofs

The main problem of the compiler is that, since the algorithm $\mathsf{VC}.\mathsf{OpenAll}_{\mathsf{pp}}$ is very time-consuming, the time complexity of $\mathsf{VC}'.\mathsf{UpdateAllProofs}$ will be extremely large every other $\sqrt{n}$ updates. It is not realistic in practice to implement our compiler in this naive way due to the instability of the time cost. In this subsection, we will show how to de-amortize the computation to make $\mathsf{VC}'.\mathsf{UpdateAllProofs}$ need time $O(\sqrt{n} \log n)$ in the worst case.

3.2.1 Separation of VC.OpenAll

Our plan is to try to separate the computation in the $O(n \log n)$-time algorithm $\mathsf{VC}.\mathsf{OpenAll}$ as long as it can be separated into $\sqrt{n}$ $O(\sqrt{n} \log n)$-time sub-steps. In fact, we examined some recent VC schemes that can serve as input to our compiler, such as [42, 18], and found that their $\mathsf{VC}.\mathsf{OpenAll}$ algorithm can indeed be separated. We formalize this as follows.

Definition 3.1 (Separation of $\mathsf{VC}.\mathsf{OpenAll}$)**.** We say a set of algorithms

$$\{\mathsf{VC}.\mathsf{OpenAllStep}_{\mathsf{i}} \ : \ 0 \le i < \sqrt{n}\}$$

is a valid separation of $\mathsf{VC}.\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{m}) \to (\pi_0, \pi_1, \ldots, \pi_{n-1})$ if the following conditions hold:

(1) $\mathsf{VC}.\mathsf{OpenAllStep}_0(\mathsf{pp}, \mathbf{m}) \to \mathsf{out}_0$ runs in $O(\sqrt{n} \log n)$ time.

(2) Every $\mathsf{VC}.\mathsf{OpenAllStep}_{\mathsf{i}}(\mathsf{out}_{\mathsf{i}-1}) \to \mathsf{out}_{\mathsf{i}}$ for $1 \le i < \sqrt{n} - 1$ runs in $O(\sqrt{n} \log n)$ time.

(3) $\mathsf{VC}.\mathsf{OpenAllStep}_{\sqrt{\mathsf{n}}-1}(\mathsf{out}_{\sqrt{\mathsf{n}}-2}) \to (\pi_0, \ldots, \pi_{n-1})$ runs in $O(\sqrt{n} \log n)$ time and outputs the same proofs as $\mathsf{VC}.\mathsf{OpenAll}$.

Implementing separation in practice. Intuitively, there are two methods to implement this kind of separation in practice. The first is to rewrite the original algorithm in a separated way. Take aSVC[42] as an example. The $\mathsf{VC}.\mathsf{OpenAll}$ algorithm of aSVC in $O(n \log n)$ time is basically the technique from FK20[16], which contains several single loops including FFT[14] (say there

are $k = O(1)$ single loops). Each single loop has loop size at most n and needs at most $O(n \log n)$ operations. We can then separate each single loop into $\sqrt{n}$ small loops, where each small loop requires $O(\sqrt{n} \log n)$ operations. Finally, we have $k\sqrt{n}$ small loops and each of them can be configured into one step $\mathsf{VC.OpenAllStep_i}$.

While the first method is efficient and reasonable, there might be some situations where it is not convenient or even feasible to separate the single loops easily. The second method is more generalized. It simply focuses on the operations with the highest cost in the algorithm. This type of operation could be, for instance, group operations on elliptic curves. Since there are at most $O(n \log n)$ operations with the highest cost, we can use a counter to count how many operations we have done so far. As soon as the counter reaches $O(\sqrt{n} \log n)$ we should log everything, save the current configuration locally, and exit the algorithm temporarily. In next round, we can start a new counter and do those things again. In this way, we can finish the whole algorithm in exactly $\sqrt{n}$ rounds where each round requires almost equal time to complete.

3.2.2 Updating all proofs with de-amortization

Assume now we have a valid separation $\{\mathsf{VC.OpenAllStep_i} : 0 \leq i < \sqrt{n}\}$ for $\mathsf{VC.OpenAll}$. We show how to make some slight changes to our compiler to de-amortize $\mathsf{UpdateAllProofs}$. In the beginning, the compiler has an empty record list and $L = \emptyset$. When receiving an update request (i, δ), the compiler updates the commitment C as before if needed and appends (i, δ) to the end of record list L.

1. If $|L| = \sqrt{n}$, the compiler calls

   $$\mathsf{VC.OpenAllStep_0}(\mathsf{pp}, \mathbf{m}')$$

   and saves $\mathsf{out_0}$. Note that $\mathbf{m}'$ here should be the original vector applied by $\sqrt{n}$ update request in L;

2. If $|L| = \sqrt{n} + i,\ 1 \leq i < \sqrt{n} - 1$, the compiler calls $\mathsf{VC.OpenAllStep_i}(\mathsf{out_{i-1}})$ and saves $\mathsf{out_i}$.

3. If $|L| = 2\sqrt{n} - 1$, the compiler calls $\mathsf{VC.OpenAllStep}_{\sqrt{n}-1}(\mathsf{out}_{\sqrt{n}-2})$ and receives proofs for all positions. Then it replaces the old proofs it stored with the newly received ones (this is actually the behavior of "reading n old proofs and outputting n new proofs"), and removes the first $\sqrt{n}$ update requests in the record list. Now $|L| = \sqrt{n} - 1$.

Therefore, $|L|$ will never reach $2\sqrt{n}$. Actually, in each round $\mathsf{VC'.UpdateProofs_{pp}}(i, \delta)$ runs in at most $O(\sqrt{n} \log n)$ time according to the previous definition of valid separation.

4 Bucketing BalanceProofs

In addition to the balance of the time complexity between updating all proofs and querying single proofs, we can also balance between the time complexity and proof size. The intuitive idea is bucketing, i.e., to separate the original vector into p parts. Then we can perform updates and aggregation inside each part. The resulting batch proof size is $O(p)$. In practice we can choose the bucketing size $p = n^{1/3}$ or $p = n^{1/4}$ to get best performance. It is not useful to choose too small p like $p = \log n$ since that would be make no change on updating and querying asymptotically, or too large p like $p = \frac{n}{\log n}$ since the resulting batch proof size will be too large.

We must modify how the proofs look for each bucket to make this tradeoff. In this section, we will first present the definition of the new proofs when using the bucketing method. Next, we

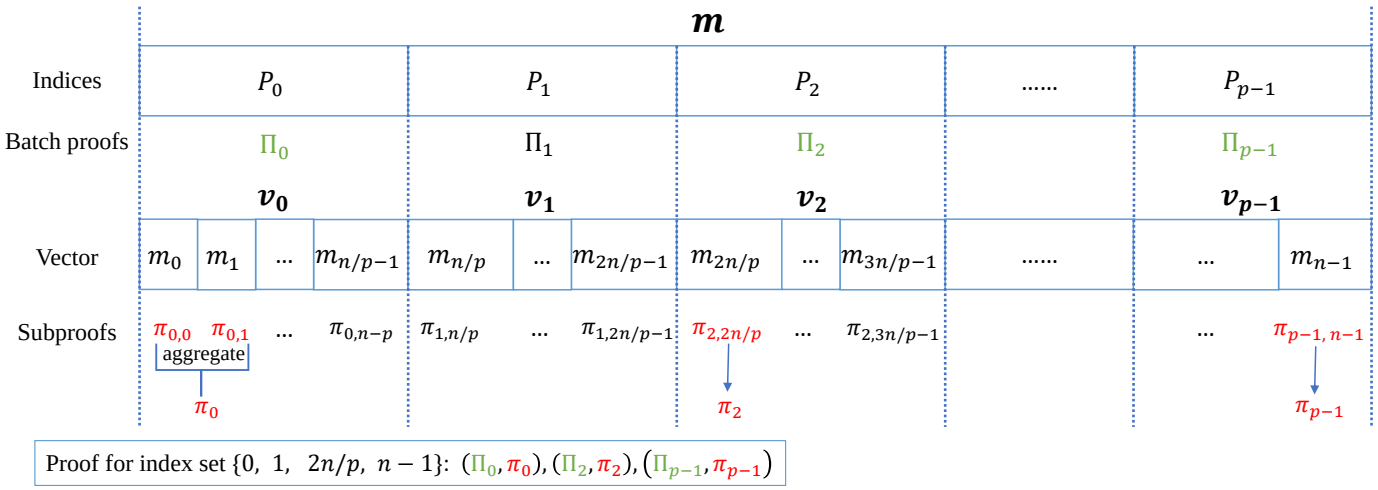


Figure 3: Aggregating example for aSVC-bucketing.

will show how this tradeoff works taking aSVC [42] as an example of the input VC scheme (see Section 2.2 for the details of aSVC algorithms).

In aSVC, proofs are just KZG proofs [22] (one group element for both individual proofs and batch proofs). Now, we still consider a vector $\mathbf{m} = [m_0, \ldots, m_{n-1}]$ and its Lagrange interpolation $\phi(x) = \sum_{i\in[0,n)} m_i \mathcal{L}_i(x)$. Say $n = 2^L$ and we are given $(n-1)$-SBDH public parameters [19] $(g^{\tau^i})_{i\in[0,n)}$. The commitment for the whole vector is still KZG commitment: $C = g^{\phi(\tau)}$.

4.1 First method: aSVC-bucketing

In this subsection, we introduce aSVC-bucketing, which uses aSVC technique to deal with batch proofs for each bucket. We will give the fully-detailed scheme of aSVC-bucketing BalanceProofs taking aSVC as an input VC scheme in Appendix A.

We choose some $L' \in [1, L)$ and set the bucketing size $p = 2^{L'}$, e.g. choose $L' = L/3$ to set $p = n^{1/3}$. Note that $\frac{n}{p} = 2^{L-L'}$. We 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, based on this partition of indices, we can cut $\mathbf{m}$ into p subvectors $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{p-1}$ and for each $\mathbf{v}_i$ we have $|\mathbf{v}_i| = \frac{n}{p}$. Specifically,

$$\mathbf{v}_i = \left[m_{i\cdot\frac{n}{p}}, m_{i\cdot\frac{n}{p}+1}, \ldots, m_{(i+1)\cdot\frac{n}{p}-1}\right], \ i \in [0, p).$$

Similarly, we can write Lagrange interpolations $\phi_i(x)$ for each $\mathbf{v}_i$:

$$\phi_i(x) = \sum_{j\in P_i} \mathcal{L}_{i,j}(x) \cdot m_j, \text{ where } \mathcal{L}_{i,j}(x) = \prod_{k\in P_i, k\neq j} \frac{x - \omega^k}{\omega^j - \omega^k} \tag{6}$$

and derive from division that

$$\phi(x) = \phi_i(x) + q_i(x) \prod_{j\in P_i} (x - \omega^j) \tag{7}$$

for some polynomial $q_i(x)$. Thus $\Pi_i := g^{q_i(\tau)}$ is in fact a KZG batch proof for the index set P_i of vector $\mathbf{m}$.

We can still provide individual proofs inside any subvector $\mathbf{v}_i$ based on the equation:

$$\begin{aligned}\phi_i(x) &= q_{i,j}(x)(x-\omega^j) + \phi_i(\omega^j) \\ &= q_{i,j}(x)(x-\omega^j) + \phi(\omega^j),\ j \in P_i.\end{aligned} \tag{8}$$

Namely $\pi_{i,j} := g^{q_{i,j}(\tau)}$ is in fact an individual KZG proof for position j of subvector $\mathbf{v}_i$.

Individual bucketing proofs. When we need to compute one individual bucketing proof for one position i, we should first find j such that $i \in P_j$, then we provide both the batch proof Π_j for index set P_j inside vector $\mathbf{m}$ and the individual proof $\pi_{j,i}$ for position i inside vector $\mathbf{v}_j$. Therefore, the resulting individual bucketing proof for position i is $(\Pi_j, \pi_{j,i})$.

A verifier who has the commitment C, the claimed evaluation $(i, \phi(\omega^i) = z)$ 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 individual bucketing proof (w_0, w_1) by checking the following equation (say $i \in P_{i_0}$):

$$e(C/g^z, g) = e\left(w_0, g^{a_{i_0}(\tau)}\right) \cdot e\left(w_1, g^\tau / g^{\omega^i}\right) \tag{9}$$

Although the verifier needs $O(p)$ size verification key, we can reduce it to constant size if we use the partitions $(P_i)_{i\in[0,p)}$ in Section 4.2. More specifically, with the new partitions, $a_k(x) = \prod_{j\in P_k}(x-\omega^j) = x^{n/p} - (\omega^k)^{n/p}$, thus we only need $g^{\tau^{n/p}}$ and g as verification keys to compute $g^{a_k(\tau)}$.

Batch bucketing proofs. When we need to compute one batch bucketing proof for an index set I, we first partition I using $(P_i)_{i\in[0,p)}$, i.e., find $j_0, \ldots, j_{k-1}$ and $I_0, \ldots, I_{k-1}$ such that $j_u \neq j_v$ if $u \neq v$, $I = \bigcup_{t\in[0,k)} I_t$ and $I_t \subseteq P_{j_t},\ \forall t \in [0,k)$. Note that here $1 \leq k \leq p$. Then we provide k batch proofs Π_{j_t} for each $P_{j_t},\ t \in [0,k)$, and k other batch proofs π_{j_t} for each I_t inside vector $\mathbf{v}_{j_t},\ t \in [0,k)$. Our final batch bucketing proof for index set I is $(\Pi_{j_t}, \pi_{j_t})_{t\in[0,k)}$, which contains $O(k)$ group elements, and is of size $O(\min\{|I|, p\})$.

To verify the batch bucketing proofs, the verifier just needs to verify each pair (Π_{j_t}, π_{j_t}) is a correct batch proof for positions in I_t. The verifier checks if the following holds when given one claimed proof (w_0, w_1) for $j_t,\ t \in [0,k)$:

$$e(C/g^{c_{j_t}(\tau)}, g) = e\left(w_0, g^{a_{j_t}(\tau)}\right) \cdot e\left(w_1, g^{b_{j_t}(\tau)}\right) \tag{10}$$

where $b_{j_t}(x) = \prod_{h\in I_t}(x-\omega^h)$ and $c_{j_t}(\tau)$ is the Lagrange interpolation over I_t and claimed evaluations.

Aggregating individual bucketing proofs. Given the definitions above, we can aggregate multiple individual bucketing proofs to one batch bucketing proof naturally. See the example in Figure 3: The index set is $I = \{0, 1, 2n/p, n-1\}$ and we are given four individual bucketing proofs $(\Pi_0, \pi_{0,0})$, $(\Pi_0, \pi_{0,1})$, $(\Pi_2, \pi_{2,2n/p})$, $(\Pi_{p-1}, \pi_{p-1,n-1})$. In this case we can determine $k = 3$, $I_0 = \{0,1\} \subseteq P_0$, $I_1 = \{2n/p\} \subseteq P_2$, and $I_2 = \{n-1\} \subseteq P_{p-1}$. We only need to aggregate $\pi_{0,0}$ and $\pi_{0,1}$ to one small batch proof π_0 inside $\mathbf{v}_0$ and the aggregation result is one batch bucketing proof
$((\Pi_0, \pi_0), (\Pi_2, \pi_2), (\Pi_{p-1}, \pi_{p-1}))$ where $\pi_2 = \pi_{2,2n/p}$ and $\pi_{p-1} = \pi_{p-1,n-1}$.

Applying our compiler in subvectors. Based on the bucketing technique, we can now apply our compiler inside those subvectors. More specifically, in order to update all the individual bucketing proofs, we maintain p update record lists $(L_i)_{i\in[0,p)}$ for each subvector. When receiving an update request (i, δ), we first determine which subvector position i belongs to, assuming it

is $\mathbf{v}_j$. Next, we require $O(p)$ time to update all the batch proofs $(\Pi_i)_{i\in[0,p)}$. Then we append (i,δ) to the end of the list L_j. If now the size of L_j reaches $\sqrt{\frac{n}{p}}$, we call $\mathsf{VC.OpenAll}_{\mathsf{pp}}(\mathbf{v}_j, \mathsf{aux})$ (de-amortized or not) to get and replace all individual proofs for $\mathbf{v}_j$, and then empty L_j. The behavior to query any latest proof at any time is similar for each subvector with the help of record lists $(L_i)_{i\in[0,p)}$.

4.1.1 Updating all batch proofs $(\Pi_i)_{i\in[0,p)}$

We mentioned that we can update all p batch proofs in $O(p)$ time after receiving an update request (i,δ). Here we show how we can update each batch proof Π_j in constant time for any $j \in [0,p)$. Recall that after this update, the polynomial $\phi(x)$ is updated to $\phi'(x) = \phi(x)+\delta\cdot\mathcal{L}_i(x)$. Say that $i \in P_{i_0}$, then we also have $\phi'_{i_0}(x) = \phi_{i_0}(x) + \delta\cdot\mathcal{L}_{i_0,i}(x)$.

The $i \in P_j$ $(i_0 = j)$ Case. The polynomial $\phi_j(x)$ is also updated in the way that $\phi'_j(x) = \phi_j(x) + \delta\cdot\mathcal{L}_{j,i}(x)$. Consider the quotient polynomial $q_j(x)$ from definition:

$$q'_j(x) = \frac{\phi'(x)-\phi'_j(x)}{\prod_{k\in P_j}(x-\omega^k)} = q_j(x) + \frac{\delta\cdot(\mathcal{L}_i(x)-\mathcal{L}_{j,i}(x))}{\prod_{j\in P_i}(x-\omega^j)}. \tag{11}$$

Thus we can pre-compute $r_i(x) = \frac{\mathcal{L}_i(x)-\mathcal{L}_{j,i}(x)}{\prod_{k\in P_j}(x-\omega^k)}$ and save update parameters $(g^{r_i(\tau)})_{i\in[0,n)}$ in the generation algorithm. The proof Π_j can be updated to $\Pi'_j = \Pi_j\cdot(g^{r_i(\tau)})^\delta$.

The $i \notin P_j$ $(i_0 \neq j)$ Case. Since $i \notin P_j$, there is no change of the polynomial $\phi_j(x)$. Consider the quotient polynomial $q_j(x)$ again:

$$\begin{aligned} q'_j(x) &= \frac{\phi'(x)-\phi_j(x)}{\prod_{k\in P_j}(x-\omega^k)} = \frac{(\phi(x)-\phi_j(x))+\delta\cdot\mathcal{L}_i(x)}{\prod_{k\in P_j}(x-\omega^k)} \\ &= q_j(x) + \frac{\delta\cdot\mathcal{L}_i(x)}{\prod_{k\in P_j}(x-\omega^k)}. \end{aligned} \tag{12}$$

Thus we can just pre-compute $r_{i,j}(x) = \frac{\mathcal{L}_i(x)}{\prod_{k\in P_j}(x-\omega^k)}$ and save update parameters $(g^{r_{i,j}(\tau)})_{i\in[0,n),j\in[0,p)}$ in the generation algorithm. The proof Π_j can be updated to $\Pi'_j = \Pi_j\cdot(g^{r_{i,j}(\tau)})^\delta$.

4.1.2 Updating individual proofs inside subvectors

After receiving an update (i,δ), assuming $i \in P_j$, we only need to update individual proofs inside $\mathbf{v}_j$ since for any $k \neq j$, $\phi_k(x)$ does not change at all. In order to update individual proofs inside $\mathbf{v}_j$, recall that there is an update record list L_j for $\mathbf{v}_j$. We append (i,δ) to the end of L_j and check if $|L_j| \geq \sqrt{\frac{n}{p}}$. If so, we re-open all the individual proofs for $\mathbf{v}_j$ and empty L_j.

4.2 Second method: AMT-bucketing

While the first method is very direct and simple, it requires $O(np)$ size public update keys, which could be $O(n^{4/3})$ if we set $p = n^{1/3}$. This extra cost of public parameters seems to be inherent when we are only using aSVC for bucketing. In this subsection, we will propose AMT-like [41] batch proofs for bucketing together with same subproofs inside subvectors as before. This method requires $O(n\log n)$-size public update keys and it has better performance ($O(\log n)$ time) to update all batch proofs at the same time.

The key point here is to use *polynomial multipoint evaluation* techniques [45] and make our index set $(P_i)_{i\in[0,p)}$ consistent with the index sets of AMT proofs defined in Section 2.3. Thus we need a new partition of $[0,n)$ here:

$$P_i = \left\{ kp + i \ : \ k \in \left[0, \frac{n}{p}\right) \right\}, \ \forall i \in [0,p).$$

Based on this partition of indices, we can also rearrange $\mathbf{m}$ into p subvectors $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{p-1}$:

$$\mathbf{v}_i = [m_i, m_{i+p}, \ldots, m_{i+n-p}], \ i \in [0,p).$$

Again we write Lagrange interpolations $\phi_i(x)$ for each new $\mathbf{v}_i$ and derive that

$$\begin{aligned} \phi(x) &= \phi_i(x) + q_i(x) \prod_{j\in P_i} (x - \omega^j) \\ &= \phi_i(x) + q_i(x) \prod_{k\in[0,\frac{n}{p})} (x - \omega^{kp+i}) \\ &= \phi_i(x) + q_i(x) \left(x^{\frac{n}{p}} - (\omega^i)^{\frac{n}{p}} \right). \end{aligned} \tag{13}$$

Note that here the polynomial $x^{\frac{n}{p}} - (\omega^i)^{\frac{n}{p}}$ can be perfectly factorized, which leads us to the following AMT-like batch proofs.

AMT-like batch proofs Π_j. We re-define the same index sets as Section 2.3:

$$\begin{aligned} &P_{0,0} = [0,n) \ldots \\ &P_{i,j} = \left\{ j + k \cdot 2^i \ : \ k \in \left[0, \frac{n}{2^i}\right) \right\}, \forall i \in [0, L'], \ j \in [0, 2^i). \end{aligned}$$

Recall that $p = 2^{L'}$ and P_j in previous sections is actually $P_{L',j}$. Similarly, we define $\phi_{i,j}(x)$ as the Lagrange interpolation[10, 3] over points in $P_{i,j}$, and also $\phi_{L',j}(x) = \phi_j(x)$. Then we can rewrite $\phi(x)$ in the following way for any $\phi_{L',j}(x), \ j \in [0,p)$:

$$\phi(x) = \phi_{L',j}(x) + \sum_{k\in[0,L')} q_{k,j_k}(x)(x^{u_k} - (\omega^{j_{k+1}})^{u_k}) \tag{14}$$

where $u_k = \frac{n}{2^{k+1}}$ and $j_k = j \mod 2^k, \ k \in [0, L']$. Therefore, naturally we define the AMT-like batch proof for $P_j = P_{L',j}$ to be $\Pi_j = (g^{q_{k,j_k}(\tau)})_{k\in[0,L')}$, and one **individual AMT-bucketing proof** for position i is $(\Pi_j, \pi_{j,i})$ where $i \in P_j$. The verification of individual AMT-bucketing proofs is the combination of verifying AMT proofs and verifying individual aSVC-bucketing proofs: The verifier checks if the following holds when given one claimed proof $((w_{0,k})_{k\in[0,L')}, w_1)$ with claimed evaluation $(i, \phi(\omega^i) = z)$ and $i \in P_j$:

$$e(C/g^z, g) = e\left(w_1, g^\tau / g^{\omega^i}\right) \cdot \prod_{k\in[0,L')} e\left(w_{0,k}, g^{\tau^{u_k}} / g^{\omega^{j_{k+1}u_k}}\right) \tag{15}$$

Aggregating individual AMT-bucketing proofs. Given the definitions above, we can aggregate individual AMT-like bucketing proofs for an index set I using the same way as in Section 4.1. The only difference is that the same item in $\Pi_{j_1}, \Pi_{j_2}, j_1 \neq j_2$ can be provided just once, so that the size of the final batch proof will be $O(\min\{|I| \cdot L', p\})$. See Figure 4 for a brief example. In this case, $L' = 2, p = 4$, The batch proof for P_4 is $\Pi_4 = (g^{q_{0,0}(\tau)}, g^{q_{1,0}(\tau)}, g^{q_{2,0}(\tau)})$ and the batch proof for P_2 or P_6 is $\Pi_2 = \Pi_6 = (g^{q_{0,0}(\tau)}, g^{q_{1,0}(\tau)}, g^{q_{2,2}(\tau)})$. Then the aggregated result of Π_2,Π_4 and Π_6 would be $(g^{q_{0,0}(\tau)}, g^{q_{1,0}(\tau)}, g^{q_{2,0}(\tau)}, g^{q_{2,2}(\tau)})$. The verification of batch AMT-like bucketing proofs is just a combination of Eq 10 and Eq 15.

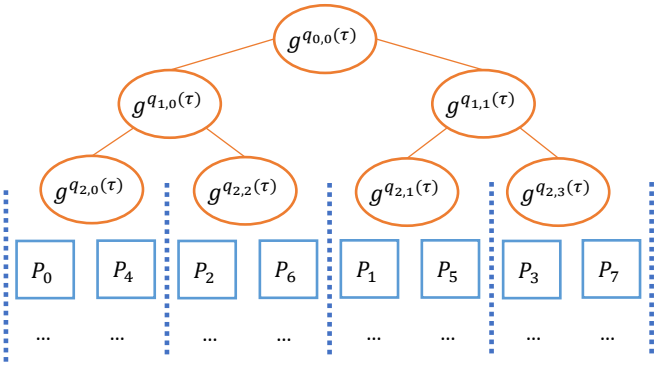


Figure 4: Batch proof example for AMT-bucketing

4.2.1 Updating all batch proofs $(\Pi_i)_{i\in[0,p)}$

With the help of tree-structure batch proofs, we can update all batch proofs in $O(\log p) = O(L')$ time using the same technique in AMT [41]. Recall the same notations in Section 2.3. After receiving an update request (i, δ), $\phi'(x) = \phi(x) + \delta\mathcal{L}_i(x)$, we can update those proofs using

$$g^{q'_{j,k}(\tau)} = g^{q_{j,k}(\tau)} \cdot (\mathsf{upkTree}_{i,j,k})^{\delta} \tag{16}$$

4.3 Analysis

Correctness and security. Correctness follows directly by inspection. We prove that our bucketing technique can generate sound VC schemes in Appendix C.

5 Evaluation

In this section, we measure the performance of BalanceProofs. We analyze the time and space complexity asymptotically first. For the experiments, we fully implemented three versions of our compiler taking aSVC as input VC scheme: basic BalanceProofs (no-bucketing), aSVC-bucketing BalanceProofs and AMT-bucketing BalanceProofs. We take go-kzg [34] as a reference to implement KZG proofs. We also implemented the de-amortization version for UpdateAllProofs.

5.1 Asymptotics

Recall that the vector size is $n = 2^L$. The bucketing size is $p = 2^{L'}$ and each subvector has size $\frac{n}{p} = 2^{L-L'}$. I is the index set to be aggregated.

Public parameters. For BalanceProofs, the public parameters are the same as aSVC, which is $O(n)$ size and needs $O(n \log n)$ time to generate. For aSVC-bucketing, we need $O(np)$ size public parameters and the main extra cost comes from the keys to update all the batch proofs: $(g^{r_{i,j}(\tau)})_{i\in[0,n),j\in[0,p)}$, which requires $O(n^{4/3} \log n)$ time to generate. For AMT-bucketing, we need $O(n \log n)$ size extra public parameters to update the AMT-like batch proofs, which requires $O(n \log^2 n)$ time to generate [41].

Committing. For all of our schemes, the commitment is a KZG commitment. Given commitments to Lagrange polynomials as $O(n)$-size public parameters we can compute the commitment in $O(n)$ time without DFT and interpolation [9, 42, 14].

Opening and updating all proofs. For all of our schemes, it takes $O(n \log n)$ time to open all proofs using FK20[16] technique. According to the analysis in previous sections, UpdateAllProofs() takes $O(\sqrt{n} \log n)$ time for BalanceProofs, $O(p+\sqrt{n/p} \log(n/p))$ time for aSVC-bucketing and $O(\log n+\sqrt{n/p} \log(n/p))$ time for AMT-bucketing. Therefore, if the bucketing size p is large, it is better to use AMT-bucketing in practice to get smaller time of updating all the proofs.

Querying the latest individual proofs. This algorithm depends on the size of the auxiliary information. In the worst case, it takes $O(\sqrt{n})$ time for BalanceProofs, and $O(\sqrt{n/p})$ time for bucketing versions.

Aggregation. For BalanceProofs, Aggregate() takes $O(|I| \log^2 |I|)$ time (same as aSVC) [42, 44, 46, 45]. For the bucketing versions, we just grab those batch proofs together and aggregate inside each subvectors, which also results in $O(|I| \log^2 |I|)$ time (considering the worst case when $I \subseteq P_j$ for some j).

Proof size. For BalanceProofs, both individual proofs and batch proofs have size $O(1)$ (same as aSVC). For aSVC-bucketing, the individual proofs contain 2 group elements, which is also $O(1)$ size, while the batch proof size relies on the relation between I and $(P_i)_{i\in[0,p)}$, which in worst case is $O(\min\{k, p\})$. Similarly, for AMT-bucketing, the individual proof size is $O(\log n)$ and batch proof size is $O(\min\{k \log n, p\})$.

Verification. For BalanceProofs, Verify() takes $O(1)$ time for individual proofs and $O(|I| \log^2 |I|)$ time for batch proofs (same as aSVC) [42, 45]. For aSVC-bucketing, Verify() takes $O(1)$ time for individual proofs and $O(|I| \log^2 |I|)$ time for batch proofs (also considering the worst case directly). Similarly, for AMT-bucketing, Verify() takes $O(\log n)$ time for individual proofs and $O(|I| \log^2 |I|)$ time for batch proofs. The verification key is $O(|I|)$ size.

5.2 Experimental settings

Our implementation is in Golang. We choose BLS12-381 [23], a pairing friendly elliptic curve to implement pairings, which is also the elliptic curve used in Hyperproofs and offers 128 bits of security. We run each experiment 8 times and report the average.

Hardware. We ran all the experiments on an AWS EC2 m5d.4xlarge instance with Intel(R) Xeon(R) Platinum 8259CL CPU with 2.50GHz, 8 cores and 64GB of RAM. Although our current implementation is not parallelized and we only utilize a single CPU core in our experiments, all of our algorithms are parallelizable.

5.3 Benchmarks

We benchmark the performance of BalanceProofs of all three versions in Table 2 (without bucketing) and Table 3 (with bucketing). For the two bucketing versions, aSVC-bucketing and AMT-bucketing, their experimental results are almost equal except for UpdateAllProofs() and aggregated proof sizes, so we combine their results in Table 3 where each row without "AMT-like" tag is for aSVC-bucketing by default.

Committing. We commit to vectors of size $n = 2^L$ where L ranges from 20 to 30. There is no difference between the three versions of BalanceProofs. For $L = 28$ it takes roughly 144 minutes to compute the commitment.

Opening all the proofs. It takes hour-level time to open all the proofs so that we can only run experiments when $L < 24$. For bucketing versions, we first compute $n^{1/3}$ batch proofs and then compute $n^{1/3} \cdot n^{2/3}$ subproofs inside $n^{1/3}$ subvectors in $O(n^{1/3} \cdot n^{2/3} \log(n^{2/3})) = O(n \log n)$ time with smaller constants than no-bucketing version.

$L = \log_2 n$	20	22	24	26	28	30
Commit (min)	0.57	2.12	8.74	35.62	143.58	*
OpenAll (hrs)	0.87	4.27	*	*	*	*
UpdAllProofs (s)	3.03	6.63	14.35	30.55	64.49	135.69
Query Indiv. (s)	0.09	0.17	0.39	0.81	1.53	3.09
Indiv. Verify (ms)	1.47	1.50	1.51	1.49	1.51	1.51
Aggregate (s)	1.58	1.57	1.58	1.57	1.58	1.59
Agg. Verify (ms)	120.5	122.7	120.3	123.7	122.6	120.9
Indiv. proof size	48 bytes					
Agg. proof size	48 bytes					

Table 2: Single-thread benchmarks for basic BalanceProofs taking aSVC as input. Running times with an asterisk(*) are too long to test (more than 5 hours).

$L = \log_2 n$	20	22	24	26	28	30
Commit (min)	0.57	2.12	8.74	35.62	143.58	*
OpenAll (hrs)	0.59	2.51	*	*	*	*
UpdAllProofs (s)	0.39	0.57	0.82	1.19	1.70	2.52
–AMT-bucketing	0.26	0.39	0.58	0.84	1.21	1.77
Query Indiv. (s)	0.012	0.017	0.025	0.051	0.069	0.098
Indiv. Verify (ms)	1.96	1.96	1.99	1.95	1.97	1.99
Aggregate (s)	0.12	0.12	0.10	0.11	0.11	0.10
Agg. Verify (ms)	5.15	4.12	4.75	5.16	4.51	4.93
Indiv. proof size	96 bytes					
Agg. proof size	6KB	12KB	24KB	24KB	48KB	96KB
–AMT-bucketing	12KB	24KB	48KB	48KB	96KB	192KB

Table 3: Single-thread benchmarks for BalanceProofs (aSVC-bucketing and AMT-bucketing). Numbers are for aSVC-bucketing by default. Running times with an asterisk(*) are too long to test (more than 5 hours).

Updating all the proofs. Since we have implemented the de-amortization version for UpdateAllProofs, we do not need to run at least $O(\sqrt{n})$ updates to measure the real update time correctly. We choose a batch of 1024 random updates and measure their running time on average. Although for no-bucketing version we need about 135 seconds to update all proofs when $L = 30$, we can reduce the time to second-level ($1 \sim 2$ s) with the help of bucketing, which is acceptable and realistic in practice.

Also, AMT-bucketing reduces $20\% \sim 30\%$ of the time to update all the proofs comparing to aSVC-bucketing. Based on previous analysis, we believe that if we set the bucketing size p much larger the improvement of AMT-bucketing will be more significant comparing to aSVC-bucketing.

Querying individual proofs. It requires about 3s to query one proof on average for $L = 30$ in no-bucketing version, and 0.1s to query one proof on average for $L = 30$ in bucketing versions. Note that when $L = 30$, update record list size is at most $n^{1/2} \approx 2^{15} \approx 30 \cdot 2^{10}$ in no-bucketing version and at most $n^{1/3} \approx 2^{10}$ in bucketing versions.

Proof size and verification time. The individual proof size is 1 $\mathbb{G}_1$ element for no-bucketing, 2 $\mathbb{G}_1$ elements for aSVC-bucketing and $\frac{1}{3}L+1$ $\mathbb{G}_1$ elements for AMT-bucketing. The verification of one individual proof needs 2 pairings without bucketing, 3 pairings for aSVC-bucketing and $\frac{1}{3}L + 2$ pairings for AMT-bucketing. We optimize those pairings to computation of Miller loops and final exponentiations.

Aggregation. We aggregate 1024 individual proofs in our experiments, since 1024 is a common average number of transactions in one block of cryptocurrencies [31, 20]. Therefore, the time of Aggregate in Table 2 and Table 3 remains almost unchanged when l ranges from 20 to 30 because the time complexity of Aggregate is $O(|I| \log^2 |I|)$. In the next subsection, we will show that our aggregation can be 1000× faster than Hyperproofs.

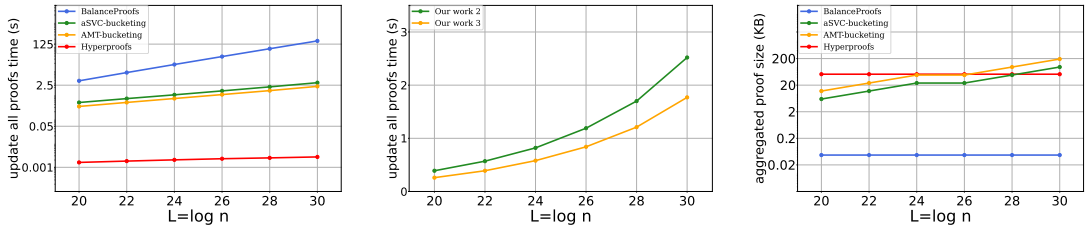
Size and verification time of aggregated proofs. The aggregated proof size is 1 $\mathbb{G}_1$ element without bucketing, $2^{1+L/3}$ $\mathbb{G}_1$ elements for aSVC-bucketing and $2^{2+L/3}$ $\mathbb{G}_1$ elements for AMT-bucketing. The verification of one aggregated proof needs 2 pairings without bucketing, 3 pairings for aSVC-bucketing and $\frac{1}{3}L + 2$ pairings for AMT-bucketing, together with $O(|I| \log^2 |I|)$ time to do polynomial calculations.

5.4 Comparison with Hyperproofs

In this subsection, we compare BalanceProofs with other VC schemes. We do not compare with VC schemes that are not aggregatable, such as Lattice-based VC[32, 35, 33] and AMT[41]. We also do not compare with VC schemes that use at least linear time to update all proofs, since it is not meaningful to update all the proofs in $\Omega(n)$ time where $n > 2^{20}$. Thus we choose Hyperproofs and do all the experiments on the same machine. Both implementations are in Golang and use BLS12-381[23] as elliptic curves. The code of Hyperproofs we used is cloned from Github[37]. We show some of comparison results in Figure 5.

Opening proofs. Hyperproofs involve computing a multilinear tree (MLT) to open all proofs, which needs $O(n \log n)$ time asymptotically and about 2.5 hours in experiments when $L = 24$. While our schemes may require 10+ hours to open all proofs when $L = 24$, in practice we rarely run the whole OpenAll algorithm frequently.

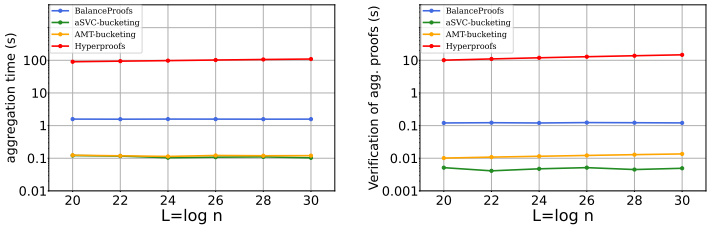
Updating all proofs. We choose a batch of 1024 random updates and evaluate their average time. The time required by Hyperproofs to update all proofs is relatively small since their proofs are in tree-structure and they just need $O(L) = O(\log n)$ group operations to update all proofs. Although our UpdateAllProofs() runs in slower time, the numbers are all reasonable in practice (up to 3s for $L = 30$) if using the bucketing technique. Hyperproofs can query one latest individual proof by grabbing proofs from the proof tree without any group or field operations,



(a) Time to update all proofs

(b) Time to update all proofs (comparing bucketing versions)

(c) Aggregated proof size



(d) Aggregation time

(e) Verification time for aggregated proofs

Figure 5: Comparison between BalanceProofs and Hyperproofs. Including time to update all proofs (average time of 1024 updates), aggregated proof size (aggregating 1024 individual proofs), time to aggregate 1024 proofs and time to verify aggregated proofs.

while our schemes needs to go through the auxiliary information (up to 0.12s for $L = 30$ in bucketing versions).

Aggregation. The size of individual proofs to be aggregated is 1024. Due to the black-box use of IPA argument [8], Hyperproofs needs about $90 \sim 110$s to aggregate 1024 proofs and about $10 \sim 15$s to verify the aggregated proofs, with L ranging from 20 to 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, in all of our schemes the aggregation time is at most 1.58s and the verification is even millisecond-level, which is very efficient for practical use.

For aggregated proof size, basic BalanceProofs has $1000\times$ smaller aggregated proof size, while bucketing versions has almost same-level aggregated proof size comparing to Hyperproofs. However, the size of aggregated proofs in Hyperproofs depends on the smallest power of two $\geq \log(|I| \log n) = \log |I| + \log \log n$, which is 16 if $|I| = 1024$ and L ranges from 20 to 30. If $|I| \geq 4096 = 2^{12}$ and L still ranges from 20 to 30, this power of two will be 32 and the aggregated proof size will be doubled, i.e., 103KB, while our bucketing versions will remain no change in $O(2^{L/3}) = O(n^{1/3})$ size.

Parameterization. Furthermore, we want to stress that with different versions BalanceProofs are more flexible comparing to Hyperproofs. In practice, if you want an application involving many aggregating operations and aggregated proof size matters a lot, you can choose the scheme without bucketing to achieve constant-size aggregated proof. If you want an application involving many update requests and aggregation is a useful but not commonly used operation, you can choose the scheme in the bucketing versions and determine the bucketing size p on your choice. However, Hyperproofs provide just one option where you can update all proofs quickly but

$L = 20$	Our scheme	Hyperproofs	Merkle trees
UpdAllProofs	2.51 ms	1.59 ms	~ 14 μs
Indiv. proof size	0.18 KB	0.96 KB	0.63 KB
$L = 30$	Our scheme	Hyperproofs	Merkle trees
UpdAllProofs	3.75 ms	2.68 ms	~ 21 μs
Indiv. proof size	0.28 KB	1.44 KB	0.94 KB

Table 4: Comparison between our schemes with Verkle trees, Hyperproofs and Merkle trees

aggregate proofs inefficiently in practice.

5.5 Applying our compiler to Verkle trees

As we mentioned in the last subsection, although the running time of UpdateAllProof() in our schemes is realistic and applicable in practice, it is still about $1000\times$ of time of UpdateAllProof() in Hyperproofs when $L = 30$. In this subsection, we can try to improve the time of UpdateAllProof() in our schemes even further with the help of Verkle trees[24] *if we do not need to do aggregation among individual proofs*. Additionally, Verkle trees also have a great feature that it can reduce the proof size from $\log_2 n$ to $\log_b n$ with branching factor b, comparing to other tree-structure proofs like Merkle trees[29]. If we apply aSVC-bucketing BalanceProofs (with *constant* individual proof size) to Verkle trees, we can not only improve the performance of UpdateAllProof() but reduce the individual proof size to $O(\log_b n)$ as well.

Since one of our main goals is to reduce the individual proof size as much as possible (from $O(\log_2 n)$ to $O(\log_b n)$), we do not apply any VC scheme with log-size individual proof (including AMT-bucketing BalanceProofs) to Verkle trees (otherwise there will be no change of the individual proof size: $O(\log_b n) \cdot O(\log_2 b) = O(\log_2 n)$).

Experimental settings. We apply aSVC-bucketing BalanceProofs to Verkle trees, and compare its performance with Hyperproofs and Merkle trees. We take go-verkle [17] as a reference to implement Verkle trees. We evaluate everything on $L = 20, 30$, i.e., we are computing and updating proofs for a vector of size $n = 2^{20}$ and $n = 2^{30}$. For our scheme, we choose $b = 2^{10}$. Thus the height of the Verkle Tree is $\log_b n = 2$ or 3, so there are only 2 or 3 vector commitments from the root to any leaf. We use SHA256 as the hash function to hash each commitment in the internal node to bytes. Those hashed bytes will be converted into one field element and involve vector commitment computation in their parent nodes. For the bucketing size of our scheme, we choose $p = \sqrt{b} = 2^5$.

For Merkle trees, we also choose SHA256 as the hash function to conduct fair comparisons.

Discussion. The experiment results are shown in Table 4. We are only interested in the time of UpdAllProofs and individual proof size. For the time of UpdAllProofs, our scheme has very close performance to Hyperproofs (less than twice as theirs). More importantly, the time of UpdAllProofs in our scheme will increase logarithmically when $n = 2^L$ increases, since the size of the vector commitment in the Verkle Tree b does not change at all. This can make our scheme much more useful if we may deal with datasets of size 2^{60} or even larger in the future. Furthermore, both the public parameter size (including update keys) and the generation time for our scheme is relatively small, comparing to Hyperproofs ($b^{4/3} = 10000 \ll n = 2^{30}$). For the individual proof size, our scheme is about $0.2\times$ as Hyperproofs and $0.3\times$ as Merkle trees.

We can also cut the individual proof size in half if we choose basic BalanceProofs as the vector commitment in the Verkle trees with reasonable increases in the update time.

6 Conclusion

We presented BalanceProofs, a new compiler that produces efficiently *maintainable* and *aggregatable* VC schemes. We also presented two bucketing variants of BalanceProofs which support making the tradeoff between time complexity and proof size. We showed that aSVC-bucketing BalanceProofs has practical UpdateAllProofs() time and $10\times$ to $100\times$ better performance than Hyperproofs. Also, with the help of Verkle trees and ignoring aggregation, our scheme has close performance and much smaller public parameter size and proof size than Hyperproofs.

Future work. We picked aSVC [42] as the input VC to our compiler. It would be very interesting to input some other VC schemes, such as Pointproofs[18], BBF[5], etc, to achieve possible improvements in public parameters, update time and proof sizes. Also, we can try using multilinear trees and PST commitments [39, 50, 49] instead of AMTs[41] and KZG proofs[22] in the bucketing technique to see if there is any other improvement. Lastly, the idea of bookkeeping to balance the time of updating and querying may be applicable in some other cryptographic building blocks.

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] Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-knowledge sets with short proofs. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, pages 433–450, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

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

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

[17] gballet. Implementation of Verkle trees in Go. https://github.com/gballet/go-verkle, 2021. Accessed: 2022-04.

[18] 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.

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

[20] 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.

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

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

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

[24] John Kuszmaul. Verkle Trees: V(ery short M)erkle Trees. https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf, 2018.

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

[26] 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.

[27] Benoît Libert and Moti Yung. Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs. In *TCC'10*, 2010.

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

[29] 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.

[30] 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.

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

[32] 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.

[33] Chris Peikert, Zachary Pepin, and Chad Sharp. Vector and functional commitments from lattices. Cryptology ePrint Archive, Report 2021/1254, 2021. `https://ia.cr/2021/1254`.

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

[35] 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.

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

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

[38] 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.

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

[40] Peter Todd. Making utxo set growth irrelevant with low-latency delayed txo commitments, 2016. `https://petertodd.org/2016/delayed-txo-commitments`.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] Joachim von zur Gathen and Jurgen Gerhard. Newton iteration. In *Modern Computer Algebra*, chapter 9, pages 257–292. Cambridge University Press, 3rd edition, 2013.

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

[48] 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`.

[49] 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.

[50] 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 aSVC-bucketing BalanceProofs

We present the whole aSVC-bucketing BalanceProofs by taking aSVC[42] as the input VC scheme. Note that since original aSVC does not use any auxiliary information, we just remove it in their interface to simplify notations.

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

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

Returns $\mathsf{pp}' = (\mathsf{pp}, (g^{r_i(\tau)})_{i\in[0,n)}, (g^{r_{i,j}(\tau)})_{i\in[0,n),j\in[0,p)})$.

(2) $\mathsf{VC}'.\mathsf{Commit}_{\mathsf{pp}'}(\mathbf{m}) \to (\mathsf{C}, \mathsf{aux})$: Calls $\mathsf{VC}.\mathsf{Commit}_{\mathsf{pp}}(\mathbf{m}) \to \mathsf{C}$. Calls $\mathsf{VC}'.\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{m})$ and returns $(\Pi_j, (\pi_{j,k})_{k\in P_j})_{j\in[0,p)}$. Initializes empty lists $(L_j)_{j\in[0,p)}$. Returns

$$(\mathsf{C}, [(L_j)_{j\in[0,p)}; (\Pi_j, (\pi_{j,k})_{k\in P_j})_{j\in[0,p)}])$$

.

(3) $\mathsf{VC}'.\mathsf{Open}_{\mathsf{pp}'}(i, \mathbf{m}, \mathsf{aux}) \to (\Pi_{j_0}, \pi_{j_0,i})$: Parses $\mathsf{aux} = [(L_j)_{j\in[0,p)}; (\Pi_j, (\pi_{j,k})_{k\in P_j})_{j\in[0,p)}]$. Finds j_0 such that $i \in P_{j_0}$. If $L_{j_0} = \emptyset$, outputs $(\Pi_{j_0}, \pi_{j_0,i})$. Otherwise for each update request (j, δ_j) in L_{j_0} calls $\mathsf{VC}'.\mathsf{UpdateProof}_{\mathsf{pp}}(j, \delta_j, i, (\Pi_{j_0}, \pi_{j_0,i}))$ in turn and finally returns the correct latest proof $(\Pi_{j_0}, \pi_{j_0,i})$.

(4) $\mathsf{VC}'.\mathsf{OpenAll}_{\mathsf{pp}'}(\mathbf{m}) \to (\Pi_j, (\pi_{j,k})_{k\in P_j})_{j\in[0,p)}$: Computes batch proofs $(\Pi_j)_{j\in[0,p)}$ from $\mathbf{m}$. Parses $\mathbf{m} = (\mathbf{v}_0, \ldots, \mathbf{v}_{p-1})$. Calls $\mathsf{VC}.\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{v}_j) \to (\pi_{j,k})_{k\in P_j}$ for all $j \in [0, p)$ and returns $(\Pi_j, (\pi_{j,k})_{k\in P_j})_{j\in[0,p)}$.

(5) $\mathsf{VC}'.\mathsf{Agg}_{\mathsf{pp}'}(I, (m_i, (\Pi_{j_i}, \pi_{j_i,i}))_{i\in I}) \to \pi_I$: First, partitions I as $I_0, \ldots, I_{k-1}$ and finds $j_0, \ldots, j_{k-1}$ such that $I_t \subseteq P_{j_t}$, $\forall t \in [0, k)$. For all $t \in [0, k)$, calls $\mathsf{VC}.\mathsf{Agg}_{\mathsf{pp}}(I_t, (m_c, \pi_{j_t,c})_{c\in I_t}) \to \pi_{I_t}$ and returns $(\Pi_{j_t}, \pi_{I_t})_{t\in[0,k)}$.

(6) $\mathsf{VC}'.\mathsf{Verify}_{\mathsf{pp}'}(\mathsf{C}, I, (m_i)_{i\in I}, \pi_I) \to \{0, 1\}$: If $|I| = 1$, then parses π_I as $(\Pi_j, \pi_{j,i})$ and then checks if Equation 9 holds; If $|I| > 1$, then parses π_I as pairs of form (Π_{j_t}, π_{j_t}) and then checks if Equation 10 holds.

(7) $\mathsf{VC}'.\mathsf{UpdateCom}_{\mathsf{pp}'}(i, \delta, \mathsf{C}) \to \mathsf{C}'$: Calls $\mathsf{VC}.\mathsf{UpdateCom}_{\mathsf{pp}}(i, \delta, \mathsf{C}) \to \mathsf{C}'$ and returns C'.

(8) $\mathsf{VC}'.\mathsf{UpdateAllProofs}_{\mathsf{pp}'}(i, \delta, (\Pi_j, (\pi_{j,k})_{k\in P_j})_{j\in[0,p)}, \mathsf{aux}) \to ((\Pi'_j, (\pi'_{j,k})_{k\in P_j})_{j\in[0,p)}, \mathsf{aux}')$: Parses $\mathsf{aux} = [(L_j)_{j\in[0,p)}; \ldots]$. Uses $(g^{r_{i,j}(\tau)})_{i\in[0,n),j\in[0,p)}$ to update all batch proofs to $(\Pi'_j)_{j\in[0,p)}$. Parses $\mathbf{m}' = (\mathbf{v}_0, \ldots, \mathbf{v}_{p-1})$. Finds j_0 such that $i \in P_{j_0}$. Set $(\pi'_{k,l})_{k\neq j_0} = (\pi_{k,l})_{k\neq j_0}$. Appends (i, δ) to L_{j_0}. If $|L_{j_0}| \geq \sqrt{\frac{n}{p}}$, then calls $\mathsf{VC}.\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{v}_{j_0})$ to get all new single proofs inside $\mathbf{v}_{j_0}$: $(\pi'_{j_0,l})_{l\in P_{j_0}}$ and empties L_{j_0}; otherwise set $(\pi'_{j_0,l})_{l\in P_{j_0}} = (\pi_{j_0,l})_{l\in P_{j_0}}$. Lets aux' collect all the new lists and proofs. Returns $((\Pi'_j, (\pi'_{j,k})_{k\in P_j})_{j\in[0,p)}, \mathsf{aux}')$.

(9) $\mathsf{VC}'.\mathsf{UpdateProof}_{\mathsf{pp}'}(i, \delta, j, (\Pi_{j_0}, \pi_{j_0,j})) \to (\Pi'_{j_0}, \pi'_{j_0,j})$:If $i \in P_{j_0}$, then calls $\mathsf{VC}.\mathsf{UpdateProof}_{\mathsf{pp}}$ $(i, \delta, j, \pi_{j_0,j}) \to \pi'_{j_0,j}$ and returns $(\Pi_{j_0}, \pi'_{j_0,j}))$; otherwise, returns $(\Pi'_{j_0} = \Pi_{j_0} \cdot (g^{r_{i,j_0}(\tau)})^\delta, \pi_{j_0,j}))$.

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 $a \in \mathbb{Z}_p / \{-\tau\}$:

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

Next, we show the q-SBDH assumption [19] which will be used to give soundness proofs for our VC schemes. q-SBDH assumption is a variant of q-SDH (q-Strong Diffie-Hellman) 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 $a \in \mathbb{Z}_p/\{-\tau\}$:

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

C Security Proofs

In this section, we show security proofs for BalanceProofs in bucketing versions.

C.1 Soundness for aSVC-bucketing proofs.

For aSVC-bucketing technique, we present the soundness proof for individual proofs first.

Theorem C.1. Our individual aSVC-bucketing proofs from Section 4.1 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.

$\mathcal{A}$ should output some $i \in P_{i_0}, w_0, w_0', w_1, w_1', z_i, z_i'$ such that we have the following, where $a_{i_0}(x) = \prod_{j \in P_{i_0}}(x - \omega^j)$:

$$e(C/g^{z_i}, g) = e\left(w_0, g^{a_{i_0}(\tau)}\right) \cdot e\left(w_1, g^\tau / g^{\omega^i}\right) \tag{17}$$

$$e(C/g^{z_i'}, g) = e\left(w_0', g^{a_{i_0}(\tau)}\right) \cdot e\left(w_1', g^\tau / g^{\omega^i}\right) \tag{18}$$

Divide the two equations:

$$e(g^{z_i'-z_i}, g) = e\left(w_0/w_0', g^{a_{i_0}(\tau)}\right) \cdot e\left(w_1/w_1', g^{\tau-\omega^i}\right) \tag{19}$$

Rewrite $a_{i_0}(x) = a'_{i_0}(x) \cdot (x - \omega^i)$, then we have

$$\begin{aligned} e(g,g)^{z_i'-z_i} &= e\left(w_0/w_0', g^{a'_{i_0}(\tau)}\right)^{\tau-\omega^i} \cdot e\left(w_1/w_1', g\right)^{\tau-\omega^i} \\ &= \left(e\left(w_0/w_0', g^{a'_{i_0}(\tau)}\right) \cdot e\left(w_1/w_1', g\right)\right)^{\tau-\omega^i} \end{aligned} \tag{20}$$

Finally we have

$$e(g,g)^{\frac{1}{\tau-\omega^i}} = \left(e\left(w_0/w_0', g^{a'_{i_0}(\tau)}\right) \cdot e\left(w_1/w_1', g\right)\right)^{\frac{1}{z_i'-z_i}}, \tag{21}$$

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

Then we present the soundness proof for batch proofs.

Theorem C.2. Our batch aSVC-bucketing proofs from Section 4.1 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 $|I| > 1$. We show how to break $(n-1)$-SBDH assumption.

$\mathcal{A}$ should output some I, J where some $i \in I \cap J$ is the position $\mathcal{A}$ will forge. Suppose $i \in P_{i_0}$ and let $I_0 = I \cap P_{i_0}, J_0 = J \cap P_{i_0}$. $\mathcal{A}$ should also provide $w_0, w_0', w_1, w_1', c(x), c'(x)$ such that we have the following:

$$e(C/g^{c(\tau)}, g) = e\left(w_0, g^{a_{i_0}(\tau)}\right) \cdot e\left(w_1, g^{b(\tau)}\right) \tag{22}$$

$$e(C/g^{c'(\tau)}, g) = e\left(w_0', g^{a_{i_0}(\tau)}\right) \cdot e\left(w_1', g^{b'(\tau)}\right) \tag{23}$$

where $b(x) = \prod_{k \in I_0}(x - \omega^k)$, $b'(x) = \prod_{k \in J_0}(x - \omega^k)$, $c(x)$ is interpolation of claimed values for I_0, $c'(x)$ is interpolation of claimed values for J_0 and $c(\omega^i) \neq c'(\omega^i)$.

Divide the two equations:

$$e(g^{c'(\tau)-c(\tau)}, g) = e\left(w_0/w_0', g^{a_{i_0}(\tau)}\right) \cdot \frac{e\left(w_1, g^{b(\tau)}\right)}{e\left(w_1', g^{b'(\tau)}\right)} \tag{24}$$

Rewrite $a_{i_0}(x) = a'_{i_0}(x) \cdot (x - \omega^i)$:

$$e(g^{c'(\tau)-c(\tau)}, g) = e\left(w_0/w_0', g^{a'_{i_0}(\tau)}\right)^{\tau-\omega^i} \cdot \frac{e\left(w_1, g^{b(\tau)}\right)}{e\left(w_1', g^{b'(\tau)}\right)} \tag{25}$$

Let $c_i = c(\omega^i)$ and $c_i' = c'(\omega^i)$, then we can write $c(x) = d(x)(x - \omega^i) + c_i$ and $c'(x) = d'(x)(x - \omega^i) + c_i'$:

$$\begin{aligned} e(g^{d'(\tau)-d(\tau)}, g)^{\tau-\omega^i} \cdot e(g, g)^{c_i'-c_i} = \\ e\left(w_0/w_0', g^{a'_{i_0}(\tau)}\right)^{\tau-\omega^i} \cdot \frac{e\left(w_1, g^{b(\tau)}\right)}{e\left(w_1', g^{b'(\tau)}\right)} \end{aligned} \tag{26}$$

Let $b(x) = b_0(x)(x - \omega^i)$ and $b'(x) = b_0'(x)(x - \omega^i)$:

$$\begin{aligned} e(g^{d'(\tau)-d(\tau)}, g)^{\tau-\omega^i} \cdot e(g, g)^{c_i'-c_i} = \\ e\left(w_0/w_0', g^{a'_{i_0}(\tau)}\right)^{\tau-\omega^i} \cdot \frac{e\left(w_1, g^{b_0(\tau)}\right)^{\tau-\omega^i}}{e\left(w_1', g^{b_0'(\tau)}\right)^{\tau-\omega^i}} \end{aligned} \tag{27}$$

Finally we have

$$e(g, g)^{\frac{1}{\tau-\omega^i}} = \left(\frac{e\left(w_0/w_0', g^{a'_{i_0}(\tau)}\right) e\left(w_1, g^{b_0(\tau)}\right)}{e\left(g^{d'(\tau)-d(\tau)}, g\right) e\left(w_1', g^{b_0'(\tau)}\right)}\right)^{\frac{1}{c_i'-c_i}}, \tag{28}$$

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

C.2 Soundness for AMT-like bucketing proofs

For AMT-like bucketing technique, we present the soundness proof for individual proofs first.

Theorem C.3. Our individual AMT-bucketing proofs from Section 4.2 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.

$\mathcal{A}$ should output some forged position $i \in P_j$. $\mathcal{A}$ should also provide $w_0 = (w_{0,k})_{k\in[0,L')}$, $w_0' = (w'_{0,k})_{k\in[0,L')}$, w_1, w_1', z_i, z_i' such that we have the following (we use the same notations as Section 4.2, i.e., $u_k = \frac{n}{2^{k+1}}$ and $j_k = j \mod 2^k,\ k \in [0, L']$):

$$e(C/g^{z_i}, g) = e\left(w_1, g^{\tau}/g^{\omega^i}\right) \cdot \prod_{k\in[0,L')} e\left(w_{0,k}, g^{\tau^{u_k} - \omega^{j_{k+1}u_k}}\right) \tag{29}$$

$$e(C/g^{z_i'}, g) = e\left(w_1', g^{\tau}/g^{\omega^i}\right) \cdot \prod_{k\in[0,L')} e\left(w'_{0,k}, g^{\tau^{u_k} - \omega^{j_{k+1}u_k}}\right) \tag{30}$$

Divide the two equations:

$$e(g^{z_i'-z_i}, g) = e\left(w_1/w_1', g^{\tau-\omega^i}\right) \cdot \prod_{k\in[0,L')} e\left(w_{0,k}/w'_{0,k}, g^{\tau^{u_k} - \omega^{j_{k+1}u_k}}\right) \tag{31}$$

Note that for each $k \in [0, L')$, $x^{u_k} - \omega^{j_{k+1}u_k}$ has a factor $x - \omega^i$, then we can rewrite $x^{u_k} - \omega^{j_{k+1}u_k} = b_k(x)(x - \omega^i)$:

$$e(g^{z_i'-z_i}, g) = e\left(w_1/w_1', g^{\tau-\omega^i}\right) \cdot \prod_{k\in[0,L')} e\left(w_{0,k}/w'_{0,k}, g^{b_k(\tau)}\right)^{\tau-\omega^i} \tag{32}$$

$$e(g,g)^{z_i'-z_i} = e\left(w_1/w_1', g\right)^{\tau-\omega^i} \left(\prod_{k\in[0,L')} e\left(w_{0,k}/w'_{0,k}, g^{b_k(\tau)}\right)\right)^{\tau-\omega^i} \tag{33}$$

Finally we have

$$e(g,g)^{\frac{1}{\tau-\omega^i}} = \left(e\left(w_1/w_1', g\right) \prod_{k\in[0,L')} e\left(w_{0,k}/w'_{0,k}, g^{b_k(\tau)}\right)\right)^{\frac{1}{z_i'-z_i}} \tag{34}$$

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

Then we present the soundness theorem for batch proofs, whose proof can be derived directly from the proofs of Theorem C.2 and Theorem C.3.

Theorem C.4. Our batch AMT-bucketing proofs from Section 4.2 are sound as per Definition 2.3 under q-SBDH assumption.