SoK: Vector Commitments

Anca Nitulescu

Protocol Labs

1 Definitions

First Notions. Vector commitments (VC) [CFM08,LY10,CF13] allow to commit to a sequence of values and later on reveal one or many values at a specific position and prove it consistent with the initial commitment. Vector commitments are used to reduce storage costs in a variety of applications: instead of storing a vector of values, one can store only the commitment and receive the values together with their proofs as needed. Vector commitments allow applications to trade off storage (of all values) for bandwidth (taken up by revealed values and proofs). This means that the commitment and the proofs of opening should have a reduced size.

Subvector Openings. Moreover, vector commitments with subvector openings (SVC) were proposed in two independent works by Boneh, Bünz and Fisch [BBF19] and Lai and Malavolta [LM19]. The SVC schemes allow one to open a committed vector at a set of positions with an opening of size independent of both the vector's length and the number of opened positions.

Updatability. In [CF13] the authors introduced the notion of updatable vector commitments and proposed a few applications of vector commitments to: updatable zero-knowledge sets, accumulators and publicly verifiable databases. In a nutshell, a VC is said to be updatable if from a commitment C to a vector $\mathbf{m}$, one can compute a new commitment C' to a vector that updates one position of the vector $\mathbf{m}$ to a different value. In particular, the interesting aspect of this feature is that the computation of such C' can be performed without knowing the entire vector and more efficiently than recomputing the commitment from scratch. In addition to the ability of updating a commitment, an updatable VC must also provide a method to update an opening for position i that is valid for a commitment C into a new opening for the same position that is valid for the new commitment C'. Similarly to the commitment case, this new opening should be computable by only knowing the portion of the vector that is supposed to change and in time faster than recomputing the opening from scratch.

Key-Updatability. Usually, updating commitments and openings requires some auxiliary information related to the positions that are changed. We will call this static auxiliary information the *update key*. Some schemes, like [AR20] do not require at all update keys, while others require computing the update key from some constant-sized public parameters. If this key derivation is efficient, we will consider the scheme *keyless* updatable, while if the key is not efficiently-derivable we will classify the scheme as *key-updatable.*

Hint-Updatability. A weaker variant of this definition requires the algorithms that update the commitment and the opening to take as input an opening for the position in which the vector update occurs. This is the definition used in some of the recent constructions [BBF19,CFG$^+$20].

Aggregatable SVCs. In an SVC, aggregation models the ability of computing an opening for a set of positions I and J starting from two openings and sets of positions I and J, respectively.

One-Hop Aggregation. Aggregation is said to be one-hop if one can aggregate only once, or alternatively if it is not possible to aggregate openings obtained through the aggregate algorithm.

VC Scheme	Group	Setup	Commitment	Proof	Verification	Update	SVC	Aggreg.
	$\mathbb{G}/\mathbb{G}_?/\mathbb{L}$	$\vert \mathsf{pp}\vert$	$\vert C\vert$	$\vert \pi\vert$	time			
Merkle Tree	-	$1\vert H\vert$	$1\vert H\vert$	$\log n\vert H\vert$	$(\log n)H$	hint	$\times$	via SNARKs
[LY10]	bilinear	$n\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$1(\mathbb{G}+\mathbb{F})$	key	$\times$	$\times$
[CF13]$^{(1)}$	bilinear	$n^2\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$1(\mathbb{G}+\mathbb{F})$	key	$\times$	$\times$
[CF13]$^{(2)}$	RSA	$n\vert\mathbb{G}_?\vert$	$1\vert\mathbb{G}_?\vert$	$1\vert\mathbb{G}_?\vert$	$1(\mathbb{G}_?+\mathbb{F})$	key	$\times$	$\times$
[PSTY13]	lattice	$1\vert\mathbb{L}\vert$	$1\vert\mathbb{L}\vert$	$\log n\vert\mathbb{L}\vert$	$\log n\mathbb{L}$	keyless	$\times$	$\times$
Verkle Tree	bilinear	$1\vert H\vert + q\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$\log_q n\vert\mathbb{G}\vert$	$(\log_q n)\mathbb{G}$	hint	$\times$	via SNARKs
[LRY16]	bilinear$_N$	$n\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$1(\mathbb{G}+\mathbb{F})$	key	$\times$	$\times$
[LM19]$^{(1)}$	bilinear	$n^2\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$m(\mathbb{G}+\mathbb{F})$	key	✓	$\times$
[LM19]$^{(2)}$	RSA	$n\vert\mathbb{G}_?\vert$	$1\vert\mathbb{G}_?\vert$	$1\vert\mathbb{G}_?\vert$	$\ell m(\mathbb{G}_?+\mathbb{F})$	key	✓	$\times$
[BBF19]	RSA	$1\vert\mathbb{G}_?\vert$	$1\vert\mathbb{G}_?\vert$	$5\vert\mathbb{G}_?\vert + 1\vert\mathbb{F}\vert$	$(\ell m\log(\ell n))\mathbb{F}+\lambda\mathbb{G}_?$	hint	✓	one-hop
[TAB$^+$20]	bilinear	$n\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$m\mathbb{G}+(m\log^2 m)\mathbb{F}$	key	✓	one-hop
[Tom20]	bilinear	$n\log n\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$\log n\vert\mathbb{G}\vert$	$(m\log^2 m)\mathbb{G}$	key	✓	one-hop
[GRWZ20]	bilinear	$3n\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$m(\mathbb{G}+\mathbb{F})$	hint	✓	cross one-hop
[CFG$^+$20]$^{(1)}$	RSA	$1\vert\mathbb{G}_?\vert$	$1\vert\mathbb{G}_?\vert$	$2\vert\mathbb{G}_?\vert$	$(\ell\cdot m\log m)\mathbb{G}_?$	hint	✓	incremental
[CFG$^+$20]$^{(2)}$	RSA	$3\vert\mathbb{G}_?\vert$	$4\vert\mathbb{G}_?\vert + 2\vert\mathbb{F}\vert$	$3\vert\mathbb{G}_?\vert$	$(\ell m\log(\ell n))\mathbb{F}+\lambda\mathbb{G}_?$	hint	✓	incremental
[AR20]	RSA	$1\vert\mathbb{G}_?\vert$	$2\vert\mathbb{G}_?\vert$	$3\vert\mathbb{G}_?\vert$	$m(\mathbb{G}_?+\mathbb{F})$	keyless	✓	one-hop
[TXN20]	RSA	$1\vert\mathbb{G}_?\vert$	$2\vert\mathbb{G}_?\vert$	$2\vert\mathbb{G}_?\vert$	$(\ell m\log m)\mathbb{G}_?$	key	✓	cross increm.
[SCP$^+$21]	bilinear	$n\vert\mathbb{G}\vert$	$1\vert\mathbb{G}\vert$	$\log n\vert\mathbb{G}_?\vert$	$\log(m\log n)\mathbb{G}_?$	key	✓	cross one-hop

Table 1. VC schemes and their properties. $|\mathbb{G}|$: size of group element, $|H|$: size of a hash, m: number of positions, $\mathbb{G}$: group operation, $\mathbb{F}$: field operation, H: hash operation, n: size of the vector, ℓ: bit-length of vector elements, λ: security parameter. All values are asymptotic.

Incremental Aggregation. Aggregation is said to be incremental if it can be performed an unbounded number of times, and if one can also disaggregate any opening, i.e., from an opening for I one can compute an opening for any subset $K \subset I$.

Campanelli et al. [CFG$^+$20] showed two constructions of incrementally aggregatable SVCs, that have constant-size parameters and work over groups of unknown order.

Cross-Commitment Aggregation. The notion of Cross-Commitment Aggregation allows to compute a succinct proof of opening for a set of positions from different vectors committed separately.

Gorbunov et al. [GRWZ20] show how to extend the VC scheme of [LY10] to allow for cross-commitment aggregation. Their final SVC requires public parameters whose size is linear in the size of the committed vector, while cross-commitment aggregation allow for splitting up a long vector into shorter ones and simply aggregate the proofs. However, this approach increases the total size of the commitments, which cannot be aggregated.

Tomescu et al. [TAB$^+$20] showed how to realize an *updatable* SVC with one-hop aggregation based on the n-Strong Bilinear Diffie-Hellman assumption in bilinear groups. Their scheme has linear-sized public parameters, and it supports commitment updates, proof updates from a static update key tied only to the updated position, in contrast with the dynamic update *hints* required by related works.

Finally, [TXN20] proposes cross-commitment, **incremental** aggregation. This enables one to disaggregate cross-aggregated proofs, update them, and re-aggregate them. These augmented features for VC can be useful for smart contracts when we need stateless validation.

Functional Vector Commitments. A non-interactive functional commitment allows committing to a message m in such a way that the committer has the flexibility of only revealing a function $f(m)$ of the committed message during the opening phase. In functional VCs, messages are vectors $\mathbf{m}$ and commitments can later be opened to a specific function $f(\mathbf{m}) = y$ of the vector coordinates. The security requirement for such primitive is *function binding*, showing that it is infeasible to open a commitment to two different evaluations y, y' for the same function.

Arguments of Knowledge of Subvector Opening. AoK for a Vector Commitment scheme comes in different flavours:

1. **Classic AoK:** It proves knowledge of the subvector that opens a commitment at a public set of positions. An immediate application of the classic AoK is a keyless proof of storage (PoS) protocol with compact proofs.
2. **Extension of AoK:** It can be used to prove that two commitments share a common subvector.
3. **Commit Version of AoK:** This is like the classic one, except that the subvector one proves knowledge of is never revealed, this subvector is also committed: one can create two vector commitments C and C' together with a short proof that C' is a commitment to a subvector of the vector committed in C.

1.1 Algorithms for Vector Commitments

To give a more formal definition for Vector Commitments, we consider the tuple of algorithms $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ with the following syntax:

$\mathsf{Setup}(1^\lambda, \mathcal{M}, n) \to \mathsf{pp}$ The setup algorithm on input the security parameter λ, the message space $\mathcal{M}$ for the elements in the vector, and the vector length n and outputs the public parameters pp.

$\mathsf{Com}(\mathsf{pp}, \mathbf{v}) \to (C, \mathsf{aux})$ On input pp and a vector $\mathbf{v} \in \mathcal{M}^n$, the commit algorithm returns a commitment C, and an auxiliary information aux needed for creating openings.

$\mathsf{Open}(\mathsf{pp}, i, \mathsf{aux}) \to \pi_i$ On input the public parameters pp, an index $i \in [n]$, and an auxiliary information aux, the opening algorithm outputs an opening π_i.

$\mathsf{Ver}(\mathsf{pp}, C, i, y, \pi_i) \to b \in \{0, 1\}$ On input the public parameters pp, a commitment C, an index $i \in [n]$, a value $y \in \mathcal{M}$ and an opening π_i, it outputs either accept (1) or reject (2).

These algorithms satisfy the properties of *correctness*, *position binding*, in the sense that it is not possible to open a commitment to two different values at the same position, and *conciseness*.

Position Binding. Informally, this insures that it should be infeasible for any polynomially bounded adversary (with knowledge of pp) to come up with two proofs that certify the opening of a fixed commitment C to two different values for the same position.

There are two flavours of this security notion:

- **weak binding:** The position binding definition holds only for *honestly generated* commitments C.
- **strong binding:** The binding property should be true even for commitments C that are *adversarially generated*.

Weak-binding is sufficient for stateless validation for example, where commitments are always honestly produced through (Byzantine) agreement on a sequence of updates. Also, weakly-binding VCs are very easy to obtain from cryptographic accumulators. Strong binding is useful in more adversarial settings like transparency logs, where commitments are produced adversarially by log servers.

Conciseness. This asks that both the commitment and the opening for a position i should be of constant size independent of the vector's length.

Remark that Merkle trees, and other tree-based constructions [PSTY13,Tom20], are vector commitments with a relaxed conciseness property, as the commitment is of constant size but the opening of a Merkle tree path is $O(\log n)$.

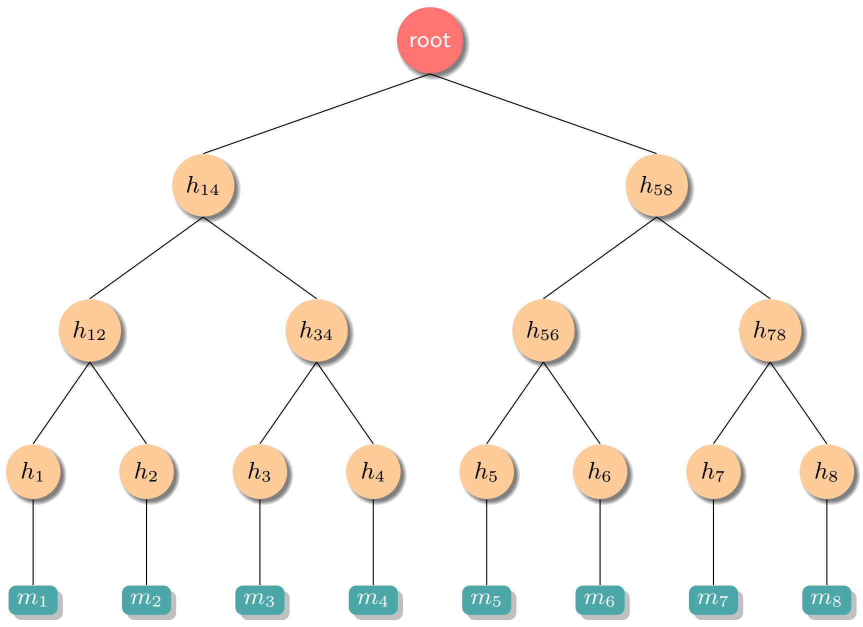

Fig. 1. A Merkle tree commiting to the vector $\mathbf{m}$. Each inner node of the tree is the hash value of the concatenation of its two children.

1.2 Background and State-of-the-art

Table 1 gives a summary for a selection of existing vector commitments taking into account the size of their public parameters, the size of the commitment and opening proof, the verifier's time, whether they are updatable and in which model, if they support subvector openings (SVC) or aggregation.

The best known solution for building vector commitments uses Merkle trees where the k independent openings are aggregated via a general-purpose SNARK, or using a SNARK-friendly collision-resistant hash for the Merkle tree.

Merkle tree construction uses a collision resistant hash function (CRHF) H to compute a succinct commitment to $(m_1, m_2, \ldots m_n)$ of n elements where $n = 2^k$ and later to locally open to any position of m_i in the vector. An example of such a Merkle tree is illustrated in Figure 1.

The structure of the Merkle tree as shown in Figure 1 allows for efficiently mapping the values in the vector recursively to a single root value. To commit to a vector, one computes a tree in which the leaves are the elements of the vector and every internal node is the hash of its two child nodes and the root hash is the compact commitment to the vector.

This tree structure enables succinct proofs of openings, that allow to verify that the hashing of data is consistent all the way up the tree and in the correct position without having to actually look at the entire set of hashes. Instead, demonstrating that a leaf node is a part of a given binary hash tree requires computing a number of hashes proportional to the logarithm of the number of leaves (vector elements); this contrasts with hash lists, where the number is proportional to the number of leaf nodes itself. In more details, an opening of the kth position of the vector consists in all the sibling values of all the nodes in the path from m_k until the root, and the verifier can be convinced of the authenticity of m_k by using the siblings to recompute the nodes in the path, and by checking that the last node is indeed the same as the initial commitment root. The security of this construction can be reduced to the collision resistance of the hash function. Briefly stated, a collision-resistant hash function (CRHF) is a function for which it is hard to find two inputs that map to the same output.

2 Motivation: Vector Commitments for Proof of Space

Vector Commitments are essential in some applications such as Proof of Space (Proof of Storage). Combined with Arguments of Knowledge of Subvector Opening (AoK) with constant-size proofs, the Vector Commitment scheme can lead to an efficient construction of Proof of Space for decentralized usages.

Proof of Space (PoS) protocols allow a client to verify that a server is storing intactly a file via a short communication challenge-response protocol. More precisely, in a PoS protocol we have two main steps:

– **Initialization** (Setup phase): on public input N, an *advice* (eg, vector of random data) of size N is created and committed to. The advice is stored by the prover, while the verifier knows only a commitment to the advice.
– **Execution** (Audit phase): the verifier and the prover run a protocol and the verifier outputs reject/accept. Accept means that the verifier is convinced that the prover stores the advice. This phase can be repeated many times.

We require that the verifier is highly efficient in both phases, whereas the prover is highly efficient in the execution phase providing they are honest and had stored the data, meaning they have random access to the data.

A PoS is sound if a verifier interacting with a malicious prover who stores a fraction of the advice that has size $N' < N$, and runs in at most T steps during the execution phase, outputs accept with small probability (i.e., soundness error). The value $(N - N')/N$ is called the *spacegap*.

Keyless Proof of Space. A PoS is said to be keyless if no secret key is needed by clients, a property useful in open systems where the client is a set of distrustful parties (e.g., verifiers in a blockchain) and the server may even be one of these clients. A classical keyless PoS is based on Merkle trees and random spot-checks, recently generalized to work with vector commitments. A drawback of this construction is that proofs grow with the number of spot checks (and the size of the tree) and become undesirably large in some applications, e.g., if needed to be stored in a blockchain.

With our AoK we can obtain openings of fixed size, as short as 2KB, which is 40x shorter than those based on Merkle trees in a representative setting without relying on SNARKs (that would be unfeasible in terms of time and memory).

Filecoin System. As an illustrative application for a PoS, Filecoin system uses PoS to commit to an advice vector $\mathbf{A}$. Moreover, in Filecoin it is necessary to prove useful space, i.e. storage space that can be used to keep real-world data. When the vector to be committed encodes some real data $\mathbf{D}$, the terminology used is *Proof of Replication* (PoRep), where a "replica" vector $\mathbf{R}$ is defined as $\mathbf{R} = \mathbf{D} + \mathbf{A}$.

A more formal way to capture the security requirements of the Proof of Replication [Fis19] in Filecoin decentralized storage network is to ask for the replica (that is the advice that now contains encoded data) to have an extraction property. This requires the existence of an extraction algorithm that can recover the original data from the interaction with a successful prover during the execution phases.

A more detail description of algorithms used in PoS from Filecoin is given in **??**.

3 Open Questions and Challenges

3.1 Research Directions with Applications to PoS

Problem 1: Augmented Aggregation for SVC. In an SVC, the notion of aggregation models the ability of computing an opening for a set of positions I and J starting from two openings for sets of positions I and J respectively.

Moreover, Cross-Commitment Aggregation allows to compute a succinct proof of opening for a set of positions from different vectors committed separately.

Some of the directions which can be explored to improve existing constructions and understand the limits of achieving stronger notions of aggregation are:

Directions:

- *Reduce the Size of Public Parameters.* The known *pairing-based* schemes with cross-commitment aggregation rely on long public parameters. They require public parameters of size linear in the size of the committed vector. The trusted setups for these schemes seem to be compatible with some ceremonies performed in practice for pairing-based SNARKs like [Gro16] (e.g. "powers of tau" ceremonies from Zcash or from Filecoin). Reducing the size of these parameters or their dependency on the length of vectors to be committed, possibly by means of aggregation (not only for openings, but also for commitments) is a interesting direction to explore in order to improve the storage-bandwidth trade-off.
- *Aggregation for Commitments and Openings.* The RSA-based Key VC scheme from [TXN20] requires only constant-sized parameters and it achieves cross-commitments incremental aggregation for openings. The drawback of these aggregation methods is that they still require the verifier to keep multiple commitment values around. We are interested to look at different ways to achieve more compact commitments or aggregation among commitments, not only for openings. The commitments aggregation should still allow for opening positions of the committed vectors given the aggregations. Would be valuable to also understand what are the challenges, impossibilities or lower bounds in such a scenario.

Problem 2: Functional Vector Commitments. The only constructions of Functional VCs known today are for openings to linear functions, where messages are vectors $\mathbf{m}$ and commitments can later be opened to a specific linear function $f_{\mathbf{x}}(\mathbf{m}) = \sum x_i m_i = y$ of the vector coordinates. We would like to extend this property to broader classes of functions. Depending of the nature of this function, the difficulty of constructing such SVC can vary.

Directions:

- *Lower Bounds and Impossibilities.* Understand the difficulty to construct functional VC for quadratic functions, then for other classes of functions. State the requirements and lower bounds for achieving such properties.
- *Arguments of Knowledge of Subvector Function Evaluation.* We can construct a functional version of AoK. This enables one to prove knowledge of a computation on the values in a subvector: one can create two vector commitments C to $\mathbf{m}$ and C' to a subvector $\mathbf{m}_I$ of $\mathbf{m}$ together with a short proof that y is the result of applying $f(\mathbf{m}_I)$ computation to the subvector $\mathbf{m}_I$ committed in C'. Some key requirements for such a scheme are:
  - pre-processing for proving the opening can be linear in the size of $\mathbf{m}_I$
  - public parameters (opening keys) size is ideally sub-linear in size of vector $\mathbf{m}$
  - proving time should be sub-linear in the vector size

Problem 3: Improving Merkle Trees Openings. The current PoS uses Merkle trees where the k independent openings are aggregated via a general-purpose SNARK, using a SNARK-friendly collision-resistant hash for the Merkle tree (Poseidon). One interesting research direction is to optimize a SNARK for this particular type of problem.

Directions:

- *Algebraic Merkle Tree.* First idea is to eplace all the hash functions with SNARK-friendly hash functions (algebraic). Unfortunately, this would result in a loss in the cost of commiting to a vector. Trying to reduce this cost as much as possible while keeping the benefits of an algebraic hash by means of memory-computation trade-offs, pre-computations or by designing better hashes.
- *Custom SNARKs for Merkle Trees.* Trying to design a SNARK for aggregating Merkle tree paths (the path needed for opening a subvector)
- *Q-ary Merkle Tree.* Developing SNARK-friendly q-ary hash functions (rather than binary), which would reduce the height of the Merkle tree from $\log_2 n$ to $\log_k n$ and thus the number of hash function invocations in the SNARK circuit. Poseidon, to some extend, is more efficient in such a setting. But can we do better?

- *Memory-Computation Trade-off.* Finding a preprocessing memory-computation trade-off for opening Merkle tree commitments (storing some levels of the tree, portions of the path from leaves to the root etc.)

3.2 General Open Questions in VC Research

A few interesting theoretical problems that are open in the VC research area:

Problem 4: Updatability and Aggregation. Updatability it is an essential notion for SVC. It comes in different degrees: hint-updatability that use dynamic keys to make the updates, key-updatability with static keys or keyless updatability. Hint-updatability essentially requires more interaction to perform an update as a one should first obtain an opening for the position to be changed, before performing the update.

Directions:

- *Better Understand Updatability and Aggregation:* We aim on constructing VCs with more flexible updatability properties in combination with other interesting features like aggregation.
  Nevertheless, seems that updatability is actually a more nuanced notion when considered in presence of aggregation: for example, not all VCs support updating aggregated proofs. Same holds for updatability of cross-aggregated proofs. Getting more insights about the relation between Aggregation and Updatability might be also an interesting research direction.

- *Cross Incremental Aggregation and Keyless-Updatability.* Seems that building VC with *keyless-updatability* and *cross-commitment incremental aggregation* is still an open problem. No scheme with constant-size public parameters that achieves these two properties simultaneously is known to date.
  The scheme in [TXN20] is a SVC with incremental cross-commitment aggregation of proofs and hint updatability. In their scheme both commitments, initial proofs and (cross-)aggregated proofs are all hint-updatable.
  More recently, Agrawal and Raghuraman [AR20] introduce a VC scheme that is *key-updatable* (with static update keys) and *one-hop aggregatable* and requires only *constant-sized parameters.* The most complete results in this direction are the works [CFG$^+$20] and [TXN20] which are similar, but they further allow for *incremental aggregation.*

Problem 5: Assumptions and Algebraic Settings for VC. All existing concise VC schemes with constant-size opening proofs are based either on groups of unknown order or on bilinear groups.

Directions:

- *Lattice Assumptions:* There is little work done to construct VCs based on lattice assumptions. Papamanthou et al. instantiate a homomorphic Merkle tree construction using Ajtai's hash function in [PSTY13] to obtain a lattice-based VC scheme but this has still $O(\log n)$-size openings. A more challenging problem is finding a lattice-based scheme with constant-size openings. A possible approach is to start from understanding the existing lattice-based accumulators [JS14] with constant-sized proofs. Turning them into a strongly-binding vector commitment would probably require more work.

- *DLog Groups for VC:* It is still an open question to find a VC that can be based on the discrete-logarithm assumption (or a related one, like CDH) in groups without pairings. Bulletproofs is an example of such VC scheme with $O(\log n)$-size openings. The work [YLF$^+$21] builds on top of Bulletproofs to achieve a similar goal. The challenge is to obtain concise VCs with preferable constant size openings from DLog assumptions.

- *Incrementally aggregatable SVC from prime order groups:* Almost all known constructions for SVC are based on bilinear groups or group of unknown order. Would be interesting to explore constructing such schemes from discrete log assumptions in prime order groups. Such a scheme may be appealing for efficiency purposes since unknown order groups are typically less efficient than prime order groups.

References

AR20. Shashank Agrawal and Srinivasan Raghuraman. KVaC: Key-Value Commitments for blockchains and beyond. In *ASIACRYPT 2020, Part III*, LNCS, pages 839–869. Springer, Heidelberg, December 2020.

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, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, Heidelberg, August 2019.

CF13. Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 55–72. Springer, Heidelberg, February / March 2013.

CFG^{+}20. Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In *ASIACRYPT 2020, Part II*, LNCS, pages 3–35. Springer, Heidelberg, December 2020.

CFM08. Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-knowledge sets with short proofs. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 433–450. Springer, Heidelberg, April 2008.

Fis19. Ben Fisch. Tight proofs of space and replication. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 324–348. Springer, Heidelberg, May 2019.

Gro16. Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

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`.

JS14. Mahabir Prasad Jhanwar and Reihaneh Safavi-Naini. Compact accumulator using lattices. Cryptology ePrint Archive, Report 2014/1015, 2014. `https://eprint.iacr.org/2014/1015`.

LM19. Russell W. F. Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 530–560. Springer, Heidelberg, August 2019.

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, *ICALP 2016*, volume 55 of *LIPIcs*, pages 30:1–30:14. Schloss Dagstuhl, July 2016.

LY10. Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 499–517. Springer, Heidelberg, February 2010.

PSTY13. Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming authenticated data structures. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 353–370. Springer, Heidelberg, May 2013.

SCP^{+}21. Shravan Srinivasan, Alex Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. Cryptology ePrint Archive, Report 2021/599, 2021. `https://eprint.iacr.org/2021/599`.

TAB^{+}20. A. Tomescu, I. Abraham, V. Buterin, J. Drake, D. Feist, and D. Khovratovic. Aggregatable subvector commitments for stateless cryptocurrencies. In *SCN 2020*, volume 12238 of *LNCS*. Springer, 2020.

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.

TXN20. Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated dictionaries with cross-incremental proof (dis)aggregation. Cryptology ePrint Archive, Report 2020/1239, 2020. `https://eprint.iacr.org/2020/1239`.

YLF^{+}21. Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. hbACSS: How to Robustly Share Many Secrets. Cryptology ePrint Archive, Report 2021/159, 2021. `https://eprint.iacr.org/2021/159`.