Aggregatable Subvector Commitments for Stateless Cryptocurrencies

Alin Tomescu[✉,1], 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
[2] Ethereum Foundation

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, give an efficient construction in prime-order groups from constant-sized polynomial commitments and use it to bootstrap a highly-efficient stateless cryptocurrency.

Our aSVC supports (1) computing all n $O(1)$-sized proofs in $O(n \log n)$ time, (2) updating a proof in $O(1)$ time and (3) aggregating b proofs into an $O(1)$-sized subvector proof in $O(b \log^2 b)$ time. Importantly, our scheme has an $O(n)$-sized proving key, an $O(1)$-sized verification key and $O(1)$-sized update keys. In contrast, previous schemes with constant-sized proofs in prime-order groups either (1) require $O(n^2)$ time to compute all proofs, (2) require $O(n)$-sized update keys to update proofs in $O(1)$ time, or (3) do not support aggregating proofs. Furthermore, schemes based on hidden-order groups either (1) have larger concrete proof size and computation time, or (2) do not support proof updates.

We use our aSVC to obtain a stateless cryptocurrency with very low communication and computation overheads. Specifically, our constant-sized, aggregatable proofs reduce each block's proof overhead to just one group element, which is optimal. In contrast, previous work required $O(b \log n)$ group elements, where b is the number of transactions per block. Furthermore, our smaller proofs reduce the block verification time from $O(b \log n)$ pairings to just two pairings and an $O(b)$-sized multi-exponentiation. Lastly, our aSVC's smaller update keys only take up $O(b)$ block space, compared to $O(b \log n)$ in previous work. Also, their zverifiability reduces miner storage from $O(n)$ to $O(1)$. The end result is a stateless cryptocurrency that concretely and asymptotically outperforms the state of the art.

1 Introduction

In a *stateless cryptocurrency*, neither *miners* nor cryptocurrency *users* need to store the full blockchain. Instead, the blockchain *state* consisting of users' account balances is *authenticated* using an *authenticated data structure (ADS)*. This way, miners only store a succinct *digest* of the blockchain state. 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 and users can easily *synchronize* or *update* their proofs as new blocks get published.

Stateless cryptocurrencies have received increased attention [STS99, Mil12, Tod16, But17, Dry19, RMCI17, CPZ18, BBF19, GRWZ20] due to their many advantages. First, stateless transaction validation against a digest scales better than stateful validation against a stored database [CPZ18]. Second, stateless cryptocurrencies eliminate hundreds of gigabytes of storage that miners or full nodes need to validate blocks. Third, statelessness makes sharding much easier, by allowing miners to efficiently switch from one shard to another. Fourth, since validating a block is stateless and efficient, anybody can be a full node, resulting in a much more resilient, distributed cryptocurrency.

Previous work [Dry19, BBF18, CPZ18] shows how to obtain *UTXO-based* [Nak08] stateless cryptocurrencies efficiently using RSA accumulators [BdM94] or Merkle hash trees. However, for *account-based* [Woo] cryptocurrencies, current solutions either have larger-than-ideal proof sizes [CPZ18] or are based on hidden-order groups which are concretely slower [BBF18]. Thus, the focus of this paper is to improve the concrete and asymptotic complexities of *account-based*, stateless cryptocurrencies. (The advantages and disadvantages of account-based and UTXO-based designs are discussed in depth in [Zah18, But16, Cor16, Pat17, Eth17, Yan16].)

Account-based, Stateless Cryptocurrencies from Vector Commitments (VCs). Previous work pioneers the idea of building account-based stateless cryptocurrencies on top of any *vector commitment (VC)* scheme [CPZ18]. At a high level, a VC scheme allows a *prover* to compute a succinct *commitment* c of 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, an *update key* upk_j tied to the updated position j is necessary. Together, the proving, verification and update keys are known as the VC's *public parameters*.

Lastly, some schemes support computing succinct proofs for I*-subvectors* $(v_i)_{i \in I}$ where $I \subset \{0, 1, \ldots, n-1\} = [0, n)$. Such schemes are called *subvector commitment (SVC)* schemes [LM19]. 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.

1.1 Our Contributions

Stateless cryptocurrencies can be built *efficiently* from VCs if the underlying VC has (1) sublinear-sized, updatable proofs with sublinear-time verification, (2) updatable commitments and (3) sublinear-sized update keys. We say such a VC scheme has "*scalable updates*." Unfortunately, most VCs do not have scalable updates (see Section 1.2 and Tables 2 and 4) or, if they do [CPZ18, Tom20], they are not optimal in their proof and update key sizes. Lastly, while schemes in hidden-order groups could be enhanced with scalable updates, their concrete performance does not match that of schemes in prime-order groups. In this paper, we present a new VC with scalable updates that has optimal proof sizes and we use it to build an efficient stateless cryptocurrency.

Aggregatable Subvector Commitments (aSVCs) with Scalable Updates. We present a new notion of *aggregatable subvector commitment (aSVC)*, or an SVC that supports commitment updates, proof updates and aggregation of proofs into subvector proofs. Then, we construct an aSVC *with scalable updates* over pairing-friendly groups. Our aSVC outperforms previous work on several dimensions (see Table 2).

First, our aSVC has constant-sized I-subvector proofs: one group element. Second, it has constant-sized update keys: two group elements (or one group element, when used in the stateless cryptocurrency setting). Third, it can update proofs and commitments in $O(1)$ time. Fourth, it can aggregate multiple proofs into an I-subvector proof fast using $O(|I|)$ exponentiations and $O(|I| \log^2 |I|)$ field operations. Lastly, our aSVC can compute *all* proofs fast in $O(n \log n)$ time via new techniques for precomputing proofs in polynomial commitments [FK20].

At the core of our construction lies a new idea by Buterin to use *partial fraction decomposition* to aggregate proofs in polynomial commitments [But20]. We use this not only to aggregate proofs but also to reduce our update key size. Furthermore, to prove security of our aSVC we have to strengthen the security definition of KZG polynomial commitments and prove they still satisfy it. As a last remark, our aSVC could be used to improve verifiable databases with efficient updates [BGV11], updatable elementary zero-knowledge databases [CF13], anonymous credentials [KZG10] and stateless smart contract validation [GRWZ20].

Table 1. Asymptotic comparison of our work with other stateless cryptocurrencies. n is the number of cryptocurrency users, 2λ is the bit-width of a vector element in $\text{BBF}_{2\lambda}$ [BBF18], where λ is the security parameter. $N = \lambda n$, and b is the number of transactions in a block. $|\mathbb{G}|$ denotes either a group operation or a group element in a known-order, pairing-friendly group. $|\mathbb{G}_?|$ denotes the same for hidden-order groups. $|\mathbb{G}^e|$ denotes an exponentiation in a known-order group. $|\mathbb{F}|$ denotes either a field element or operation in a field of size $2^{2\lambda}$. $|\mathbb{P}|$ denotes 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. $|\pi_{\mathsf{agg}}|$ is the size of a proof aggregated from all π_i's in a block. *"Miner storage"* is the overhead for miners storing update keys. *"Vrfy. proofs time/blk."* is the time for a miner to (batch) verify all transaction proofs in a new block. *"Check digest time/blk."* is the time for a miner to check that, by "applying" the transactions from block $t+1$ to block t's digest, he obtains the correct digest for block $t+1$. *"Aggr. proofs time/blk."* is the time to aggregate b transaction proofs into a single π_{agg} for a block. *"User proof synchr. time/blk"* is the time for a user to "synchronize" or update her proof by "applying" all the transactions in a new block. We treat Pointproofs [GRWZ20] as a payments-only stateless cryptocurrency without smart contracts, where users store the $O(n)$-sized update key. Although $\text{BBF}_{2\lambda}$ [BBF18] has asymptotic performance close to ours, it suffers from concretely higher overheads due to its reliance on hidden-order groups. Also, it does not yet support updating proofs (see Appendix E.1.3). Our aggregation and verification times π_{agg} have an undesired $\log^2 b$ factor, which comes from doing $O(b \log^2 b)$ very fast field operations. However, in practice, the $O(b)$ exponentiations dominate the run time. 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	$\lvert\pi_i\rvert$	$\lvert\mathsf{upk}_i\rvert$	$\lvert\pi_{\mathsf{agg}}\rvert$	Miner storage	Vrfy. proofs time/blk.	Check digest time/blk.	Aggr. proofs time/blk.	User proof synchr. time/blk.
Edrax [CPZ18]	$\log n \lvert\mathbb{G}\rvert$	$\log n \lvert\mathbb{G}\rvert$	$\times$	n	$b\log n \lvert\mathbb{P}\rvert$	$b \lvert\mathbb{G}^e\rvert$	$\times$	$b\log n \lvert\mathbb{G}^e\rvert$
Pointproofs [GRWZ20]	$1 \lvert\mathbb{G}\rvert$	$n \lvert\mathbb{G}\rvert$	$1 \lvert\mathbb{G}\rvert$	n	$b \lvert\mathbb{G}^e\rvert + 2 \lvert\mathbb{P}\rvert$	$b \lvert\mathbb{G}^e\rvert$	$b \lvert\mathbb{G}^e\rvert$	$b \lvert\mathbb{G}^e\rvert$
$\text{BBF}_{2\lambda}$ [BBF18]	$\lambda \lg N \lvert\mathsf{bit}\rvert + 1 \lvert\mathbb{G}_?\rvert$	$1 \lvert\mathbb{G}_?\rvert$	$1 \lvert\mathbb{G}_?\rvert$	1	$b\lambda \lg N \lvert\mathbb{F}\rvert + \lambda \lvert\mathbb{G}_?\rvert$	$b\lambda \lg N \lvert\mathbb{G}_?\rvert$	$\Omega(b\lambda \lg N)$	$\times$
This work	$1 \lvert\mathbb{G}\rvert$	$1 \lvert\mathbb{G}\rvert$	$1 \lvert\mathbb{G}\rvert$	1	$b\log^2 b \lvert\mathbb{F}\rvert + b \lvert\mathbb{G}^e\rvert + 2 \lvert\mathbb{P}\rvert$	$b \lvert\mathbb{G}^e\rvert$	$b\log^2 b \lvert\mathbb{F}\rvert + b \lvert\mathbb{G}^e\rvert$	$b \lvert\mathbb{G}^e\rvert$

A Highly-Efficient Stateless Cryptocurrency. We use our aSVC to construct a stateless cryptocurrency based on the elegant VC-based design of Edrax [CPZ18]. Our stateless cryptocurrency has very low storage, communication and computation overheads (see Table 1). First, it has smaller proofs and update keys, which speeds up proof updates and proof verification. Second, it uses proof aggregation to drastically reduce block size and further speed up proof verification (see Table 1). This helps miners propose and validate new blocks faster, helps users update their proofs faster and reduces overall communication. Third, our verifiable update keys removes the need for miners to either (1) store all $O(n)$ update keys or (2) interact during transaction validation to check update keys.

1.2 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) and the RSA problem. Their RSA-based scheme is the first to support $O(1)$-sized public parameters, which can be *specialized* [CFG+20] 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.

Chepurnoy et al. [CPZ18] instantiate VCs using multivariate polynomial commitments [PST13] but with logarithmic rather than constant-sized proofs. They then build the first efficient, account-based, stateless cryptocurrency on top of their scheme. Their scheme is the first to support efficiently computing all n proofs in $O(n \log n)$ time. This is very useful for *proof serving nodes* in stateless cryptocurrencies. 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. Their scheme is the first to allows multiple proofs to be *aggregated*, 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). 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 instantiate VCs using hidden-order groups. Their scheme is the first to support *infinite aggregation* of proofs as well as *disaggregation*. They are also the first to formalize the notion of *specialization* for public parameters.

Feist and Khovratovich [FK20] introduce a technique for precomputing all *constant-sized* evaluation proofs in KZG commitments only if the evaluation points are all the n nth roots of unity. Our aSVC from Section 3.3 uses their technique to compute VC proofs fast.

Gorbunov et al. [GRWZ20] extends [LY10] with I-subvector proofs that can be aggregated from $(v_i)_{i\in I}$ proofs. Additionally, they add support for aggregating multiple I-subvector proofs *across different vector commitments* into a single, constant-sized proof. However, this versatility seems to come 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 the scheme in a stateless cryptocurrency for payments such as Edrax [CPZ18].

Concurrent with our work, Campanelli et al. [CFG+20] and Gorbunov et al. [GRWZ20] also formalize aSVCs with a stronger notion of *cross-commitment aggregation*. However, these formalizations lack update keys and support for updating proofs and/or commitments. This hides many complexities that arise in stateless cryptocurrencies, such as verifying update keys (see Section 4.2.2). Furthermore, Gorbunov et al. also enhance Lagrange-based VCs with proof aggregation via partial fraction decomposition, but they do not address the problem of updating proofs efficiently. Lastly, Gorbunov et al. show it is possible to aggregate I-subvector proofs across different commitments for Lagrange-based VCs, such as our aSVC from Section 3.

Libert et al. [LRY16] generalize VCs to *functional commitments (FCs)* which, instead of revealing v_i when opening, reveals $\sum_{i\in[0,n)} x_i v_i$, for any $\mathbf{x} = (x_i)_{i\in[0,n)}$ given as input to the opening algorithm. Lai and Malavolta [LM19] generalize FCs to *linear map commitments (LMCs)* which reveals $f(\mathbf{v})$ for any linear map $f : \mathbb{F}_p^n \to \mathbb{F}_p^q$ given as input to the opening algorithm (q is fixed at setup). 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. Proposals for UTXO-based cryptocurrencies were initially based on Merkle hash trees [Mil12, Tod16, Dry19, CPZ18] as their authenticated data structure (ADS). Chepurnoy et al. [CPZ18] present such a construction, partly inspired by Zcash's design [BCG+14]. Dryja [Dry19] gives a beautiful Merkle forest construction that significantly reduces communication. Boneh et al. [BBF18, BBF19] further reduce communication by replacing Merkle trees with 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 efficiently updatable digests and proofs. 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 proof sizes are logarithmic and thus sub-optimal. Furthermore, Edrax does not support proof aggregation, which would significantly reduce block size.

Gorbunov et al. [GRWZ20] introduce *Pointproofs*, a powerful VC scheme that supports aggregating proofs across *different* commitments. They use this power to solve a slightly different problem: stateless block validation for smart contract executions (rather than stateless validation 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 or precompute proofs efficiently. Instead they assume that all contracts have $n \leq 10^3$ 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] enable stateless validation of blocks in an account-based cryptocurrency using recursively-composable, succinct non-interactive arguments of knowledge (SNARKs) [BSCTV14]. 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. Let λ denote our security parameter. Let $\mathbb{G}_1, \mathbb{G}_2$ be 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.) Let $\mathbb{G}_?$ denote a hidden-order group. We will use multiplicative notation for the group operations in $\mathbb{G}_?, \mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$. Let $\mathbb{Z}_p$ denote a finite field of prime order p. Let ω denote a primitive nth root of unity [vzGG13a]. Let $\mathsf{poly}(\cdot)$ denote

any function upper-bounded by some univariate polynomial. Let $\mathsf{negl}(\cdot)$ denotes any negligible function. Let $\log x$ and $\lg x$ be shorthand for $\log_2 x$. Let $[i,j] = \{i, i+1, \ldots, j-1, j\}$, $[0,n) = [0, n-1]$ and $[n] = [1,n]$. Let $\mathbf{v} = (v_i)_{i\in[0,n)}$ denote 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:

$$\phi(X) = \sum_{i\in[0,n)} \mathcal{L}_i(X) y_i, \text{ where } \mathcal{L}_i(X) = \prod_{\substack{j\in[0,n)\\ j\neq i}} \frac{X - x_j}{x_i - x_j} \tag{1}$$

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 $j \neq i$. Also, keep in mind that $\mathcal{L}_i(X)$ is defined in terms of the *interpolation domain* $(x_i)_{i\in[0,n)}$. Throughout this paper, the domain 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 $\phi(X)$ based on the n-S(B)DH assumption [BB08, Goy07]. 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 n-SDH public parameters $(g^{\tau^i})_{i\in[0,n]}$ where τ denotes a trapdoor. (These parameters are computed via a *trusted setup* which can be decentralized with MPC protocols [BGG19, BGM17].) KZG is computationally-hiding under the discrete log assumption and computationally-binding under n-SDH [KZG10].

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 [KZG10]. 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 $\pi = g^{q(\tau)}$.

To verify π against $\{\phi(i)\}_{i\in I}$ and 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 unbounded if the VC is unbounded.) For simplicity of exposition, we do not give details on the consensus algorithm, on transaction signature verification and 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, users store 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 include her proof $\pi_{i,t}$ in her transaction. However, user i does not need to include the recipient j's proof $\pi_{j,t}$ in the transaction.

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 a user i 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, the user 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]. Specifically, in an aSVC anybody can aggregate b proofs for individual positions into a single constant-sized *subvector proof* for those positions. Our formalization differs from previous work [BBF18, GRWZ20, CFG+20] in that it accounts for 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]. However, we add a $\mathsf{VC.VerifyUPK}$ API for verifying update keys and a $\mathsf{VC.AggregateProofs}$ API for aggregating proofs. We stress that $\mathsf{VC.VerifyUPK}$ is necessary in distributed applications such as stateless cryptocurrencies (see Section 4.2.2).

$\mathsf{VC.KeyGen}(1^\lambda, n) \to \mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)}$. 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* $(\mathsf{upk}_j)_{j\in[0,n)}$.

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

$\mathsf{VC.ProvePos}(\mathsf{prk}, I, \mathbf{v}) \to \pi_I$. 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 $I = i \in [0,n)$.

$\mathsf{VC.VerifyPos}(\mathsf{vrk}, c, \mathbf{v}_I, I, \pi_I) \to T/F$. 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 $I = i \in [0,n)$.

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

$\mathsf{VC.UpdateComm}(c, \delta, j, \mathsf{upk}_j) \to c'$. 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.

$\mathsf{VC.UpdateProof}(\pi_i, \delta, j, \mathsf{upk}_i, \mathsf{upk}_j) \to \pi_i'$. 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.

$\mathsf{VC.AggregateProofs}(I, (\pi_i)_{i\in I}) \to \pi_I$ Deterministic algorithm that takes π_i for $v_i, \forall i \in I$ and aggregates them into an I-subvector proof π_I, which is ideally constant-sized.

Additional APIs. We add a $\mathsf{VC.EmptyCommit}$ API for committing to an "empty" vector, which is useful for initializing the authenticated state of a stateless cryptocurrency.

$\mathsf{VC.EmptyCommit}(\mathsf{eck}) \to c, \pi_0$. Deterministic algorithm that, given $\mathsf{eck} \in \{\mathsf{prk}, \mathsf{vrk}\}$, returns a commitment to the empty vector $\mathbf{v} = (0, 0, \ldots, 0)$ and a proof π_0 that $v_i = 0$, which verifies successfully for any position i.

3.2 aSVC Correctness and Security Definitions

We argue why our aSVC from Section 3 satisfies these definitions in Section 3.4.6.

Definition 1 (Aggregatable Vector Commitment 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 vector 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* $I \subseteq [0, n)$*:*

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

Definition 3 (Commitment 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* $\mathbf{u}$ *be the same vector as* $\mathbf{v}$ *except with* $v_k + \delta$ *rather than* v_k *at position* k*. Then:*

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

Definition 4 (Proof Update Correctness). $\forall$ *vectors* $\mathbf{v} = (v_j)_{j\in[0,n)}$, $\forall$ *positions* $i \in [0, n), k \in [0, n)$, $\forall$ *updates* $\delta \in \mathbb{Z}_p$*:*

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

Definition 5 (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 6 (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 7 (Update Key Uniqueness). $\forall$ *positions* $i \in [0, n)$*:*

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

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

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

Table 2. Asymptotic comparison of our aSVC with other (aS)VCs. n is the vector size and b is the subvector size. See Appendix D for a detailed explanation. The complexities in the table are *asymptotic* in terms of number of pairings, exponentiations and field operations. *"Proof upd."* is the time to update *one* proof for *one* single vector element after a change to *one* vector element. *"Com. upd."* is the time to update the commitment after a change to one vector element. We include complexities for computing a VC proof for v_i and a size-b subvector proof (in *"Prove one v_i"* and in *"Prove subv. $(v_i)_{i\in I}$"*, respectively) and for committing to a size-n vector (in *"Com."*), even though these features are not needed in a stateless cryptocurrency.

(aS)VC scheme	$\lvert\mathsf{prk}\rvert$	$\lvert\mathsf{vrk}\rvert$	$\lvert\mathsf{upk}_i\rvert$	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. $(v_i)_{i\in I}$	Aggregate	Prove each $(v_i)_{i\in[n]}$
$\mathrm{LM}_{\mathsf{cdh}}$ [CF13, LM19]	n^2	n	n	n	1	1	n	1	1	bn	b	×	n^2
KZG [KZG10]	n	b	×	$n\lg^2 n$	×	1	n	1	×	$b\lg^2 b + n\lg n$	$b\lg^2 b$	×	n^2
$\mathrm{KZG}_{\mathsf{Lagr}}$ [CDHK15]	n	n	1	$n\lg^2 n$	1	1	n	1	×	$n\lg^2 n$	$b\lg^2 b$	×	n^2
CPZ [CPZ18]	n	$\lg n$	$\lg n$	n	1	$\lg n$	n	$\lg n$	$\lg n$	×	×	×	$n\lg n$
TCZ [Tom20]	n	$\lg n + b$	$\lg n$	$n\lg n$	1	$\lg n$	$n\lg n$	$\lg n$	$\lg n$	$b\lg^2 b + n\lg n$	$b\lg^2 b$	×	$n\lg n$
LY [GRWZ20, LY10]	n	n	n	n	1	1	n	1	1	bn	b	b	n^2
Our aSVC	n	b	1	n	1	1	n	1	1	$b\lg^2 b + n\lg n$	$b\lg^2 b$	$b\lg^2 b$	$n\lg n$
Our $\mathbf{aSVC}_{\mathsf{precomp}}$	n	b	1	$n\log n$	1	1	1	1	1	$b\lg^2 b$	$b\lg^2 b$	$b\lg^2 b$	$n\lg n$

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]. We use *roots of unity* and "store" v_i as $\phi(\omega^i) = v_i$. This means that 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.

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 $\ell_i = g^{\mathcal{L}_i(\tau)}$. This means c can be computed as $c = \prod_{i=1}^{n} (\ell_i)^{v_i}$ in $O(n)$ time without interpolating $\phi(X)$. This also allows the commitment c to be easily updated to c' after adding δ to v_i. Specifically, $c' = c \cdot (\ell_i)^{\delta}$, which is just a commitment to the new updated $\phi'(X) = \phi(X) + \delta \cdot \mathcal{L}_i(X)$.

Proving. A normal proof π_i for a single element v_i is just a KZG proof for the evaluation of $\phi(X)$ at ω^i (see Section 2.2). Interestingly, we show this proof can be computed in $O(n)$ time without interpolating $\phi(X)$ (see Section 3.4.4). A subvector proof π_I for for $v_I, I \subseteq [0,n)$ is just a KZG batch proof for the evaluations of $\phi(X)$ at all $(\omega^i)_{i\in I}$.

Limitations. Next, we add support for (1) updating proofs, (2) aggregating proofs and (3) precomputing all individual proofs efficiently. For aggregation, we use known techniques for aggregating KZG proofs via partial fraction decomposition (see Section 3.4). For updating proofs efficiently, we introduce a new mechanism to reduce the update key size from linear to constant. For precomputing all proofs, we use existing techniques for computing evaluation proofs fast in KZG polynomial commitments [FK20].

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 $\left(\phi(\omega^i)\right)_{i\in I}$:

$$\mathcal{L}_i(X) = \prod_{j\in I, j\neq i} \frac{X-\omega^j}{\omega^i-\omega^j} = \frac{A_I(X)}{A_I'(\omega^i)(X-\omega^i)}, \text{ where } A_I(X) = \prod_{i\in I}(X-\omega^i) \tag{2}$$

Here, $A_I'(X) = \sum_{j\in[0,n)} A_I(X)/(X-\omega^j)$ is the formal 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:

$$\frac{1}{A_I(X)} = \frac{1}{\prod_{i\in I}(X-\omega^i)} = \sum_{i\in[0,n)} c_i \cdot \frac{1}{X-\omega^i}, \text{ where } c_i = \frac{1}{A_I'(\omega^i)} \tag{3}$$

$A_I(X)$ can be computed in $O(|I|\log^2|I|)$ 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 coefficients $c_i = \frac{1}{A'_I(\omega^i)}$ 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 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 single constant-sized proof π_I for the I-subvector of $\mathbf{v}$.

Recall that π_i is a KZG 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:

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

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

To conclude, to aggregate the π_i's into π_I, we 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 (1) $i = j$ or that (2) $i \neq j$. 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 $\phi'(X) = \phi(X) + \delta\mathcal{L}_j(X)$. 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 $v_i + \delta$:

$$q'_i(X) = \frac{\phi'(X) - (v_i+\delta)}{X-\omega^i} = \frac{(\phi(X)+\delta\mathcal{L}_i(X)) - v_i - \delta}{X-\omega^i} = \frac{\phi(X)-v_i}{X-\omega^i} - \frac{\delta(\mathcal{L}_i(X)-1)}{X-\omega^i} = q_i(X) + \delta\left(\frac{\mathcal{L}_i(X)-1}{X-\omega^i}\right) \tag{6}$$

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, which the proof updater always has. (Recall that, in this $i = j$ case, the proof updater called $\mathsf{VC.UpdateProof}(\pi_i, \delta, i, \mathsf{upk}_i, \mathsf{upk}_i)$.) 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 $\pi'_i = \pi_i \cdot (u_i)^\delta$.

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

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

In this case, the proof updater needs a KZG commitment to $\frac{\mathcal{L}_j(X)}{X-\omega^i}$. For this, we will make sure upk_j gives the proof updater *extra information* to allow them to reconstruct this commitment in $O(1)$ time. Here, our VC API differs from that of Chepurnoy et al [CPZ18] as we assume the proof updater gets both upk_i and upk_j. We stress this is reasonable in the stateless cryptocurrency setting where each user has to store their proof π_i and upk_i anyway as they process the $(\delta, j, \mathsf{upk}_j)$ updates (see Section 4).

Update Keys for the $i \neq j$ Case. To understand what extra information the proof updater with upk_i needs in order to reconstruct a commitment $u_{i,j}$ to $U_{i,j}(X) = \frac{\mathcal{L}_j(X)}{X-\omega^i}$, let us rewrite it as $U_{i,j}(X) = \frac{A(X)}{A'(\omega^j)(X-\omega^j)(X-\omega^i)}$. Next, note that since $A'(\omega^j) = n\omega^{-j}$ is a constant (see Appendix A), the proof updater need only reconstruct a KZG commitment to $W_{i,j}(X) = \frac{A(X)}{(X-\omega^j)(X-\omega^i)}$.

Our key idea is to make $a_j = g^{A(\tau)/(\tau-\omega^j)}$ part of $\mathsf{upk}_j, \forall j \in [0, n)$. This way, the proof updater, who has both upk_i and upk_j, can reconstruct a commitment $w_{i,j}$ to $W_{i,j}(X)$ using partial fraction decomposition. Specifically, the proof updater will compute $c_1, c_2 \in \mathbb{Z}_p$ such that $\frac{1}{(X-\omega^j)(X-\omega^i)} = c_1\frac{1}{X-\omega^j} + c_2\frac{1}{X-\omega^i}$ (see Section 3.4). Multiplying by $A(X)$,

we have $W_{i,j}(X) = \frac{A(X)}{(X-\omega^j)(X-\omega^i)} = c_1 \frac{A(X)}{X-\omega^j} + c_2 \frac{A(X)}{X-\omega^i}$. Thus, the proof updater can compute $w_{i,j} = a_j^{c_1} a_i^{c_2}$ and then get $u_{i,j} = (w_{i,j})^{\frac{1}{A'(\omega^j)}}$.

To summarize, the proof updater will: (1) Obtain $a_i = g^{A(\tau)/(\tau-\omega^i)}$ and $a_j = g^{A(\tau)/(\tau-\omega^j)}$ from upk_i and upk_j, respectively, (2) Compute $d_j = 1/(A'(\omega^j)) = 1/(n\omega^{-j}) = \omega^j/n$, (2) Use partial fraction decomposition to compute $w_{i,j} = g^{A(\tau)/[(\tau-\omega^j)(\tau-\omega^i)]}$ from a_i and a_j, (3) Compute $u_{i,j} = (w_{i,j})^{d_j} = g^{\frac{A(\tau)}{A'(\omega^j)(\tau-\omega^j)(\tau-\omega^i)}} = g^{\mathcal{L}_j(\tau)/(\tau-\omega^i)}$, (4) Compute the updated proof $\pi_i' = \pi_i \cdot (u_{i,j})^{\delta}$.

3.4.3 Precomputing All Proofs

The Feist-Khovratovich [FK20] technique can be used to compute all proofs $(\pi_i)_{i\in[0,n)}$ in $O(n \log n)$ time. (Note that $\phi(X)$ can be interpolated in $O(n \log n)$ time via an inverse DFT.) As a result, any subset of $(\pi_i)_{i\in I}$ proofs can be aggregated into an I-subvector proof in $O(|I| \log^2 |I|)$ time. This is useful for reducing the time to compute subvector proofs (see Table 2). Furthermore, it helps proof serving nodes in stateless cryptocurrencies compute proofs faster (see Section 4.3.2).

3.4.4 aSVC Algorithms

In this subsection, we give a full description of how our scheme implements the aSVC API from Section 3.1. To support verifying I-subvector proofs, our verification key is $O(|I|)$-sized.

$\mathsf{VC.KeyGen}(1^\lambda, n) \to \mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j\in[0,n)}$. 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 $\mathsf{upk}_i = (a_i, u_i)$. Sets $\mathsf{prk} = \left((g^{\tau^i})_{i\in[0,n]}, (\ell_i)_{i\in[0,n)}, (\mathsf{upk}_i)_{i\in[0,n)}\right)$. Sets $\mathsf{vrk} = ((g^{\tau^i})_{i\in[0,|I|]}, a)$.

$\mathsf{VC.Commit}(\mathsf{prk}, \mathbf{v}) \to c$. 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 $I = \{i\}$, $A_I(X) = X - \omega^i$ and $R_I(X) = v_i$.)

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

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

$\mathsf{VC.UpdateProof}(\pi_i, \delta, j, \mathsf{upk}_i, \mathsf{upk}_j) \to \pi_i'$. If $i = j$, returns $\pi_i' = \pi_i \cdot (u_i)^\delta$. If $i \neq j$, computes $c_1, c_2 \in \mathbb{Z}_p$ such that $\frac{1}{(X-\omega^j)(X-\omega^i)} = c_1 \frac{1}{X-\omega^j} + c_2 \frac{1}{X-\omega^i}$ (see Section 3.4). Computes $w_{i,j} = a_j^{c_1} \cdot a_i^{c_2}$ and $u_{i,j} = w_{i,j}^{1/A'(\omega^j)}$. Returns $\pi_i' = \pi_i \cdot (u_{i,j})^\delta$.

$\mathsf{VC.AggregateProofs}(I, (\pi_i)_{i\in I}) \to \pi_I$. Computes $A_I(X) = \prod_{i\in I}(X-\omega^i)$ in $O(|I| \log^2 |I|)$ time. Computes the derivative $A_I'(X)$ of $A_I(X)$ in $O(|I|)$ time. Computes $c_i = (A_I'(\omega^i))_{i\in I}$ in $O(|I| \log^2 |I|)$ time using a multipoint polynomial evaluation [vzGG13b]. Returns $\pi_I = \prod_{i\in I} \pi_i^{c_i}$.

3.4.5 Distributing the Trusted Setup

Our VC requires a trusted setup phase that computes its public parameters. To guarantee nobody learns the trapdoor τ, this phase should be distributed via MPC protocols [BGG19, BGM17, BCG+15]. Unfortunately, the most efficient MPC protocols only output (g^{τ^i})'s [BGM17]. This means we can either (1) use less efficient protocols that output our full public parameters or (2) find a way to derive the remaining public parameters from the (g^{τ^i})'s. Fortunately, all remaining public parameters are easy to derive.

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 just a bilinear accumulator membership proof for ω^i w.r.t. $A(X)$.

Thus, all a_i's can be computed in $O(n \log n)$ time via the FK technique [FK20]. Lastly, we need a way to compute all $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{X-\omega^i}}$.

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 $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{X-\omega^i}} = g^{Q_i(\tau)}$ is a KZG evaluation proof for $\mathcal{L}_i(\omega^i) = 1$, which means:

$$\mathcal{L}_i(X) = Q_i(X)(X - \omega^i) + 1 \tag{8}$$

Let $\psi_i(X) = \frac{A(X)}{X-\omega^i} = \frac{X^n-1}{X-\omega^i}$, and let $\pi_i = g^{q_i(\tau)}$ be an evaluation proof for $\psi_i(\omega^i)$ such that:

$$\psi_i(X) = q_i(X)(X - \omega^i) + \psi_i(X) \tag{9}$$

We argue that $Q_i(X) = \frac{1}{A'(\omega^i)} q_i(X)$ and thus each $u_i = g^{Q_i(\tau)}$ can be derived from $\pi_i = g^{q_i(\tau)}$ as $u_i = \pi_i^{\frac{1}{A'(\omega^i)}}$. For this, we will use the fact that $\mathcal{L}_i(X) = \frac{A(X)}{A'(\omega^i)(X-\omega^i)} = \frac{1}{A'(\omega^i)}\psi_i(X)$ and $\psi(\omega^i) = A'(\omega^i)$ (see Equation (2)):

$$\psi_i(X) = q_i(X)(X - \omega^i) + \psi_i(\omega^i) \Rightarrow \tag{10}$$

$$\frac{1}{A'(\omega^i)}\psi_i(X) = \frac{1}{A'(\omega^i)}[q_i(X)(X - \omega^i) + A'(\omega^i)] \Rightarrow \tag{11}$$

$$\mathcal{L}_i(X) = \left(\frac{1}{A'(\omega^i)} q_i(X)\right) \cdot (X - \omega^i) + 1 \Rightarrow Q_i(X) = \frac{1}{A'(\omega^i)} q_i(X) \tag{12}$$

Thus, computing all u_i's reduces to computing n evaluation proofs π_i for $\psi_i(\omega^i)$. However, since each proof π_i is for a *different* polynomial $\psi_i(X)$, all π_i's would still require $O(n^2)$ time to compute naively. We fix this next by leveraging the "structure" of $\psi_i(X)$ when divided by $X - \omega^i$. Specifically, in Appendix B, we show that:

$$q_i(X) = \sum_{j \in [0, n-2]} (j+1)(\omega^i)^j X^{(n-2)-j}, \forall i \in [0, n) \tag{13}$$

Let $H_j(X) = (j+1)X^{(n-2)-j}$. Then:

$$q_i(X) = \sum_{j \in [0, n-2]} H_j(X)\omega^{ij}, \forall i \in [0, n) \tag{14}$$

In particular, if h_j and π_i are KZG commitments to $H_j(X)$ and $q_i(X)$ respectively, we have:

$$\pi_i = \prod_{j \in [0, n-2]} h_j^{(\omega^{ij})}, \forall i \in [0, n) \tag{15}$$

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:

$$\mathsf{DFT}_n(\mathbf{a}) = \hat{\mathbf{a}} = [\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}] \in \mathbb{G}^n, \text{ where } \hat{a}_i = \prod_{j \in [0, n)} a_j^{(\omega^{ij})} \tag{16}$$

If we let $\boldsymbol{\pi} = [\pi_0, \pi_1, \ldots, \pi_{n-1}]$ and $\mathbf{h} = [h_0, h_1, \ldots, h_{n-2}, 1_{\mathbb{G}}, 1_{\mathbb{G}}]$, it follows that:

$$\boldsymbol{\pi} = \mathsf{DFT}_{\mathsf{n}}(\mathbf{h}) \tag{17}$$

Thus, we can compute all π_i's in $O(n \log n)$ time with a single DFT on the h_i's. (All n h_i's can be computed in $O(n)$ time.) Then, we can compute each $u_i = \pi_i^{\frac{1}{A'(\omega^i)}}$ in $O(n)$ time.

3.4.6 Correctness and Security

The correctness of our aSVC schemes 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 9 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 present a stateless cryptocurrency that improves over Edrax [CPZ18] in both computation and communication efficiency. We adopt 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 times 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

In this section, we re-introduce Edrax's design, which we adopt and slightly modify. 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 updatable proofs and digests are not known yet. (The key-value map commitment by Boneh et al. [BBF18] could work, but its efficiency and updatability in the context of stateless cryptocurrencies remains to be explored.) As a result, these AD-based approaches were *interactive*, 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, BGG19]. Miners need to store all $O(n)$ update keys to propose blocks and to validate blocks (see 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. Its digest d_0 is computed as $d_0, \pi_0 \leftarrow \mathsf{VC.EmptyCommit}(\mathsf{vrk})$. Initially, each user i is *unregistered* and starts with $\pi_{i,0} = \pi_0$ as their initial proof, which verifies correctly $\forall i$: $\mathsf{VC.VerifyPos}(\mathsf{vrk}, d_0, 0, i, \pi_{i,0}) = T$.

"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*. 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:

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

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 using a call to $d_{t+1} = \mathsf{VC.UpdateComm}(d_{t+1}, (\mathsf{addr}_i|0), i, \mathsf{upk}_i)$,
3. He incorporates tx in his block $t+1$ as $\mathsf{tx}' = [\texttt{INIT}, (i, \mathsf{upk}_i, \mathsf{tpk}_i)] = [\texttt{INIT}, \mathsf{FPK}_i]$.

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 $\mathsf{addr}_i = H(\mathsf{FPK}_i) = H(i, \mathsf{upk}_i, \mathsf{tpk})$, 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.

Transfering 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:

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

A miner working on block $t+1$ who receives this `SPEND` transaction, proceeds as follows:

1. He checks that $v \leq v'$ and verifies the proof $\pi_{i,t}$ that user i has v' coins via $\mathsf{VC.VerifyPos}(\mathsf{vrk}, d_t, (\mathsf{addr}_i|v'), i, \pi_{i,t})$, where $\mathsf{addr}_i = H(\mathsf{FPK}_i)$. (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 $d_{t+1} = \mathsf{VC.UpdateComm}(d_{t+1}, -v, i, \mathsf{upk}_i)$, (Note that this only sets the lower order bits of v_i corresponding to bal_i, not touching the higher order bits for addr_i.)
3. He updates j's balance in block $t+1$ 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 transactions of the form:

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

(We are ignoring `INIT` transactions, for now.) To validate this block, the miner (who has d_t) proceeds in three steps:

Step 1: Check Balances. First, for each tx, he checks that $v \leq v'$ 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 $d' = d_{t+1}$. (Finally, the miner can similarly verify and "apply" `INIT` transactions.)

Step 3: Updated Proofs, If Any. If the miner was racing to build block $t+1$ and lost, the miner can start mining block $t+2$ by "moving over" the `SPEND` transactions from his unmined block $t+1$. For this, he has to update 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.

Catching Up With New Blocks. Consider a user i who has processed the blockchain 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 $\pi_{i,t+1} = \mathsf{VC.UpdateProof}(\pi_{i,t+1}, (H(\mathsf{FPK}_j)|0), j, \mathsf{upk}_j)$. For each $[\texttt{SPEND}, t, \mathsf{FPK}_j, k, \mathsf{upk}_k, v, \pi_{j,t}, v']$, she updates her proof twice: $\pi_{i,t+1} = \mathsf{VC.UpdateProof}(\pi_{i,t+1}, -v, j, \mathsf{upk}_j)$ and $\pi_{i,t+1} = \mathsf{VC.UpdateProof}(\pi_{i,t+1}, +v, k, \mathsf{upk}_k)$.

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

Since our aSVC proofs are aggregatable, miners can aggregate all b proofs in a block of b transactions into a single, constant-sized proof using $\mathsf{VC.AggregateProofs}$. 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, adding b group elements of overhead per block (see Section 4.2.3). Nonetheless, this is still an improvement over Edrax, which had $O(b \log n)$ overhead.

Our smaller proofs are also faster to update, taking $O(1)$ time rather than $O(\log n)$. While verifying an aggregated proof 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 cryptographic pairings rather than b. This makes validating new blocks much faster.

4.2.2 Reducing Miner Storage Using Verifiable Update Keys

Recall that miners need update keys to update the digest when processing and validating `INIT` and `SPEND` transactions. Importantly, miners should validate an update key before using it. Otherwise, updating a digest with an incorrect update key will corrupt that digest, leading to a denial of service attack. To address this, Edrax miners store all $O(n)$ update keys, which makes validating any update key trivial. Alternatively, Edrax can outsource update keys to an

untrusted third party (e.g., via a Merkle tree) and miners can verifiably fetch them on demand. Unfortunately, this would require interaction *during block proposal and block validation*, which we believe is unacceptable.

Our design avoids the $O(n)$ storage and (most of) the interaction by outsourcing the update keys, but in a different fashion than Edrax. Specifically, since our update keys are verifiable, we do not need the overhead of Merkle tree-based authentication. To handle `SPEND` transactions, miners simply verify the update keys included in the transaction via $\mathsf{VC.VerifyUPK}$. In contrast, in Edrax, miners either have to store all update keys or verify them by asking for a Merkle proof from third parties, which requires interaction during block proposal and validation. Alternatively, each Edrax user can include Merkle proofs in their transaction, but this increases transaction size and makes transacting interactive, which Edrax is designed to avoid.

Furthermore, for `INIT` transactions, miners can fetch (in the background) a running window of the next k update keys needed for the next k `INIT` transactions. Importantly, this does not require any interaction during the block proposal, as the update keys are fetched in the background by upper-bounding the number of `INIT` transactions expected in the near future. This background fetching could also be implemented in Edrax, but with additional overhead from Merkle proofs. (Unless the Edrax update keys are made verifiable too, which seems possible.)

4.2.3 Smaller Update Keys

Recall that upk_i contains $a_i = g^{A(\tau)/(X-\omega^i)}$ and $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{X-\omega^i}}$ in our aSVC. We observe that 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 above, miners might actually need u_i (e.g., when a subset of i's pending transactions get included in block t and the remaining transactions for block $t+1$ must have their proofs updated). 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 $\mathbf{v}$. One way to address this is to use a hidden-order based VC which supports an unbounded number of elements. However, that would negatively impact performance. Yet another way, is to develop authenticated dictionaries *with scalable updates*. This is left as future work.

For bounded VCs such as ours, we address this by adding a cost to user registrations. A simple solution would be to register users by transferring a minimum amount of coins to their soon-to-be-registered TPKs via a different kind of `SPEND` transaction. In other words, `INIT` transactions should be more like `SPEND` transactions that send coins to a soon-to-be-registered user j. Thus, we propose a new `INITSPEND` transaction type that does exactly this:

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

User i would sign this `INITSPEND` transaction using her tpk_i, similar to a `SPEND` transaction. Miners processing this transaction would (1) first register a new user j with traditional public key tpk and (2) transfer v coins to j.

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 to be the case if new miners are to be able to join without "permission" from other miners or users. As a result, DoS attacks are severely limited, since malicious miners can only register a new account per block (which is already the case in UTXO-based cryptocurrencies [Nak08]). Furthermore, transaction fees and/or additional proof-of-work can also severely limit the frequency of `INITSPEND` transactions.

Limitations. This approach has the unfortunate side-effect of allowing user j to register multiple accounts under the same tpk. Furthermore, user j might do so accidentally, as he distributes his tpk instead of his FPK_j to other users to pay her. We believe this issue could be addressed either through a careful user interface design or by ensuring that each tpk is only registered once, perhaps via a *cryptographic accumulator* [BdM94, Ngu05] built over all TPKs. Certainly, this issue could be side-stepped if the VC is replaced with an authenticated dictionary, which does not require `INIT` transactions. We leave this as future work.

4.2.5 Minting Coins and Transaction Fees.

Support for minting new coins can be easily added by introducing a new `MINT` transaction type:

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

Here, i is the miner's user account number and v is the amount of newly minted coins. (Note that miners, just like users, must register using `INIT` transactions if they are to receive block rewards.) When this `MINT` transaction is processed by other users or miners, they update their digest d_t using $\mathsf{VC.UpdateComm}(d_t, +v, i, \mathsf{upk}_i)$. (In addition, users also update their proofs.) To support transaction fees, we can extend the `SPEND` transaction format to include a fee, which is then added with 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.)

This can be addressed by having two VCs rather than one: one VC for mapping i to addr_i and another for mapping i to $(c_i | \mathsf{bal}_i)$. Our key observation is that the two VCs should use different n-SDH params, one with base g and another with base h, such that $\mathsf{DiscreteLog}_g(h)$ is unknown. This would allow aggregating the VCs, their proofs and their update keys, so as to introduce zero computational and communication overhead in our stateless cryptocurrency.

The security of this scheme should naturally follow from the information-theoretically hiding flavor of KZG commitments [KZG10], which commits to $\phi(X)$ as $g^{\phi(\tau)}h^{r(\tau)}$ in a similar fashion. However, we leave investigating the details of such a 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. Asymptotically, 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 [CPZ18, TCZ+20], 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)*, which is a *subvector commitment (SVC)* scheme that supports aggregation and supports updating proofs and commitments. Then, we gave an efficient aSVC construction based on KZG commitments to Lagrange polynomials, which outperforms previous work. Lastly, we used our aSVC to build an efficient stateless cryptocurrency in the account model which outperforms previous work.

Future Work. Our work leaves open interesting challenges in stateless cryptocurrencies. First, the ability to aggregate update keys would further reduce block size. Second, unbounded proof aggregation [CFG+20] could be used to further "compress" stateless blockchains. Third, authenticated dictionaries *with scalable updates* would remove the limitation on the number of users and eliminate DoS attacks on user registration.

References

BB08. Dan Boneh and Xavier Boyen. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *Journal of Cryptology*, 21(2):149–177, 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 Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 561–586, Cham, 2019. Springer International Publishing.

BCG+14. E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, May 2014.

BCG+15. E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304, May 2015.

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.

BGG19. Sean Bowe, Ariel Gabizon, and Matthew D. Green. A Multi-party Protocol for Constructing the Public Parameters of the Pinocchio zk-SNARK. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 64–77, Berlin, Heidelberg, 2019. Springer Berlin Heidelberg.

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

BGV11. Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable Delegation of Computation over Large Datasets. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 111–131, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

BMRS20. Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized Cryptocurrency at Scale. Cryptology ePrint Archive, Report 2020/352, 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 Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 276–294, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

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

But16. Vitalik Buterin. Thoughts on UTXOs by Vitalik Buterin, Co-Founder of Ethereum. Medium.com, 2016. https://medium.com/@ConsenSys/thoughts-on-utxo-by-vitalik-buterin-2bb782c67e53.

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/print, 2020.

CDHK15. Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and Modular Anonymous Credentials: Definitions and Practical Constructions. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 262–288, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

CF13. Dario Catalano and Dario Fiore. Vector Commitments and Their Applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography – PKC 2013*, pages 55–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

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

CFM08. Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-Knowledge Sets with Short Proofs. In Nigel Smart, editor, *EUROCRYPT'08*, pages 433–450, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

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

Cor16. Corda. Rationale for and tradeoffs in adopting a UTXO-style model. Corda Blog, 2016. https://www.corda.net/blog/rationale-for-and-tradeoffs-in-adopting-a-utxo-style-model/.

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

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

Eth17. Ethereum Community. Design rationale: Blockchain-level protocol: Account and not utxos. Ethereum Wiki, 2017. https://github.com/ethereum/wiki/wiki/Design-Rationale#accounts-and-not-utxos.

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

Goy07. Vipul Goyal. Reducing Trust in the PKG in Identity Based Cryptosystems. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007*, pages 430–447, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

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

Jou00. Antoine Joux. A One Round Protocol for Tripartite Diffie–Hellman. In Wieb Bosma, editor, *Algorithmic Number Theory*, pages 385–393, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

KR13. Markulf Kohlweiss and Alfredo Rial. Optimally Private Access Control. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, page 37–48, New York, NY, USA, 2013. Association for Computing Machinery.

KZG10. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In Masayuki Abe, editor, *ASIACRYPT '10*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

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.

LM18. Russell W.F. Lai and Giulio Malavolta. Subvector Commitments with Application to Succinct Arguments. Cryptology ePrint Archive, Report 2018/705, 2018. `https://eprint.iacr.org/2018/705`.

LM19. Russell W. F. Lai and Giulio Malavolta. Subvector Commitments with Application to Succinct Arguments. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 530–560, Cham, 2019. Springer International Publishing.

LRY16. Benoît Libert, Somindu C. Ramanna, and Moti Yung. Functional Commitment Schemes: From Polynomial Commitments to Pairing-Based Accumulators from Simple Assumptions. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

LY10. Benoît Libert and Moti Yung. Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs. In Daniele Micciancio, editor, *Theory of Cryptography*, pages 499–517, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

Mil12. Andrew Miller. Storing UTXOs in a balanced Merkle tree (zero-trust nodes with O(1)-storage). BitcoinTalk Forums, 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 *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 80–89, New York, NY, USA, 1991. ACM.

Nak08. Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2008. Accessed: 2019-04-12.

Ngu05. Lan Nguyen. Accumulators from Bilinear Pairings and Applications. In Alfred Menezes, editor, *CT-RSA '05*, pages 275–292, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

Pat17. Patrick Dai. Why Qtum Choose UTXO Model and the Benefits. 8BTC, 2017. `http://news.8btc.com/why-qtum-choose-utxo-model-and-the-benefits`.

PST13. Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of Correct Computation. In Amit Sahai, editor, *Theory of Cryptography*, pages 222–242, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

RMCI17. Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 376–392, Cham, 2017. Springer International Publishing.

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 *2020 IEEE Symposium on Security and Privacy (SP)*, 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, New York, NY, USA, 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, New York, NY, USA, 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, New York, NY, USA, 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.

Yan16. Yangrui. Utxo model vs. account/balance model (forum thread). StackExchange Bitcoin, 2016. https://bitcoin.stackexchange.com/questions/49853/utxo-model-vs-account-balance-model.

Zah18. Joachim Zahnentferner. Chimeric Ledgers: Translating and Unifying UTXO-based and Account-based Cryptocurrencies. Cryptology ePrint Archive, Report 2018/262, 2018. https://eprint.iacr.org/2018/262.

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 $(X - \omega^i)$ you can verify that:

$$g(x) = (\omega^i)^0 X^{n-1} + (\omega^i)^1 X^{n-2} + (\omega^i)^2 X^{n-3} + \cdots + (\omega^i)^{n-2} X^1 + (\omega^i)^{n-1} X^0 \tag{18}$$

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

$$\begin{aligned} A'(\omega^i) = g(\omega^i) &= (\omega^i)^0 \omega^{i(n-1)} + (\omega^i)^1 \omega^{i(n-2)} + (\omega^i)^2 \omega^{i(n-3)} + \cdots + (\omega^i)^{n-2} \omega^{i\cdot 1} + (\omega^i)^{n-1} \omega^{i\cdot 0} && (19)\\ &= n\omega^{i(n-1)} = n(\omega^{i\cdot n - i}) = n\omega^{-i} && (20)\end{aligned}$$

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 3.4.5, we argued all u_i's can be computed in $O(n \log n)$ time. Here, we give intuition on the quotients $q_i(X)$ (see Equation (13)) and prove they are correct.

First, let us look at the quotient $q_1(X)$ obtained when dividing $\psi_1(X)$ by $X - \omega^1$, assuming $n = 8$:

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

We have to show that:

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

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

$$\begin{aligned} q_i(X)(X-\omega^i) + \psi_i(\omega_i) &= q_i(X)(X-\omega^i) + n\omega^{-i} && (22)\\ &= n\omega^{-i} + (X-\omega^i) \sum_{j\in[0,n-2]} (j+1)(\omega^i)^j X^{(n-2)-j} && (23)\\ &= n\omega^{-i} + X \cdot \sum_{j\in[0,n-2]} (j+1)(\omega^i)^j X^{(n-2)-j} - \omega^i \cdot \sum_{j\in[0,n-2]} (j+1)(\omega^i)^j X^{(n-2)-j} && (24)\\ &= n\omega^{-i} + \sum_{j\in[0,n-2]} (j+1)(\omega^i)^j X^{(n-1)-j} - \sum_{j\in[0,n-2]} (j+1)(\omega^i)^{j+1} X^{(n-2)-j} && (25)\\ &= n\omega^{-i} + \sum_{j\in[0,n-2]} (j+1)\omega^{ij} X^{(n-1)-j} - \sum_{j\in[1,n-1]} j\omega^{ij} X^{(n-2)-(j-1)} && (26)\end{aligned}$$

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

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

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

$$= \sum_{j\in[0,n-2]} \omega^{ij}X^{(n-1)-j} + \omega^{-i}X^0 \tag{30}$$

$$= \sum_{j\in[0,n-2]} \omega^{ij}X^{(n-1)-j} + \omega^{in-i}X^0 \tag{31}$$

$$= \sum_{j\in[0,n-2]} \omega^{ij}X^{(n-1)-j} + \omega^{i(n-1)}X^0 \tag{32}$$

$$= \sum_{j\in[0,n)} \omega^{ij}X^{(n-1)-j} \tag{33}$$

$$= \psi_i(X) \tag{34}$$

C Security Proofs

C.1 KZG Batch Opening Binding (Re)definition

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

Definition 9 (Batch Opening Binding). $\forall$ *adversaries* $\mathcal{A}$ *running in time* $\mathsf{poly}(\lambda)$*:*

$$\Pr\left[\begin{array}{c} pp \leftarrow \mathsf{KZG.Setup}(1^\lambda, n), \\ c, I, J, v_I(X), v_J(X), \pi_I, \pi_J \leftarrow \mathcal{A}(1^\lambda) : \\ \mathsf{KZG.VerifyEvalBatch}(pp, c, I, \pi_I, v_I(X)) = T \ \wedge \\ \mathsf{KZG.VerifyEvalBatch}(pp, c, J, \pi_J, v_J(X)) = T \ \wedge \\ \exists k \in I \cap J, \ \textit{such that } v_I(k) \neq v_J(k) \end{array}\right] \leq \mathsf{negl}(\lambda) \tag{35}$$

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

$$e(c, g) = e(\pi_I, g^{A_I(\tau)})e(g^{v_I(\tau)}, g) \tag{36}$$

$$e(c, g) = e(\pi_J, g^{A_J(\tau)})e(g^{v_J(\tau)}, g) \tag{37}$$

Divide the top equation by the bottom one to get:

$$\mathbf{1}_T = \frac{e(g^{v_I(\tau)}, g)}{e(g^{v_J(\tau)}, g)} \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \Leftrightarrow \tag{38}$$

$$\mathbf{1}_T = e(g^{v_I(\tau)-v_J(\tau)}, g) \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \Leftrightarrow \tag{39}$$

$$e(g^{v_J(\tau)-v_I(\tau)}, g) = \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \tag{40}$$

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, $v_J(X) = q_J(X)(X - k) + v'_k$.

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

$$e(g^{(\tau-k)(q_J(\tau)-q_I(\tau))+v'_k-v_k}, g) = \frac{e(\pi_I, g^{A_I(\tau)})}{e(\pi_J, g^{A_J(\tau)})} \Leftrightarrow \tag{42}$$

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

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

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

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

$$e(g^{q_J(\tau)-q_I(\tau)}, g)e(g, g)^{\frac{v'_k-v_k}{\tau-k}} = \frac{e(\pi_I, g^{a_I(\tau)})}{e(\pi_J, g^{a_J(\tau)})} \Leftrightarrow \tag{46}$$

$$e(g, g)^{\frac{v'_k-v_k}{\tau-k}} = \frac{e(\pi_I, g^{a_I(\tau)})}{e(\pi_J, g^{a_J(\tau)})e(g^{q_J(\tau)-q_I(\tau)}, g)} \Leftrightarrow \tag{47}$$

$$e(g, g)^{\frac{1}{\tau-k}} = \left(\frac{e(\pi_I, g^{a_I(\tau)})}{e(\pi_J, g^{a_J(\tau)})e(g^{q_J(\tau)-q_I(\tau)}, g)}\right)^{\frac{1}{v'_k-v_k}} \tag{48}$$

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, 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 7. 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 proofs pass verification, the following pairing equations hold:

$$e(a_i, g^\tau / g^{\omega^i}) = e(a, g) \tag{49}$$

$$e(a'_i, g^\tau / g^{\omega^i}) = e(a, g) \tag{50}$$

Thus, it follows that:

$$e(a_i, g^\tau / g^{\omega^i}) = e(a'_i, g^\tau / g^{\omega^i}) \Leftrightarrow \tag{51}$$

$$e(a_i, g) = e(a'_i, g) \Leftrightarrow \tag{52}$$

$$a_i = a'_i \tag{53}$$

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 proofs pass verification, the following pairing equations hold:

$$e(\ell_i / g^1, g) = e(u_i, g^\tau / g^{\omega^i}) \tag{54}$$

$$e(\ell_i / g^1, g) = e(u'_i, g^\tau / g^{\omega^i}) \tag{55}$$

$$\tag{56}$$

Thus, it follows that:

$$e(u_i, g^\tau / g^{\omega^i}) = e(u'_i, g^\tau / g^{\omega^i}) \tag{57}$$

$$e(u_i, g) = e(u'_i, g) \Leftrightarrow \tag{58}$$

$$u_i = u'_i \tag{59}$$

Contradiction.

Table 3. Asymptotic comparison of our aSVC with other (aS)VCs. n is the vector size and b is the subvector size. See Appendix D for a detailed explanation. The complexities in the table are *asymptotic* in terms of number of pairings, exponentiations and field operations. *"Proof upd."* is the time to update *one* proof for *one* single vector element after a change to *one* vector element. *"Com. upd."* is the time to update the commitment after a change to one vector element. We include complexities for computing a VC proof for v_i and a size-b subvector proof (in *"Prove one v_i"* and in *"Prove subv. $(v_i)_{i\in I}$"*, respectively) and for committing to a size-n vector (in *"Com."*), even though these features are not needed in a stateless cryptocurrency.

(aS)VC scheme	$\lvert\mathsf{prk}\rvert$	$\lvert\mathsf{vrk}\rvert$	$\lvert\mathsf{upk}_i\rvert$	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 $(v_i)_{i\in[n]}$
$\mathrm{LM}_{\mathsf{cdh}}$ [CF13, LM19]	n^2	n	n	n	1	1	n	1	1	bn	b	×	n^2
KZG [KZG10]	n	b	×	$n\lg^2 n$	×	1	n	1	×	$b\lg^2 b + n\lg n$	$b\lg^2 b$	×	n^2
$\mathrm{KZG}_{\mathsf{Lagr}}$ [CDHK15]	n	n	1	$n\lg^2 n$	1	1	n	1	×	$n\lg^2 n$	$b\lg^2 b$	×	n^2
CPZ [CPZ18]	n	$\lg n$	$\lg n$	n	1	$\lg n$	n	$\lg n$	$\lg n$	×	×	×	$n\lg n$
TCZ [Tom20]	n	$\lg n + b$	$\lg n$	$n\lg n$	1	$\lg n$	$n\lg n$	$\lg n$	$\lg n$	$b\lg^2 b + n\lg n$	$b\lg^2 b$	×	$n\lg n$
LY [GRWZ20, LY10]	n	n	n	n	1	1	n	1	1	bn	b	b	n^2
Our aSVC	n	b	1	n	1	1	n	1	1	$b\lg^2 b + n\lg n$	$b\lg^2 b$	$b\lg^2 b$	$n\lg n$
Our $\mathsf{aSVC}_{\mathsf{precomp}}$	n	b	1	$n\log n$	1	1	1	1	1	$b\lg^2 b$	$b\lg^2 b$	$b\lg^2 b$	$n\lg n$

D Complexities of VCs in Table 2

We explain the complexities we gave in Table 2, which we re-include above for convenience as Table 3. Each VC scheme has its own sub-section where we explain its complexities, ultimately surveying the full scheme itself. 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 $\mathrm{LM}_{\mathsf{cdh}}$ [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 $\mathsf{prk} = (h_{i,j} = g^{z_i \cdot z_j})_{i,j\in[0,n),i\neq j}$ and is $O(n^2)$ sized. The verification key is $\mathsf{vrk} = (h_i = g^{z_i})_{i\in[0,n)}$ 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:

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

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 by computing two pairings:

$$e(C/h_i^{v_i}, h_i) = e(\pi_i, g) \tag{61}$$

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:

$$\pi_I = \prod_{i\in I}\prod_{j\in[0,n)\setminus I} h_{i,j}^{v_j} = \prod_{i\in I}\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}\right)^{z_i} = \prod_{i\in I}\pi_i^* \tag{62}$$

As intuition, note that the inner product π_i^* is very similar to a proof π_i for v_i but for a vector $(v_j)_{j\in[0,n)-I}$. The proof can be computed from the $h_{i,j}$'s in the prk with $O((n - |I|)|I|)$ exponentiations (because each π_i^* can be computed in

$O(n-|I|)$ exponentiations). A subvector proof π_I can be verified using $(h_i)_{i\in I}$ from vrk by checking in $O(|I|)$ time if:

$$e\left(c/\prod_{j\in I} h_j^{v_j}, \prod_{i\in I} h_i\right) = e(\pi_I, g) \Leftrightarrow \tag{63}$$

$$e\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}, \prod_{i\in I} g^{z_i}\right) = e\left(\prod_{i\in I}\prod_{j\in[0,n)\setminus I} h_{i,j}^{v_j}, g\right) \tag{64}$$

$$e\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}, g^{\sum_{i\in I} z_i}\right) = e\left(\prod_{i\in I}\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}\right)^{z_i}, g\right) \tag{65}$$

$$e\left(\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}\right)^{\sum_{i\in I} z_i}\right) = e\left(\left(\prod_{j\in[0,n)\setminus I} h_j^{v_j}\right)^{\sum_{i\in I} z_i}, g\right) \tag{66}$$

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]

In their paper on polynomial commitment schemes [KZG10, Sec], Kate, Zaverucha and Goldberg also discuss using their scheme 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 $\mathsf{vrk} = (g, (g^{\tau^i})_{i\in|I|})$, where $|I|$ is the size of the largest subvector whose proof the verifier should be able to check, and is thus $O(|I|)$-sized. There is no support for updating commitments and proofs using update keys, although adding it is possible.

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.

Proofs for a v_i. A proof for v_i is:

$$\pi_i = g^{\frac{\phi(\tau)-v_i}{\tau-i}} = g^{q_i(\tau)} \tag{67}$$

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:

$$e(c/g^{v_i}, g) = e(\pi_i, g^\tau/g^i) \tag{68}$$

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:

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

Here, $R_I(X)$ is interpolated in $O(|I|\log^2|I|)$ 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 $O(|I|\log^2|I|)$ field operations via a *subproduct tree* [vzGG13b]. 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 $O(n-|I|)$ exponentiations (because $\deg(q_I) \le \max(\deg(\phi), \deg(R_I)) - \deg(A_I) = (n-1)-|I|$). Thus, the overall subvector proving time is $O(n\log n + |I|\log^2|I|)$.

To verify a subvector proof π_I, first, the verifier must recompute $R_I(X)$ and $A_I(X)$ in $O(|I|\log^2|I|)$ field operations. Then, the verifier uses $(g^{\tau^i})_{i\in|I|}$ from the vrk to compute KZG commitments $g^{R_I(\tau)}$ and $g^{A_I(\tau)}$ in $O(|I|)$ exponentiations. Finally, the verifier checks using two pairings 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 $O(|I|\log^2|I|)$ 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 $\mathsf{KZG}_{\mathsf{Lagr}}$ [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 $\mathcal{L}_i(X) = \prod_{j\in[0,n],j\neq i}\frac{X-j}{i-j}$ are defined over the points $[0,n]$, not $[n]$.) The verification key is $\mathsf{vrk} = (g, (g^{\mathcal{L}_i(\tau)})_{i\in[n]}, (g^{\tau^i})_{i\in[0,|I|+1]})$, where $|I|$ is the maximum size of a subvector in a subvector proof, and is thus $O(n)$-sized. There is no support for updating commitments and proofs using update keys, although adding it is possible.

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 $v_0 = 0$. The commitment can be computed with $O(n)$ exponentiations, given the Lagrange commitments and $g^{P(\tau)}$ from prk. However, since $\phi(X)$ is needed to compute proofs, this adds another $O(n\log^2 n)$ field operations. 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:

$$\pi_i = g^{\frac{(\phi(\tau)+r\cdot p(\tau))-v_i\mathcal{L}_i(\tau)}{\tau-i}} = g^{q_i(\tau)} \tag{71}$$

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:

$$e\left(c/\left(g^{\mathcal{L}_i(\tau)}\right)^{v_i}, g\right) = e(\pi_i, g^\tau/g^i) \tag{72}$$

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:

$$\pi_I = g^{\frac{\phi(\tau)+r\cdot P(\tau)-R_I(\tau)}{A_I(\tau)}} = g^{q_I(\tau)} \tag{73}$$

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 $O(|I|\log^2|I|)$ field operations via a *subproduct tree* [vzGG13b]. Given g^{τ^i}'s from the prk, committing to $q_I(X)$ takes $O(n-|I|)$ exponentiations (because $\deg(q_I) \leq \max(\deg(\phi), \deg(P), \deg(R_I)) - \deg(A_I) = n-|I|$). 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 $O(|I|)$ 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 $O(|I|\log^2|I|)$ field operations using a *subproduct tree* [vzGG13b]. Then, the verifier uses $(g^{\tau^i})_{i\in[0,|I|+1]}$ from the vrk to compute a KZG commitment to $g^{A_I(\tau)}$ in $O(|I|)$ exponentiations. Finally, the verifier checks using two pairings if:

$$e(c/g^{R_I(\tau)}, g) = e(\pi_I, g^{A_I(\tau)}) \tag{74}$$

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

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 neatly summarizes its performance, we mostly 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 it is possible. The key idea is to notice that computing n proofs separately actually repeats a lot of work. If we avoid re-doing previously-done work, all proofs can be computed in $O(n \log n)$ time. We reflect this in Table 2.

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 describes a full VC based on this technique from KZG commitments [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 $\mathsf{vrk} = (g, (g^{\tau^{2^i}})_{i\in[\lfloor \log_2(n-1)\rfloor]}, (g^{\tau^i})_{i\in|I|})$, where $|I|$ is the size of the largest subvector whose proof the verifier should be able to check, and is thus $O(|I|)$-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].

Commitment. A commitment is $c = g^{\phi(\tau)}$ where $\phi(\omega^i) = v_i, \forall i \in [0, n)$ so that $\phi(X)$ can be computed with $O(n \log n)$ field operations via an inverse Discrete Fourier Transform (DFT) [CLRS09, Ch 30.2] and $O(n)$ 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:

$$\pi_i = (g^{q_w(\tau)})_{w\in[1+\lfloor \log(n-1)\rfloor]} \tag{75}$$

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, ensuring 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:

$$e(c/g^{v_i}, g) = \prod_{w\in[1+\lfloor \log(n-1)\rfloor]} e(g^{q_w(\tau)}, g^{a_w(\tau)}) \tag{76}$$

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 $O(n \log n + |I| \log^2 |I|)$ and the subvector proof verification time is $O(|I| \log^2 |I|)$ 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 LY [GRWZ20, LY10] or "Pointproofs"

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 $e : \mathbb{G}_1 \times G_2 \to G_T$. Let g_1, g_2, g_T be generators of $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ respectively. The proving key $\mathsf{prk} = ((g_1^{\alpha^i})_{i\in[0,2n]\setminus\{n-1\}}, (g_2^{\alpha^i})_{i\in[0,n]}, g_T^{\alpha^{n+1}})$. Note that $g_1^{\alpha^{n+1}}$ is "missing" from the proving key, which is essential for security. The verification key $\mathsf{vrk} = ((g_2^{\alpha^i})_{i\in[1,n]}, g_T^{\alpha^{n+1}})$. The ith update key $\mathsf{upk}_i = g_1^{\alpha^i}$. 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 using 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:

$$\pi_i = g_1^{\sum_{j\in[n]} v_j\alpha^{j+(n+1)-i}} \tag{77}$$

$$= (c/g_1^{v_i\alpha^i})^{\alpha^{(n+1)-i}} \tag{78}$$

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:

$$e(c, g_2^{\alpha^{(n+1)-i}}) = e(\pi_i, g_2)\left(g_T^{\alpha^{n+1}}\right)^{v_i} \tag{79}$$

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

Subvector Proofs for $(v_i)_{i\in I}$ A $O(1)$-sized subvector proof for $\mathbf{v}_I$ is a product of individual, randomized proofs $\pi_i, \forall i \in I$. First, all $|I|$ proofs π_i are computed in $O(|I|n)$ 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:

$$\pi_I = \prod_{i\in I} \pi_i^{t_i} \tag{80}$$

Thus, the overall time to compute a subvector proof is $O(|I|n)$. The subvector proof can be verified in $O(|I|)$ time using $(g_2^{\alpha^{(n+1)-i}})_{i\in I}$ from vrk as:

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

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

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

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

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

$$= e\left(\prod_{i\in I}\left(\pi_i \cdot g_1^{\alpha^{n+1} v_i}\right)^{t_i}, g_2\right) \tag{86}$$

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

$$e\left(c, g_2^{\sum_{i\in I} t_i\alpha^{(n+1)-i}}\right) = e\left(\prod_{i\in I}\left((c/g_1^{v_i\alpha^i})^{\alpha^{(n+1)-i}} \cdot g_1^{\alpha^{n+1}v_i}\right)^{t_i}, g_2\right) \Leftrightarrow \tag{87}$$

$$= e\left(\prod_{i\in I}\left((c/g_1^{v_i\alpha^i}) \cdot g_1^{\frac{\alpha^{n+1}v_i}{\alpha^{(n+1)-i}}}\right)^{t_i\alpha^{(n+1)-i}}, g_2\right) \Leftrightarrow \tag{88}$$

$$= e\left(\prod_{i\in I}\left((c/g_1^{v_i\alpha^i}) \cdot g_1^{v_i\alpha^i}\right)^{t_i\alpha^{(n+1)-i}}, g_2\right) \Leftrightarrow \tag{89}$$

$$= e\left(\prod_{i\in I} c^{t_i\alpha^{(n+1)-i}}, g_2\right) \Leftrightarrow \tag{90}$$

$$= e\left(c^{\sum_{i\in I} t_i\alpha^{(n+1)-i}}, g_2\right) \Leftrightarrow \tag{91}$$

$$= e\left(c, g_2^{\sum_{i\in I} t_i\alpha^{(n+1)-i}}\right) \tag{92}$$

Aggregating Proofs and Precomputing All Proofs. Recall that a subvector proof works only through aggregating individual proofs and that the cost of aggregation was computing $|I|$ hashes and an $O(|I|)$-sized multi-exponentiation. Thus, aggregation takes $O(|I|)$ time. Finally, 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 3.4.4, this section will be briefer than previous ones.

Public Parameters. The proving key, verification key and ith update key are $O(n), O(|I|)$ and $O(1)$-sized, respectively. Similar to Appendix D.5, n is assumed to be a power of two, and $\mathcal{L}_i(X) = \prod_{j\in[0,n),j\neq i} \frac{X-\omega^j}{\omega^i-\omega^j}$ 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:

$$\pi_i = g^{\frac{\phi(\tau)-v_i}{\tau-\omega^i}} = g^{q_i(\tau)} \tag{93}$$

However, note that:

$$\frac{\phi(\tau)-\phi(\omega^i)}{\tau-\omega^i} = \frac{\sum_{j\in[0,n)} \mathcal{L}_j(\tau)v_j - v_i}{\tau-\omega^i} \tag{94}$$

$$= \frac{\sum_{j\in[0,n)\setminus\{i\}} \mathcal{L}_j(\tau)v_j}{\tau-\omega^i} + \frac{\mathcal{L}_i(\tau)v_i - v_i}{\tau-\omega^i} \tag{95}$$

$$= \sum_{j\in[0,n)\setminus\{i\}} v_j\frac{\mathcal{L}_j(\tau)}{\tau-\omega^i} + v_i\frac{\mathcal{L}_i(\tau)-1}{\tau-\omega^i} \tag{96}$$

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:

$$\pi_i = g^{q_i(\tau)} = \prod_{j\in[0,n)\setminus\{i\}} (u_{i,j})^{v_j} \cdot (u_i)^{v_i} \tag{97}$$

Table 4. 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 they are much faster, for the most part they can be safely 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.

(aS)VC scheme	$\lvert\mathsf{prk}\rvert$	$\lvert\mathsf{vrk}\rvert$	$\lvert\mathsf{upk}_i\rvert$	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	$N \lg N$	$\ell \lg N$	$\ell \lg N$	$N \lg N$	$\ell \lg N + \lambda$	×	$N \lg N$	$b\ell \lg N + \lambda$	$b\ell \lg N$	$N \lg^2 N$
CFG^1_ℓ [CFG+20]	1	1	×	$N \lg N$	×	1	$N \lg N$	$\ell \lg N + \lambda$	×	$\ell(n-b) \lg N$	$b\ell \lg N + \lambda$	$b \lg b \lg N$	$N \lg^2 N$
CFG^2_ℓ [LM19, CFG+20]	1	1	×	$N \lg n$	×	1	$N \lg n$	ℓ	×	$\ell(n-b) \lg (n-b)$	$\ell b \lg b$	$\ell b \lg^2 b$	$N \lg^2 n$
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:

$$e(c/g^{v_i}, g) = e(\pi_i, g^\tau / g^{\omega^i}) \tag{98}$$

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 $O(n \log n + |I| \log^2 |I|)$ and the subvector proof verification time is $O(|I| \log^2 |I|)$ 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 $O(|I| \log^2 |I|)$ field operations, (2) computing its derivative $A_I'(X)$ in $O(|I|)$ field operations and (3) evaluating $A_I'(X)$ at all $(\omega^i)_{i\in I}$ using a multipoint evaluation in $O(|I| \log^2 |I|)$ field operations [vzGG13b]. Then, the subvector proof can be aggregated with $O(|I|)$ exponentiations as:

$$\pi_I = \prod_{i\in I} \pi_i^{c_i} \tag{99}$$

Thus, aggregation takes $O(|I| \log^2 |I|)$ 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 Prove Time. 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 $O(|I| \log^2 |I|)$ 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 4. 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 count operations in 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, $\left(g^{\frac{1}{e_1}}\right)^b \left(g^{\frac{1}{e_2}}\right)^a = g^{\frac{be_2}{e_1 e_2}} g^{\frac{ae_1}{e_1 e_2}} = g^{\frac{ae_1 + be_2}{e_1 e_2}} = g^{\frac{1}{e_1 e_2}}$. 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 $\mathbf{BBF}_\ell$ [BBF18, 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,vrk consists of g. This scheme does not support "fixed" update keys compatible with our definitions. Instead, the ith update key w.r.t. a commitment c is "dynamic" and consists of a VC proof for v_i that verifies against c. This does not appear to be problematic as our $\mathsf{VC.VerifyUPK}$ definition (see Section 3.1) can be updated to verify the upk against the commitment c.

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 $(p_{(i-1)\cdot\ell}, p_{(i-1)\cdot\ell+1}, \ldots, p_{(i-1)\cdot\ell+(\ell-1)})$. 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:

$$c = g^{\prod_{i \in [n]} \prod_{j \in [0,\ell-1]} v_{i,j} \cdot \left(p_{(i-1)\cdot\ell+j}\right)} \tag{100}$$

$$= g^{\prod_{i \in [n]} P_i} \tag{101}$$

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.)

Before discussing updating commitments, we must first discuss how a VC proof for v_i works.

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]):

$$\pi_i^{[1]} = g^{\prod_{j \in [n], j \neq i} P_j} = c^{1/P_i} \tag{102}$$

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. First, 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 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, unlike other VC schemes, the update key upk_i is set to be the VC proof π_i that verifies against 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 many updated elements $(v_i)_{i \in I}$ with upk_i's that all verify w.r.t. c, we have to take an additional step. First, we use $O(|I|)$ *Shamir tricks* [BBF18] on all the $\pi_i^{[1]}$'s to obtain the commitment $c^{1/\prod_{i \in I} P_i}$. This commitment does not have 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 $O(|I| \ell \log(\ell n))$ group operations. We assume the $O(|I|)$ Shamir tricks can be done in $O(|I|)$ 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 $O(|I| \ell \log(\ell n))$-sized $\prod_{i \in I} P_i$ and $\prod_{i \in I} Z_i$. This does not affect the proof size, but affects the proving time in two ways.

First, computing $\pi_i^{[1]}$ can be done faster in $O(\ell(n - |I|) \log(\ell n))$ group operations. However, this speedup is negated by an increase in the time to compute its associated PoE to $O(|I| \ell \log(\ell n))$ field and group operations. Second, computing $\pi_i^{[0]}$ maintains the same asymptotic complexity, since it is dominated by computing g^β, which remains just as expensive. However, $\pi_i^{[0]}$'s size will increase to $O(|I| \ell \log(\ell n))$, 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 $O(|I| \ell \log(\ell n))$. Similarly, the PoKE proofs will speed up verification of $\pi_i^{[0]}$ to $O(\lambda)$ group operations but the $O(|I| \ell \log(\ell n))$ 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 $\Omega(|I| \ell \log(\ell n))$ since it must read all $|I|$ 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, LM18, 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 $|I|$ proofs into an I-subvector proof takes $O(|I| \log |I| \log N)$ group operations [CFG+20, Sec 5.1, pg. 23]. For CFG^2_ℓ, this takes $O(\ell |I| \log^2 |I|)$ group operations [CFG+20, Sec 5.2, pg. 32].

Updating Proofs and Commitments. The paper does not discuss updating proofs and commitments, although this seems possible.

Precomputing All Proofs. Unfortunately, [CFG+20] does not address precomputing all proofs efficiently. Nonetheless, this seems possible in CFG^1_ℓ. We believe the time will be dominated by the complexity of computing all $N = \ell n$ RSA accumulator membership witnesses in $O(N \log^2 N)$ group operations. Furthermore, since CFG^2_ℓ supports disaggregation, all proofs can be computed efficiently using a disaggregation-based divide-and-conquer approach. We estimate this will take $O(\ell n \log^2 n)$ group operations.