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]

[1] VMware Research, Palo Alto, CA, USA
{alint,iabraham}@vmware.com
[2] Ethereum Foundation, Singapore, Singapore
{vitalik,justin,dankrad,dmitry.khovratovich}@ethereum.org

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

1 Introduction

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

Stateless cryptocurrencies have received increased attention [Dry19, RMCI17, CPZ18, BBF19, GRWZ20] due to several advantages. First, stateless cryptocurrencies eliminate hundreds of gigabytes of miner storage needed to validate blocks. Second, statelessness makes scaling consensus via *sharding* much easier, by allowing miners to efficiently switch from one shard to another [KJG+18].

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

C. Galdi and V. Kolesnikov (Eds.): SCN 2020, LNCS 12238, pp. 45–64, 2020.
https://doi.org/10.1007/978-3-030-57990-6_3

Third, statelessness lowers the barrier to entry for full nodes, resulting in a much more resilient, distributed cryptocurrency.

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

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

Our Contributions. In this paper, we formalize a new *aggregatable subvector commitment (aSVC)* notion that supports commitment updates, proof updates and aggregation of proofs into subvector proofs. Then, we construct an aSVC *with scalable updates* over pairing-friendly groups. Compared to other pairing-based VCs, our aSVC has constant-sized, aggregatable proofs that can be updated with constant-sized update keys (see Table 2). Furthermore, our aSVC supports computing all proofs in quasilinear time. We prove our aSVC secure under q-SBDH [Goy07] in the extended version of our paper [TAB+20].

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

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

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

1.1 Related Work

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

Boneh et al. [BBF19] instantiate VCs using hidden-order groups. They are the first to support aggregating multiple proofs (under certain conditions). They are also the first to have constant-sized public parameters, without the need to specialize them into $O(n)$-sized ones. However, their VC uses update hints (rather than keys), which is less suitable for stateless cryptocurrencies. Campanelli et al. [CFG+20] also formalize SVCs with a more powerful notion of *infinite (dis)aggregation* of proofs. In contrast, our aSVC only supports "one hop" aggregation and does not support disaggregation. They also formalize a notion of updatable, distributed VCs as Verified Decentralized Storage (VDS). However, their use of hidden-order groups leads to larger concrete proof sizes.

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

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

Stateless Cryptocurrencies. The concept of stateless validation appeared early in the cryptocurrency community [Mil12, Tod16, But17] and later on in the academic community [RMCI17, Dry19, CPZ18, BBF19, GRWZ20]. Initial proposals for UTXO-based cryptocurrencies used Merkle hash trees [Mil12, Tod16, Dry19, CPZ18]. In particular, Dryja [Dry19] gives a beautiful Merkle forest construction that significantly reduces communication. Boneh et al. [BBF19] further reduce communication by using RSA accumulators.

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

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

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

2 Preliminaries

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

Lagrange 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* in $O(n \log^2 n)$ time [vzGG13b] as $\phi(X) = \sum_{i \in [0,n)} \mathcal{L}_i(X) y_i$, where $\mathcal{L}_i(X) = \prod_{j \in [0,n), j \ne i} \frac{X - x_j}{x_i - x_j}$. Recall that a *Lagrange polynomial* $\mathcal{L}_i(X)$ has the property that $\mathcal{L}_i(x_i) = 1$ and $\mathcal{L}_i(x_j) = 0, \forall i, j \in [0, n)$ with $j \ne i$. Note that $\mathcal{L}_i(X)$ is defined in terms of the x_i's which, throughout this paper, will be either $(\omega^i)_{i \in [0,n)}$ or $(\omega^i)_{i \in I}, I \subset [0, n)$.

2.1 KZG Polynomial Commitments

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

Committing. Let $\phi(X)$ denote a polynomial of degree $d \le n$ 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 $e(C/g^y, g) = e(\pi, g^\tau/g^a) \Leftrightarrow e(g^{\phi(\tau)-y}, g) = e(g^{q(\tau)}, g^{\tau-a}) \Leftrightarrow \phi(\tau) - y = q(\tau)(\tau - a)$.

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

2.2 Account-Based Stateless Cryptocurrencies

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

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 Sect. 4.3.1.) Importantly, miners and users are *stateless*: they do not store the state, just its *digest* d_t at the latest block t they are aware of. Additionally, each user i stores a proof $\pi_{i,t}$ for their account balance that verifies against d_t.

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

Once the miner has a set V of valid transactions, he can use them to create the next block $t + 1$ and propose it. The miner obtains this new block's digest

d_{t+1} by "applying" all transactions in V to d_t. When other miners receive this new block $t + 1$, they can validate its transactions from V against d_t and check that the new digest d_{t+1} was produced correctly from d_t by "reapplying" all the transactions from V.

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

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

3 Aggregatable Subvector Commitment (aSVC) Schemes

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

3.1 aSVC API

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

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

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

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

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

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

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

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

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

3.2 aSVC Correctness and Security Definitions

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

Definition 2 (Correctness). *$\forall$ honestly generated $\mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_j)_{j \in [0,n)}$ via $\mathsf{VC.KeyGen}$, $\forall$ vectors $\mathbf{v} = (v_j)_{j \in [0,n)}$ with commitment c obtained via $\mathsf{VC.Commit}$ and, optionally, $\mathsf{VC.UpdateComm}$ calls, $\forall I \subseteq [0, n)$, if π_I is a (sub)vector proof for $\boldsymbol{v}_I = (v_i)_{i \in I}$ obtained via any valid interleaving of $\mathsf{VC.ProvePos}$, $\mathsf{VC.AggregateProofs}$ and $\mathsf{VC.UpdateProof}$, then $\mathsf{VC.VerifyPos}(\mathsf{vrk}, c, \boldsymbol{v}_I, I, \pi_I)$ returns true. Furthermore, $\mathsf{VC.VerifyUPK}(\mathsf{vrk}, i, \mathsf{upk}_i) = T, \forall i \in [0, n)$.*

Definition 3 (Update Key Uniqueness). *$\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$:*

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

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

Definition 4 (Position Binding Security). *$\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$, if $\mathbf{v}_I = (v_i)_{i \in I}$ and $\mathbf{v}'_J = (v'_j)_{j \in J}$, then:*

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

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

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

3.3 aSVC from KZG Commitments to Lagrange Polynomials

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

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 Sect. 2.1). Similar to previous work [CDHK15], the proving key includes commitments to all Lagrange polynomials $\ell_i=g^{\mathcal{L}_i(\tau)}$. Thus, we can compute $c=\prod_{i=1}^{n}(\ell_i)^{v_i}$ in $O(n)$ time without interpolating $\phi(X)$ and update it as $c'=c\cdot(\ell_i)^{\delta}$ after adding δ to v_i. Note that c' is just a commitment to an updated $\phi'(X)=\phi(X)+\delta\cdot\mathcal{L}_i(X)$.

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

3.4 Partial Fraction Decomposition

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

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

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

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

$A_I(X)$ can be computed in $O(|I|\log^2|I|)$ time [vzGG13b]. Its derivative, $A_I'(X)$, can be computed in $O(|I|)$ time and evaluated at all ω^i's in $O(|I|\log^2|I|)$ time [vzGG13b]. Thus, all c_i's can be computed in $O(|I|\log^2|I|)$ time. For the special case of $I=[0,n)$, we have $A_I(X)=A(X)=\prod_{i\in[0,n)}(X-\omega^i)=X^n-1$ and $A'(\omega^i)=n\omega^{-i}$ [TAB+20, Appendix A]. In this case, any c_i can be computed in $O(1)$ time.

3.4.1 Aggregating Proofs

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

Recall that π_i is a commitment to $q_i(X)=\frac{\phi(X)-v_i}{X-\omega^i}$ and π_I is a commitment to $q(X)=\frac{\phi(X)-R(X)}{A_I(X)}$, where $A_I(X)=\prod_{i\in I}(X-\omega^i)$ and $R(X)$ is interpolated such that $R(\omega^i)=v_i, \forall i\in I$. Our goal is to find coefficients $c_i\in\mathbb{Z}_p$ such that $q(X)=\sum_{i\in I}c_iq_i(X)$ and thus aggregate $\pi_I=\prod_{i\in I}\pi_i^{c_i}$. We observe that:

$$\begin{aligned}
q(X) &= \phi(X)\frac{1}{A_I(X)} - R(X)\frac{1}{A_I(X)} && (3)\\
&= \phi(X)\sum_{i\in I}\frac{1}{A_I'(\omega^i)(X-\omega^i)} - \left(A_I(X)\sum_{i\in I}\frac{v_i}{A_I'(\omega^i)(X-\omega^i)}\right)\cdot\frac{1}{A_I(X)} && (4)\\
&= \sum_{i\in I}\frac{\phi(X)}{A_I'(\omega^i)(X-\omega^i)} - \sum_{i\in I}\frac{v_i}{A_I'(\omega^i)(X-\omega^i)} = \sum_{i\in I}\frac{1}{A_I'(\omega^i)}\cdot\frac{\phi(X)-v_i}{X-\omega^i} && (5)\\
&= \sum_{i\in I}\frac{1}{A_I'(\omega^i)}\cdot q_i(X) && (6)
\end{aligned}$$

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

3.4.2 Updating Proofs

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

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

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

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

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

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

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

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

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

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

3.4.3 aSVC Algorithms

Having established the intuition for our aSVC, we can now describe it in detail using the aSVC API from Sect. 3.1.

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

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

$\mathsf{VC.ProvePos}(\mathsf{prk}, I, \mathbf{v}) \rightarrow \pi_I$. 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 [TAB+20, Appendix D.7] for the $|I| = 1$ case.)

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

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

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

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

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

3.4.4 Distributing the Trusted Setup

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

Computing All u_i's Fast. Inspired by the FK technique [FK20], we show how to compute all n u_i's in $O(n \log n)$ time using a single DFT on group elements. First, note that $u_i = g^{\frac{\mathcal{L}_i(\tau)-1}{X-\omega^i}}$ is a KZG evaluation proof for $\mathcal{L}_i(\omega^i) = 1$. Thus, $u_i = g^{Q_i(\tau)}$ where $Q_i(X) = \frac{\mathcal{L}_i(X)-1}{X-\omega^i}$. Second, let $\psi_i(X) = A'(\omega^i)\mathcal{L}_i(X) =$

$\frac{X^n-1}{X-\omega^i}$. Then, let $\pi_i = g^{q_i(\tau)}$ be an evaluation proof for $\psi_i(\omega^i) = A'(\omega^i)$ where $q_i(X) = \frac{\psi_i(X)-A'(\omega^i)}{X-\omega^i}$ and note that $Q_i(X) = \frac{1}{A'(\omega^i)}q_i(X)$. Thus, computing all u_i's reduces to computing all π_i's. However, since each proof π_i is for a *different* polynomial $\psi_i(X)$, directly applying FK does not work. Instead, we give a new algorithm that leverages the structure of $\psi_i(X)$ when divided by $X - \omega^i$. Specifically, in [TAB+20, Appendix B], we show that:

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

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

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

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

3.4.5 Correctness and Security

The correctness of our aSVC scheme follows naturally from Lagrange interpolation. Aggregation and proof updates are correct by the arguments laid out in Sects. 3.4.1 and 3.4.2, respectively. Subvector proofs are correct by the correctness of KZG batch proofs [KZG10]. We prove our aSVC is position binding and has update key uniqueness in the extended version [TAB+20, Appendix C].

4 A Highly-Efficient Stateless Cryptocurrency

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

4.1 From VCs to Stateless Cryptocurrencies

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

Trusted Setup. To support up to n users, public parameters $(\mathsf{prk}, \mathsf{vrk}, (\mathsf{upk}_i)_{i\in[0,n)}) \leftarrow \mathsf{VC.KeyGen}(1^\lambda, n)$ are generated via a *trusted setup*, which can be decentralized using MPC protocols [BGM17]. Miners need to store all $O(n)$ update keys to propose blocks and to validate blocks (which we fix in Sect. 4.2.2). The prk is only needed for *proof-serving nodes* (see Sect. 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 Sect. 4.3.1.) Importantly, since $p \approx 2^{256}$, the first 224 bits of v_i are used for addr_i and the last 32 bits for bal_i. The genesis block's state is the all zeros vector with digest d_0 (e.g., in our aSVC, $d_0 = g^0$). Initially, each user i is *unregistered* and starts with a proof $\pi_{i,0}$ that their $v_i = 0$.

"Full" vs. "Traditional" Public Keys. User i's address is computed as $\mathsf{addr}_i = H(\mathsf{FPK}_i)$, where $\mathsf{FPK}_i = (i, \mathsf{upk}_i, \mathsf{tpk}_i)$ is her *full public key.* 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 via $d_{t+1} = \mathsf{VC.UpdateComm}(d_{t+1}, (\mathsf{addr}_i|0), i, \mathsf{upk}_i)$,
3. He incorporates tx in block $t+1$ as $\mathsf{tx}' = [\texttt{INIT}, (i, \mathsf{upk}_i, \mathsf{tpk}_i)] = [\texttt{INIT}, \mathsf{FPK}_i]$.

The full public key with upk_i is included so other users can correctly update their VC when they process tx'. Note that to compute $\mathsf{addr}_i = H(\mathsf{FPK}_i)$, the miner needs to have the correct upk_i which requires $O(n)$ storage. We discuss how to avoid this in Sect. 4.2.2.

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

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

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

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

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

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

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

Step 1: Check Balances. First, for each tx, he checks that $v \leq v'$ 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 $d' = d_{t+1}$.

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

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

4.2 Efficient Stateless Cryptocurrencies from aSVCs

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

4.2.1 Smaller, Faster, Aggregatable Proofs

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

4.2.2 Reducing Miner Storage Using Verifiable Update Keys

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

4.2.3 Smaller Update Keys

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

4.2.4 Addressing DoS Attacks on User Registrations.

Unfortunately, the registration process based on `INIT` transactions is susceptible to Denial of Service (DoS) attacks: an attacker can simply send a large number of `INIT` transactions and quickly exhaust the free space in the vector $\mathbf{v}$.

There are several ways to address this. First, one can use an aSVC from hidden-order groups, which supports an unbounded number of elements [CFG+20]. However, that would negatively impact performance. Second, as future work, one could develop and use unbounded, authenticated dictionaries *with scalable updates*. Third, one could simply use multiple bounded aSVCs together with cross-commitment proof aggregation, which our aSVC supports [GRWZ20]. Lastly, one can add a cost to user registrations via a new `INITSPEND` transaction that registers a user j by having user i send her some coins:

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

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

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

4.2.5 Minting Coins and Transaction Fees

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

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

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

4.3 Discussion

4.3.1 Making Room for Transaction Counters

As mentioned in Sect. 2.2, to prevent transaction replay attacks, account-based stateless cryptocurrencies such as Edrax should actually map a user i to $v_i = (\mathsf{addr}_i | c_i | \mathsf{bal}_i)$, where c_i is her *transaction counter*. This change is trivial, but does leave less space in v_i for addr_i, depending on how many bits are needed for c_i and bal_i. (Recall that $v_i \in \mathbb{Z}_p$ typically has ≈ 256 bits.) To address this, we propose using one aSVC for mapping i to addr_i and another aSVC for mapping i to $(c_i | \mathsf{bal}_i)$. Our key observation is that if the two aSVCs use different n-SDH parameters (e.g., (g^{τ^i})'s and (h^{τ^i})'s, such that $\log_g h$ is unknown), then we could aggregate commitments, proofs and update keys so as to introduce zero computational and communication overhead in our stateless cryptocurrency. Security of this scheme could be argued similar to security of perfectly hiding KZG commitments [KZG10], which commit to $\phi(X)$ as $g^{\phi(\tau)}h^{r(\tau)}$ in an analogous fashion. We leave investigating the details of this scheme to future work.

4.3.2 Overhead of Synchronizing Proofs

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

5 Conclusion

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

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

References

[BB08] Boneh, D., Boyen, X.: Short signatures without random oracles and the SDH assumption in bilinear groups. J. Cryptol. **21**(2), 149–177 (2007). https://doi.org/10.1007/s00145-007-9005-7

[BBF19] Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to IOPs and stateless blockchains. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11692, pp. 561–586. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26948-7_20

[BGM17] Bowe, S., Gabizon, A., Miers, I.: Scalable multi-party computation for zk-SNARK parameters in the random beacon model (2017). https://eprint.iacr.org/2017/1050

[BMRS20] Bonneau, J., Meckler, I., Rao, V., Shapiro, E.: Coda: Decentralized Cryptocurrency at Scale (2020). https://eprint.iacr.org/2020/352

[BSCTV14] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. Algorithmica **79**(4), 1102–1160 (2016). https://doi.org/10.1007/s00453-016-0221-0

[But17] Buterin, V.: The stateless client concept. ethresear.ch (2017). https://ethresear.ch/t/the-stateless-client-concept/172

[But20] Buterin, V.: Using polynomial commitments to replace state roots (2020). https://ethresear.ch/t/using-polynomial-commitments-to-replace-state-roots/7095

[CDHK15] Camenisch, J., Dubovitskaya, M., Haralambiev, K., Kohlweiss, M.: Composable and modular anonymous credentials: definitions and practical constructions. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9453, pp. 262–288. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48800-3_11

[CF13] Catalano, D., Fiore, D.: Vector commitments and their applications. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 55–72. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36362-7_5

[CFG+20] Campanelli, M., Fiore, D., Greco, N., Kolonelos, D., Nizzardo, L.: Vector Commitment Techniques and Applications to Verifiable Decentralized Storage (2020). https://eprint.iacr.org/2020/149

[CFM08] Catalano, D., Fiore, D., Messina, M.: Zero-knowledge sets with short proofs. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 433–450. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78967-3_25

[CPZ18] Chepurnoy, A., Papamanthou, C., Zhang, Y.: Edrax: A Cryptocurrency with Stateless Transaction Validation (2018). https://eprint.iacr.org/2018/968

[Dry19] Dryja, T.: Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set (2019). https://eprint.iacr.org/2019/611

[FK20] Feist, D., Khovratovich, D.: Fast amortized Kate proofs (2020). https://github.com/khovratovich/Kate

[Goy07] Goyal, V.: Reducing trust in the PKG in identity based cryptosystems. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 430–447. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74143-5_24

[GRWZ20] Gorbunov, S., Reyzin, L., Wee, H., Zhang, Z.: Pointproofs: Aggregating Proofs for Multiple Vector Commitments (2020). https://eprint.iacr.org/2020/419

[KJG+18] Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: OmniLedger: a secure, scale-out, decentralized ledger via sharding. In: IEEE S&P 2018, May 2018

[KR13] Kohlweiss, M., Rial, A.: Optimally private access control. In: ACM WPES 2013 (2013)

[KZG10] Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 177–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_11

[LM19] Lai, R.W.F., Malavolta, G.: Subvector commitments with application to succinct arguments. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11692, pp. 530–560. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26948-7_19

[LY10] Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In: Micciancio, D. (ed.) TCC 2010. LNCS, vol. 5978, pp. 499–517. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11799-2_30

[Mil12] Miller, A.: Storing UTXOs in a balanced Merkle tree (zero-trust nodes with O(1)-storage) (2012). https://bitcointalk.org/index.php?topic=101734.msg1117428

[Nak08] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008). https://bitcoin.org/bitcoin.pdf

[PST13] Papamanthou, C., Shi, E., Tamassia, R.: Signatures of correct computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 222–242. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_13

[RMCI17] Reyzin, L., Meshkov, D., Chepurnoy, A., Ivanov, S.: Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In: Kiayias, A. (ed.) FC 2017. LNCS, vol. 10322, pp. 376–392. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70972-7_21

[TAB+20] Tomescu, A., Abraham, I., Buterin, V., Drake, J., Feist, D., Khovratovich, D.: Aggregatable Subvector Commitments for Stateless Cryptocurrencies (2020). https://eprint.iacr.org/2020/527

[Tod16] Todd, P.: Making UTXO set growth irrelevant with low-latency delayed TXO commitments (2016). https://petertodd.org/2016/delayed-txo-commitments

[Tom20] Tomescu, A.: How to Keep a Secret and Share a Public Key (Using Polynomial Commitments). PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2020)

[Vir17] Virza, M.: On Deploying Succinct Zero-Knowledge Proofs. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2017)

[vzGG13a] von zur Gathen, J., Gerhard, J.: Fast multiplication. In: Modern Computer Algebra, 3rd edn, chapter 8, pp. 221–254. Cambridge University Press, Cambridge (2013)

[vzGG13b] von zur Gathen, J., Gerhard, J.: Fast polynomial evaluation and interpolation. In: Modern Computer Algebra, 3rd edn, chapter 10, pp. 295–310. Cambridge University Press, Cambridge (2013)

[Woo] Wood, G.: Ethereum: A Secure Decentralised Generalised Transaction Ledger (2014). http://gavwood.com/paper.pdf