Aggregatable Subvector Commitments for Stateless Cryptocurrencies

Alin Tomescu[✉,1,[0000−0002−6896−1380]], Ittai Abraham[1], Vitalik Buterin[2], Justin Drake[2], Dankrad Feist[2], and Dmitry Khovratovich[2]
{alint,iabraham}@vmware.com, {vitalik,justin,dankrad,dmitry.khovratovich}@ethereum.org

[1] VMware Research, Palo Alto, CA, US
[2] Ethereum Foundation, Singapore, Singapore

Abstract. An *aggregatable subvector commitment (aSVC)* scheme is a *vector commitment (VC)* scheme that can aggregate multiple proofs into a single, small subvector proof. In this paper, we formalize aSVCs and give a construction from constant-sized polynomial commitments. Our construction is unique in that it has linear-sized public parameters, it can compute all constant-sized proofs in quasilinear time, it updates proofs in constant time and it can aggregate multiple proofs into a constant-sized subvector proof. Furthermore, our concrete proof sizes are small due to our use of pairing-friendly groups. We use our aSVC to obtain a payments-only stateless cryptocurrency with very low communication and computation overheads. Specifically, our constant-sized, aggregatable proofs reduce each block's proof overhead to a single group element, which is optimal. Furthermore, our subvector proofs speed up block verification and our smaller public parameters further reduce block size.

1 Introduction

In a *stateless cryptocurrency*, neither *miners* nor cryptocurrency *users* need to store the full *ledger state*. Instead, this state consisting of users' account balances is split among all users using an *authenticated data structure*. This way, miners only store a succinct *digest* of the ledger state and each user stores their account balance. Nonetheless, miners can still validate transactions sent by users, who now include *proofs* that they have sufficient balance. Furthermore, miners can still propose new *blocks* of transactions and users can easily *synchronize* or *update* their proofs as new blocks get published.

Stateless cryptocurrencies have received increased attention [Dry19, RMCI17, CPZ18, BBF19, GRWZ20, STS99, Mil12, Tod16, But17] due to several advantages. First, stateless cryptocurrencies eliminate hundreds of gigabytes of miner storage needed to validate blocks. Second, statelessness makes scaling consensus via *sharding* much easier, by allowing miners to efficiently switch from one shard to another [KJG$^+$18,Com16]. Third, statelessness lowers the barrier to entry for full nodes, resulting in a much more resilient, distributed cryptocurrency.

Stateless Cryptocurrencies from VCs. At a high level, a VC scheme allows a *prover* to compute a succinct *commitment* c to a *vector* $\mathbf{v} = [v_0, v_1, \ldots, v_{n-1}]$ of n *elements* where $v_i \in \mathbb{Z}_p$. Importantly, the prover can generate a *proof* π_i that v_i is the element at position i in $\mathbf{v}$, and any *verifier* can check it against the commitment c. The prover needs a *proving key* prk to commit to vectors and to compute proofs, while the verifier needs a *verification key* vrk to verify proofs. (Usually $|\mathsf{vrk}| \ll |\mathsf{prk}|$.) Some VC schemes support *updates*: if one or more elements in the vector change, the commitment and proofs can be updated efficiently. For this, a static *update key* upk_j tied only to the updated position j is necessary. Alternatively, some schemes require dynamic *update hints* uph_j, typically consisting of the actual proof π_j. The proving, verification and update keys comprise the VC's *public parameters*. Lastly, *subvector commitment (SVC)* schemes [LM19] support computing succinct proofs for *I-subvectors* $(v_i)_{i\in I}$ where $I \subset [0, n)$. Furthermore, some schemes are *aggregatable*: multiple proofs π_i for $v_i, \forall i \in I$ can be aggregated into a single, succinct I-subvector proof.

Chepurnoy, Papamanthou and Zhang pioneered the idea of building *account-based* [Woo], stateless cryptocurrencies on top of any *vector commitment (VC)* scheme [CPZ18]. Ideally, such a VC would have (1) sublinear-sized, updatable proofs with sublinear-time verification, (2) updatable commitments and (3) sublinear-sized update keys. In particular, static update keys (rather than dynamic update hints) help reduce interaction and thus simplify the design (see Section 4.1). We say such a VC has "*scalable updates*." Unfortunately, most VCs do not have scalable updates (see Section 1.1 and Tables 2 and 3) or, if they do [CPZ18,Tom20], they are not optimal in their proof and update key sizes. Lastly, while some schemes in hidden-order groups have scalable updates [CFG$^+$20], they suffer from larger concrete proof sizes and are likely to require more computation in practice.

Our Contributions. In this paper, we formalize a new *aggregatable subvector commitment (aSVC)* notion that supports commitment updates, proof updates and aggregation of proofs into subvector proofs. Then, we construct

An errata for this paper can be found at `https://github.com/alinush/asvc-paper`.

Table 1. Asymptotic comparison of our work with other stateless cryptocurrencies. n is the number of users, λ is the security parameter, and b is the number of transactions in a block. $\mathbb{G}$ is an *exponentiation* in a known-order group. $\mathbb{G}_?$ is a (slower) *exponentiation* (of size 2λ bits) in a hidden-order group. $\mathbb{P}$ is a pairing computation. $|\pi_i|$ is the size of a proof for a user's account balance. $|\mathsf{upk}_i|$ is the size of user i's update key. $|\pi_I|$ is the size of a proof aggregated from all π_i's in a block. We give each *Miner's storage* in terms of VC public parameters (e.g., update keys). A miner takes: (1) *Check digest time*, to check that, by "applying" the transactions from block $t+1$ to block t's digest, he obtains the correct digest for block $t+1$, (2) *Aggr. proofs time*, to aggregate b transaction proofs, and (3) *Vrfy.* $|\pi_I|$ *time*, to verify the aggregated proof. A user takes *Proof synchr. time* to "synchronize" or update her proof by "applying" all the transactions in a new block. We treat [GRWZ20] and [CFG+20] as a payments-only stateless cryptocurrency without smart contracts. Our aggregation and verification times have an extra $b\log^2 b\ \mathbb{F}$ term, consisting of very fast field operations. A detailed analysis of the underlying VCs can be found in Appendices D.4, D.6, D.7 and E.1.

Account-based stateless cryptocurrencies	Edrax [CPZ18]	Pointproofs [GRWZ20]	2nd VC of [CFG+20]	**Our work**
$\|\pi_i\|$	$\log n\ \|\mathbb{G}\|$	$1\ \|\mathbb{G}\|$	$1\ \|\mathbb{G}_?\|$	$1\ \|\mathbb{G}\|$
$\|\mathsf{upk}_i\|$	$\log n\ \|\mathbb{G}\|$	$n\ \|\mathbb{G}\|$	$1\ \|\mathbb{G}_?\|$	$1\ \|\mathbb{G}\|$
$\|\pi_I\|$	$b\log n\ \|\mathbb{G}\|$	$1\ \|\mathbb{G}\|$	$1\ \|\mathbb{G}_?\|$	$1\ \|\mathbb{G}\|$
Miner's storage	$n\ \|\mathbb{G}\|$	$n\ \|\mathbb{G}\|$	$1\ \|\mathbb{G}_?\|$	$b\ \|\mathbb{G}\|$
Vrfy. $\|\pi_I\|$ time	$b\log n\ \mathbb{P}$	$2\ \mathbb{P} + b\ \mathbb{G}$	$b\log b\ \mathbb{G}_?$	$2\ \mathbb{P} + b\ \mathbb{G} + b\lg^2 b\ \mathbb{F}$
Check digest time	$b\ \mathbb{G}$	$b\ \mathbb{G}$	$b\ \mathbb{G}_?$	$b\ \mathbb{G}$
Aggr. proofs time	$\times$	$b\ \mathbb{G}$	$b\log^2 b\ \mathbb{G}_?$	$b\ \mathbb{G} + b\lg^2 b\ \mathbb{F}$
Proof synchr. time	$b\log n\ \mathbb{G}$	$b\ \mathbb{G}$	$b\ \mathbb{G}_?$	$b\ \mathbb{G}$

an aSVC *with scalable updates* over pairing-friendly groups. Compared to other pairing-based VCs, our aSVC has constant-sized, aggregatable proofs that can be updated with constant-sized update keys (see Table 2). Furthermore, our aSVC supports computing all proofs in quasilinear time. We prove security of our aSVC by strengthening (and re-proving) the security definition of KZG polynomial commitments [KZG10].

A Highly-Efficient Stateless Cryptocurrency. We use our aSVC to construct a stateless cryptocurrency based on the elegant design of Edrax [CPZ18]. Our stateless cryptocurrency has very low storage, communication and computation overheads (see Table 1). First, our constant-sized update keys have a smaller impact on block size and help users update their proofs faster. Second, our proof aggregation drastically reduces block size and speeds up block validation. Third, our verifiable update keys remove the need for miners to either (1) store all $O(n)$ update keys or (2) interact during transaction validation to check update keys.

1.1 Related Work

Vector Commitments (VCs). The notion of VCs appears early in [CFM08, LY10, KZG10] but Catalano and Fiore [CF13] are the first to formalize it. They introduce schemes based on the Computational Diffie-Hellman (CDH), with $O(n^2)$-sized public parameters, and on the RSA problem, with $O(1)$-sized public parameters, which can be *specialized* into $O(n)$-sized ones when needed. Lai and Malavolta [LM19] formalize *subvector commitments (SVCs)* and extend both constructions from [CF13] with constant-sized I-subvector proofs. Camenisch et al. [CDHK15] build VCs from KZG commitments [KZG10] to Lagrange polynomials (see Section 2.1) that are not only *binding* but also *hiding*. However, their scheme intentionally prevents aggregation of proofs as a security feature. Feist and Khovratovich [FK20] introduce a technique for precomputing all *constant-sized* evaluation proofs in KZG commitments when the evaluation points are all roots of unity. We use their technique to compute VC proofs fast. Chepurnoy et al. [CPZ18] instantiate VCs using multivariate polynomial commitments [PST13] but with logarithmic rather than constant-sized proofs. Then, they build the first efficient, account-based, stateless cryptocurrency on top of their scheme. Later on, Tomescu [Tom20] presents a very similar scheme but from univariate polynomial commitments [KZG10] which supports subvector proofs.

Boneh et al. [BBF19] instantiate VCs using hidden-order groups. They are the first to support aggregating multiple proofs (under certain conditions; see [BBF18, Sec. 5.2, p. 20]). They are also the first to have constant-sized public parameters, without the need to specialize them into $O(n)$-sized ones. However, their VC uses update hints (rather than keys), which is less suitable for stateless cryptocurrencies. Furthermore, they introduce *key-value map commitments (KVCs)*, which support a larger set of positions from $[0, 2^{2\lambda})$ rather than $[0, n)$, where λ is a security parameter. They argue their KVC can be used for account-based stateless cryptocurrencies, but do not explore a construction in depth. Campanelli et al. [CFG+20] also formalize SVCs with a more powerful notion of *infinite (dis)aggregation* of proofs.

In contrast, our aSVC only supports "one hop" aggregation and does not support disaggregation. They also formalize a notion of updatable, distributed VCs as Verified Decentralized Storage (VDS). However, their use of hidden-order groups leads to larger concrete proof sizes. Both of their schemes have $O(1)$-sized public parameters and can compute all proofs efficiently in quasilinear time. One scheme supports update hints while the other supports update keys.

Concurrent with our work, Gorbunov et al. [GRWZ20] also formalize aSVCs with a stronger notion of *cross-commitment aggregation*. However, their formalization lacks (verifiable) update keys, which hides many complexities that arise in stateless cryptocurrencies (see Section 4.2.2). Their VC scheme extends [LY10] with (1) aggregating proofs into I-subvector proofs and (2) aggregating multiple I-subvector proofs *with respect to different VCs* into a single, constant-sized proof. However, this versatility comes at the cost of (1) losing the ability to precompute all proofs fast, (2) $O(n)$-sized update keys for updating proofs, and (3) $O(n)$-sized verification key. This makes it difficult to apply their scheme in a stateless cryptocurrency for payments such as Edrax [CPZ18]. Furthermore, Gorbunov et al. also enhance KZG-based VCs with proof aggregation, but they do not consider proof updates. Lastly, they show it is possible to aggregate I-subvector proofs across different commitments for KZG-based VCs.

Kohlweiss and Rial [KR13] extend VCs with zero-knowledge protocols for proving correct computation of a new commitment, for opening elements at secret positions, and for proving secret updates of elements at secret positions.

Stateless Cryptocurrencies. The concept of stateless validation appeared early in the cryptocurrency community [Mil12, Tod16, But17] and later on in the academic community [RMCI17, Dry19, CPZ18, BBF19, GRWZ20].

UTXO-based. Initial proposals for UTXO-based cryptocurrencies used Merkle hash trees [Mil12, Tod16, Dry19, CPZ18]. In particular, Dryja [Dry19] gives a beautiful Merkle forest construction that significantly reduces communication. Boneh et al. [BBF19] further reduce communication by using RSA accumulators [BdM94, LLX07].

Account-based. Reyzin et al. [RMCI17] introduce a Merkle-based construction for account-based stateless cryptocurrencies. Unfortunately, their construction relies on *proof-serving nodes*: every user sending coins has to fetch the recipient's Merkle proof from a node and include it with her own proof in the transaction. Edrax [CPZ18] obviates the need for proof-serving nodes by using a vector commitment (VC) with update keys (rather than update hints like Merkle trees). Nonetheless, proof-serving nodes can still be used to assist users who do not want to manually update their proofs (which is otherwise very fast). Unfortunately, Edrax's (non-aggregatable) proofs are logarithmic-sized and thus sub-optimal.

Gorbunov et al. [GRWZ20] introduce *Pointproofs*, a versatile VC scheme which can aggregate proofs across *different* commitments. They use this power to solve a slightly different problem: stateless block validation for smart contract executions (rather than for payments as in Edrax). Unfortunately, their approach requires miners to store a different commitment for each smart contract, or around 4.5 GBs of (dynamic) state in a system with 10^8 smart contracts. This could be problematic in applications such as sharded cryptocurrencies, where miners would have to download part of this large state from one another when switching shards. Lastly, the verification key in Pointproofs is $O(n)$-sized, which imposes additional storage requirements on miners. Furthermore, Gorbunov et al. do not discuss how to update nor precompute proofs efficiently. Instead they assume that all contracts have $n \le 10^3$ memory locations and users can compute all proofs in $O(n^2)$ time. In contrast, our aSVC can compute all proofs in $O(n \log n)$ time [FK20]. Nonetheless, their approach is a very promising direction for supporting smart contracts in stateless cryptocurrencies.

Bonneau et al. [BMRS20] use recursively-composable, succinct non-interactive arguments of knowledge (SNARKs) [BSCTV14] for stateless validation. However, while block validators do not have to store the full state in their system, miners who propose blocks still have to. In contrast, in previous stateless cryptocurrencies (including ours), even miners who propose blocks are stateless.

2 Preliminaries

Notation. λ is our security parameter. $\mathbb{G}_1, \mathbb{G}_2$ are groups of prime order p endowed with a *pairing* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ [MVO91, Jou00]. (We assume symmetric pairings where $\mathbb{G}_1 = \mathbb{G}_2$ for simplicity of exposition.) $\mathbb{G}_?$ is a hidden-order group. We use multiplicative notation for all groups. ω is a primitive nth root of unity in $\mathbb{Z}_p$ [vzGG13a]. $\mathsf{poly}(\cdot)$ is any function upper-bounded by some univariate polynomial. $\mathsf{negl}(\cdot)$ is any negligible function. $\log x$ and $\lg x$ are shorthand for $\log_2 x$. $[i, j] = \{i, i+1, \dots, j-1, j\}$, $[0, n) = [0, n-1]$ and $[n] = [1, n]$. $\mathbf{v} = (v_i)_{i \in [0,n)}$ is a vector of size n with elements $v_i \in \mathbb{Z}_p$.

2.1 Lagrange Polynomial Interpolation

Given n pairs $(x_i, y_i)_{i \in [0,n)}$, we can find or *interpolate* the *unique* polynomial $\phi(X)$ of degree $< n$ such that $\phi(x_i) = y_i, \forall i \in [0, n)$ using *Lagrange interpolation* [BT04] in $O(n \log^2 n)$ time [vzGG13b] as $\phi(X) = \sum_{i \in [0,n)} \mathcal{L}_i(X) y_i$, where

$\mathcal{L}_i(X) = \prod_{j\in[0,n),j\neq i} \frac{X-x_j}{x_i-x_j}$. Recall that a *Lagrange polynomial* $\mathcal{L}_i(X)$ has the property that $\mathcal{L}_i(x_i) = 1$ and $\mathcal{L}_i(x_j) = 0, \forall i, j \in [0, n)$ with $j \neq i$. Note that $\mathcal{L}_i(X)$ is defined in terms of the x_i's which, throughout this paper, will be either $(\omega^i)_{i\in[0,n)}$ or $(\omega^i)_{i\in I}, I \subset [0, n)$.

2.2 KZG Polynomial Commitments

Kate, Zaverucha and Goldberg (KZG) proposed a *constant-sized* commitment scheme for degree n polynomials $\phi(X)$. Importantly, an *evaluation proof* for any $\phi(a)$ is constant-sized and constant-time to verify; it does not depend in any way on the degree of the committed polynomial. KZG requires public parameters $(g^{\tau^i})_{i\in[0,n]}$, which can be computed via a decentralized MPC protocol [BGM17] that hides the *trapdoor* τ. KZG is computationally-hiding under the discrete log assumption and computationally-binding under n-SDH [BB08].

Committing. Let $\phi(X)$ denote a polynomial of degree $d \leq n$ with coefficients $c_0, c_1, \ldots, c_d$ in $\mathbb{Z}_p$. A KZG commitment to $\phi(X)$ is a single group element $C = \prod_{i=0}^{d} \left(g^{\tau^i}\right)^{c_i} = g^{\sum_{i=0}^{d} c_i\tau^i} = g^{\phi(\tau)}$. Committing to $\phi(X)$ takes $\Theta(d)$ time.

Proving One Evaluation. To compute an *evaluation proof* that $\phi(a) = y$, KZG leverages the polynomial remainder theorem, which says $\phi(a) = y \Leftrightarrow \exists q(X)$ such that $\phi(X) - y = q(X)(X - a)$. The proof is just a KZG commitment to $q(X)$: a single group element $\pi = g^{q(\tau)}$. Computing the proof takes $\Theta(d)$ time. To verify π, one checks (in constant time) if $e(C/g^y, g) = e(\pi, g^\tau/g^a) \Leftrightarrow e(g^{\phi(\tau)-y}, g) = e(g^{q(\tau)}, g^{\tau-a}) \Leftrightarrow e(g, g)^{\phi(\tau)-y} = e(g, g)^{q(\tau)(\tau-a)} \Leftrightarrow \phi(\tau) - y = q(\tau)(\tau - a)$.

Proving Multiple Evaluations. Given a set of points I and their evaluations $\{\phi(i)\}_{i\in I}$, KZG can prove all evaluations with a constant-sized *batch proof* rather than $|I|$ individual proofs. The prover computes an *accumulator polynomial* $a(X) = \prod_{i\in I}(X - i)$ in $\Theta(|I| \log^2 |I|)$ time and computes $\phi(X)/a(X)$ in $\Theta(d \log d)$ time, obtaining a quotient $q(X)$ and remainder $r(X)$. The batch proof is $\pi_I = g^{q(\tau)}$. To verify π_I and $\{\phi(i)\}_{i\in I}$ against C, the verifier first computes $a(X)$ from I and interpolates $r(X)$ such that $r(i) = \phi(i), \forall i \in I$ in $\Theta(|I| \log^2 |I|)$ time (see Section 2.1). Next, she computes $g^{a(\tau)}$ and $g^{r(\tau)}$. Finally, she checks if $e(C/g^{r(\tau)}, g) = e(g^{q(\tau)}, g^{a(\tau)})$. We stress that batch proofs are only useful when $|I| \leq d$. Otherwise, if $|I| > d$, the verifier can interpolate $\phi(X)$ directly from the evaluations, which makes verifying any $\phi(i)$ trivial.

2.3 Account-based Stateless Cryptocurrencies

In a stateless cryptocurrency based on VCs [CPZ18], there are *miners* running a permissionless consensus algorithm [Nak08] and *users*, numbered from 0 to $n - 1$ who have *accounts* with a *balance* of coins. (n can be ∞ if the VC is unbounded.) For simplicity of exposition, we do not give details on the consensus algorithm, on transaction signature verification nor on monetary policy. These all remain the same as in previous stateful cryptocurrencies.

The (Authenticated) State. The *state* is an *authenticated data structure (ADS)* mapping each user i's *public key* to their account balance bal_i. (In practice, the mapping is also to a *transaction counter* c_i, which is necessary to avoid transaction replay attacks. We address this in Section 4.3.1.) Importantly, miners and users are *stateless*: they do not store the state, just its *digest* d_t at the latest block t they are aware of. Additionally, each user i stores a proof $\pi_{i,t}$ for their account balance that verifies against d_t.

Miners. Despite miners being stateless, they can still validate transactions, assemble them into a new *block*, and propose that block. Specifically, a miner can verify every new transaction spends valid coins by checking the sending user's balance against the latest digest d_t. This requires each user i who sends coins to j to include her proof $\pi_{i,t}$ in her transaction. Importantly, user i should not have to include the recipient's proof $\pi_{j,t}$ in the transaction, since that would require interacting with *proof-serving nodes* (see Section 4.3.2)

Once the miner has a set V of valid transactions, he can use them to create the next block $t + 1$ and propose it. The miner obtains this new block's digest d_{t+1} by "applying" all transactions in V to d_t. When other miners receive this new block $t + 1$, they can validate its transactions from V against d_t and check that the new digest d_{t+1} was produced correctly from d_t by "reapplying" all the transactions from V.

Users. When creating a transaction tx for block $t + 1$, user i includes her proof $\pi_{i,t}$ for miners to verify she has sufficient balance. When she sees a new block $t + 1$, she can update her proof $\pi_{i,t}$ to a new proof $\pi_{i,t+1}$, which verifies against the new digest d_{t+1}. For this, she will look at all changes in balances $(j, \Delta\mathsf{bal}_j)_{j\in J}$, where J is the set of users with transactions in block $t + 1$, and "apply" those changes to her proof. Similarly, miners can also update proofs of pending transactions which did not make it in block t and now need a proof w.r.t. d_{t+1}

Users assume that the consensus mechanism produces correct blocks. As a result, they do *not* need to verify transactions in the block; they only need to update their own proof. Nonetheless, since block verification is stateless and fast, users could easily participate as block validators, should they choose to.

3 Aggregatable Subvector Commitment (aSVC) Schemes

In this section, we introduce the notion of *aggregatable subvector commitments (aSVCs)* as a natural extension to *subvector commitments (SVCs)* [LM19] where anybody can aggregate b proofs for individual positions into a single constant-sized *subvector proof* for those positions. Our formalization differs from previous work [BBF19, GRWZ20] in that it accounts for (static) update keys as the *verifiable* auxiliary information needed to update commitments and proofs. This is useful in distributed settings where the public parameters of the scheme are split amongst many participants, such as in stateless cryptocurrencies. In Section 3.3, we introduce an efficient aSVC construction *with scalable updates* from KZG commitments to Lagrange polynomials.

3.1 aSVC API

Our API resembles the VC API by Chepurnoy et al. [CPZ18] and the SVC API by Lai and Malavolta [LM19], extended with an API for verifying update keys (see Section 4.2.2) and an API for aggregating proofs. Unlike [CPZ18], our $\mathsf{VC.UpdateProof}$ API receives both upk_i and upk_j as input. This is reasonable in the stateless setting, since each user has to store their upk_i anyway and they extract upk_j from the transactions (see Section 4).

$\mathsf{VC.KeyGen}(1^\lambda, n) \rightarrow \mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)}$. Randomized algorithm that, given a security parameter λ and an upper-bound n on vector size, returns a *proving key* prk, a *verification key* vrk and *update keys* $(\mathsf{upk}_j)_{j\in[0,n)}$.

$\mathsf{VC.Commit}(\mathsf{prk}, \mathbf{v}) \rightarrow c$. Deterministic algorithm that returns a commitment c to any vector $\mathbf{v}$ of size $\le n$.

$\mathsf{VC.ProvePos}(\mathsf{prk}, I, \mathbf{v}) \rightarrow \pi_I$. Deterministic algorithm that returns a proof π_I that $\mathbf{v}_I = (v_i)_{i\in I}$ is the I-subvector of $\mathbf{v}$. For notational convenience, I can be either an index set $I \subseteq [0, n)$ or an individual index $I = i \in [0, n)$.

$\mathsf{VC.VerifyPos}(\mathsf{vrk}, c, \mathbf{v}_I, I, \pi_I) \rightarrow T/F$. Deterministic algorithm that verifies the proof π_I that $\mathbf{v}_I$ is the I-subvector of the vector committed in c. As before, I can be either an index set $I \subseteq [0, n)$ or an individual index $I = i \in [0, n)$.

$\mathsf{VC.VerifyUPK}(\mathsf{vrk}, i, \mathsf{upk}_i) \rightarrow T/F$. Deterministic algorithm that verifies that upk_i is indeed the ith update key.

$\mathsf{VC.UpdateComm}(c, \delta, j, \mathsf{upk}_j) \rightarrow c'$. Deterministic algorithm that returns a new commitment c' to $\mathbf{v'}$ obtained by updating v_j to $v_j + \delta$ in the vector $\mathbf{v}$ committed in c. Needs upk_j associated with the updated position j.

$\mathsf{VC.UpdateProof}(\pi_i, \delta, i, j, \mathsf{upk}_i, \mathsf{upk}_j) \rightarrow \pi'_i$. Deterministic algorithm that updates an old proof π_i for the ith element v_i, given that the jth element was updated to $v_j + \delta$. Note that i can be equal to j.

$\mathsf{VC.AggregateProofs}(I, (\pi_i)_{i\in I}) \rightarrow \pi_I$ Deterministic algorithm that, given proofs π_i for $v_i, \forall i \in I$, aggregates them into a succinct I-subvector proof π_I.

3.2 aSVC Correctness and Security Definitions

We argue why our aSVC from Section 3 satisfies these definitions in Section 3.4.5.

Definition 1 (Aggregatable SVC Scheme). ($\mathsf{VC.KeyGen}$, $\mathsf{VC.Commit}$, $\mathsf{VC.ProvePos}$, $\mathsf{VC.VerifyPos}$, $\mathsf{VC.VerifyUPK}$, $\mathsf{VC.UpdateComm}$, $\mathsf{VC.UpdateProof}$, $\mathsf{VC.AggregateProofs}$) *is a secure aggregatable subvector commitment scheme if* $\forall$ *upper-bounds* $n = \mathsf{poly}(\lambda)$ *it satisfies the following properties:*

Definition 2 (Opening Correctness). $\forall$ *vectors* $\mathbf{v} = (v_j)_{j\in[0,n)}$, $\forall$ *index sets* $I \subseteq [0, n)$*:*

$$\Pr\begin{bmatrix} \mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)} \leftarrow \mathsf{VC.KeyGen}(1^\lambda, n), \\ c \leftarrow \mathsf{VC.Commit}(\mathsf{prk}, \mathbf{v}), \\ \pi_I \leftarrow \mathsf{VC.ProvePos}(\mathsf{prk}, I, \mathbf{v}) : \\ \mathsf{VC.VerifyPos}(\mathsf{vrk}, c, \mathbf{v}_I, I, \pi_I) = T \end{bmatrix} \ge 1 - \mathsf{negl}(\lambda)$$

Definition 3 (Commitment and Proof Update Correctness). *$\forall$ vectors $\mathbf{v} = (v_j)_{j\in[0,n)}$, $\forall$ positions $i, k \in [0, n)$, $\forall$ updates $\delta \in \mathbb{Z}_p$, let $\mathbf{u} = (u_j)_{j\in[0,n)}$ be the same vector as $\mathbf{v}$ except with $v_k + \delta$ rather than v_k at position k. Then:*

$$\Pr\left[\begin{array}{c}\mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)} \leftarrow \mathsf{VC.KeyGen}(1^\lambda, n),\\ c \leftarrow \mathsf{VC.Commit}(\mathsf{prk}, \mathbf{v}),\\ \hat{c} \leftarrow \mathsf{VC.UpdateComm}(c, \delta, k, \mathsf{upk}_k),\\ c' \leftarrow \mathsf{VC.Commit}(\mathsf{prk}, \mathbf{u}) :\\ c' = \hat{c}\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

$$\Pr\left[\begin{array}{c}\mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)} \leftarrow \mathsf{VC.KeyGen}(1^\lambda, n),\\ c \leftarrow \mathsf{VC.Commit}(\mathsf{prk}, \mathbf{v}),\\ c' \leftarrow \mathsf{VC.UpdateComm}(c, \delta, k, \mathsf{upk}_k),\\ \pi_i \leftarrow \mathsf{VC.ProvePos}(\mathsf{prk}, i, \mathbf{v}),\\ \pi'_i \leftarrow \mathsf{VC.UpdateProof}(\pi_i, \delta, i, k, \mathsf{upk}_i, \mathsf{upk}_k) :\\ \mathsf{VC.VerifyPos}(\mathsf{vrk}, c', u_i, i, \pi'_i) = T\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Definition 4 (Aggregation Correctness). *$\forall$ vectors $\mathbf{v} = (v_j)_{j\in[0,n)}$, $\forall$ index sets $I \subseteq [0, n)$:*

$$\Pr\left[\begin{array}{c}\mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)} \leftarrow \mathsf{VC.KeyGen}(1^\lambda, n),\\ c \leftarrow \mathsf{VC.Commit}(\mathsf{prk}, \mathbf{v}),\\ (\pi_i \leftarrow \mathsf{VC.ProvePos}(\mathsf{prk}, i, \mathbf{v}))_{i\in I},\\ \pi_I \leftarrow \mathsf{VC.AggregateProofs}(I, (\pi_i)_{i\in I}) :\\ \mathsf{VC.VerifyPos}(\mathsf{vrk}, c, \mathbf{v}_I, I, \pi_I) = T\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Definition 5 (Update Key Correctness). *$\forall$ positions $i \in [0, n)$:*

$$\Pr\left[\begin{array}{c}\mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)} \leftarrow \mathsf{VC.KeyGen}(1^\lambda, n) :\\ \mathsf{VC.VerifyUPK}(\mathsf{vrk}, i, \mathsf{upk}_i) = T\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Definition 6 (Update Key Uniqueness). *$\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$:*

$$\Pr\left[\begin{array}{c}\mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)} \leftarrow \mathsf{VC.KeyGen}(1^\lambda, n),\\ i, \mathsf{upk}, \mathsf{upk}' \leftarrow \mathcal{A}(1^\lambda, \mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)}) :\\ \mathsf{VC.VerifyUPK}(\mathsf{vrk}, i, \mathsf{upk}) = T \wedge\\ \mathsf{VC.VerifyUPK}(\mathsf{vrk}, i, \mathsf{upk}') = T \wedge\\ \mathsf{upk} \neq \mathsf{upk}'\end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: Definitions that allow for *dynamic* update hints rather than *unique* update keys are possible too, but would be less simple to state and less useful for stateless cryptocurrencies (see Section 4).

Definition 7 (Position Binding Security). *$\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$, if $\mathbf{v}_I = (v_i)_{i\in I}$ and $\mathbf{v}'_J = (v'_j)_{j\in J}$, then:*

$$\Pr\left[\begin{array}{c}\mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_i)_{i\in[0,n)} \leftarrow \mathsf{VC.KeyGen}(1^\lambda, n),\\ (c, I, J, \mathbf{v}_I, \mathbf{v}'_J, \pi_I, \pi_J) \leftarrow \mathcal{A}(1^\lambda, \mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_i)_{i\in[0,n)}) :\\ \mathsf{VC.VerifyPos}(\mathsf{vrk}, c, \mathbf{v}_I, I, \pi_I) = T \wedge\\ \mathsf{VC.VerifyPos}(\mathsf{vrk}, c, \mathbf{v}'_J, J, \pi_J) = T \wedge\\ \exists k \in I \cap J,\ \textit{such that } v_k \neq v'_k\end{array}\right] \leq \mathsf{negl}(\lambda)$$

3.3 aSVC From KZG Commitments to Lagrange Polynomials

In this subsection, we present our aSVC from KZG commitments to Lagrange polynomials. Similar to previous work, we represent a vector $\mathbf{v} = [v_0, v_1, \ldots, v_{n-1}]$ as a polynomial $\phi(X) = \sum_{i\in[0,n)} \mathcal{L}_i(X)v_i$ in Lagrange basis [KZG10, CDHK15, Tom20, GRWZ20]. However, unlike previous work, we add support for efficiently updating and aggregating proofs. For aggregation, we use known techniques for aggregating KZG proofs via *partial fraction decomposition* [But20]. For updating proofs, we introduce a new mechanism to reduce the update key size from linear to constant. We use *roots of unity* and "store" v_i as $\phi(\omega^i) = v_i$, which means our Lagrange polynomials are $\mathcal{L}_i(X) = \prod_{j\in[0,n),j\neq i} \frac{X-\omega^j}{\omega^i-\omega^j}$. For this to work *efficiently*, we assume without loss of generality that n is a power of two.

Table 2. Asymptotic comparison of our aSVC with other (aS)VCs based on prime-order groups. n is the vector size and b is the subvector size. See Appendix D for a more detailed analysis. All schemes have $O(n)$-sized parameters (except [LM19] has $O(n^2)$ and [CFG$^+$20] has $O(1)$); can update commitments in $O(1)$ time (except for [KZG10]); have $O(1)$-sized proofs that verify in $O(1)$ time (except [CPZ18] and [Tom20] proofs are $O(\lg n)$). *Com.* is the time to commit to a size-n vector. *Proof upd.* is the time to update *one* individual proof π_i after a change to *one* vector element v_j. *Prove one*, *Prove subv.* and *Prove each* are the times to compute a proof π_i for one v_i, a size-b subvector proof π_I and proofs for all $(v_i)_{i\in[0,n)}$, respectively.

(aS)VC scheme	$\lvert\mathsf{vrk}\rvert$	$\lvert\mathsf{upk}_i\rvert$	Com.	Prove one	Proof upd.	Prove subv.	Verify subv.	Aggr-egate	Prove each
[LM19]	n	n	n	n	1	bn	b	$\times$	n^2
[KZG10]	b	$\times$	$n\lg^2 n$	n	$\times$	$b\lg^2 b + n\lg n$	$b\lg^2 b$	$\times$	n^2
[CDHK15]	n	n	$n\lg^2 n$	n	1	$n\lg^2 n$	$b\lg^2 b$	$\times$	n^2
[CPZ18]	$\lg n$	$\lg n$	n	n	$\lg n$	$\times$	$\times$	$\times$	n^2
[Tom20]	$\lg n + b$	$\lg n$	$n\lg n$	$n\lg n$	$\lg n$	$b\lg^2 b + n\lg n$	$b\lg^2 b$	$\times$	$n\lg n$
[GRWZ20]	n	n	n	n	1	bn	b	b	n^2
[CFG$^+$20]	1	1	$n\lg n$	$n\lg n$	1	$(n-b)\lg(n-b)$	b	$b\lg^2 b$	$n\lg n$
Our work	b	1	n	n	1	$b\lg^2 b + n\lg n$	$b\lg^2 b$	$b\lg^2 b$	$n\lg n$
Our work*	b	1	$n\lg n$	1	1	$b\lg^2 b$	$b\lg^2 b$	$b\lg^2 b$	$n\lg n$

Committing. A commitment to $\mathbf{v}$ is just a KZG commitment $c = g^{\phi(\tau)}$ to $\phi(X)$, where τ is the trapdoor of the KZG scheme (see Section 2.2). Similar to previous work [CDHK15], the proving key includes commitments to all Lagrange polynomials $\ell_i = g^{\mathcal{L}_i(\tau)}$. Thus, we can compute $c = \prod_{i=1}^{n}(\ell_i)^{v_i}$ in $O(n)$ time without interpolating $\phi(X)$ and update it as $c' = c\cdot(\ell_i)^{\delta}$ after adding δ to v_i. Note that c' is just a commitment to an updated $\phi'(X) = \phi(X) + \delta\cdot\mathcal{L}_i(X)$.

Proving. A proof π_i for a single element v_i is just a KZG evaluation proof for $\phi(\omega^i)$. A subvector proof π_I for for $v_I, I\subseteq[0,n)$ is just a KZG batch proof for all $\phi(\omega^i)_{i\in I}$ evaluations. Importantly, we use the Feist-Khovratovich (FK) [FK20] technique to compute all proofs $(\pi_i)_{i\in[0,n)}$ in $O(n\log n)$ time. This allows us to aggregate I-subvector proofs faster in $O(|I|\log^2|I|)$ time (see Table 2).

3.4 Partial Fraction Decomposition

A key ingredient in our aSVC scheme is *partial fraction decomposition* [Wik19], which we re-explain from the perspective of Lagrange interpolation. First, let us rewrite the Lagrange polynomial for interpolating $\phi(X)$ given all $\left(\phi(\omega^i)\right)_{i\in I}$:

$$\mathcal{L}_i(X) = \prod_{j\in I, j\neq i}\frac{X-\omega^j}{\omega^i-\omega^j} = \frac{A_I(X)}{A_I'(\omega^i)(X-\omega^i)},\ \text{where } A_I(X) = \prod_{i\in I}(X-\omega^i) \tag{1}$$

Here, $A_I'(X) = \sum_{j\in[0,n)} A_I(X)/(X-\omega^j)$ is the derivative of $A_I(X)$ [vzGG13b]. Next, for any $\phi(X)$, we can rewrite the Lagrange interpolation formula as $\phi(X) = A_I(X)\sum_{i\in[0,n)}\frac{y_i}{A_I'(\omega^i)(X-\omega^i)}$. In particular, for $\phi(X)=1$, this implies $\frac{1}{A_I(X)} = \sum_{i\in[0,n)}\frac{1}{A_I'(\omega^i)(X-\omega^i)}$. In other words, we can decompose $A_I(X)$ as:

$$\frac{1}{A_I(X)} = \frac{1}{\prod_{i\in I}(X-\omega^i)} = \sum_{i\in[0,n)} c_i\cdot\frac{1}{X-\omega^i},\ \text{where } c_i = \frac{1}{A_I'(\omega^i)} \tag{2}$$

$A_I(X)$ can be computed in $O(|I|\log^2|I|)$ time using a *subproduct tree* and DFT-based polynomial multiplication [vzGG13b]. Its derivative, $A_I'(X)$, can be computed in $O(|I|)$ time and evaluated at all ω^i's in $O(|I|\log^2|I|)$ time [vzGG13b]. Thus, all c_i's can be computed in $O(|I|\log^2|I|)$ time. For the special case of $I=[0,n)$, we have $A_I(X) = A(X) = \prod_{i\in[0,n)}(X-\omega^i) = X^n - 1$ and $A'(\omega^i) = n\omega^{-i}$ (see Appendix A). In this case, any c_i can be computed in $O(1)$ time.

3.4.1 Aggregating Proofs

We build upon Drake and Buterin's observation [But20] that partial fraction decomposition (see Section 3.4) can be used to aggregate KZG evaluation proofs. Since our VC proofs are KZG proofs, we show how to aggregate a set of proofs $(\pi_i)_{i\in I}$ for elements v_i of $\mathbf{v}$ into a constant-sized I-subvector proof π_I for $(v_i)_{i\in I}$.

Recall that π_i is a commitment to $q_i(X) = \frac{\phi(X)-v_i}{X-\omega^i}$ 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 interpolated such that $R(\omega^i) = v_i, \forall i\in I$. Our goal is to find coefficients $c_i\in\mathbb{Z}_p$

such that $q(X) = \sum_{i \in I} c_i q_i(X)$ and thus aggregate $\pi_I = \prod_{i \in I} \pi_i^{c_i}$. We observe that:

$$q(X) = \phi(X) \frac{1}{A_I(X)} - R(X) \frac{1}{A_I(X)} \tag{3}$$

$$= \phi(X) \sum_{i \in I} \frac{1}{A_I'(\omega^i)(X - \omega^i)} - \left(A_I(X) \sum_{i \in I} \frac{v_i}{A_I'(\omega^i)(X - \omega^i)} \right) \cdot \frac{1}{A_I(X)} \tag{4}$$

$$= \sum_{i \in I} \frac{\phi(X)}{A_I'(\omega^i)(X - \omega^i)} - \sum_{i \in I} \frac{v_i}{A_I'(\omega^i)(X - \omega^i)} = \sum_{i \in I} \frac{1}{A_I'(\omega^i)} \cdot \frac{\phi(X) - v_i}{X - \omega^i} \tag{5}$$

$$= \sum_{i \in I} \frac{1}{A_I'(\omega^i)} \cdot q_i(X) \tag{6}$$

Thus, we can compute all $c_i = 1/A_I'(\omega^i)$ using $O(|I| \log^2 |I|)$ field operations (see Section 3.4) and compute $\pi_I = \prod_{i \in I} \pi_i^{c_i}$ with an $O(|I|)$-sized multi-exponentiation.

3.4.2 Updating Proofs

When updating π_i after a change to v_j, it could be that either $i = j$ or $i \neq j$. First, recall that π_i is a KZG commitment to $q_i(X) = \frac{\phi(X) - v_i}{X - \omega^i}$. Second, recall that, after a change δ to v_j, the polynomial $\phi(X)$ is updated to $\phi'(X) = \phi(X) + \delta \cdot \mathcal{L}_j(X)$. We refer to the party updating their proof π_i as the *proof updater*.

The $i = j$ Case. Consider the quotient polynomial $q_i'(X)$ in the updated proof π_i' after v_i changed to $v_i + \delta$:

$$q_i'(X) = \frac{\phi'(X) - (v_i + \delta)}{X - \omega^i} = \frac{(\phi(X) + \delta \mathcal{L}_i(X)) - v_i - \delta}{X - \omega^i} \tag{7}$$

$$= \frac{\phi(X) - v_i}{X - \omega^i} + \frac{\delta(\mathcal{L}_i(X) - 1)}{X - \omega^i} = q_i(X) + \delta \left(\frac{\mathcal{L}_i(X) - 1}{X - \omega^i} \right) \tag{8}$$

This means the proof updater needs a KZG commitment to $\frac{\mathcal{L}_i(X) - 1}{X - \omega^i}$, which is just a KZG evaluation proof that $\mathcal{L}_i(\omega^i) = 1$. This can be addressed very easily by making this commitment part of upk_i. To conclude, to update π_i, the proof updater obtains $u_i = g^{\frac{\mathcal{L}_i(\tau) - 1}{\tau - \omega^i}}$ from upk_i and computes $\pi_i' = \pi_i \cdot (u_i)^\delta$. (Remember that the proof updater, who calls $\mathsf{VC.UpdateProof}(\pi_i, \delta, i, i, \mathsf{upk}_i, \mathsf{upk}_i)$, has upk_i.)

The $i \neq j$ Case. Now, consider the quotient polynomial $q_i'(X)$ after v_j changed to $v_j + \delta$:

$$q_i'(X) = \frac{\phi'(X) - v_i}{X - \omega^i} = \frac{(\phi(X) + \delta \mathcal{L}_j(X)) - v_i}{X - \omega^i} \tag{9}$$

$$= \frac{\phi(X) - v_i}{X - \omega^i} + \frac{\delta \mathcal{L}_j(X)}{X - \omega^i} = q_i(X) + \delta \left(\frac{\mathcal{L}_j(X)}{X - \omega^i} \right) \tag{10}$$

In this case, the proof updater will need to construct a KZG commitment to $\frac{\mathcal{L}_j(X)}{X - \omega^i}$. For this, we put enough information in upk_i and upk_j, which the proof updater has (see Section 3.1), to help her do so.

Since $U_{i,j}(X) = \frac{A(X)}{A'(\omega^j)(X - \omega^j)(X - \omega^i)}$ and $A'(\omega^j) = n\omega^{-j}$ (see Appendix A), it is sufficient to reconstruct a KZG commitment to $W_{i,j}(X) = \frac{A(X)}{(X - \omega^j)(X - \omega^i)}$, which can be decomposed as $W_{i,j}(X) = A(X) \left(c_i \frac{1}{X - \omega^i} + c_j \frac{1}{X - \omega_j} \right) = c_i \frac{A(X)}{X - \omega^i} + c_j \frac{A(X)}{X - \omega^j}$, where $c_i = 1/(\omega^i - \omega^j)$ and $c_j = 1/(\omega^j - \omega^i)$ (see Section 3.4). Thus, if we include $a_j = g^{A(\tau)/(\tau - \omega^j)}$ in each upk_j, the proof updater can first compute $w_{i,j} = a_i^{c_i} a_j^{c_j}$, then compute $u_{i,j} = (w_{i,j})^{\frac{1}{A'(\omega^j)}}$ and finally update the proof as $\pi_i' = \pi_i \cdot (u_{i,j})^\delta$.

3.4.3 aSVC Algorithms

Having established the intuition for our aSVC, we can now describe it in detail using the aSVC API from Section 3.1. To support verifying I-subvector proofs, our verification key is $O(|I|)$-sized.

$\mathsf{VC.KeyGen}(1^\lambda, n) \rightarrow \mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j \in [0,n)}$. Generates n-SDH public parameters $g, g^\tau, g^{\tau^2}, \ldots, g^{\tau^n}$. Computes $a = g^{A(\tau)}$, where $A(X) = X^n - 1$. Computes $a_i = g^{A(\tau)/(X - \omega^i)}$ and $\ell_i = g^{\mathcal{L}_i(\tau)}, \forall i \in [0, n)$. Computes KZG proofs $u_i = g^{\frac{\mathcal{L}_i(\tau) - 1}{X - \omega^i}}$ for $\mathcal{L}_i(\omega^i) = 1$. Sets $\mathsf{upk}_i = (a_i, u_i)$, $\mathsf{prk} = \left((g^{\tau^i})_{i \in [0,n]}, (\ell_i)_{i \in [0,n)}, (\mathsf{upk}_i)_{i \in [0,n)} \right)$ and $\mathsf{vrk} = ((g^{\tau^i})_{i \in [0,|I|]}, a)$.

$\mathsf{VC.Commit}(\mathsf{prk}, \mathbf{v}) \to c$. Returns $c = \prod_{i\in[0,n)}(\ell_i)^{v_i}$.

$\mathsf{VC.ProvePos}(\mathsf{prk}, I, \mathbf{v}) \to \pi_I$. Computes $A_I(X) = \prod_{i\in I}(X - \omega^i)$ in $O(|I|\log^2|I|)$ time. Divides $\phi(X)$ by $A_I(X)$ in $O(n\log n)$ time, obtaining a quotient $q(X)$ and a remainder $r(X)$. Returns $\pi_I = g^{q(\tau)}$. (We give an $O(n)$ time algorithm in Appendix D.7 for the $|I| = 1$ case.)

$\mathsf{VC.VerifyPos}(\mathsf{vrk}, c, \mathbf{v}_I, I, \pi_I) \to T/F$. 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 $R_I(i) = v_i, \forall i \in I$ in $O(|I|\log^2|I|)$ time and commits to it as $g^{R_I(\tau)}$ in $O(|I|)$ time. Returns T iff. $e(c/g^{R_I(\tau)}, g) = e(\pi_I, g^{A_I(\tau)})$. (When $I = \{i\}$, we have $A_I(X) = X - \omega^i$ and $R_I(X) = v_i$.)

$\mathsf{VC.VerifyUPK}(\mathsf{vrk}, i, \mathsf{upk}_i) \to T/F$. Checks that ω^i is a root of $X^n - 1$ (which is committed in a) via $e(a_i, g^\tau/g^{(\omega^i)}) = e(a, g)$. Checks that $\mathcal{L}_i(\omega^i) = 1$ via $e(\ell_i/g^1, g) = e(u_i, g^\tau/g^{(\omega^i)})$, where $\ell_i = a_i^{1/A'(\omega^i)} = g^{\mathcal{L}_i(\tau)}$.

$\mathsf{VC.UpdateComm}(c, \delta, j, \mathsf{upk}_j) \to c'$. Returns $c' = c \cdot (\ell_j)^\delta$, where $\ell_j = a_j^{1/A'(\omega^j)}$.

$\mathsf{VC.UpdateProof}(\pi_i, \delta, i, j, \mathsf{upk}_i, \mathsf{upk}_j) \to \pi'_i$. If $i = j$, returns $\pi'_i = \pi_i \cdot (u_i)^\delta$. If $i \neq j$, computes $w_{i,j} = a_i^{1/(\omega^i - \omega^j)} \cdot a_j^{1/(\omega^j - \omega^i)}$ and $u_{i,j} = w_{i,j}^{1/A'(\omega^j)}$ (see Section 3.4.2) and returns $\pi'_i = \pi_i \cdot (u_{i,j})^\delta$.

$\mathsf{VC.AggregateProofs}(I, (\pi_i)_{i\in I}) \to \pi_I$. Computes $A_I(X) = \prod_{i\in I}(X - \omega^i)$, its derivative $A'_I(X)$ and all $c_i = (A'_I(\omega^i))_{i\in I}$ in $O(|I|\log^2|I|)$ time. Returns $\pi_I = \prod_{i\in I}\pi_i^{c_i}$.

3.4.4 Distributing the Trusted Setup

Our aSVC requires a centralized, trusted setup phase that computes its public parameters. We can decentralize this phase using highly-efficient MPC protocols that generate (g^{τ^i})'s in a distributed fashion [BGM17]. Then, we can derive the remaining parameters from the (g^{τ^i})'s, which has the advantage of keeping our parameters *updatable*. First, the commitment $a = g^{A(\tau)}$ to $A(X) = X^n - 1$ can be computed in $O(1)$ time via an exponentiation. Second, the commitments $\ell_i = g^{\mathcal{L}_i(\tau)}$ to Lagrange polynomials can be computed via a single DFT on the (g^{τ^i})'s [Vir17, Sec 3.12.3, pg. 97]. Third, each $a_i = g^{A(\tau)/(\tau-\omega^i)}$ is a bilinear accumulator membership proof for ω^i w.r.t. $A(X)$ and can all be computed in $O(n\log n)$ time using FK [FK20]. But what about computing each $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{X-\omega^i}}$?

Computing All u_i's Fast. Inspired by the FK technique [FK20], we show how to compute all n u_i's in $O(n\log n)$ time using a single DFT on group elements. First, note that $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{X-\omega^i}}$ is a KZG evaluation proof for $\mathcal{L}_i(\omega^i) = 1$. Thus, $u_i = g^{Q_i(\tau)}$ where $Q_i(X) = \frac{\mathcal{L}_i(X)-1}{X-\omega^i}$. Second, let $\psi_i(X) = A'(\omega^i)\mathcal{L}_i(X) = \frac{X^n-1}{X-\omega^i}$. Then, let $\pi_i = g^{q_i(\tau)}$ be an evaluation proof for $\psi_i(\omega^i) = A'(\omega^i)$ where $q_i(X) = \frac{\psi_i(X)-A'(\omega^i)}{X-\omega^i}$ and note that $Q_i(X) = \frac{1}{A'(\omega^i)}q_i(X)$. Thus, computing all u_i's reduces to computing all π_i's. However, since each proof π_i is for a *different* polynomial $\psi_i(X)$, directly applying FK does not work. Instead, we give a new algorithm that leverages the structure of $\psi_i(X)$ when divided by $X - \omega^i$. Specifically, in Appendix B, we show that:

$$q_i(X) = \sum_{j\in[0,n-2]} H_j(X)\omega^{ij}, \forall i \in [0,n), \text{ where } H_j(X) = (j+1)X^{(n-2)-j} \tag{11}$$

If we let h_j be a KZG commitment to $H_j(X)$, then we have $\pi_i = \prod_{j\in[0,n-2]} h_j^{(\omega^{ij})}$, $\forall i \in [0,n)$. Next, recall that the Discrete Fourier Transform (DFT) *on a vector of group elements* $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}] \in \mathbb{G}^n$ is:

$$\mathsf{DFT}_n(\mathbf{a}) = \hat{\mathbf{a}} = [\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}] \in \mathbb{G}^n, \text{ where } \hat{a}_i = \prod_{j\in[0,n)} a_j^{(\omega^{ij})} \tag{12}$$

If we let $\boldsymbol{\pi} = [\pi_0, \pi_1, \ldots, \pi_{n-1}]$ and $\mathbf{h} = [h_0, h_1, \ldots, h_{n-2}, 1_\mathbb{G}, 1_\mathbb{G}]$, then $\boldsymbol{\pi} = \mathsf{DFT_n}(\mathbf{h})$. Thus, computing all n h_i's takes $O(n)$ time and computing all n π_i's takes an $O(n\log n)$ time DFT. As a result, computing all u_i's from the (g^{τ^i})'s takes $O(n\log n)$ time overall.

3.4.5 Correctness and Security

The correctness of our aSVC scheme follows naturally from Lagrange interpolation. Aggregation and proof updates are correct by the arguments laid out in Sections 3.4.1 and 3.4.2, respectively. Subvector proofs are correct by the correctness of KZG batch proofs [KZG10].

The security of our aSVC schemes does *not* follow naturally from the security of KZG polynomial commitments. Specifically, as pointed out in [GRWZ20], two inconsistent subvector proofs do *not* lead to a direct break of KZG's *batch evaluation binding*, as defined in [KZG10, Sec. 3.4]. To address this, we propose a stronger batch evaluation binding definition (see Definition 8 in Appendix C.1) and prove KZG satisfies it under n-SBDH. This new definition is directly broken by two inconsistent subvector proofs, which implies our aSVC is secure under n-SBDH. Lastly, we prove update key uniqueness holds unconditionally in Appendix C.2.

4 A Highly-efficient Stateless Cryptocurrency

In this section, we enhance Edrax's elegant design by replacing their VC with our secure *aggregatable* subvector commitment (aSVC) scheme from Section 3.3. As a result, our stateless cryptocurrency has smaller, aggregatable proofs and smaller update keys. This leads to smaller, faster-to-verify blocks for miners and faster proof synchronization for users (see Table 1). Furthermore, our verifiable update keys reduce the storage overhead of miners from $O(n)$ update keys to $O(1)$. We also address a denial of service (DoS) attack in Edrax's design.

4.1 From VCs to Stateless Cryptocurrencies

Edrax pioneered the idea of building account-based, stateless cryptocurrencies on top of any VC scheme [CPZ18]. In contrast, previous approaches were based on *authenticated dictionaries (ADs)* [RMCI17, But17], for which efficient constructions with static update keys are not known. In other words, these AD-based approaches used *dynamic update hints* uph_j consisting of the proof for position j. This complicated their design, requiring user i to ask a *proof-serving node* for user j's proof in order to create a transaction sending money to j.

Trusted Setup. To support up to n users, public parameters $(\mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_i)_{i\in[0,n)}) \leftarrow \mathsf{VC.KeyGen}(1^\lambda, n)$ are generated via a *trusted setup*, which can be decentralized using MPC protocols [BGM17]. Miners need to store all $O(n)$ update keys to propose blocks and to validate blocks (which we fix in Section 4.2.2). The prk is only needed for *proof-serving nodes* (see Section 4.3.2).

The (Authenticated) State. The state is a vector $\mathbf{v} = (v_i)_{i\in[0,n)}$ of size n that maps user i to $v_i = (\mathsf{addr}_i|\mathsf{bal}_i) \in \mathbb{Z}_p$, where bal_i is her balance and addr_i is her *address*, which we define later. (We discuss including transaction counters for preventing replay attacks in Section 4.3.1.) Importantly, since $p \approx 2^{256}$, the first 224 bits of v_i are used for addr_i and the last 32 bits for bal_i. The genesis block's state is the all zeros vector with digest d_0 (e.g., in our aSVC, $d_0 = g^0$). Initially, each user i is *unregistered* and starts with a proof $\pi_{i,0}$ that their $v_i = 0$.

"Full" vs. "Traditional" Public Keys. User i's address is computed as $\mathsf{addr}_i = H(\mathsf{FPK}_i)$, where $\mathsf{FPK}_i = (i, \mathsf{upk}_i, \mathsf{tpk}_i)$ is her *full public key* and H is a collision-resistant hash function. Here, tpk_i denotes a *"traditional" public key* for a digital signature scheme, with corresponding secret key tsk_i used to authorize user i's transactions. To avoid confusion, we will clearly refer to public keys as either "full" or "traditional."

Registering via `INIT` Transactions. `INIT` transactions are used to *register* new users and assign them a unique, ever-increasing number from 1 to n. For this, each block t stores a *count of users registered so far* cnt_t. To register, a user generates a *traditional secret key* tsk with a corresponding *traditional public key* tpk. Then, she broadcasts an `INIT` transaction:

$$\mathsf{tx} = [\mathtt{INIT}, \mathsf{tpk}]$$

A miner working on block $t+1$ who receives tx, proceeds as follows.

1. He sets $i = \mathsf{cnt}_{t+1}$ and increments the count cnt_{t+1} of registered users,
2. He updates the VC via $d_{t+1} = \mathsf{VC.UpdateComm}(d_{t+1}, (\mathsf{addr}_i|0), i, \mathsf{upk}_i)$,
3. He incorporates tx in block $t+1$ as $\mathsf{tx}' = [\mathtt{INIT}, (i, \mathsf{upk}_i, \mathsf{tpk}_i)] = [\mathtt{INIT}, \mathsf{FPK}_i]$.

The full public key with upk_i is included so other users can correctly update their VC when they process tx'. (The index i is not necessary, since it can be computed from the block's cnt_{t+1} and the number of `INIT` transactions processed in the block so far.) Note that to compute $\mathsf{addr}_i = H(\mathsf{FPK}_i)$, the miner needs to have the correct upk_i which requires $O(n)$ storage. We discuss how to avoid this in Section 4.2.2.

Transferring Coins via `SPEND` Transactions. When transferring v coins to user j, user i (who has $v' \geq v$ coins) must first obtain $\mathsf{FPK}_j = (j, \mathsf{upk}_j, \mathsf{tpk}_j)$. This is similar to existing cryptocurrencies, except the (full) public key is now slightly larger. Then, user i broadcasts a `SPEND` transaction, signed with her tsk_i:

$$\mathsf{tx} = [\mathtt{SPEND}, t, \mathsf{FPK}_i, j, \mathsf{upk}_j, v, \pi_{i,t}, v']$$

A miner working on block $t+1$ processes this `SPEND` transaction as follows:

1. He checks that $v \le v'$ and verifies the proof $\pi_{i,t}$ that user i has v' coins via $\mathsf{VC.VerifyPos}(\mathsf{vrk}, d_t, (\mathsf{addr}_i|v'), i, \pi_{i,t})$. (If the miner receives another transaction from user i, it needs to carefully account for i's new $v' - v$ balance.)
2. He updates i's balance in block $t+1$ with $d_{t+1} = \mathsf{VC.UpdateComm}(d_{t+1}, -v, i, \mathsf{upk}_i)$, which only sets the lower order bits of v_i corresponding to bal_i, without touching the higher order bits for addr_i.
3. He does the same for j with $d_{t+1} = \mathsf{VC.UpdateComm}(d_{t+1}, +v, j, \mathsf{upk}_j)$.

Validating Blocks. Suppose a miner receives a new block $t+1$ with digest d_{t+1} that has b `SPEND` transactions:

$$\mathsf{tx} = [\texttt{SPEND}, t, \mathsf{FPK}_i, j, \mathsf{upk}_j, v, \pi_{i,t}, v']$$

To validate this block, the miner (who has d_t) proceeds in three steps (`INIT` transactions can be handled analogously):

Step 1: Check Balances. First, for each tx, he checks that $v \le v'$ and that user i has balance v' via $\mathsf{VC.VerifyPos}(\mathsf{vrk}, d_t, (\mathsf{addr}_i|v'), i, \pi_{i,t}) = T$. Since the sending user i might have multiple transactions in the block, the miner has to carefully keep track of each sending user's balance to ensure it never goes below zero.

Step 2: Check Digest. Second, he checks d_{t+1} has been computed correctly from d_t and from the new transactions in block $t+1$. Specifically, he sets $d' = d_t$ and for each tx, he computes $d' = \mathsf{VC.UpdateComm}(d', -v, i, \mathsf{upk}_i)$ and $d' = \mathsf{VC.UpdateComm}(d', +v, j, \mathsf{upk}_j)$. Then, he checks that $d' = d_{t+1}$.

Step 3: Update Proofs, If Any. If the miner lost the race to build block $t+1$, he can start mining block $t+2$ by "moving over" the `SPEND` transactions from his unmined block. For this, he updates all proofs in those `SPEND` transactions, so they are valid against the new digest d_{t+1}. Similarly, the miner must also "move over" all `INIT` transactions, since block $t+1$ might have registered new users.

User Proof Synchronization. Consider a user i who has processed the ledger up to time t and has digest d_t and proof $\pi_{i,t}$. Eventually, she receives a new block $t+1$ with digest d_{t+1} and needs to update her proof so it verifies against d_{t+1}. Initially, she sets $\pi_{i,t+1} = \pi_{i,t}$. For each $[\texttt{INIT}, \mathsf{FPK}_j]$ transaction, she updates her proof $\pi_{i,t+1} = \mathsf{VC.UpdateProof}(\pi_{i,t+1}, (H(\mathsf{FPK}_j)|0), i, j, \mathsf{upk}_i, \mathsf{upk}_j)$. For each $[\texttt{SPEND}, t, \mathsf{FPK}_j, k, \mathsf{upk}_k, v, \pi_{j,t}, v']$, she updates her proof twice: $\pi_{i,t+1} = \mathsf{VC.UpdateProof}(\pi_{i,t+1}, -v, i, j, \mathsf{upk}_i, \mathsf{upk}_j)$ and $\pi_{i,t+1} = \mathsf{VC.UpdateProof}(\pi_{i,t+1}, +v, i, k, \mathsf{upk}_i, \mathsf{upk}_k)$. We stress that users can safely be offline and miss new blocks. Eventually, when a user comes back online, she downloads the missed blocks, updates her proof and is ready to transact.

4.2 Efficient Stateless Cryptocurrencies from aSVCs

In this subsection, we explain how replacing the Edrax VC with our aSVC from Section 3.3 results in a more efficient stateless cryptocurrency (see Table 1). Then, we address a denial of service attack on user registrations in Edrax.

4.2.1 Smaller, Faster, Aggregatable Proofs

Our aSVC enables miners to aggregate all b proofs in a block of b transactions into a single, constant-sized proof. This drastically reduces Edrax's per-block proof overhead from $O(b \log n)$ group elements to just one group element. Unfortunately, the b update keys cannot be aggregated, but we still reduce their overhead from $O(b \log n)$ to b group elements per block (see Section 4.2.3). Our smaller proofs are also faster to update, taking $O(1)$ time rather than $O(\log n)$. While verifying an aggregated proof in our aSVC is $O(b \log^2 b)$ time, which is asymptotically slower than the $O(b)$ time for verifying b individual ones, it is still *concretely* faster as it only requires two, rather than $O(b)$, cryptographic pairings. This makes validating new blocks much faster in practice.

4.2.2 Reducing Miner Storage Using Verifiable Update Keys

We stress that miners must validate update keys before using them to update a digest. Otherwise, they risk corrupting that digest, which results in a denial of service. Edrax miners sidestep this problem by simply storing all $O(n)$ update keys. Alternatively, Edrax proposes outsourcing update keys to an untrusted third party via a static Merkle tree. Unfortunately, this would either require interaction *during block proposal and block validation* or would double the update key size. For example, miners would need to fetch the correct update key and/or its Merkle proof to process a `SPEND` transaction. Our implicitly-verifiable update keys avoid these pitfalls, since miners can directly verify the update keys in a `SPEND` transaction via $\mathsf{VC.VerifyUPK}$. Furthermore, for `INIT` transactions, miners can fetch (in the background) a running window of the update keys needed for the next k registrations. By carefully upper-bounding the number of registrations expected in the near future, we can avoid interaction during the block proposal. This background fetching could be implemented in Edrax too, either with a small overhead via Merkle proofs or by making their update keys verifiable (which seems possible).

4.2.3 Smaller Update Keys

Although, in our aSVC, upk_i contains $a_i = g^{A(\tau)/(X-\omega^i)}$ and $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{X-\omega^i}}$, miners only need to include a_i in the block. This is because of two reasons. First, user i already has u_i to update her own proof after changes to her own balance. Second, no other user $j \neq i$ will need u_i to update her proof π_j. However, as hinted in Section 4.1, miners actually need u_i when only a subset of i's pending transactions get included in block t. In this case, the excluded transactions must have their proofs updated using u_i so they can be included in block $t+1$. Fortunately, this is not a problem, since miners always receive u_i with user i's transactions. The key observation is that they do not have to include u_i in the mined block, since users do not need it.

4.2.4 Addressing DoS Attacks on User Registrations.

Unfortunately, the registration process based on `INIT` transactions is susceptible to Denial of Service (DoS) attacks: an attacker can simply send a large number of `INIT` transactions and quickly exhaust the free space in the vector $\mathbf{v}$. There are several ways to address this. First, one can use an aSVC from hidden-order groups, which supports an unbounded number of elements [CFG+20]. However, that would negatively impact performance. Second, as future work, one could develop and use unbounded, authenticated dictionaries *with scalable updates*. Third, one could simply use multiple bounded aSVCs together with cross-commitment proof aggregation, which our aSVC supports [GRWZ20]. Lastly, one can add a cost to user registrations via a new `INITSPEND` transaction that registers a user j by having user i send her some coins:

$$[\texttt{INITSPEND}, t, \mathsf{FPK}_i, \mathsf{tpk}, v, \pi_{i,t}, v'],\ \text{where } 0 < v \le v'$$

Miners processing this transaction would first register a new user j with traditional public key tpk and then transfer her v coins. We stress that this is how existing cryptocurrencies operate anyway: in order to join, one has to be transferred some coins from existing users. Lastly, we can ensure that each tpk is only registered once by including in each `INIT/INITSPEND` transaction a non-membership proof for tpk in a Merkle prefix tree of all TPKs. We leave a careful exploration of this to future work.

Finally, miners (and only miners) will be allowed to create a *single* $[\texttt{INIT}, \mathsf{FPK}_i]$ transaction per block to register themselves. This has the advantage of letting new miners join, without "permission" from other miners or users, while severely limiting DoS attacks, since malicious miners can only register a new user per block. Furthermore, transaction fees and/or additional proof-of-work can also severely limit the frequency of `INITSPEND` transactions.

4.2.5 Minting Coins and Transaction Fees.

Support for minting new coins can be added with a new `MINT` transaction type:

$$\mathsf{tx} = [\texttt{MINT}, i, \mathsf{upk}_i, v]$$

Here, i is the miner's user account and v is the amount of newly minted coins. (Note that miners must register as users using `INIT` transactions if they are to receive block rewards.) To support transaction fees, we can extend the `SPEND` transaction format to include a fee, which is then added to the miner's block reward specified in the `MINT` transaction.

4.3 Discussion

4.3.1 Making Room for Transaction Counters

As mentioned in Section 2.3, to prevent transaction replay attacks, account-based stateless cryptocurrencies such as Edrax should actually map a user i to $v_i = (\mathsf{addr}_i | c_i | \mathsf{bal}_i)$, where c_i is her *transaction counter*. This change is trivial, but does leave less space in v_i for addr_i, depending on how many bits are needed for c_i and bal_i. (Recall that $v_i \in \mathbb{Z}_p$ typically has ≈ 256 bits.) To address this, we propose using one aSVC for mapping i to addr_i and another aSVC for mapping i to $(c_i | \mathsf{bal}_i)$. Our key observation is that if the two aSVCs use different n-SDH parameters (e.g., (g^{τ^i})'s and (h^{τ^i})'s, such that $\log_g h$ is unknown), then we could aggregate commitments, proofs and update keys so as to introduce zero computational and communication overhead in our stateless cryptocurrency. Security of this scheme could be argued similar to security of perfectly hiding KZG commitments [KZG10], which commit to $\phi(X)$ as $g^{\phi(\tau)} h^{r(\tau)}$ in an analogous fashion. We leave investigating the details of this scheme to future work.

4.3.2 Overhead of Synchronizing Proofs

In a stateless cryptocurrency, users need to keep their proofs updated w.r.t. the latest block. For example, in our scheme, each user spends $O(b \cdot \Delta t)$ time updating her proof, if there are Δt new blocks of b transactions each. Fortunately, when the underlying VC scheme supports precomputing all n proofs fast [Tom20], this overhead can be shifted to untrusted third parties called *proof-serving nodes* [CPZ18]. Specifically, a proof-serving node would have access to the proving key prk and periodically compute all proofs for all n users. Then, any user with an out-of-sync proof could ask a node for their proof and then manually update it, should it be slightly out of date with the latest block. Proof-serving nodes save users a significant amount of proof update work, which is important for users running on constrained devices such as mobile phones.

5 Conclusion

In this paper, we formalized a new cryptographic primitive called an *aggregatable subvector commitment (aSVC)* that supports aggregating and updating proofs (and commitments) using only constant-sized, static auxiliary information referred to as an "update key." We constructed an efficient aSVC from KZG commitments to Lagrange polynomials which, compared to other pairing-based schemes, can precompute, aggregate and update proofs efficiently and, compared to schemes from hidden-order groups, has smaller proofs and should perform better in practice. Lastly, we continued the study of stateless validation initiated by Chepurnoy et al., improving block validation time and block size, while addressing attacks and limitations. We hope our work will ignite further research into stateless validation for payments and smart contracts and lead to improvements both at the theoretical and practical level.

Acknowledgements

The authors want to thank Madars Virza for pointing out the Lagrange-based approach to VCs and the DFT technique for computing all KZG commitments to Lagrange polynomials. We also thank Leonid Reyzin and Dimitris Kolonelos for corrections and productive conversations that helped improve this paper.

References

BB08. Dan Boneh and Xavier Boyen. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *Journal of Cryptology*, Apr 2008.

BBF18. Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. Cryptology ePrint Archive, Report 2018/1188, 2018. `https://eprint.iacr.org/2018/1188`.

BBF19. Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In *CRYPTO'19*, 2019.

BdM94. 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.

BGM17. Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model, 2017. `https://eprint.iacr.org/2017/1050`.

BMRS20. Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized Cryptocurrency at Scale, 2020. `https://eprint.iacr.org/2020/352`.

BSCTV14. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable Zero Knowledge via Cycles of Elliptic Curves. In *CRYPTO'14*, 2014.

BT04. J. Berrut and L. Trefethen. Barycentric Lagrange Interpolation. *SIAM Review*, 46(3):501–517, 2004.

But17. Vitalik Buterin. The stateless client concept. ethresear.ch, 2017. `https://ethresear.ch/t/the-stateless-client-concept/172`.

But20. Vitalik Buterin. Using polynomial commitments to replace state roots. `https://ethresear.ch/t/using-polynomial-commitments-to-replace-state-roots/7095`, 2020.

CDHK15. Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and Modular Anonymous Credentials: Definitions and Practical Constructions. In *ASIACRYPT'15*, 2015.

CF13. Dario Catalano and Dario Fiore. Vector Commitments and Their Applications. In *PKC'13*, 2013.

CFG$^+$20. Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Vector Commitment Techniques and Applications to Verifiable Decentralized Storage, 2020. `https://eprint.iacr.org/2020/149`.

CFM08. Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-Knowledge Sets with Short Proofs. In *EUROCRYPT'08*, 2008.

CLRS09. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

Com16. Ethereum Community. Sharding FAQ. `https://github.com/ethereum/wiki/wiki/Sharding-FAQ`, 2016.

CPZ18. Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. Edrax: A Cryptocurrency with Stateless Transaction Validation, 2018. `https://eprint.iacr.org/2018/968`.

Dry19. Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set, 2019. `https://eprint.iacr.org/2019/611`.

FK20. Dankrad Feist and Dmitry Khovratovich. Fast amortized Kate proofs, 2020. `https://github.com/khovratovich/Kate`.

GRWZ20. Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating Proofs for Multiple Vector Commitments, 2020. `https://eprint.iacr.org/2020/419`.

Jou00. Antoine Joux. A One Round Protocol for Tripartite Diffie–Hellman. In *Algorithmic Number Theory*, 2000.

KJG$^+$18. E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *IEEE S&P'18*, May 2018.

KR13. Markulf Kohlweiss and Alfredo Rial. Optimally Private Access Control. In *ACM WPES'13*, 2013.

KZG10. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT'10*, 2010.

LLX07. Jiangtao Li, Ninghui Li, and Rui Xue. Universal Accumulators with Efficient Nonmembership Proofs. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 253–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

LM19. Russell W. F. Lai and Giulio Malavolta. Subvector Commitments with Application to Succinct Arguments. In *CRYPTO'19*, 2019.

LY10. Benoît Libert and Moti Yung. Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs. In *TCC'10*, 2010.

Mil12. 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`.

MVO91. Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. In *ACM STOC*, 1991.

Nak08. Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2008.

PST13. Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of Correct Computation. In *TCC'13*, 2013.

RMCI17. Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies. In *FC'17*, 2017.

Sha81. Adi Shamir. On the generation of cryptographically strong pseudo-random sequences. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming*, pages 544–550, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

STS99. Tomas Sander and Amnon Ta-Shma. Auditable, Anonymous Electronic Cash. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 555–572, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

STSY01. Tomas Sander, Amnon Ta-Shma, and Moti Yung. Blind, Auditable Membership Proofs. In Yair Frankel, editor, *Financial Cryptography*, pages 53–71, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

TCZ+20. 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.

Tod16. Peter Todd. Making UTXO set growth irrelevant with low-latency delayed TXO commitments, 2016. `https://petertodd.org/2016/delayed-txo-commitments`.

Tom20. 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.

Vir17. Madars Virza. *On Deploying Succinct Zero-Knowledge Proofs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2017.

vzGG13a. Joachim von zur Gathen and Jurgen Gerhard. Fast Multiplication. In *Modern Computer Algebra*, chapter 8, pages 221–254. Cambridge University Press, 3rd edition, 2013.

vzGG13b. 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.

vzGG13c. Joachim von zur Gathen and Jurgen Gerhard. Newton iteration. In *Modern Computer Algebra*, chapter 9, pages 257–292. Cambridge University Press, 3rd edition, 2013.

Wik19. Wikipedia contributors. Partial fraction decomposition — Wikipedia, the free encyclopedia, 2019. [Online; accessed 11-April-2020].

Woo. Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. `http://gavwood.com/paper.pdf`.

A Closed-form Formula for Evaluating the Derivative of $X^n - 1$ at Roots of Unity

Let $A(X) = X^n - 1$ and recall that $\mathcal{L}_i(X) = \frac{A(X)}{A'(\omega^i)\cdot(X-\omega^i)}$ (see Section 3.4). Let $A'(X)$ be the derivative of $X^n - 1$ and let $g(x) = A(X)/(X - \omega^i)$.

First, note that $A'(\omega^i) = g(\omega^i)$. Second, by carrying out the division of $X^n - 1$ by $(X - \omega^i)$, one can verify that:

$$g(x) = (\omega^i)^0 X^{n-1} + (\omega^i)^1 X^{n-2} + (\omega^i)^2 X^{n-3} + \cdots + (\omega^i)^{n-2} X^1 + (\omega^i)^{n-1} X^0 \tag{13}$$

Third, evaluating $A'(X)$ at $X = \omega^i$ gives:

$$A'(\omega^i) = g(\omega^i) = (\omega^i)^0 \omega^{i(n-1)} + (\omega^i)^1 \omega^{i(n-2)} + (\omega^i)^2 \omega^{i(n-3)} + \cdots + (\omega^i)^{n-2} \omega^{i\cdot 1} + (\omega^i)^{n-1} \omega^{i\cdot 0} \tag{14}$$

$$= n\omega^{i(n-1)} = n(\omega^{i\cdot n - i}) = n\omega^{-i} \tag{15}$$

B Computing all $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{\tau-\omega^i}}$ in $O(n\log n)$ time from g^{τ^i}'s

In Section 3.4.4, we argued all u_i's can be computed in $O(n \log n)$ time. Here, we prove correctness of the formula for the $q_i(X)$'s from Equation (11). As an example, let us look at the quotient $q_1(X)$ obtained when dividing $\psi_1(X)$ by $X - \omega^1$, assuming $n = 8$:

$$
\begin{array}{r|rrrrrrrr}
 & X^6 & +2\omega X^5 & +3\omega^2 X^4 & +4\omega^3 X^3 & +5\omega^4 X^2 & +6\omega^5 X & +7\omega^6 & \\
\hline
X - \omega & X^7 & +\omega X^6 & +\omega^2 X^5 & +\omega^3 X^4 & +\omega^4 X^3 & +\omega^5 X^2 & +\omega^6 X & +\omega^7 \\
 & -X^7 & +\omega X^6 & & & & & & \\
\cline{2-3}
 & & 2\omega X^6 & +\omega^2 X^5 & & & & & \\
 & & -2\omega X^6 & +2\omega^2 X^5 & & & & & \\
\cline{3-4}
 & & & 3\omega^2 X^5 & +\omega^3 X^4 & & & & \\
 & & & -3\omega^2 X^5 & +3\omega^3 X^4 & & & & \\
\cline{4-5}
 & & & & 4\omega^3 X^4 & +\omega^4 X^3 & & & \\
 & & & & -4\omega^3 X^4 & +4\omega^4 X^3 & & & \\
\cline{5-6}
 & & & & & 5\omega^4 X^3 & +\omega^5 X^2 & & \\
 & & & & & -5\omega^4 X^3 & +5\omega^5 X^2 & & \\
\cline{6-7}
 & & & & & & 6\omega^5 X^2 & +\omega^6 X & \\
 & & & & & & -6\omega^5 X^2 & +6\omega^6 X & \\
\cline{7-8}
 & & & & & & & 7\omega^6 X & +\omega^7 \\
 & & & & & & & -7\omega^6 X & +7\omega^7 \\
\cline{8-9}
 & & & & & & & & 8\omega^7
\end{array}
$$

In general, we want to show that:

$$q_i(X) = \sum_{j\in[0,n-2]} (j+1)(\omega^i)^j X^{(n-2)-j}, \forall i \in [0, n) \tag{16}$$

We do this by showing that the polynomial remainder theorem holds; i.e., $\psi_i(X) = q_i(X)(X - \omega^i) + \psi_i(\omega_i)$:

$$q_i(X)(X - \omega^i) + \psi_i(\omega_i) = \tag{17}$$

$$= q_i(X)(X - \omega^i) + n\omega^{-i} \tag{18}$$

$$= n\omega^{-i} + (X - \omega^i) \sum_{j\in[0,n-2]} (j+1)(\omega^i)^j X^{(n-2)-j} \tag{19}$$

$$= n\omega^{-i} + X \cdot \sum_{j\in[0,n-2]} (j+1)(\omega^i)^j X^{(n-2)-j} - \omega^i \cdot \sum_{j\in[0,n-2]} (j+1)(\omega^i)^j X^{(n-2)-j} \tag{20}$$

$$= n\omega^{-i} + \sum_{j\in[0,n-2]} (j+1)(\omega^i)^j X^{(n-1)-j} - \sum_{j\in[0,n-2]} (j+1)(\omega^i)^{j+1} X^{(n-2)-j} \tag{21}$$

$$= n\omega^{-i} + \sum_{j\in[0,n-2]} (j+1)\omega^{ij} X^{(n-1)-j} - \sum_{j\in[1,n-1]} j\omega^{ij} X^{(n-2)-(j-1)} \tag{22}$$

$$= n\omega^{-i} + \sum_{j\in[0,n-2]} (j+1)\omega^{ij} X^{(n-1)-j} - \sum_{j\in[1,n-1]} j\omega^{ij} X^{(n-1)-j} \tag{23}$$

$$= n\omega^{-i} + \left(X^{n-1} + \sum_{j\in[1,n-2]} (j+1)\omega^{ij} X^{(n-1)-j}\right) - \left((n-1)\omega^{i(n-1)} X^0 + \sum_{j\in[1,n-2]} j\omega^{ij} X^{(n-1)-j}\right) \tag{24}$$

$$= \left(X^{n-1} + \sum_{j\in[1,n-2]} (j+1)\omega^{ij} X^{(n-1)-j} - \sum_{j\in[1,n-2]} j\omega^{ij} X^{(n-1)-j}\right) - \left((n-1)\omega^{i(n-1)} X^0 - n\omega^{-i}\right) \tag{25}$$

$$= \left(X^{n-1} + \sum_{j\in[1,n-2]} \omega^{ij} X^{(n-1)-j}\right) - \left((n-1)\omega^{-i} X^0 - n\omega^{-i} X^0\right) \tag{26}$$

$$= \sum_{j\in[0,n-2]} \omega^{ij} X^{(n-1)-j} + \omega^{-i} X^0 \tag{27}$$

$$= \sum_{j\in[0,n-2]} \omega^{ij} X^{(n-1)-j} + \omega^{in-i} X^0 \tag{28}$$

$$= \sum_{j\in[0,n-2]} \omega^{ij} X^{(n-1)-j} + \omega^{i(n-1)} X^0 \tag{29}$$

$$= \sum_{j\in[0,n)} \omega^{ij} X^{(n-1)-j} \tag{30}$$

$$= \psi_i(X) \tag{31}$$

C Security Proofs

C.1 KZG Batch Opening Binding (Re)definition

We strengthen the *batch opening binding definition* of KZG [KZG10, Sec. 3.4, pg. 9] and prove KZG still satisfies it.

Definition 8 (Batch Opening Binding). $\forall$ *adversaries* $\mathcal{A}$ *running in time* $\mathsf{poly}(\lambda)$*:*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{KZG.Setup}(1^\lambda, n), \\ c, I, J, v_I(X), v_J(X), \pi_I, \pi_J \leftarrow \mathcal{A}(\mathsf{pp}, 1^\lambda) : \\ \mathsf{KZG.VerifyEvalBatch}(\mathsf{pp}, c, I, \pi_I, v_I(X)) = T \wedge \\ \mathsf{KZG.VerifyEvalBatch}(\mathsf{pp}, c, J, \pi_J, v_J(X)) = T \wedge \\ \exists k \in I \cap J,\ \textit{such that } v_I(k) \neq v_J(k) \end{array}\right] \leq \mathsf{negl}(\lambda) \tag{32}$$

Suppose an adversary breaks the definition. Let $A_I(X) = \prod_{i\in I}(X - i)$. Then, the following holds:

$$e(c, g) = e(\pi_I, g^{A_I(\tau)}) e(g^{v_I(\tau)}, g) \tag{33}$$

$$e(c, g) = e(\pi_J, g^{A_J(\tau)}) e(g^{v_J(\tau)}, g) \tag{34}$$

Divide the top equation by the bottom one to get:

$$\mathbf{1}_T = \frac{e(g^{v_I(\tau)}, g)}{e(g^{v_J(\tau)}, g)} \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \Leftrightarrow \tag{35}$$

$$\mathbf{1}_T = e(g^{v_I(\tau)-v_J(\tau)}, g) \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \Leftrightarrow \tag{36}$$

$$e(g^{v_J(\tau)-v_I(\tau)}, g) = \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \tag{37}$$

Let $v_k = v_I(k)$ and $v'_k = v_J(k)$. We can rewrite $v_I(X)$ using the polynomial remainder theorem as $v_I(X) = q_I(X)(X - k) + v_k$. Similarly, $v_J(X) = q_J(X)(X-k) + v'_k$.

$$e(g^{q_J(\tau)(\tau-k)+v'_k - q_I(\tau)(\tau-k)-v_k}, g) = \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \Leftrightarrow \tag{38}$$

$$e(g^{(\tau-k)(q_J(\tau)-q_I(\tau))+v'_k-v_k}, g) = \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \Leftrightarrow \tag{39}$$

$$e(g^{(\tau-k)(q_J(\tau)-q_I(\tau))}, g)e(g^{v'_k-v_k}, g) = \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \Leftrightarrow \tag{40}$$

$$e(g^{q_J(\tau)-q_I(\tau)}, g)^{\tau-k} e(g,g)^{v'_k-v_k} = \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \tag{41}$$

Factor out $(X-k)$ in $A_I(X)$ to get $A_I(X) = a_I(X)(\tau - k)$. Similarly, $A_J(X) = a_J(X)(\tau-k)$.

$$e(g^{q_J(\tau)-q_I(\tau)}, g)^{\tau-k} e(g,g)^{v'_k-v_k} = \left(\frac{e(\pi_I, g^{a_I(\tau)})}{e(\pi_J, g^{a_J(\tau)})}\right)^{\tau-k} \Leftrightarrow \tag{42}$$

$$e(g^{q_J(\tau)-q_I(\tau)}, g)e(g,g)^{\frac{v'_k-v_k}{\tau-k}} = \frac{e(\pi_I, g^{a_I(\tau)})}{e(\pi_J, g^{a_J(\tau)})} \Leftrightarrow \tag{43}$$

$$e(g,g)^{\frac{v'_k-v_k}{\tau-k}} = \frac{e(\pi_I, g^{a_I(\tau)})}{e(\pi_J, g^{a_J(\tau)})e(g^{q_J(\tau)-q_I(\tau)}, g)} \Leftrightarrow \tag{44}$$

$$e(g,g)^{\frac{1}{\tau-k}} = \left(\frac{e(\pi_I, g^{a_I(\tau)})}{e(\pi_J, g^{a_J(\tau)})e(g^{q_J(\tau)-q_I(\tau)}, g)}\right)^{\frac{1}{v'_k-v_k}} \tag{45}$$

Since the commitments to $a_I(X), a_J(X), q_I(X), q_J(X)$ can be easily reconstructed from $v_I(X), v_J(X), I$ and J, and since $v'_k \neq v_k$, this constitutes a direct break of n-SBDH.

C.2 Update Key Uniqueness

We prove that our aSVC scheme from Section 3 has *Update Key Uniqueness* as defined in Definition 6. Let a be the commitment to $A(X) = X^n - 1$ from the verification key vrk. Suppose an adversary outputs two update keys $\mathsf{upk}_i = (a_i, u_i)$ and $\mathsf{upk}'_i = (a'_i, u'_i)$ at position i that both pass $\mathsf{VC.VerifyUPK}$ but $\mathsf{upk}_i \neq \mathsf{upk}'_i$. Then, it must be the case that either $a_i \neq a'_i$ or that $u_i \neq u'_i$.

$a_i \neq a'_i$ Case: Since both update keys pass verification, the following pairing equations hold:

$$e(a_i, g^\tau / g^{\omega^i}) = e(a, g) \tag{46}$$

$$e(a'_i, g^\tau / g^{\omega^i}) = e(a, g) \tag{47}$$

Thus, it follows that:

$$e(a_i, g^\tau / g^{\omega^i}) = e(a'_i, g^\tau / g^{\omega^i}) \Leftrightarrow \tag{48}$$

$$e(a_i, g) = e(a'_i, g) \Leftrightarrow \tag{49}$$

$$a_i = a'_i \tag{50}$$

Contradiction.

$u_i \neq u'_i$ Case: Let $A'(X)$ denote the derivative of $A(X) = X^n - 1$. Let $\ell_i = a_i^{1/A'(\omega^i)} = g^{\mathcal{L}_i(\tau)}$. Since both update keys pass verification, the following pairing equations hold:

$$e(\ell_i / g^1, g) = e(u_i, g^\tau / g^{\omega^i}) \tag{51}$$

$$e(\ell_i / g^1, g) = e(u'_i, g^\tau / g^{\omega^i}) \tag{52}$$

Thus, it follows that:

$$e(u_i, g^\tau / g^{\omega^i}) = e(u'_i, g^\tau / g^{\omega^i}) \tag{53}$$

$$e(u_i, g) = e(u'_i, g) \Leftrightarrow \tag{54}$$

$$u_i = u'_i \tag{55}$$

Contradiction.

D Complexities of Pairing-based VCs in Table 2

We survey each *pairing-based* VC scheme from Table 2 and explain its complexities. In Appendix E, we do the same for VCs based on hidden-order groups. Despite our best efforts to understand the complexities of each scheme, we recognize there could be better upper bounds for some of them.

D.1 Complexities of CDH-based [LM19]

This scheme was originally proposed by Catalano and Fiore [CF13] and extended by Lai and Malavolta to support subvector proofs [LM19].

Public Parameters. The proving key is $\mathsf{prk} = (h_{i,j})_{i,j\in[0,n)}$ and is $O(n^2)$ sized. Here, $h_{i,j} = g^{z_i \cdot z_j}$ when $i \neq j$ and $h_{i,i} = h_i = g^{z_i}$, with each $z_i \in \mathbb{Z}_p$ picked uniformly at random. The verification key is $\mathsf{vrk} = (h_i)_{i\in[0,n)}$ and is $O(n)$-sized. The ith update key is $\mathsf{upk}_i = (h_i, (h_{i,j})_{j\in[0,n)})$. Note that $h_{i,j} = h_j^{z_i} = h_i^{z_j}$.

Commitment. A commitment is $c = \prod_{i\in[0,n)} h_i^{v_i}$ and can be computed with $O(n)$ exponentiations. If any vector element v_j changes to $v_j + \delta$, the commitment can be updated in $O(1)$ time using h_j from upk_j as $c' = c \cdot (h_j)^\delta$.

Proofs for a v_i. A proof for v_i is:

$$\pi_i = \prod_{j\in[0,n)\setminus\{i\}} h_{i,j}^{v_j} = \left(\prod_{j\in[0,n)\setminus\{i\}} h_j^{v_j}\right)^{z_i} \tag{56}$$

The proof is $O(1)$-sized and can be computed from the $h_{i,j}$'s in the prk with $O(n)$ exponentiations. It can be verified in $O(1)$ time using h_i from the vrk as:

$$e(C/h_i^{v_i}, h_i) = e(\pi_i, g) \tag{57}$$

If any vector element $v_j, j \neq i$ changes to $v_j + \delta$, the proof π_i can be updated in $O(1)$ time using $h_{i,j}$ from upk_j as $\pi_i' = \pi_i \cdot \left(h_{i,j}^\delta\right)$. This new π_i' will verify against the updated c' commitment defined earlier. If v_i changes to $v_i + \delta$, the proof π_i need not be updated.

Subvector Proofs for $(v_i)_{i\in I}$ A $O(1)$-sized subvector proof for $\mathbf{v}_I$ is:

$$\pi_I = \prod_{i\in I}\prod_{j\in[0,n)\setminus I} h_{i,j}^{v_j} = \prod_{i\in I}\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}\right)^{z_i} = \prod_{i\in I}\pi_i^* \tag{58}$$

As intuition, note that the inner product $\pi_i^* = \left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}\right)^{z_i}$ is very similar to a proof $\pi_i = \left(\prod_{j\in[0,n)\setminus\{i\}} h_j^{v_j}\right)^{z_i}$ for v_i. Let $b = |I|$. The proof can be computed from the $h_{i,j}$'s in the prk with $O(b(n-b))$ exponentiations (because each π_i^* can be computed in $O(n-b)$ exponentiations). A subvector proof π_I can be verified using $(h_i)_{i\in I}$ from vrk by checking in $O(b)$ time if:

$$e\left(c/\prod_{j\in I} h_j^{v_j}, \prod_{i\in I} h_i\right) = e(\pi_I, g) \Leftrightarrow \tag{59}$$

$$e\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}, \prod_{i\in I} g^{z_i}\right) = e\left(\prod_{i\in I}\prod_{j\in[0,n)\setminus I} h_{i,j}^{v_j}, g\right) \tag{60}$$

$$e\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}, g^{\sum_{i\in I} z_i}\right) = e\left(\prod_{i\in I}\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}\right)^{z_i}, g\right) \tag{61}$$

$$e\left(\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}\right)^{\sum_{i\in I} z_i}, g\right) = e\left(\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}\right)^{\sum_{i\in I} z_i}, g\right) \tag{62}$$

Aggregating Proofs and Precomputing All Proofs. Aggregating proofs is not discussed in [CF13, LM19], but it seems possible. Finally, precomputing all proofs efficiently is not discussed. Naively, it can be done inefficiently in $O(n^2)$ time.

D.2 Complexities of KZG [KZG10]

Kate, Zaverucha and Goldberg also discuss using their polynomial commitment scheme [KZG10] to commit to a sequence of messages, thus implicitly obtaining a VC scheme. Although they do not analyze its complexity in their paper, we do so here.

Public Parameters. The proving key is $\mathsf{prk} = (g^{\tau^i})_{i\in[0,n-1]}$ and is $O(n)$ sized. The verification key is $\mathsf{vrk} = (g, (g^{\tau^i})_{i\in b})$, where $b = |I|$ is the size of the largest subvector whose proof the verifier should be able to check, and is thus $O(b)$-sized. There is no support for updating commitments and proofs using update keys, although our work shows this is possible (see Section 3.3).

Commitment. A commitment is $c = g^{\phi(\tau)}$ where $\phi(X) = \sum_{i\in[0,n)} \mathcal{L}_i(X)v_i$ and can be computed with $O(n\log^2 n)$ field operations (see Section 2.1) and $O(n)$ exponentiations. Commitment updates are not discussed, but the scheme could be modified to support them (see Section 3.3).

Proofs for a $\boldsymbol{v_i}$. A proof for v_i is:

$$\pi_i = g^{\frac{\phi(\tau)-v_i}{\tau-i}} = g^{q_i(\tau)} \tag{63}$$

The proof is $O(1)$-sized and can be computed by dividing $\phi(X)$ by $(X - i)$ in $O(n)$ field operations, obtaining $q_i(X)$, and committing to $q_i(X)$ using the g^{τ^i}'s in the prk with $O(n)$ exponentiations. The proof can be verified in $O(1)$ time using g^τ from the vrk by computing two pairings:

$$e(c/g^{v_i}, g) = e(\pi_i, g^\tau/g^i) \tag{64}$$

Proof updates are not discussed, but the scheme could be modified to support them (see Section 3.3).

Subvector Proofs for $\boldsymbol{(v_i)_{i\in I}}$ A $O(1)$-sized subvector proof for $\mathbf{v}_I$ is:

$$\pi_I = g^{\frac{\phi(\tau)-R_I(\tau)}{A_I(\tau)}} = g^{q_I(\tau)} \tag{65}$$

Here, $R_I(X)$ of degree $\le b-1$ is interpolated in $O(b\log^2 b)$ field operations so that $R_I(i) = v_i, \forall i \in I$ (see Section 2.1). Also, $A_I(X) = \prod_{i\in I}(X - i)$ is computed in $O(b\log^2 b)$ field operations via a *subproduct tree* [vzGG13b]. The proof leverages the fact that dividing $\phi(X)$ by $A_I(X)$ gives quotient $q_I(X)$ and remainder $R_I(X)$. The quotient $q_I(X)$ can be obtained in $O(n\log n)$ field operations via a DFT-based division [vzGG13c]. Given g^{τ^i}'s from the prk, committing to $q_I(X)$ takes $O(n-b)$ exponentiations, since $\deg(q_I) = \deg(\phi) - \deg(A_I) \le (n-1) - b$. Thus, the overall subvector proving time is $O(n\log n + b\log^2 b)$.

To verify a subvector proof π_I, first, the verifier must recompute $R_I(X)$ and $A_I(X)$ in $O(b\log^2 b)$ field operations. Then, the verifier uses $(g^{\tau^i})_{i\in b}$ from the vrk to compute KZG commitments $g^{R_I(\tau)}$ and $g^{A_I(\tau)}$ in $O(b)$ exponentiations. Finally, the verifier checks using two pairings if:

$$e(c/g^{R_I(\tau)}, g) = e(\pi_I, g^{A_I(\tau)}) \tag{66}$$

Thus, the overall subvector proof verification time is $O(b\log^2 b)$ time.

Aggregating Proofs and Precomputing All Proofs. Aggregating proofs is not discussed, but the scheme can be modified to support them (see Section 3.4.1). Finally, precomputing all proofs efficiently is not discussed, but is possible (see Section 3.3). Naively, it can be done inefficiently in $O(n^2)$ time.

D.3 Complexity of CDHK [CDHK15]

In this scheme, we assume the vector $\mathbf{v} = [v_1, v_2, \ldots, v_n]$ is indexed from 1 to n. This scheme is similar to a KZG-based VC, except (1) it is randomized, (2) it computes proofs in a slightly different way and (3) it *willfully* prevents aggregation of proofs as a security feature.

Public Parameters. The proving key is $\mathsf{prk} = \left((g^{\tau^i})_{i\in[0,n+1]}, (g^{\mathcal{L}_i(\tau)})_{i\in[0,n]}, g^{P(\tau)}\right)$, where $P(X) = X\cdot\prod_{i\in[n]}(X-i)$ and is $O(n)$ sized. (Note that the Lagrange polynomials $\mathcal{L}_i(X) = \prod_{j\in[0,n],j\ne i}\frac{X-j}{i-j}$ are defined over the points $[0,n]$, not $[n]$.) The verification key is $\mathsf{vrk} = (g, (g^{\mathcal{L}_i(\tau)})_{i\in[n]}, (g^{\tau^i})_{i\in[0,b+1]})$, where $b = |I|$ is the size of the largest subvector whose proof the verifier should be able to check. Unfortunately, the verification key is $O(n)$-sized. There is no support for updating commitments and proofs using update keys, although adding it is possible via our techniques (see Section 3.3). As a result, we treat this scheme as if it used an $O(n)$-sized update key $\mathsf{upk}_i = \left(g^{\mathcal{L}_i(\tau)}, \left(g^{\frac{\mathcal{L}_j(\tau)}{\tau-i}}\right)_{j\in[n]}\right)$

Commitment. A commitment is $c = \prod_{i\in[n]} \left(g^{\mathcal{L}_i(\tau)}\right)^{v_i} \left(g^{P(\tau)}\right)^r = g^{\phi(\tau)+r\cdot P(\tau)}$ where $\phi(X) = \sum_{i\in[0,n]} \mathcal{L}_i(X)v_i$, with $v_0 = 0$. To compute the commitment, $\phi(X)$ must first be interpolated using $O(n\log^2 n)$ field operations. Then, c can be computed with $O(n)$ exponentiations, given the Lagrange commitments and $g^{P(\tau)}$ from prk. Commitment updates are not discussed, but they can be trivially implemented by setting $\mathsf{upk}_i = g^{\mathcal{L}_i(\tau)}$ and having $c' = c \cdot \left(g^{\mathcal{L}_j(\tau)}\right)^\delta$ be the new commitment after a change δ to v_j. We reflect this in Table 2.

Proofs for a v_i. A proof for v_i is:

$$\pi_i = g^{\frac{(\phi(\tau)+r\cdot P(\tau))-v_i\mathcal{L}_i(\tau)}{\tau-i}} = g^{q_i(\tau)} \tag{67}$$

The proof is $O(1)$-sized and can be computed by dividing $\phi(X)+r\cdot P(X)-v_i\mathcal{L}_i(X)$ by $(X-i)$ in $O(n)$ field operations, obtaining $q_i(X)$, and committing to $q_i(X)$ using the g^{τ^i}'s in the prk with $O(n)$ exponentiations. The proof can be verified in $O(1)$ time using $g^{\mathcal{L}_i(\tau)}$ from the vrk by computing two pairings:

$$e\left(c/\left(g^{\mathcal{L}_i(\tau)}\right)^{v_i}, g\right) = e(\pi_i, g^\tau/g^i) \tag{68}$$

Proof updates are not discussed, but the scheme could be modified to support them (see Section 3.3).

Subvector Proofs for $(v_i)_{i\in I}$ A $O(1)$-sized subvector proof for $\mathbf{v}_I$ is:

$$\pi_I = g^{\frac{\phi(\tau)+r\cdot P(\tau)-R_I(\tau)}{A_I(\tau)}} = g^{q_I(\tau)} \tag{69}$$

Here, $R_I(X)$ is defined so that $R_I(i) = v_i, \forall i \in I$ **and** $R_I(i) = 0, \forall i \in [0,n] \setminus I$. (In particular, this means $R_I(0) = 0$.) Interpolating $R_I(X)$ takes $O(n\log^2 n)$ field operations. Also, $A_I(X) = X\prod_{i\in I}(X-i)$ is computed in $O(b\log^2 b)$ field operations via a *subproduct tree* [vzGG13b]. Given g^{τ^i}'s from the prk, committing to $q_I(X)$ takes $O(n-b)$ exponentiations (because $\deg(q_I) = \deg(\phi) - \deg(A_I) \le n-(b+1)$). Thus, the overall subvector proving time is $O(n\log^2 n)$.

To verify a subvector proof π_I, first, the verifier recomputes the commitment to $g^{R_I(\tau)} = \sum_{i\in I}\left(g^{\mathcal{L}_i(\tau)}\right)^{v_i}$ using $O(b)$ exponentiations. (Recall that $\mathcal{L}_i(X)$ is defined over $[0,n]$ and has its KZG commitment in the vrk.) Then, he computes $A_I(X)$ in $O(b\log^2 b)$ field operations using a *subproduct tree* [vzGG13b]. Then, the verifier uses $(g^{\tau^i})_{i\in[0,b+1]}$ from the vrk to compute a KZG commitment to $g^{A_I(\tau)}$ in $O(b)$ exponentiations. Finally, the verifier checks if:

$$e(c/g^{R_I(\tau)}, g) = e(\pi_I, g^{A_I(\tau)}) \tag{70}$$

Thus, the overall subvector proof verification time is $O(b\log^2 b)$.

Aggregating Proofs and Precomputing All Proofs. Aggregating proofs is *willfully* prevented by this scheme, as a security feature. Finally, precomputing all proofs efficiently is not discussed, but it can be done inefficiently in $O(n^2)$ time. Importantly, because the proofs are slightly different from KZG, they are not amenable to known techniques for precomputing all n proofs in $O(n\log n)$ time [FK20].

D.4 Complexities of CPZ [CPZ18]

Since the Edrax paper clearly summarizes its performance, we refer the reader to [CPZ18, Table 1], with one exception discussed below.

Aggregating Proofs and Precomputing All Proofs. Aggregating proofs is not discussed and it is unclear if the scheme can be modified to support it. Precomputing all proofs efficiently is not discussed either to the best of our knowledge, but could be possible.

D.5 Complexities of TCZ [TCZ+20, Tom20]

In their paper on scaling threshold cryptosystems, Tomescu et al. [TCZ+20] present a technique for computing n *logarithmic-sized* evaluation proofs for a KZG committed polynomial of degree t in $O(n\log t)$ time. Later on, Tomescu extends these results to obtain a full VC scheme [Tom20, Sec 9.2].

Public Parameters. The proving key is $\mathsf{prk} = ((g^{\tau^i})_{i\in[0,n-1]}, (g^{\mathcal{L}_i(\tau)})_{i\in[0,n)})$ and is $O(n)$ sized. Importantly, n is assumed to be a power of two, and $\mathcal{L}_i(X) = \prod_{j\in[0,n),j\neq i} \frac{X-\omega^j}{\omega^i-\omega^j}$ where ω is a primitive nth root of unity [vzGG13a]. The verification key is $\mathsf{vrk} = (g, (g^{\tau^{2^i}})_{i\in[\lfloor\log_2(n-1)\rfloor]}, (g^{\tau^i})_{i\in[b]})$, where $b = |I|$ is the size of the largest subvector whose proof the verifier should be able to check, and is thus $O(b)$-sized. The ith update key upk_i is the *authenticated multipoint evaluation tree (AMT)* of $\mathcal{L}_i(X)$ at all points $(\omega^i)_{i\in[0,n)}$ (see [TCZ+20, Sec III-B] and [Tom20, Ch 9]). This AMT will be $O(\log n)$-sized, consisting of a single path of non-zero quotient commitments leading to the evaluation of $\mathcal{L}_i(\omega^i)$ [Tom20, Sec 9.2.2], since all other evaluations $\mathcal{L}_i(\omega^j), j \neq i$ are zero.

Commitment. A commitment is $c = g^{\phi(\tau)}$ where $\phi(\omega^i) = v_i, \forall i \in [0,n)$. Note that $\phi(X)$ can be computed with $O(n\log n)$ field operations via an inverse Discrete Fourier Transform (DFT) [CLRS09, Ch 30.2]. Then, computing c requires $O(n)$ exponentiations. Commitment updates remain the same as in the KZG-based scheme from Appendix D.3: $c' = c \cdot \left(g^{\mathcal{L}_j(\tau)}\right)^{\delta}$, where δ is the change at position j in the vector and the Lagrange polynomial commitment can be obtained from upk_j.

Proofs for a v_i. A proof for v_i is:

$$\pi_i = (g^{q_w(\tau)})_{w\in[1+\lfloor\log(n-1)\rfloor]} \tag{71}$$

Here, each $q_w(X)$ is a quotient polynomial along the AMT tree path to $\phi(\omega^i)$. The proof is $O(\log n)$-sized and can be computed by "repeatedly" dividing $\phi(X)$ by accumulator polynomials of ever-decreasing sizes $n/2, \ldots, 4, 2, 1$ in $T(n) = O(n\log n) + T(n/2) = O(n\log n)$ field operations, and committing to each $q_w(X)$ using the g^{τ^i}'s in the prk with $T'(n) = O(n) + T'(n/2) = O(n)$ exponentiations. ("Repeatedly" dividing means we first divide $\phi(X)$ by a degree $n/2$ accumulator. Then, we take the remainder of this division and divide it by the degree $n/4$ accumulator. We then take this remainder and divide it by a degree $n/8$ accumulator. And so on. This ensures the remainder degrees always halve.) The proof can be verified in $O(\log n)$ time using the $g^{\tau^{2^i}}$'s from the vrk:

$$e(c/g^{v_i}, g) = \prod_{w\in[1+\lfloor\log(n-1)\rfloor]} e(g^{q_w(\tau)}, g^{a_w(\tau)}) \tag{72}$$

Here, the $a_w(X)$'s denote the accumulator polynomials along the AMT path to $\phi(\omega^i)$, which are always of the form $X^{2^i} - c$ for some constant c and some $i \in [0, \lfloor\log(n-1)\rfloor]$.

Proof Updates. If any vector element v_j changes to $v_j + \delta$, the proof π_i can be updated in $O(\log n)$ time. (It could be that $j = i$.) The idea is to consider the quotient commitments $g^{q_w(\tau)}$ along π_i's AMT path and the $g^{u_w(\tau)}$ commitments along upk_j's AMT path. For all locations w where the two paths intersect, the quotient commitments are combined in constant time as $g^{q'_w(\tau)} = g^{q_w(\tau)} \cdot \left(g^{u_w(\tau)}\right)^{\delta}$. Since there are at most $O(\log n)$ locations w to intersect in, this takes $O(\log n)$ exponentiations. This new π'_i with quotient commitments $g^{q'_w(\tau)}$ will verify against the updated c' commitment defined earlier.

Subvector Proofs for $(v_i)_{i\in I}$ This scheme uses the same subvector proof as the original KZG-based scheme in Appendix D.2. Thus, the subvector proving time is $O(n\log n + b\log^2 b)$ and the verification time is $O(b\log^2 b)$ time.

Aggregating Proofs and Precomputing All Proofs. Aggregating proofs is not discussed and it is unclear if the scheme can be modified to support it. Precomputing all *logarithmic-sized* proofs efficiently is possible via the AMT technique in $O(n\log n)$ time.

D.6 Complexity of Pointproofs [GRWZ20, LY10]

Gorbunov et al. [GRWZ20] enhance the VC by Libert and Yung [LY10] with the ability to aggregate multiple VC proofs into a subvector proof. Additionally, they also enable aggregation of subvector proofs across different vector commitments, which they show is useful for stateless smart contract validation in cryptocurrencies. In this scheme, we assume the vector $\mathbf{v} = [v_1, v_2, \ldots, v_n]$ is indexed from 1 to n.

Public Parameters. Their scheme works over Type III pairings $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Let g_1, g_2, g_T be generators of $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ respectively. The proving key $\mathsf{prk} = (g_1, (g_1^{\alpha^i})_{i\in[1,2n]\setminus\{n+1\}}, g_2, (g_2^{\alpha^i})_{i\in[1,n]}, g_T^{\alpha^{n+1}})$. Note that $g_1^{\alpha^{n+1}}$ is "missing" from the proving key, which is essential for security. The verification key $\mathsf{vrk} = ((g_2^{\alpha^i})_{i\in[1,n]}, g_T^{\alpha^{n+1}})$ is $O(n)$ -sized. The ith update key is $\mathsf{upk}_i = g_1^{\alpha^i}$. They only support updating commitments, but proofs could be made updatable at the cost of linear-sized update keys.

Commitment. A commitment is $c = \prod_{i\in[n]} \left(g_1^{\alpha^i}\right)^{v_i} = g_1^{\sum_{i\in[n]} v_i\alpha^i}$ and can be computed with $O(n)$ exponentiations. If any vector element v_j changes to $v_j + \delta$, the commitment can be updated in $O(1)$ time as $c' = c \cdot (\mathsf{upk}_j)^\delta = c \cdot (g_1^{\alpha^j})^\delta$.

Proofs for a v_i. A proof for v_i is obtained by re-committing to v so that v_i "lands" at position $n+1$ (i.e., has coefficient α^{n+1}) rather than position i (i.e., has coefficient α^i). Furthermore, this commitment will **not** contain v_i: it cannot, since that would require having $g_1^{\alpha^{n+1}}$. To get position i to $n+1$, we must "shift" it (and every other position) by $(n+1) - i$. Thus, the proof is:

$$\pi_i = g_1^{\sum_{j\in[n]\setminus\{i\}} v_j\alpha^{j+(n+1)-i}} \tag{73}$$

$$= g_1^{\sum_{j\in[n]\setminus\{i\}} v_j\alpha^j\alpha^{(n+1)-i}} \tag{74}$$

$$= \left(g_1^{\sum_{j\in[n]\setminus\{i\}} v_j\alpha^j}\right)^{\alpha^{(n+1)-i}} \tag{75}$$

$$= \left(\frac{g_1^{\sum_{j\in[n]} v_j\alpha^j}}{g_1^{v_i\alpha^i}}\right)^{\alpha^{(n+1)-i}} \tag{76}$$

$$= (c/g_1^{v_i\alpha^i})^{\alpha^{(n+1)-i}} \tag{77}$$

The proof is constant-sized and can be computed with $O(n)$ exponentiations. It can be verified in $O(1)$ time using $g_2^{\alpha^{(n+1)-i}}$ from vrk:

$$e(c, g_2^{\alpha^{(n+1)-i}}) = e(\pi_i, g_2)\left(g_T^{\alpha^{n+1}}\right)^{v_i} \tag{78}$$

Updating the proof is not discussed but can be done in $O(1)$ time, if the update keys are tweaked to be linear rather than constant-sized.

Subvector Proofs for $(v_i)_{i\in I}$ An $O(1)$-sized subvector proof for $\mathbf{v}_I$ is just a random linear combination of all proofs $\pi_i, \forall i \in I$. First, all b proofs π_i are computed in $O(bn)$ exponentiations as described above. Second, for each $i \in I$, $t_i = H(c, I, (v_i)_{i\in I})$ is computed using a random oracle $H : \{0,1\}^* \to \mathbb{Z}_p$. Third, the subvector proof π_I is computed as:

$$\pi_I = \prod_{i\in I} \pi_i^{t_i} \tag{79}$$

If computed this way, a subvector proof would take $O(bn)$ exponentiations. However, Gorbunov et al. observe that π_I can be computed with an $O(n)$-sized multi-exponentiation on a subset of the $2n$ generators $(g_1^{\alpha^i})_{i\in[0,2n]\setminus\{n+1\}}$. The exponents will be a combination of the messages and the t_i's (see [GRWZ20, Sec 4.1] for more details). However, they do not bound the time to compute these exponents, which appears to be $O(bn)$ field operations in the worst-case.

The subvector proof can be verified in $O(b)$ time using $(g_2^{\alpha^{(n+1)-i}})_{i\in I}$ from vrk as:

$$e\left(c, \prod_{i\in I}\left(g_2^{\alpha^{(n+1)-i}}\right)^{t_i}\right) = e(\pi_I, g_2)\left(g_T^{\alpha^{n+1}}\right)^{\sum_{i\in I} v_i t_i} \Leftrightarrow \tag{80}$$

$$e\left(c, g_2^{\sum_{i\in I} t_i\alpha^{(n+1)-i}}\right) = e\left(\prod_{i\in I}\pi_i^{t_i}, g_2\right) g_T^{\alpha^{n+1}\sum_{i\in I} v_i t_i} \Leftrightarrow \tag{81}$$

$$= e\left(\prod_{i\in I}\pi_i^{t_i}, g_2\right) e\left(g_1^{\alpha^{n+1}\sum_{i\in I} v_i t_i}, g_2\right) \Leftrightarrow \tag{82}$$

$$= e\left(\prod_{i\in I}\pi_i^{t_i} \cdot g_1^{\alpha^{n+1}\sum_{i\in I} v_i t_i}, g_2\right) \Leftrightarrow \tag{83}$$

$$= e\left(\prod_{i\in I}\pi_i^{t_i} \cdot \prod_{i\in I} g_1^{\alpha^{n+1} v_i t_i}, g_2\right) \Leftrightarrow \tag{84}$$

$$= e\left(\prod_{i\in I}\left(\pi_i \cdot g_1^{\alpha^{n+1} v_i}\right)^{t_i}, g_2\right) \tag{85}$$

Recall that $\pi_i = (c/g_1^{v_i \alpha^i})^{\alpha^{(n+1)-i}}$.

$$
\begin{aligned}
e\left(c, g_2^{\sum_{i\in I} t_i \alpha^{(n+1)-i}}\right) &= e\left(\prod_{i\in I} \left((c/g_1^{v_i\alpha^i})^{\alpha^{(n+1)-i}} \cdot g_1^{\alpha^{n+1} v_i}\right)^{t_i}, g_2\right) \Leftrightarrow & (86)\\
&= e\left(\prod_{i\in I} \left((c/g_1^{v_i\alpha^i}) \cdot g_1^{\frac{\alpha^{n+1} v_i}{\alpha^{(n+1)-i}}}\right)^{t_i \alpha^{(n+1)-i}}, g_2\right) \Leftrightarrow & (87)\\
&= e\left(\prod_{i\in I} \left((c/g_1^{v_i\alpha^i}) \cdot g_1^{v_i\alpha^i}\right)^{t_i \alpha^{(n+1)-i}}, g_2\right) \Leftrightarrow & (88)\\
&= e\left(\prod_{i\in I} c^{t_i \alpha^{(n+1)-i}}, g_2\right) \Leftrightarrow & (89)\\
&= e\left(c^{\sum_{i\in I} t_i \alpha^{(n+1)-i}}, g_2\right) \Leftrightarrow & (90)\\
&= e\left(c, g_2^{\sum_{i\in I} t_i \alpha^{(n+1)-i}}\right) & (91)
\end{aligned}
$$

Aggregating Proofs and Precomputing All Proofs. A subvector proof requires b hash computations and an $O(b)$-sized multi-exponentiation and thus takes $O(b)$ time. Precomputing all proofs efficiently is not discussed. Naively, it can be done in $O(n^2)$ time.

D.7 Complexity of our Lagrange-based aSVC from Section 3.3

Our scheme builds upon previous VCs using KZG commitments [CDHK15, KZG10]. Since we give its full algorithmic description in Section 3.4.3, this section will be briefer than previous ones.

Public Parameters. The proving key, verification key and ith update key are $O(n), O(b)$ and $O(1)$-sized, respectively. Similar to Appendix D.5, n is assumed to be a power of two, and $\mathcal{L}_i(X) = \prod_{j\in[0,n), j\ne i} \frac{X-\omega^j}{\omega^i-\omega^j}$ where ω is a primitive nth root of unity [vzGG13a].

Commitment. A commitment is $c = \prod_{i\in[0,n)} \ell_i^{v_i} = g^{\phi(\tau)}$ where $\phi(X) = \sum_{i\in[0,n)} \mathcal{L}_i(X) v_i$ and $\phi(\omega^i) = v_i$. If any vector element v_j changes to $v_j + \delta$, the commitment can be updated in $O(1)$ time using as $c' = c \cdot (\mathsf{upk}_j)^\delta = c \cdot (\ell_j)^\delta$.

Proofs for a v_i. A proof for v_i is:

$$\pi_i = g^{\frac{\phi(\tau)-v_i}{\tau-\omega^i}} = g^{q_i(\tau)} \tag{92}$$

However, note that:

$$
\begin{aligned}
\frac{\phi(\tau)-\phi(\omega^i)}{\tau-\omega^i} &= \frac{\sum_{j\in[0,n)} \mathcal{L}_j(\tau) v_j - v_i}{\tau-\omega^i} & (93)\\
&= \frac{\sum_{j\in[0,n)\setminus\{i\}} \mathcal{L}_j(\tau) v_j}{\tau-\omega^i} + \frac{\mathcal{L}_i(\tau) v_i - v_i}{\tau-\omega^i} & (94)\\
&= \sum_{j\in[0,n)\setminus\{i\}} v_j \frac{\mathcal{L}_j(\tau)}{\tau-\omega^i} + v_i \frac{\mathcal{L}_i(\tau)-1}{\tau-\omega^i} & (95)
\end{aligned}
$$

Recall from Section 3.4.2 that (1) the ith update key contains a KZG commitment u_i to $\frac{\mathcal{L}_i(\tau)-1}{\tau-\omega^i}$ and that (2) the a_i's and a_j's from upk_i and upk_j can be used to compute in $O(1)$ time a KZG commitment $u_{i,j}$ to $\frac{\mathcal{L}_j(\tau)}{\tau-\omega^i}$. (Note that the partial fraction decomposition only requires evaluating a degree-1 polynomial at two points. Also, computing $A'(\omega^j)$ can be done in $O(1)$ time as explained in Appendix A.) Thus, the proof π_i can be computed in $O(n)$ field operations and $O(n)$ exponentiations as:

$$\pi_i = g^{q_i(\tau)} = \prod_{j\in[0,n)\setminus\{i\}} (u_{i,j})^{v_j} \cdot (u_i)^{v_i} \tag{96}$$

Table 3. Asymptotic comparison of our aSVC with (aS)VCs based on hidden-order groups. n is the vector size, b is the subvector size, ℓ is the length in bits of vector elements, $N = n\ell$ and λ is the security parameter. For schemes based on hidden-order groups, the complexities in the table are *asymptotic* in terms group operations rather than exponentiations. This gives a better sense of performance, since exponents cannot be "reduced" in hidden-order groups as they can in known-order groups. We try to account for field operations (of size 2λ bits), but quantifying them precisely in these schemes can be very cumbersome. Also, since field operations are much faster, they can be mostly ignored. For our aSVC scheme, we give the same complexities in terms of group *exponentiations*, pairings and field operations (see Appendix D.7 for details). Because of this, the reader must be careful when comparing our scheme with the other schemes in this table: a group exponentiation in our scheme is roughly equivalent to $O(\lambda)$ group operations in the hidden-order group schemes. (*Updating the commitment in CFG_ℓ^1 only works in a weaker security model where the commitment is not produced adversarially.)

(aS)VC scheme	$\lvert\mathsf{prk}\rvert$	$\lvert\mathsf{vrk}\rvert$	$\lvert\mathsf{upk}_i\rvert$ or $\lvert\mathsf{uph}_i\rvert$	Com.	Com. upd.	$\lvert\pi_i\rvert$	Prove one v_i	Verify one v_i	Proof upd.	Prove subv. $(v_i)_{i\in I}$	Verify subv. $(v_i)_{i\in I}$	Aggr-egate	Prove each $(v_i)_{i\in[n]}$
BBF_ℓ [BBF19]	1	1	1	$N\lg N$	$\ell\lg N$	$\ell\lg N$ bits	$N\lg N$	$\ell\lg N+\lambda$	×	$N\lg N$	$b\ell\lg N+\lambda$	$b\ell\lg N$	$N\lg^2 N$
CFG_ℓ^1 [CFG+20]	1	1	1	$N\lg N$	1*	1 $\lvert\mathbb{G}_?\rvert$	$N\lg N$	$\ell\lg N+\lambda$	1	$\ell(n-b)\lg N$	$b\ell\lg N+\lambda$	$b\lg b\lg N$	$N\lg^2 N$
CFG_ℓ^2 [CFG+20]	1	1	1	$N\lg n$	1	1 $\lvert\mathbb{G}_?\rvert$	$N\lg n$	ℓ	1	$\ell(n-b)\lg(n-b)$	ℓb	$\ell b\lg^2 b$	$N\lg n$
Our aSVC	n	b	1	n	1	1	n	1	1	$b\lg^2 b+n\lg n$	$b\lg^2 b$	$b\lg^2 b$	$n\lg n$

The proof can be verified in $O(1)$ time using g^τ from the vrk by computing two pairings:

$$e(c/g^{v_i}, g) = e(\pi_i, g^\tau/g^{\omega^i}) \tag{97}$$

Proof Updates. If any vector element $v_j, j \neq i$ changes to $v_j + \delta$, the proof π_i can be updated in $O(1)$ time using a_i, a_j from $\mathsf{upk}_i, \mathsf{upk}_j$. First, one computes $u_{i,j}$ in $O(1)$ time as described in the previous paragraph. Then, one updates $\pi_i' = \pi_i \cdot (u_{i,j})^\delta$ in $O(1)$ time. This new π_i' will verify against the updated c' commitment defined earlier. If v_i changes to $v_i + \delta$, the proof π_i is updated in $O(1)$ time using u_i from upk_i as $\pi_i' = \pi_i \cdot (u_i)^\delta$ (see Section 3.4.2).

Subvector Proofs for $(v_i)_{i\in I}$ We use the same style of subvector proofs as in Appendix D.2. Thus, the subvector proving time is $O(n\log n + b\log^2 b)$ and the subvector proof verification time is $O(b\log^2 b)$ time.

Aggregating Proofs and Precomputing All Proofs. Aggregating all proofs $(\pi_i)_{i\in I}$ requires computing coefficients $c_i = 1/A_I'(\omega^i), \forall i \in I$ using partial fraction decomposition (see Section 3.4.1). This can be done by (1) computing $A_I(X) = \prod_{i\in I}(X - \omega^i)$ in $O(b\log^2 b)$ field operations, (2) computing its derivative $A_I'(X)$ in $O(b)$ field operations and (3) evaluating $A_I'(X)$ at all $(\omega^i)_{i\in I}$ using a multipoint evaluation in $O(b\log^2 b)$ field operations [vzGG13b]. Then, the subvector proof can be aggregated with $O(b)$ exponentiations as:

$$\pi_I = \prod_{i\in I} \pi_i^{c_i} \tag{98}$$

Thus, aggregation takes $O(b\log^2 b)$ time.

Finally, precomputing all proofs can be done efficiently in $O(n\log n)$ time using the FK technique [FK20].

Slower Commitment Time for Faster (Subvector) Proofs. When comitting to a vector, we can use the FK technique to precompute all n proofs in $O(n\log n)$ time and store them as *auxiliary information*. Then, we can serve any individual proof π_i in $O(1)$ time and any subvector proof in $O(b\log^2 b)$ time by aggregating it from the π_i's.

E Complexity of VCs Based on Hidden-order Groups

We give complexities of VCs based on hidden-order groups in Table 3. These can be challenging to describe succinctly due to the many variable-length integer operations that arise. In an effort to keep complexities simple without leaving out too much detail, we often measure (and even approximate) complexities in terms of operations in a finite field of size $2^{2\lambda}$ (e.g., additions, multiplications, computing Bézout coefficients, Shamir tricks), where λ is our security parameter. Another reason to do so is for fairness with VC schemes in known-order groups, which also use finite fields of size $2^{2\lambda}$. Otherwise, a 2λ-bit multiplication would be counted as $O(\lambda\log\lambda)$ in schemes such as BBF_ℓ [BBF18][3] and as $O(1)$ time for schemes like KZG (see Appendix D.2).

The Shamir Trick. The "Shamir Trick" [Sha81, BBF18] can be used to compute an eth root of g given an e_1th root and an e_2th root where $e = e_1 e_2$ and $\gcd(e_1, e_2) = 1$. The idea is to compute Bézout coefficients a, b such that $ae_1 + be_2 = 1$. Then, $\left(g^{\frac{1}{e_1}}\right)^b \left(g^{\frac{1}{e_2}}\right)^a = g^{\frac{be_2}{e_1e_2}} g^{\frac{ae_1}{e_1e_2}} = g^{\frac{ae_1+be_2}{e_1e_2}} = g^{\frac{1}{e_1e_2}}$. Note that $|a| \approx |e_2|$ and $|b| \approx |e_1|$.

[3] Assuming recent progress on multiplying b-bit integers in $O(b\log b)$ time.

E.1 Complexity of $\mathbf{BBF}_\ell$ [BBF19]

In this scheme, we assume the vector $\mathbf{v} = [v_1, v_2, \ldots, v_n]$ is indexed from 1 to n.

Public Parameters. Let ℓ denote the size of vector elements in bits. Let n denote the number of vector elements. Let $N = \ell n$. Let $\mathbb{G}_?$ denote a hidden-order group and g be a random group element in $\mathbb{G}_?$. Let $H : [N] \to \mathsf{Primes}$ be a bijective function that on input i outputs the ith prime number p_i. (Note that $|p_N| = \log(\ell n)$ bits.) The prk and vrk consist of only g. This scheme does not use "fixed" update keys compatible with our definitions. Instead, this scheme uses *"dynamic" update hints*: the ith update hint w.r.t. a commitment c is a VC proof for v_i that verifies against c. In this sense, similar to Merkle trees, this scheme is less suitable for account-based stateless cryptocurrencies [CPZ18], since it requires user i to fetch user j's proof too, before sending her money.

Commitment. An ℓ-bit vector element v_i can be written as a vector of ℓ bits $(v_{i,j})_{j\in[0,\ell-1]}$ Then, each bit $v_{i,j}$ is mapped to the unique prime $p_{(i-1)\cdot\ell+j}$. Put differently, each v_i is mapped to ℓ unique primes $(p_{(i-1)\cdot\ell+0}, p_{(i-1)\cdot\ell+1}, \ldots, p_{(i-1)\cdot\ell+(\ell-1)})$. Then, for each v_i, take the product of all primes corresponding to non-zero bits as $P_i = \prod_{j\in[0,\ell-1]} v_{i,j} \cdot \left(p_{(i-1)\cdot\ell+j}\right)$. Note that $|P_i| = O(\ell \log(\ell n))$. A commitment to the vector $\mathbf{v} = (v_i)_{i\in[n]}$ will be an RSA accumulator over these P_i's:

$$c = g^{\prod_{i\in[n]}\prod_{j\in[0,\ell-1]} v_{i,j}\cdot\left(p_{(i-1)\cdot\ell+j}\right)} \tag{99}$$

$$= g^{\prod_{i\in[n]} P_i} \tag{100}$$

The exponent of c is a product of at most ℓn primes, with the biggest prime having size $O(\log(\ell n))$. Thus, computing the $O(1)$-sized commitment c takes $O(\ell n \log(\ell n))$ group operations. (Note that, for hidden-order groups, we are counting group operations rather than exponentiations. This is to give a better sense of performance, which varies with the exponent size, since exponents cannot be "reduced" in hidden-order groups.)

Since updating commitments requires update hints, which are VC proofs, we must first discuss VC proofs.

E.1.1 Proofs for a v_i

A proof π_i for v_i must show two things:

1. That P_i corresponding to all non-zero bits is accumulated in c.
2. That $Z_i = \prod_{j\in[0,\ell-1]}(1 - v_{i,j}) \cdot \left(p_{(i-1)\cdot\ell+j}\right)$ corresponding to all zero bits is *not* accumulated in c. (Note that $|Z_i| = |P_i| = O(\ell \log(\ell n))$.)

Proving "One" Bits are Accumulated. To prove P_i is "in", an $O(1)$-sized RSA accumulator subset proof w.r.t. c can be computed with $O(\ell n \log(\ell n))$ group operations (via $A.\mathsf{NonMemWitCreate}^*$ in [BBF18, Sec 4.2, pg. 15]):

$$\pi_i^{[1]} = g^{\prod_{j\in[n], j\neq i} P_j} = c^{1/P_i} \tag{101}$$

To speed up the verification of this (part of) the proof, a constant-sized *proof of exponentiation (PoE)* [BBF18] is computed in $O(\ell \log(\ell n))$ field and group operations. We discuss this later in Appendix E.1.2.

Proving "Zero" Bits are Accumulated. To prove Z_i is "out", an $O(\ell \log(\ell n))$-sized disjointness proof $\pi_i^{[0]}$ can be computed w.r.t. c (via $A.\mathsf{NonMemWitCreate}$ in [BBF18, Sec 4.1, pg. 14]). First, Z_i must be computed, but we assume this can be done in $O(\ell \log(\ell n))$ field operations. Second, Bézout coefficients are computed such that $\alpha \prod_{i\in n} P_i + \beta Z_i = 1$. Then, the disjointness proof is $\pi_i^{[0]} = (g^\beta, \alpha)$. Since $|\alpha| \approx |Z_i|$, the proof is $O(\ell \log(\ell n))$-sized. Although this disjointness proof can be made $O(1)$-sized via *proofs of knowledge of exponent (PoKE)* proofs, this seems to break the ability to aggregate VC proofs in BBF_ℓ [BBF18, Sec 5.2, pg. 20]. However, the prover can still include two constant-sized PoE proofs for $(g^\beta)^{Z_i}$ and for c^α to make the verifier's job easier, which costs him only $O(\ell \log(\ell n))$ field and group operations.

To analyze the time complexity of computing $\pi_i^{[0]}$, recall that:

1. The asymptotic complexity of computing Bézout coefficients on b-bit numbers is $O(b \log^2 b)$ time.
2. $b = \left|\prod_{i\in n} P_i\right| = O(n\ell \log(\ell n))$.

As a result, the Bézout coefficients take $O((n\ell \log(\ell n)) \log^2(n\ell \log(\ell n)) = O(n\ell \log(\ell n)(\log n\ell + \log\log(\ell n))^2) = O(n\ell \log^3(\ell n))$ time. However, since these are bit operations, we will count them as $O(n\ell \log(\ell n))$ field operations. Furthermore, computing g^β, where $|\beta| \approx |\prod_{i\in[n]} P_i| = O(n\ell \log(\ell n))$ takes $O(n\ell \log(\ell n))$ group operations.

Overall, the time to compute π_i is $O(\ell n \log(\ell n)) = O(\ell n \log n)$.

E.1.2 Verifying a Proof for v_i

To verify $\pi_i = (\pi_i^{[0]}, \pi_i^{[1]})$, the verifier proceeds as follows. First, he computes P_i in $O(\ell \log(\ell n))$ field operations. Second, he checks that $\left(\pi_i^{[1]}\right)^{P_i} = c$ via the PoE proof in $\pi_i^{[1]}$ using $O(\lambda)$ group operations and $O(\ell \log n)$ field operations. Third, he parses (g^β, α) from $\pi_i^{[0]}$ and checks if $(g^\beta)^{Z_i} c^\alpha = g$. Since the prover included PoE proofs, this can be verified with $O(\lambda)$ group operations and $O(\ell \log(\ell n))$ field operations.

E.1.3 Updates

Updating Commitments. Suppose v_i changes to v_i'. For message bits that are changed from 0 to 1, updating the commitment c involves "accumulating" new primes associated with those bits in c. For message bits that are changed from 1 to 0, updating c involves removing the primes associated with those bits from c. Recall that, in BBF_ℓ, the ith update hint uph_i is actually the VC proof π_i for v_i w.r.t. c. Also recall that $\pi_i^{[1]} = c^{1/P_i}$ from π_i is exactly the commitment c without any of the primes associated with v_i. Thus, to update the commitment, we can compute $P_i' = \prod_{j\in[0,\ell-1]} v_{i,j}' p_{(i-1)\cdot\ell+j}$ in $O(\ell)$ field operations and set $c' = \left(\pi_i^{[1]}\right)^{P_i'}$ in $O(\ell \log(\ell n))$ group operations.

To process several updates for b updated elements $(v_i)_{i\in I}$ with uph_i's that all verify w.r.t. c, we have to take an additional step. First, we use $O(b)$ Shamir tricks on the $\pi_i^{[1]}$'s from uph_i to obtain the commitment $c^{1/\prod_{i\in I} P_i}$, which no longer accumulates any primes associated with the old elements $(v_i)_{i\in I}$. Then, we can add back the new primes P_i' associated with the new elements $(v_i')_{i\in I}$ in $O(b\ell \log(\ell n))$ group operations. We assume the $O(b)$ Shamir tricks can be done in $O(b)$ field operations.

Updating Proofs. Proof updates are not discussed in [BBF19], but seem possible. We leave it to future work to describe them and their complexity.

E.1.4 Subvector Proofs for $(v_i)_{i\in I}$

Recall that a normal VC proof for v_i reasons about which primes associated with v_i are (not) accumulated in c. A subvector proof will do the same, except it will reason about primes associated with all $(v_i)_{i\in I}$. Thus, instead of reasoning about two $O(\ell \log(\ell n))$-sized P_i and Z_i, it will reason about two $O(b\ell \log(\ell n))$-sized $\prod_{i\in I} P_i$ and $\prod_{i\in I} Z_i$. Specifically, an I-subvector proof consists of:

$$\pi_I^{[1]} = g^{\prod_{j\in[n]\setminus I} P_j} = c^{1/\prod_{i\in I} P_i} \tag{102}$$

$$\pi_I^{[0]} = \left(g^\beta, \alpha\right) \text{ such that } (g^\beta)^{\prod_{i\in I} Z_i} c^\alpha = g \tag{103}$$

Let us analyze the proving time and the proof size. First, computing $\pi_I^{[1]}$ can be done in $O(\ell(n-b)\log(\ell n))$ group operations, which is slightly faster than the $O(\ell n \log(\ell n))$ time for computing $\pi_i^{[1]}$ in an individual proof for v_i (see Equation (101)). Second, computing the PoE for $\left(\pi_I^{[1]}\right)^{\prod_{i\in I} P_i} = c$ can be done in $O(b\ell \log(\ell n))$ field and group operations. Third, computing $\pi_I^{[0]}$ maintains the same asymptotic complexity, since it is dominated by the time to compute g^β, which remains just as expensive. However, $\pi_I^{[0]}$'s size would increase to $O(b\ell \log(\ell n))$, since the Bézout coefficient α will be roughly of size $|\prod_{i\in I} Z_i|$. Fortunately, the prover can avoid this by giving c^α rather than α along with a PoKE proof (i.e., one group element and one 2λ-bit integer), while maintaining the same asymptotic complexity. As before, the prover also gives a PoE proof for $\left(g^\beta\right)^{Z_i}$ to speed up the verifier's job.

Because of the PoE proof, verification of $\pi_I^{[1]}$ only requires $O(\lambda)$ group operations as before, but the number of field operations increases to $O(b\ell \log(\ell n))$. Similarly, the PoKE proof will speed up verification of $\pi_I^{[0]}$ to $O(\lambda)$ group operations, but the $O(b\ell \log(\ell n))$ field operations remain for verifying the PoE proof for $\left(g^\beta\right)^{Z_i}$.

E.1.5 Aggregating Proofs

Since aggregating RSA membership and non-membership witnesses is possible [BBF18], and BBF_ℓ VC proofs consist of one RSA membership (subset) proof and one non-membership (disjointness) proof, it follows that aggregating proofs is possible. We leave it to future work to analyze the complexity of aggregation, which has to be at least $\Omega(b\ell \log(\ell n))$ since it must read all b VC proofs as input, which are each $O(\ell \log(\ell n))$-sized.

E.1.6 Precomputing All Proofs

Computing all membership and non-membership witnesses for an RSA accumulators over N elements is possible in $O(N \log N)$ exponentiations [BBF18,STSY01]. Since for BBF_ℓ we have $N = \ell n$ and an exponentiation costs $O(\log(\ell n))$ group operations, this would take $O(\ell n \log^2(\ell n))$ group operations. We are ignoring (1) the overhead of aggregating membership and non-membership witnesses and (2) the overhead of computing PoE proofs, which we assume is dominated by the cost to compute the witnesses.

E.2 Complexity of $\mathbf{CFG}^1_\ell$ [CFG+20] and $\mathbf{CFG}^2_\ell$ [CF13, LM19, CFG+20]

We refer the reader to [CFG+20, Table 1, pg. 35] for most of these these complexities.

Aggregating Proofs. For CFG^1_ℓ, aggregating b proofs into an I-subvector proof takes $O(b \log b \log N)$ group operations [CFG+20, Sec 5.1, pg. 23]. For CFG^2_ℓ, this takes $O(\ell b \log^2 b)$ group operations [CFG+20, Sec 5.2, pg. 32].

Precomputing All Proofs. The paper gives a generic technique of using incremental (dis)aggregation to precompute auxiliary information for serving proofs fast. This technique can also be used to precompute all proofs fast in quasilinear time. In CFG^1_ℓ, we believe this will take $O(N \log^2 N)$ group operations, dominated by the complexity of computing all $N = \ell n$ RSA accumulator membership witnesses. In CFG^2_ℓ, we estimate this will take $O(\ell n \log n)$ group operations (since disaggregating a proof of size m into two proofs of size $m/2$ takes $O(\ell m)$ group operations).