Incrementally Aggregatable Vector Commitments and Applications to Verifiable Decentralized Storage[⋆]

Matteo Campanelli[1], Dario Fiore[2], Nicola Greco[4], Dimitris Kolonelos[2,3], and Luca Nizzardo[4]

[1] Aarhus University
matteo@cs.au.dk[†]
[2] IMDEA Software Institute, Madrid, Spain
{dario.fiore,dimitris.kolonelos}@imdea.org
[3] Universidad Politecnica de Madrid, Spain
[4] Protocol Labs
{nicola,luca}@protocol.ai

Abstract. Vector commitments with subvector openings (SVC) [Lai-Malavolta, Boneh-Bunz-Fisch; CRYPTO'19] 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.
We continue the study of SVC with two goals in mind: improving their efficiency and making them more suitable to decentralized settings. We address both problems by proposing a new notion for VC that we call *incremental aggregation* and that allows one to merge openings in a succinct way an *unbounded* number of times. We show two applications of this property. The first one is immediate and is a method to generate openings in a distributed way. The second application is an algorithm for faster generation of openings via preprocessing.
We then proceed to realize SVC with incremental aggregation. We provide two constructions in groups of unknown order that, similarly to that of Boneh et al. (which supports aggregating only once), have constant-size public parameters, commitments and openings. As an additional feature, for the first construction we propose efficient arguments of knowledge of subvector openings which immediately yields a keyless proof of storage with compact proofs.
Finally, we address a problem closely related to that of SVC: storing a file efficiently in completely decentralized networks. We introduce and construct *verifiable decentralized storage* (VDS), a cryptographic primitive that allows to check the integrity of a file stored by a network of nodes in a distributed and decentralized way. Our VDS constructions rely on our new vector commitment techniques.

1 Introduction

Commitment schemes are one of the most fundamental cryptographic primitives. They have two basic properties. *Hiding* guarantees that a commitment reveals no information about the underlying message. *Binding* instead ensures that one cannot change its mind about the committed message; namely, it is not possible to open a commitment to two distinct values $m \neq m'$.

Vector commitments (VC) [LY10, CF13] are a special class of commitment schemes in which one can commit to a vector $\vec{v}$ of length n and to later open the commitment at

[†] Work done while author was at IMDEA Software Institute.
[⋆] A full version of this paper can be found at https://ia.cr/2020/149

any position $i \in [n]$. The distinguishing feature of VCs is that both the commitment and an opening for a position i have size independent of n. In terms of security, VCs should be *position binding*, i.e., one cannot open a commitment at position i to two distinct values $v_i \neq v'_i$.

VCs were formalized by Catalano and Fiore [CF13] who proposed two realizations based on the CDH assumption in bilinear groups and the RSA assumption respectively. Both schemes have constant-size commitments and openings but suffer from large public parameters that are $O(n^2)$ and $O(n)$ for the CDH- and RSA-based scheme respectively. Noteworthy is that Merkle trees [Mer88] are VCs with $O(\log n)$-size openings.

Two recent works [BBF19, LM19] proposed new constructions of vector commitments that enjoy a new property called *subvector openings* (also called *batch openings* in [BBF19]). A VC with subvector openings (called SVC, for short) allows one to open a commitment at a collection of positions $I = \{i_1, \ldots, i_m\}$ with a constant-size proof, namely of size independent of the vector's length n and the subvector length m. This property has been shown useful for reducing communication complexity in several applications, such as PCP/IOP-based succinct arguments [LM19, BBF19] and keyless Proofs of Retrievability (PoR) [Fis18].

In this work we continue the study of VCs with subvector openings with two main goals: (1) improving their efficiency, and (2) enabling their use in decentralized systems.

With respect to efficiency, although the most attractive feature of SVCs is the constant size of their opening proofs, a drawback of all constructions is that generating each opening takes at least time $O(n)$ (i.e., as much as committing). This is costly and may harm the use of SVCs in applications such as the ones mentioned above.

When it comes to decentralization, VCs have been proposed as a solution for integrity of a distributed ledger (e.g., blockchains in the account model [BBF19]): the commitment is a succinct representation of the ledger, and a user responsible for the i-th entry can hold the corresponding opening and use it to prove validity of v_i. In this case, though, it is not obvious how to create a succinct subvector opening for, say, m positions held by *different* users each responsible *only* of its own position/s in the vector. We elaborate more on the motivation around this problem in Section 1.2.

1.1 A new notion for SVCs: incremental aggregation

To address these concerns, we define and investigate a new property of vector commitments with subvector openings called *incremental aggregation*. In a nutshell, aggregation means that different subvector openings (say, for sets of positions I and J) can be merged together into a single *concise* (i.e., constant-size) opening (for positions $I \cup J$). This operation must be doable *without* knowing the entire committed vector. Moreover, aggregation is incremental if aggregated proofs can be further aggregated (e.g., two openings for $I \cup J$ and K can be merged into one for $I \cup J \cup K$, and so on an unbounded number of times) and disaggregated (i.e., given an opening for set I one can create one for any $K \subset I$).

While a form of aggregation is already present in the VC of Boneh et al. [BBF19], in [BBF19] this can be performed only once. In contrast, *we define (and construct) the first VC schemes where openings can be aggregated an unbounded number of times*. This incremental property is key to address efficiency and decentralized applications of SVCs, as we detail below.

Incremental aggregation for efficiency. To overcome the barrier of generating each opening in linear time[5] $O_\lambda(n)$, we propose an alternative preprocessing-based method. The idea is to precompute at commitment time an auxiliary information consisting of n/B openings, one for each batch of B positions of the vector. Next, to generate an opening for an arbitrary subset of m positions, one uses the incremental aggregation property in order to disaggregate the relevant subsets of precomputed openings, and then further aggregate for the m positions. Concretely, with this method, in our construction we can do the preprocessing in time $O_\lambda(n \log n)$ and generate an opening for m positions in time roughly $O_\lambda(mB \log n)$.

With the VC of [BBF19], a limited version of this approach is also viable: one precomputes an opening for each bit of the vector in $O_\lambda(n \log n)$ time; and then, at opening time, one uses their one-hop aggregation to aggregate relevant openings in time roughly $O_\lambda(m \log n)$. This however comes with a huge drawback: one must store one opening (of size $p(\lambda) = \mathsf{poly}(\lambda)$ where λ is the security parameter) *for every bit* of the vector, which causes a prohibitive storage overhead, i.e., $p(\lambda) \cdot n$ bits in addition to storing the vector $\vec{v}$ itself.

With incremental aggregation, we can instead tune the chunk size B to obtain flexible time-memory tradeoffs. For example, with $B = \sqrt{n}$ one can use $p(\lambda)\sqrt{n}$ bits of storage to get $O_\lambda(m\sqrt{n} \log n)$ opening time. Or, by setting $B = p(\lambda)$ as the size of one opening, we can obtain a storage overhead of exactly n bits and opening time $O_\lambda(m \log n)$.

Incremental aggregation for decentralization. Essentially, by its definition, incremental aggregation enables generating subvector openings in a distributed fashion. Consider a scenario where different parties each hold an opening of some subvector; using aggregation they can create an opening for the union of their subvectors, moreover the incremental property allows them to perform this operation in a non-coordinated and asynchronous manner, i.e. without the need of a central aggregator. We found this application of incrementally aggregatable SVCs to decentralized systems worth exploring in more detail. To fully address this application, we propose a new cryptographic primitive called verifiable decentralized storage which we discuss in Section 1.2.

Constructing VCs with incremental aggregation. Turning to realizing SVC schemes with our new incremental aggregation property, we propose two SVC constructions that work in hidden-order groups [DK02] (instantiatable using classical RSA groups, class groups [BH01] or the recently proposed groups from Hyperelliptic Curves [DG20]).

Our first SVC has constant-size public parameters and constant-size subvector openings, and its security relies on the Strong RSA assumption and an argument of knowledge in the generic group model. Asymptotically, its efficiency is similar to the SVC of Boneh et al. [BBF19], but concretely we outperform [BBF19]. We implement our new SVC and show it can obtain very fast opening times thanks to the preprocessing method described earlier: opening time reduces by several orders of magnitude for various choices of vector and opening sizes, allowing us to obtain practical opening times—of the order of seconds—that would be impossible without preprocessing—of

[5] We use the notation $O_\lambda(\cdot)$ to include the factor depending on the security parameter λ. Writing "$O_\lambda(t)$" essentially means "$O(t)$ cryptographic operations".

the order of hundred of seconds. In a file of 1 Mibit (2^{20} bits), preprocessing reduces the time to open 2048 bits from one hour to less than 5 seconds!

For the second construction, we show how to modify the RSA-based SVC of [LM19] (which in turn extends the one of [CF13] to support subvector openings) in order to make it with *constant-size* parameters and to achieve incremental aggregation. Compared to the first construction, it is more efficient and based on more standard assumptions, in the standard model.

Efficient Arguments of Knowledge of Subvector Opening. As an additional result, we propose efficient arguments of knowledge (AoK) with *constant-size* proofs for our first VC. In particular, we can prove knowledge of the subvector that opens a commitment at a public set of positions. An immediate application of this AoK is a *keyless proof of storage* (PoS) protocol with compact proofs. PoS allows a client to verify that a server is storing intactly a file via a short-communication challenge-response protocol. A PoS is said *keyless* if no secret key is needed by clients (e.g., mutually distrustful verifiers in a blockchain) and the server may even be one of these clients. 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). For lack of space, these AoK results appear in the full version.

1.2 Verifiable Decentralized Storage (VDS)

We now turn our attention to the problem of preserving storage integrity in a highly decentralized context which some of the distributed features of our VCs (i.e. incremental aggregation) can help us address. We are interested in studying the security of the emerging trend of decentralized and open alternatives to traditional cloud storage and hosting services: *decentralized storage networks* (DSNs). Filecoin (built on top of IPFS), Storj, Dat, Freenet and general-purpose blockchains like Ethereum[6] are some emerging projects in this space.

Background on DSNs. Abstracting from the details of each system, a DSN consists of participants called *nodes*. These can be either storage providers (aka *storage nodes*) or simple *client nodes*. Akin to centralized cloud storage, a client can outsource[7] the storage of large data. However, a key difference with DSN is that storage is provided by, and distributed across, a collection of nodes that can enter and leave the system at will. To make these systems viable it is important to tackle certain basic security questions. DSNs can have some reward mechanism to economically incentivize storage nodes. This means, for example, that there are economic incentives to *fake* storing a file. A further challenge for security (and for obtaining it efficiently) is that these systems are *open* and *decentralized*: anyone can enter the system (and participate as either a service provider or a consumer) and the system works without any central management or trusted parties.

[6] `filecoin.io`, `storj.io`, `datproject.org`, `freenetproject.org`, `ethereum.org`

[7] We point out that in systems like Filecoin some nodes do not effectively outsource anything. Yet they participate (for economic rewards) verifying that others are actually storing for some third party node.

In this work we focus on the basic problem of ensuring that the storage nodes of the DSN are doing their job properly, namely: *How can any client node check that the whole DSN is storing correctly its data (in a distributed fashion)?*

While this question is well studied in the centralized setting where the storage provider is a single server, for decentralized systems the situation is less satisfactory.

The Problem of Verifiable Decentralized Storage in DSNs. Consider a client who outsources the storage of a large file F, consisting of blocks $(F_1, \ldots, F_N)$, to a collection of storage nodes. A storage node can store a portion of F and the network is assumed to be designed in order to self-coordinate so that the whole F is stored, and to be fault-resistant (e.g., by having the same data block stored on multiple nodes). Once the file is stored, clients can request to the network to retrieve or modify a data block F_i (or more), as well as to append (resp. delete) blocks to (resp. from) the file.

In this scenario, our goal is to formalize a cryptographic primitive that can provide clients with the guarantee of *integrity of the outsourced data and its modifications*. The basic idea of VDS is that: (i) the client retains a short *digest* δ_F that "uniquely" points to the file F; (ii) any operation performed by the network, a retrieval or a file modification, can be proven by generating a short *certificate* that is publicly verifiable given δ_F.

This problem is similar in scope to the one addressed by authenticated data structures (ADS) [Tam03]. But while ADS is centralized, VDS is not. In VDS nodes act as storage in a distributed and uncoordinated fashion. This is more challenging as VDS needs to preserve some basic properties of the DSN:

Highly Local. The file is stored across multiple nodes and no node is required to hold the entire F: in VDS every node should function with only its own local view of the system, which should be much smaller than the whole F. Another challenge is dynamic files: in VDS both the digest and the local view must be *locally* updatable, possibly with the help of a short and publicly verifiable update advice from the node holding the modified data blocks.

Decentralized Keyless Clients. In a decentralized system the notion of a client who outsources the storage of a file is blurry. It may for example be a set of mutually distrustful parties (even the entire DSN), or a collection of storage nodes themselves that decide to make some data available to the network. This comes with two implications:

1. *VDS must work without any secret key* on the clients side, so that everyone in the network can delegate and verify storage. This *keyless* setting captures not only clients requiring no coordination, but also a stronger security model. Here the attacker may control both the storage node and the client, yet it must not be able to cheat when proving correctness of its storage. The latter is crucial in DSNs with economic rewards for well-behaving nodes[8].
2. *In VDS a file F exists as long as some storage nodes provide its storage* and a pointer to the file is known to the network through its digest. When a file F is modified into F' and its digest δ_F is updated into $\delta_{F'}$, both versions of the file may coexist. Forks

[8] Since in a decentralized system a storage node may also be a client, an attacker could "delegate storage to itself" and use the client's secret key to cheat in the proof in order to steal rewards (akin to the so-called "generation attack" in Filecoin [Lab17]).

are possible and it is left to each client (or the application) to choose which digest to track: the old one, the new one, or both.

Non-Coordinated Certificates Generation. There are multiple ways in which data retrieval queries can be answered in a DSN. In some cases (e.g., Freenet [CSWH01] or the original Gnutella protocol), data retrieval is also answered in a peer-to-peer non-coordinated fashion. When a query for blocks $i_1, \ldots, i_m$ propagates through the network, every storage node replies with the blocks that it owns and these answers are aggregated and propagated in the network until they reach the client who asked for them. To accommodate arbitrary aggregation strategies, in VDS we consider the incremental aggregation of query certificates in an arbitrary and bandwidth-efficient fashion. For example, short certificates for file blocks F_i and F_j should be mergeable into a *short* certificate for (F_i, F_j) and this aggregation process should be carried on and on. Noteworthy that having certificates that stay short after each aggregation keeps the communication overhead of the VDS integrity mechanism at a minimum.[9]

A new cryptographic primitive: VDS. To address the problem described above, we put forward the definition of a new cryptographic primitive called *verifiable decentralized storage* (VDS). In a nutshell, VDS is a collection of algorithms that can be used by clients and storage nodes to maintain the system. The basic ideas are the following: every file F is associated to a succinct digest δ_F; a storage node can answer and certify retrieval queries for subportions of F that it stores, as well as to push updates of F that enable anyone else to update the digest accordingly. Moreover, certified retrieval results can be arbitrarily aggregated. With respect to security, VDS guarantees that malicious storage nodes (even a full coalition of them) cannot create certificates for falsified data blocks that pass verification. For efficiency, the key property of VDS is that digests and every certificate are at most $O(\log |F|)$, and that every node in the system works with storage and running time that depends at most logarithmically in F's size. We discuss our definition of VDS in Section 5.

Constructing VDS. We propose two constructions of VDS in hidden-order groups. Both our VDS schemes are obtained by extending our first and second SVC scheme respectively, in order to handle updates and to ensure that all such update operations can be performed locally. We show crucial use of the new properties of our constructions: subvector openings, incremental aggregation and disaggregation, and arguments of knowledge for sub-vector commitments (the latter for the first scheme only).

Our two VDS schemes are based on the Strong RSA [BP97] and Strong distinct-prime-product root [LM19], and Low Order [BBF18] assumptions and have similar performances. The second scheme has the interesting property that the storage node can perform and propagate updates by running in time that is independent of even its total local storage.

Finally, we note that VDS shares similarities with the notion of updatable VCs [CF13] extended with incrementally aggregatable subvector openings. There are two main differences. First, in VDS updates can be applied with the help of a short advice created by the party who created the update, whereas in updatable VC this is possible

[9] The motivation of this property is similar to that of sequential aggregate signatures, see e.g., [LMRS04, BGR12].

having only the update's description. The second difference is that in VDS the public parameters must be short, otherwise nodes could not afford storing them. This is not necessarily the case in VCs and in fact, to the best of our knowledge, there exists no VC construction with short parameters that is updatable (according to the updatability notion of [CF13]) and has incrementally aggregatable subvector openings. We believe this is an interesting open problem.

1.3 Concurrent Work

In very recent concurrent works, Gorbunov et al. [GRWZ20] and Tomescu et al. [TAB$^+$20] study similar problems related to aggregation properties of vector commitments. In [TAB$^+$20], Tomescu et al. study a vector commitment scheme based on the Kate et al. polynomial commitment [KZG10]: they show how it can be made both updatable and aggregatable, and propose an efficient Stateless Cryptocurrency based on it. In Pointproofs [GRWZ20] they propose the notion of Cross-Commitment Aggregation, which enables aggregating opening proofs for different commitments, and show how this notion is relevant to blockchain applications. The VC schemes in both [TAB$^+$20] and [GRWZ20] work in bilinear groups and have linear-size public parameters. Also, these constructions do not support incremental aggregation or disaggregation. In contrast, our VCs work in hidden-order groups, which likely makes them concretely less efficient, but they have constant-size parameters, and they support incremental aggregation and disaggregation. Finally, we note that by using techniques similar to [GRWZ20] we can extend our constructions to support cross-commitment aggregation; we leave formalizing this extension for future work.

1.4 Preliminaries

In the paper we use rather standard cryptographic notation and definitions that for completeness are recalled in the full version. More specific to this paper we denote by $\mathsf{Primes}(\lambda)$ the set of all prime integers less than 2^λ.

Groups of Unknown Order and Computational Assumptions. Our constructions use a group $\mathbb{G}$ of unknown (aka hidden) order [DK02], in which the Low Order assumption [BBF18] and the Strong RSA assumption [BP97] hold. We let $\mathsf{Ggen}(1^\lambda)$ be a probabilistic algorithm that generates such a group $\mathbb{G}$ with order in a specific range $[\mathsf{ord}_{min}, \mathsf{ord}_{max}]$ such that $\frac{1}{\mathsf{ord}_{min}}, \frac{1}{\mathsf{ord}_{max}}, \frac{1}{\mathsf{ord}_{max} - \mathsf{ord}_{min}} \in \mathsf{negl}(\lambda)$. As discussed in [BBF18, BBF19, LM19], two concrete instantiations of $\mathbb{G}$ are class groups [BH01] and the quotient group $\mathbb{Z}_N^*/\{1, -1\}$ of an RSA group [Wes18]. See the full version for the formal definitions of the assumptions and for a recall of Shamir's trick [Sha83] that we use extensively in our constructions.

2 Vector Commitments with Incremental Aggregation

In this section, we recall vector commitments with subvector openings [CF13, LM19, BBF19] and then we formally define our new incremental aggregation property.

2.1 Vector Commitments with Subvector Openings

In our work we consider the generalization of vector commitments proposed by Lai and Malavolta [LM19] called *VCs with subvector openings*[10] (we call them SVCs for brevity) in which one can open the commitment to an ordered collection of positions with a short proof. Below is a brief recap of their definition.

Let $\mathcal{M}$ be a set, $n \in \mathbb{N}$ be a positive integer and $I = \{i_1, \ldots, i_{|I|}\} \subseteq [n]$ be an ordered index set. The I-subvector of a vector $\vec{v} \in \mathcal{M}^n$ is $\vec{v}_I := (v_{i_1}, \ldots, v_{i_{|I|}})$. Let $I, J \subseteq [n]$ be two sets, and let $\vec{v}_I, \vec{v}_J$ be two subvectors of some $\vec{v} \in \mathcal{M}^n$. The *ordered union* of $\vec{v}_I$ and $\vec{v}_J$ is the subvector $\vec{v}_{I \cup J}$, where $I \cup J$ is the ordered union of I and J.

A vector commitment scheme with subvector openings (SVC) is a tuple of algorithms $\mathsf{VC} = (\mathsf{VC.Setup}, \mathsf{VC.Com}, \mathsf{VC.Open}, \mathsf{VC.Ver})$ that work as follows. The probabilistic setup algorithm, $\mathsf{VC.Setup}(1^\lambda, \mathcal{M}) \rightarrow \mathsf{crs}$, which given the security parameter λ and description of a message space $\mathcal{M}$ for the vector components, outputs a common reference string crs; the committing algorithm, $\mathsf{VC.Com}(\mathsf{crs}, \vec{v}) \rightarrow (C, \mathsf{aux})$, which on input crs and a vector $\vec{v} \in \mathcal{M}^n$, outputs a commitment C and an auxiliary information aux; the opening algorithm, $\mathsf{VC.Open}(\mathsf{crs}, I, \vec{y}, \mathsf{aux}) \rightarrow \pi_I$ which on input the CRS crs, a vector $\vec{y} \in \mathcal{M}^m$, an ordered index set $I \subset \mathbb{N}$ and auxiliary information aux, outputs a proof π_I that $\vec{y}$ is the I-subvector of the committed message; the verification algorithm, $\mathsf{VC.Ver}(\mathsf{crs}, C, I, \vec{y}, \pi_I) \rightarrow b \in \{0, 1\}$, which on input the CRS crs, a commitment C, an ordered set of indices $I \subset \mathbb{N}$, a vector $\vec{y} \in \mathcal{M}^m$ and a proof π_I, accepts (i.e., it outputs 1) only if π_I is a valid proof that C was created to a vector $\vec{v} = (v_1, \ldots, v_n)$ such that $\vec{y} = \vec{v}_I$. We require three properties from a vector commitment: *correctness* (verification acts as expected on honestly generated commitments and openings); *position binding* (no adversary can produce two valid openings for different subvectors); *conciseness* (if its commitments and openings are of size independent of $|\vec{v}|$).

Vector Commitments with Specializable Universal CRS. The notion of VCs defined above slightly generalizes the previous ones in which the generation of public parameters (aka common reference string) depends on a bound n on the length of the committed vectors. In contrast, in our notion $\mathsf{VC.Setup}$ is length-independent. To highlight this property, we also call this primitive *vector commitments with universal CRS*.

Here we formalize a class of VC schemes that lies in between VCs with universal CRS (as defined above) and VCs with length-specific CRS (as defined in [CF13]). Inspired by the recent work of Groth et al. [GKM$^+$18], we call these schemes VCs with *Specializable* (Universal) CRS. In a nutshell, these are schemes in which the algorithms $\mathsf{VC.Com}$, $\mathsf{VC.Open}$ and $\mathsf{VC.Ver}$ work on input a length-specific CRS crs_n. However, this crs_n is generated in two steps: (i) a *length-independent, probabilistic* setup $\mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M})$, and (ii) a *length-dependent, deterministic* specialization $\mathsf{crs}_n \leftarrow \mathsf{VC.Specialize}(\mathsf{crs}, n)$. The advantage of this model is that, being $\mathsf{VC.Specialize}$ deterministic, it can be executed by anyone, and it allows to re-use the same crs for multiple vectors lengths.

See the full version for the formal definition of VCs with specializable CRS.

[10] This is also called VCs with batchable openings in an independent work by Boneh et al. [BBF19] and can be seen as a specialization of the notion of *functional vector commitments* [LRY16].

2.2 Incrementally Aggregatable Subvector Openings

In a nutshell, aggregation means that different proofs of different subvector openings can be merged together into a single *short* proof which can be created *without* knowing the entire committed vector. Moreover, this aggregation is composable, namely aggregated proofs can be further aggregated. Following a terminology similar to that of aggregate signatures, we call this property *incremental aggregation* (but can also be called *multi-hop aggregation*). In addition to aggregating openings, we also consider the possibility to "disaggregate" them, namely from an opening of positions in the set I one can create an opening for positions in a set $K \subset I$.

We stress on the two main requirements that make aggregation and disaggregation non-trivial: all openings must remain short (independently of the number of positions that are being opened), and aggregation (resp. disaggregation) must be computable locally, i.e., without knowing the whole committed vector. Without such requirements, one could achieve this property by simply concatenating openings of single positions.

Definition 2.1 (Aggregatable Subvector Openings). *A vector commitment scheme* VC *with subvector openings is called* aggregatable *if there exists algorithms* $\mathsf{VC.Agg}$, $\mathsf{VC.Disagg}$ *working as follows:*

$\mathsf{VC.Agg}(\mathsf{crs}, (I, \vec{v}_I, \pi_I), (J, \vec{v}_J, \pi_J)) \to \pi_K$ *takes as input triples* $(I, \vec{v}_I, \pi_I), (J, \vec{v}_J, \pi_J)$ *where* I *and* J *are sets of indices,* $\vec{v}_I \in \mathcal{M}^{|I|}$ *and* $\vec{v}_J \in \mathcal{M}^{|J|}$ *are subvectors, and* π_I *and* π_J *are opening proofs. It outputs a proof* π_K *that is supposed to prove opening of values in positions* $K = I \cup J$.

$\mathsf{VC.Disagg}(\mathsf{crs}, I, \vec{v}_I, \pi_I, K) \to \pi_K$ *takes as input a triple* $(I, \vec{v}_I, \pi_I)$ *and a set of indices* $K \subset I$*, and it outputs a proof* π_K *that is supposed to prove opening of values in positions* K.

The aggregation algorithm $\mathsf{VC.Agg}$ *must guarantee the following two properties:*

Aggregation Correctness. *Aggregation is (perfectly) correct if for all* $\lambda \in \mathbb{N}$*, all honestly generated* $\mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M})$*, any commitment* C *and triple* $(I, \vec{v}_I, \pi_I)$ *s.t.* $\mathsf{VC.Ver}(\mathsf{crs}, C, I, \vec{v}_I, \pi_I) = 1$*, the following two properties hold:*

1. *for any triple* $(J, \vec{v}_J, \pi_J)$ *such that* $\mathsf{VC.Ver}(\mathsf{crs}, C, J, \vec{v}_J, \pi_J) = 1$,

$$\Pr\left[\mathsf{VC.Ver}(\mathsf{crs}, C, K, \vec{v}_K, \pi_K) = 1 \ : \ \pi_K \leftarrow \mathsf{VC.Agg}(\mathsf{crs}, (I, \vec{v}_I, \pi_I), (J, \vec{v}_J, \pi_J))\right] = 1$$

   where $K = I \cup J$ *and* $\vec{v}_K$ *is the ordered union* $\vec{v}_{I \cup J}$ *of* $\vec{v}_I$ *and* $\vec{v}_J$*;*
2. *for any subset of indices* $K \subset I$,

$$\Pr\left[\mathsf{VC.Ver}(\mathsf{crs}, C, K, \vec{v}_K, \pi_K) = 1 \ : \ \pi_K \leftarrow \mathsf{VC.Disagg}(\mathsf{crs}, I, \vec{v}_I, \pi_I, K)\right] = 1$$

   where $\vec{v}_K = (v_{i_l})_{i_l \in K}$*, for* $\vec{v}_I = (v_{i_1}, \ldots, v_{i_{|I|}})$.

Aggregation Conciseness. *There exists a fixed polynomial* $p(\cdot)$ *in the security parameter such that all openings produced by* $\mathsf{VC.Agg}$ *and* $\mathsf{VC.Disagg}$ *have length bounded by* $p(\lambda)$.

We remark that the notion of specializable CRS can apply to aggregatable VCs as well. In this case, we let $\mathsf{VC.Agg}^\star$ (resp. $\mathsf{VC.Disagg}^\star$) be the algorithm that works on input the specialized crs_n instead of crs.

3 Applications of Incremental Aggregation

We discuss two general applications of the SVC incremental aggregation property.

One application is generating subvector openings in a distributed and decentralized way. Namely, assume a set of parties hold each an opening of some subvector. Then it is possible to create a (concise) opening for the union of their subvectors by using the VC.Agg algorithm. Moreover, the incremental (aka multi-hop) aggregation allows these users to perform this operation in an arbitrary order, hence no coordination or a central aggregator party are needed. This application is particularly useful in our extension to verifiable decentralized storage.

The second application is to generate openings in a faster way via preprocessing. As we mentioned in the introduction, this technique is useful in the scenario where a user commits to a vector and then must generate openings for various subvectors, which is for example the use case when the VC is used for proofs of retrievability and IOPs [BBF19].

So, here the goal is to achieve a method for computing subvector openings in time sub-linear in the total size of the vector, which is the barrier in all existing constructions. To obtain this speedup, the basic idea is to (A) compute and store openings for all the positions at commitment time, and then (B) use the aggregation property to create an opening for a specific set of positions. In order to obtain efficiency using this approach it is important that both steps (A) and (B) can be computed efficiently. In particular, step (A) is challenging since typically computing one opening takes linear time, hence computing all of them would take quadratic time.

In this section, we show how steps (A) and (B) can benefit from disaggregation and aggregation respectively. As a preliminary for this technique, we begin by describing two generic extensions of (incremental) aggregation (resp. disaggregation) that support many inputs (resp. outputs). Then we show how these extended algorithms can be used for committing and opening with preprocessing.

3.1 Divide-and-Conquer Extensions of Aggregation and Disaggregation

We discuss how the incremental property of our aggregation and disaggregation can be used to define two extended versions of these algorithms. The first one is an algorithm that can aggregate many openings for different sets of positions into a single opening for their union. The second one does the opposite, namely it disaggregates one opening for a set I into many openings for partitions of I.

Aggregating Many Openings We consider the problem of aggregating several openings for sets of positions $I_1, \ldots, I_m$ into a single opening for $\bigcup_{j=1}^{m} I_j$. Our syntax in Definition 2.1 only considers pairwise aggregation. This can be used to handle many aggregations by executing the pairwise aggregation in a sequential (or arbitrary order) fashion. Sequential aggregation might however be costly since it would require executing VC.Agg on increasingly growing sets. If $f_a(k)$ is the complexity of VC.Agg on two sets of total size k, then the complexity of the sequential method is $\sum_{j=2}^{m} f(\sum_{l=1}^{j-1} |I_l| + |I_j|)$, which for example is quadratic in m, for $f_a(k) = \Theta(k)$.

```
$\mathsf{VC.AggManyToOne}(\mathsf{crs}, (I_j, \vec{v}_{I_j}, \pi_j)_{j \in [m]})$

if $m = 1$ return $\pi_1$
$m' \leftarrow m/2$
$L \leftarrow \cup_{j=1}^{m'} I_j, \quad R \leftarrow \cup_{j=m'+1}^{m} I_j,$
$\pi_L \leftarrow \mathsf{VC.AggManyToOne}(\mathsf{crs}, (I_j, \vec{v}_{I_j}, \pi_j)_{j=1,\dots,m'})$
$\pi_R \leftarrow \mathsf{VC.AggManyToOne}(\mathsf{crs}, (I_j, \vec{v}_{I_j}, \pi_j)_{j=m'+1,\dots,m})$
$\pi_{L \cup R} \leftarrow \mathsf{VC.Agg}(\mathsf{crs}, (L, \vec{v}_L, \pi_L), (R, \vec{v}_R, \pi_R))$
return $\pi_{L \cup R}$
```

```
$\mathsf{VC.DisaggOneToMany}(\mathsf{crs}, B, I, \vec{v}_I, \pi_I)$

if $n = |I| = B$ return $\pi_I$
$n' \leftarrow n/2$
$L \leftarrow \cup_{j=1}^{n'} i_j, \quad R \leftarrow \cup_{j=n'+1}^{m} i_j,$
$\pi'_L \leftarrow \mathsf{VC.Disagg}(\mathsf{crs}, I, \vec{v}_I, \pi_I, L)$
$\pi'_R \leftarrow \mathsf{VC.Disagg}(\mathsf{crs}, I, \vec{v}_I, \pi_I, R)$
$\vec{\pi}_L \leftarrow \mathsf{VC.DisaggOneToMany}(\mathsf{crs}, B, L, \vec{v}_L, \pi'_L)$
$\vec{\pi}_R \leftarrow \mathsf{VC.DisaggOneToMany}(\mathsf{crs}, B, R, \vec{v}_R, \pi'_R)$
return $\vec{\pi}_L || \vec{\pi}_R$
```

Fig. 1. Extensions of Aggregation and Disaggregation

In Fig. 1, we show an algorithm, VC.AggManyToOne, that is a nearly optimal solution for aggregating m openings based on a divide-and-conquer methodology. Assuming for simplicity that all I_j's have size bounded by some s, then the complexity of VC.AggManyToOne is given by the recurrence relation $T(m) = 2T\left(\frac{m}{2}\right) + f_a(s \cdot m)$, which solves to $\Theta(s \cdot m \log m)$ if $f_a(n) \in \Theta(n)$, or to $\Theta(s \cdot m \log(sm) \log m)$ if $f_a(n) \in \Theta(n \log n)$.

Disaggregating from One to Many Openings We consider the problem that is dual to the one above, namely how to disaggregate an opening for a set I into several openings for sets $I_1, \dots, I_m$ that form a partition of I. Our syntax in Definition 2.1 only considers disaggregation from I to one subset K of I. Similarly to aggregation, disaggregating from one set to many subsets can be trivially obtained via a sequential application of VC.Disagg on all pairs (I, I_j). This however can be costly if the number of partitions approaches the size of I, e.g., if we want to disaggregate to all the elements of I.

In Fig. 1, we show a divide-and-conquer algorithm, VC.DisaggOneToMany, for disaggregating an opening for a set I of size m into $m' = m/B$ openings, each for a partition of size B. For simplicity, we assume that m is a power of 2, and $B \mid m$. Let $f_d(|I|)$ be the complexity of VC.Disagg. The complexity of VC.DisaggOneToMany is given by the recurrence relation $T(m) = 2T\left(\frac{m}{2}\right) + 2f_d(m/2)$, which solves to $\Theta(m \log(m/B))$ if $f_d(n) \in \Theta(n)$, or to $\Theta(m \log m \log(m/B))$ if $f_d(n) \in \Theta(n \log n)$.

3.2 Committing and Opening with Precomputation

We present a construction of committing and opening algorithms (denoted VC.PPCom and VC.FastOpen respectively) that works generically for any SVC with incremental aggregation and that, by relying on preprocessing, can achieve fast opening time.

Our preprocessing method works with a flexible choice of a parameter B that allows for different time-memory tradeoffs. In a nutshell, ranging from 1 to n, a larger B reduces memory but increases opening time while a smaller B (e.g., $B = 1$) requires larger storage overhead but gives the fastest opening time.

Let B be an integer that divides n, and let $n' = n/B$. The core of our idea is that, during the commitment stage, one can create openings for $n' = n/B$ subvectors of $\vec{v}$ that cover the whole vector (e.g., B contiguous positions). Let $\pi_{P_1}, \dots, \pi_{P_{n'}}$ be such openings; these elements are stored as advice information.

Next, in the opening phase, in order to compute the opening for a subvector $\vec{v}_I$ of m positions, one should: (i) fetch the subset of openings π_{P_j} such that, for some S,

$I \subseteq \cup_{j \in S} P_j$, (ii) possibly disaggregate some of them and then aggregate in order to compute π_I.

The two algorithms $\mathsf{VC.PPCom}$ and $\mathsf{VC.FastOpen}$ are described in detail in Fig. 2.

```
$\mathsf{VC.PPCom}(\mathsf{crs}, B, \vec{v})$

$(C, \mathsf{aux}) \leftarrow \mathsf{VC.Com}(\mathsf{crs}, \vec{v})$
$\pi^* \leftarrow \mathsf{VC.Open}(\mathsf{crs}, [n], \vec{v}, \mathsf{aux})$
$\vec{\pi} \leftarrow \mathsf{VC.DisaggOneToMany}(\mathsf{crs}, B, [n], \vec{v}, \pi^*)$
$\mathsf{aux}^* := (\pi_1, \ldots, \pi_{n'}, \vec{v})$
return $C, \mathsf{aux}^*$
```

```
$\mathsf{VC.FastOpen}(\mathsf{crs}, B, \mathsf{aux}^*, I)$

Let $P_j := \{(j-1)B + i : i \in [B]\}, \forall j \in [n']$
Let $I := \{i_1, \ldots, i_m\}$
Let $S$ minimal set s.t. $\bigcup_{j \in S} P_j \supseteq I$
for $j \in S$ do :
  $I_j \leftarrow I \cap P_j$
  $\pi'_j \leftarrow \mathsf{VC.Disagg}(\mathsf{crs}, P_j, \vec{v}_{P_j}, \pi_j, I_j)$
endfor
$\pi_I \leftarrow \mathsf{VC.AggManyToOne}(\mathsf{crs}, ((I_j, \vec{v}_{I_j}, \pi'_j))_{j \in S})$
return $\pi_I$
```

Fig. 2. Generic algorithms for committing and opening with precomputation.

In terms of auxiliary storage, in addition to the vector $\vec{v}$ itself, one needs at most $(n/B)p(\lambda)$ bits, where $p(\lambda)$ is the polynomial bounding the conciseness of the SVC scheme. In terms of time complexity, $\mathsf{VC.PPCom}$ requires one execution of $\mathsf{VC.Com}$, one execution of $\mathsf{VC.Open}$, and one execution of $\mathsf{VC.DisaggOneToMany}$, which in turn depends on the complexity of $\mathsf{VC.Disagg}$; $\mathsf{VC.FastOpen}$ requires to perform (at most) $|S|$ disaggregations (each with a set $|I_j|$ such that their sum is $|I|$)[11], and one execution of $\mathsf{VC.AggManyToOne}$ on $|S|$ openings. Note that $\mathsf{VC.FastOpen}$'s running time depends only on the size m of the set I and size B of the buckets P_j, and thus offers various tradeoffs by adjusting B.

More specific running times depend on the complexity of the algorithms $\mathsf{VC.Com}$, $\mathsf{VC.Open}$, $\mathsf{VC.Agg}$, and $\mathsf{VC.Disagg}$ of the given SVC scheme. See Section 4.3 and the full version for these results for our constructions.

4 Our Realizations of Incrementally Aggregatable SVCs

In this section we describe our new SVC realizations.

4.1 Our First SVC Construction

AN OVERVIEW OF OUR TECHNIQUES. The basic idea underlying our VC can be described as a generic construction from any accumulator with union proofs. Consider a vector of bits $\vec{v} = (v_1, \ldots, v_n) \in \{0,1\}^n$. In order to commit to this vector we produce two accumulators, Acc_0 and Acc_1, on two partitions of the set $S = \{1, \ldots, n\}$. Each accumulator Acc_b compresses the set of positions i such that $v_i = b$. In other words, Acc_b compresses the set $S_{=b} := \{i \in S : v_i = b\}$ with $b \in \{0,1\}$. In order to open to

[11] Note that for $B = 1$ the disaggregation step can be skipped.

```
$\mathsf{Setup}(1^\lambda)$ : run $\mathbb{G} \leftarrow_\$ \mathsf{Ggen}(1^\lambda)$, $g_1, g_2, g_3 \leftarrow_\$ \mathbb{G}$, set $\mathsf{crs} := (\mathbb{G}, g_1, g_2, g_3)$.
Prover's input: $(\mathsf{crs}, (Y, C), (a, b))$. Verifier's input: $(\mathsf{crs}, (Y, C))$.

  $\mathsf{V} \to \mathsf{P}$: $\ell \leftarrow_\$ \mathsf{Primes}(\lambda)$
  $\mathsf{P} \to \mathsf{V}$: $\pi := ((Q_Y, Q_C), r_a, r_b)$ computed as follows
    – $(q_a, q_b, q_c) \leftarrow (\lfloor a/\ell \rfloor, \lfloor b/\ell \rfloor, \lfloor ab/\ell \rfloor)$
    – $(r_a, r_b) \leftarrow (a \mod \ell, b \mod \ell)$
    – $(Q_Y, Q_C) := (g_1^{q_a} g_2^{q_b}, g_3^{q_c})$
  $\mathsf{V}(\mathsf{crs}, (Y, C), \ell, \pi)$:
    – Compute $r_c \leftarrow r_a \cdot r_b \mod \ell$
    – Output 1 iff $r_a, r_b \in [\ell] \wedge Q_Y^\ell g_1^{r_a} g_2^{r_b} = Y \wedge Q_C^\ell g_3^{r_c} = C$
```

Fig. 3. PoProd_2 protocol

bit b at position i, one can create an accumulator membership proof for the statement $i \in \tilde{S}_b$ where we denote by $\tilde{S}_b$ the alleged set of positions that have value b.

However, if the commitment to $\vec{v}$ is simply the pair of accumulators $(\mathsf{Acc}_0, \mathsf{Acc}_1)$ we do not achieve position binding as an adversary could for example include the same element i in both accumulators. To solve this issue we set the commitment to be the pair of accumulators plus a succinct non-interactive proof π_S that the two sets $\tilde{S}_0, \tilde{S}_1$ they compress constitute together a *partition* of S. Notably, this proof π_S guarantees that each index i is in either $\tilde{S}_0$ or $\tilde{S}_1$, and thus prevents an adversary from also opening the position i to the complement bit $1 - b$.

The construction described above could be instantiated with any accumulator scheme that admits an efficient and succinct proof of union. We, though, directly present an efficient construction based on RSA accumulators [Bd94, BP97, CL02, Lip12, BBF19] as this is efficient and has some nice extra properties like aggregation and constant-size parameters. Also, part of our technical contribution to construct this VC scheme is the construction of efficient and succinct protocols for proving the union of two RSA accumulators built with different generators.

Succinct AoK Protocols for Union of RSA Accumulators Let $\mathbb{G}$ be a hidden order group as generated by Ggen, and let $g_1, g_2, g_3 \in \mathbb{G}$ be three honestly sampled random generators. We propose a succinct argument of knowledge for the following relation

$$R_{\mathsf{PoProd}_2} = \left\{ ((Y, C), (a, b)) \in \mathbb{G}^2 \times \mathbb{Z}^2 \ : \ Y = g_1^a g_2^b \wedge C = g_3^{a \cdot b} \quad \right\}$$

Our protocol (described in Fig. 3) is inspired by a similar protocol of Boneh et al. [BBF19], PoDDH, for a similar relation in which there is only one generator (i.e., $g_1 = g_2 = g_3$, namely for DDH tuples (g^a, g^b, g^{ab})). Their protocol has a proof consisting of 3 groups elements and 2 integers of λ bits.

As we argue later PoProd_2 is still sufficient for our construction, i.e., for the goal of proving that $C = g_3^c$ is an accumulator to a set that is the union of sets represented by two accumulators $A = g_1^a$ and $B = g_2^b$ respectively. The idea is to invoke PoProd_2 on (Y, C) with $Y = A \cdot B$.

To prove the security of our protocol we rely on the adaptive root assumption and, in a non-black-box way, on the knowledge extractability of the PoKRep and PoKE^*

protocols from [BBF19]. The latter is proven in the generic group model for hidden order groups (where also the adaptive root assumption holds), therefore we state the following theorem.

Theorem 4.1. *The* PoProd_2 *protocol is an argument of knowledge for* R_{PoProd_2} *in the generic group model.*

For space reasons the full proof is in the full version. The basic intuition is to use the extractors of PoKRep and PoKE^* to extract (a, b, c) such that $Y = g_1^a g_2^b \wedge C = g_3^{a \cdot b}$. Then $c = a \cdot b$ comes from the fact that ℓ is randomly chosen, which makes the equality $r_c = r_a \cdot r_b \mod \ell$ happen with negligible probability if $c \neq a \cdot b$.

In the full version we also give a protocol PoProd that proves $g_1^a = A \wedge g_2^b = B$ instead of $g_1^a g_2^b = Y$ (i.e., a version of PoDDH with different generators). Despite being conceptually simpler, it is slightly less efficient than PoProd_2, and thus we use the latter in our VC construction.

HASH TO PRIME FUNCTION AND NON-INTERACTIVE PoProd_2. Our protocols can be made non-interactive by applying the Fiat-Shamir transform. For this we need an hash function that can be modeled as a random oracle and that maps arbitrary strings to prime numbers, i.e., $\mathsf{H}_{\mathsf{prime}} : \{0,1\}^* \to \mathsf{Primes}(2\lambda)$[12]. A simple way to achieve such a function is to apply a standard hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{2\lambda}$ to an input $\vec{y}$ together with a counter i, and if $p_{y,i} = \mathsf{H}(\vec{y}, i)$ is prime then output $p_{y,i}$, otherwise continue to $\mathsf{H}(\vec{y}, i+1)$ and so on, until a prime is found. Due to the distribution of primes, the expected running time of this method is $O(\lambda)$, assuming that H's outputs are uniformly distributed. For more discussion on hash-to-prime functions we refer to [GHR99, CMS99, CS99, BBF19, OWB19].

Our First SVC Construction Now we are ready to describe our SVC scheme. For an intuition we refer the reader to the beginning of this section. Also, we note that while the intuition was given for the case of committing to a vector of bits, our actual VC construction generalizes this idea to vectors where each item is a *block of* k *bits*. This is done by creating $2k$ accumulators, each of them holding sets of indices i for specific positions inside each block v_j.

Notation and Building Blocks.

– Our message space is $\mathcal{M} = \{0,1\}^k$. Then for a vector $\vec{v} \in \mathcal{M}^n$, we denote with $i \in [n]$ the vector's position, i.e., $v_i \in \mathcal{M}$, and with $j \in [k]$ the position of its j'th bit. So $v_{i,j}$ denotes the j-th bit in position i.
– We make use of a deterministic collision resistant function $\mathsf{PrimeGen}$ that maps integers to primes. In our construction we do not need its outputs to be random (see e.g., [BBF19] for possible instantiations).
– As a building block, we use the PoProd_2 AoK from the previous section.

[12] As pointed out in [BBF18], although for the interactive version of such protocols the prime can be of size λ, the non-interactive version requires at least a double-sized prime 2λ, as an explicit square root attack was presented.

– $\mathsf{PartndPrimeProd}(I, \vec{y}) \to ((a_{I,1}, b_{I,1}), \ldots, (a_{I,k}, b_{I,k}))$: given a set of indices $I = \{i_1, \ldots, i_m\} \subseteq [n]$ and a vector $\vec{y} \in \mathcal{M}^m$, this function computes

$$(a_{I,j}, b_{I,j}) := \left(\prod_{l=1: y_{l,j}=0}^{m} p_{i_l}, \prod_{l=1: y_{l,j}=1}^{m} p_{i_l} \right) \quad \text{for } j = 1, \ldots, k$$

where $p_i \leftarrow \mathsf{PrimeGen}(i)$ for all i.
Basically, for every bit position $j \in [k]$, the function computes the products of primes that correspond to, respectively, 0-bits and 1-bits.
In the special case where $I = [n]$, we omit the set of indices from the notation of the outputs, i.e., $\mathsf{PartndPrimeProd}([n], \vec{v})$ outputs a_j and b_j.

– $\mathsf{PrimeProd}(I) \to u_I$: given a set of indices I, this function outputs the product of all primes corresponding to indices in I. Namely, it returns $u_I := \prod_{i \in I} p_i$. In the special case $I = [n]$, we denote the output of $\mathsf{PrimeProd}([n])$ as u_n.
Notice that by construction, for any I and $\vec{y}$, it always holds $a_{I,j} \cdot b_{I,j} = u_I$.

SVC Scheme. We describe our SVC scheme and then show its incremental aggregation.

$\mathsf{VC.Setup}(1^\lambda, \{0,1\}^k) \to \mathsf{crs}$ generates a hidden order group $\mathbb{G} \leftarrow \mathsf{Ggen}(1^\lambda)$ and samples three generators $g, g_0, g_1 \leftarrow \mathbb{G}$. It also determines a deterministic collision resistant function PrimeGen that maps integers to primes.
Returns $\mathsf{crs} = (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen})$

$\mathsf{VC.Specialize}(\mathsf{crs}, n) \to \mathsf{crs}_n$ computes $u_n \leftarrow \mathsf{PrimeProd}([n])$ and $U_n = g^{u_n}$, and returns $\mathsf{crs}_n \leftarrow (\mathsf{crs}, U_n)$. One can think of U_n as an accumulator to the set $[n]$.

$\mathsf{VC.Com}^\star(\mathsf{crs}_n, \vec{v}) \to (C^\star, \mathsf{aux}^\star)$ does the following:

1. Compute $((a_1, b_1), \ldots, (a_k, b_k)) \leftarrow \mathsf{PartndPrimeProd}([n], \vec{v})$; next,

$$\text{for all } j \in [k] \text{ compute } A_j = g_0^{a_j} \text{ and } B_j = g_1^{b_j}$$

   One can think of each (A_j, B_j) as a pair of RSA accumulators for two sets that constitute a partition of $[n]$ done according to the bits of $v_{1j}, \ldots, v_{nj}$. Namely A_j and B_j accumulate the sets $\{i \in [n] : v_{i,j} = 0\}$ and $\{i \in [n] : v_{i,j} = 1\}$ respectively.

2. For all $j \in [k]$, compute $C_j = A_j \cdot B_j \in \mathbb{G}$ and a proof $\pi_{\mathsf{prod}}^{(j)} \leftarrow \mathsf{PoProd_2.P}(\mathsf{crs}, (C_j, U_n), (a_j, b_j))$. Such proof ensures that the sets represented by A_j and B_j are a partition of the set represented by U_n. Since U_n is part of the CRS (i.e., it is trusted), this ensures the well-formedness of A_j and B_j.

Return $C^\star := \left(\{A_1, B_1, \ldots, A_k, B_k\}, \left\{ \pi_{\mathsf{prod}}^{(1)}, \ldots, \pi_{\mathsf{prod}}^{(k)} \right\} \right)$ and $\mathsf{aux}^\star := \vec{v}$.

$\mathsf{VC.Open}^\star(\mathsf{crs}_n, I, \vec{y}, \mathsf{aux}^\star) \to \pi_I$ proceeds as follows:

– let $J = [n] \backslash I$ and compute $((a_{J,1}, b_{J,1}), \ldots, (a_{J,k}, b_{J,k})) \leftarrow \mathsf{PartndPrimeProd}(J, \vec{v}_J)$;

– for all $j \in [k]$ compute $\mathit{\Gamma}_{I,j} := g_0^{a_{J,j}}$ and $\mathit{\Delta}_{I,j} = g_1^{b_{J,j}}$.

Notice that $a_{J,j} = a_j, / a_{I,j}$ and $b_{J,j} = b_j, / b_{I,j}$. Also $\mathit{\Gamma}_{I,j}$ is a membership witness for the set $\{i_l \in I : y_{l,j} = 0\}$ in the accumulator A_j, and similarly for $\mathit{\Delta}_{I,j}$.
Return $\pi_I := \{\pi_{I,1}, \ldots, \pi_{I,k}\} \leftarrow \{(\mathit{\Gamma}_{I,1}, \mathit{\Delta}_{I,1}), \ldots, (\mathit{\Gamma}_{I,k}, \mathit{\Delta}_{I,k})\}$

$\mathsf{VC.Ver}^\star(\mathsf{crs}_n, C^\star, I, \vec{y}, \pi_I) \to b$ computes $((a_{I,1}, b_{I,1}), \ldots, (a_{I,k}, b_{I,k}))$ using $\mathsf{PartndPrimeProd}(I, \vec{y})$, and then returns $b \leftarrow b_{acc} \wedge b_{prod}$ where:

$$b_{acc} \leftarrow \bigwedge_{j=1}^{k} \left(\Gamma_{I,j}^{a_{I,j}} = A_j \wedge \Delta_{I,j}^{b_{I,j}} = B_j \right) \tag{1}$$

$$b_{prod} \leftarrow \bigwedge_{j=1}^{k} \left(\mathsf{PoProd}_2.\mathsf{V}(\mathsf{crs}, (A_j \cdot B_j, U_n), \pi_{\mathsf{prod}}^{(j)}) \right) \tag{2}$$

Remark 4.1. For more efficient verification, $\mathsf{VC.Open}^\star$ can be changed to include $2k$ (non-interactive) proofs of exponentiation PoE (which using the PoKCR aggregation from [BBF19] add only k elements of $\mathbb{G}$). This reduces the exponentiations cost in $\mathsf{VC.Ver}^\star$. As noted in [BBF19], although the asymptotic complexity is the same, the operations are in $\mathbb{Z}_{2^{2\lambda}}$ instead of $\mathbb{G}$, which concretely makes up an improvement.

The correctness of the vector commitment scheme described above is obvious by inspection (assuming correctness of PoProd_2).

Incremental Aggregation. We show incremental aggregation of our SVC scheme.

$\mathsf{VC.Disagg}(\mathsf{crs}, I, \vec{v}_I, \pi_I, K) \to \pi_K$. Let $L := I \setminus K$, and $\vec{v}_L$ be the subvector of $\vec{v}_I$ at positions in L. Then compute $\{a_{L,j}, b_{L,j}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(L, \vec{v}_L)$, and for each $j \in [k]$ set: $\Gamma_{K,j} \leftarrow \Gamma_{I,j}^{a_{L,j}}$, $\Delta_{K,j} \leftarrow \Delta_{I,j}^{b_{L,j}}$ and return $\pi_K := \{\pi_{K,1}, \ldots, \pi_{K,k}\} := \{(\Gamma_{K,1}, \Delta_{K,1}), \ldots, (\Gamma_{K,k}, \Delta_{K,k})\}$

$\mathsf{VC.Agg}(\mathsf{crs}, (I, \vec{v}_I, \pi_I), (J, \vec{v}_J, \pi_J)) \to \pi_K := \{(\Gamma_{K,1}, \Delta_{K,1}), \ldots, (\Gamma_{K,k}, \Delta_{K,k})\}$.

1. Let $L := I \cap J$. If $L \neq \emptyset$, set $I' := I \setminus L$ and compute $\pi_{I'} \leftarrow \mathsf{VC.Disagg}(\mathsf{crs}, I, \vec{v}_I, \pi_I, I')$; otherwise let $\pi_{I'} = \pi_I$.
2. Compute $\{a_{I',j}, b_{I',j}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(I', \vec{v}_{I'})$ and $\{a_{J,j}, b_{J,j}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(J, \vec{v}_J)$.
3. Parse $\pi_{I'} := \{(\Gamma_{I',j}, \Delta_{I',j})\}_{j=1}^k$, $\pi_J := \{(\Gamma_{J,j}, \Delta_{J,j})\}_{j=1}^k$, and for all $j \in [k]$, compute $\Gamma_{K,j} \leftarrow \mathbf{ShamirTrick}(\Gamma_{I',j}, \Gamma_{J,j}, a_{I',j}, a_{J,j})$ and $\Delta_{K,j} \leftarrow \mathbf{ShamirTrick}(\Delta_{I',j}, \Delta_{J,j}, b_{I',j}, b_{J,j})$.

Note that our algorithms above can work directly with the universal CRS crs, and do not need the specialized one crs_n.

Aggregation Correctness. The second property of aggregation correctness (the one about $\mathsf{VC.Disagg}$) is straightforward by construction:
if we let $\{a_{K,j}, b_{K,j}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(K, \vec{v}_K)$, then $a_{I,j} = a_{L,j} \cdot a_{K,j}$, and thus $A_j = \Gamma_{I,j}^{a_{I,j}} = \Gamma_{I,j}^{a_{L,j} \cdot a_{K,j}} = \Gamma_{K,j}^{a_{K,j}}$ (and similarly for $\Delta_{K,j}$).

The first property instead follows from the correctness of Shamir's trick if the integer values provided as input are coprime; however since $I' \cap J = \emptyset$, $a_{I',j}$ and $a_{J,j}$ (resp. $b_{I',j}$ and $b_{J,j}$) are coprime unless a collision occurs in $\mathsf{PrimeGen}$.

Security. The security of our SVC scheme, i.e., position binding, can be reduced to the Strong RSA and Low Order assumptions in the hidden order group $\mathbb{G}$ used in the construction and to the knowledge extractability of PoProd_2.

A bit more in detail the steps of the proof are as follows. Let an adversary to the position binding output $(C, I, \vec{y}, \pi, \vec{y}', \pi')$. First from knowledge extractability of PoProd_2 it comes that $A_j B_j = g_1^{a_j} g_2^{b_j}$ and $g^{a_j b_j} = U_n = g^{u_n}$. However, this does not necessarily means that $a_j b_j = u_n$ over the integers and to prove it we need the Low Order assumptions, under which it holds. Afterwards we prove that since $A_j B_j = g_1^{a_j} g_2^{b_j}$ no different proofs π, π' for the same positions can pass the verification under the strong RSA assumption, which is the core of our proof. The main caveat of the proof is that instead of knowing that $A_j = g_1^{a_j}$ and $B_j = g_2^{b_j}$ we know only that $A_j B_j = g_1^{a_j} g_2^{b_j}$. The former case would directly reduce to RSA Accumulator's security (strong RSA assumption). For this we first need to prove an intermediate lemma which shows that specifically for our case $A_j B_j = g_1^{a_j} g_2^{b_j}$ is enough, since the choice of the primes p_i in the exponent is restricted to a polynomially bounded set.

For lack of space, the proof is in the full version. For an intuition we refer to the overview given at the beginning of this section.

Theorem 4.2 (Position-Binding). *Let* Ggen *be the generator of hidden order groups where the Strong RSA and Low Order assumptions hold, and let* PoProd_2 *be an argument of knowledge for* R_{PoProd_2}*. Then the subVector Commitment scheme defined above is position binding.*

On concrete instantiation. Our SVC construction is described generically from a hidden order group $\mathbb{G}$, an AoK PoProd_2, and a mapping to primes $\mathsf{PrimeGen}$. The concrete scheme we analyze is the one where PoProd_2 is instantiated with the non-interactive version of the PoProd_2 protocol described in Sec. 4.1. The non-interactive version needs a hash-to-prime function $\mathsf{H}_{\mathsf{prime}}$. We note that the same function can be used to instantiate $\mathsf{PrimeGen}$, though for the sake of $\mathsf{PrimeGen}$ we do not need its randomness properties. One can choose a different mapping to primes for $\mathsf{PrimeGen}$ and even just a bijective mapping (which is inherently collision resistant) would be enough: this is actually the instantiation we consider in our efficiency analysis. Finally, see Section 1.4 for a discussion on possible instantiations of $\mathbb{G}$.

We note that by using the specific PoProd_2 protocol given in Sec. 4.1 we are assuming adversaries that are generic with respect to the group $\mathbb{G}$. Therefore, our SVC is ultimately position binding in the generic group model.

4.2 Our Second SVC Construction

In this section we propose another SVC scheme with constant-size parameters and incremental aggregation. This scheme builds on the SVC of [LM19] based on the RSA assumption, which in turn extends the VC of [CF13] to support subvector openings. Our technical contribution is twofold. First, we show that the SVC of [CF13, LM19] can be modified in order to have public parameters and verification time independent of the vector's length. Second, we propose new algorithms for (incremental) aggregation and disaggregation for this SVC.

Our second SVC Construction. Let us start by giving a brief overview of the [CF13] VC scheme and of the basic idea to turn it into one with succinct parameters and verification time. In brief, in [CF13] a commitment to a vector $\vec{v}$ is $C = S_1^{v_1} \cdots S_n^{v_n}$, where each $S_i := g^{\prod_{j \in [n] \setminus \{i\}} e_j}$ with $g \in \mathbb{G}$ a random generator and e_j being distinct prime numbers (which can be deterministically generated using a suitable map-to-primes). The opening for position i is an element Λ_i such that $\Lambda_i^{e_i} \cdot S_i^{v_i} = C$ and the key idea is that such Λ_i is an e_i-th root that can be publicly computed as long as one does it for the correct position i and value v_i. Also, as it can be seen, the element S_i is necessary to verify an opening of position i, and thus $(S_1, \ldots, S_n)$ were included in the public parameters. Catalano and Fiore observed that one can remove the S_i-s from crs if the verifier opts for recomputing S_i at verification time *at the price of linear-time verification*. Our goal though is to obtain constant-size parameters *and* constant-time verification. To do that we let the prover compute S_i and include it in the opening for position i. To prevent adversaries from providing false S_i's, we store in the public parameters $U_n = g^{\prod_{i \in [n]} e_i}$ (i.e., an accumulator to all positions) so that the verifier can verify the correctness of S_i in constant-time by checking $S_i^{e_i} = U_n$. This technique easily generalizes to subvector openings.

In the following, we describe the scheme in details and then propose our incremental aggregation algorithms. To simplify our exposition, we use the following notation: for a set of indices $I \subseteq [n]$, $e_I := \prod_{i \in I} e_i$ denotes the product of all primes corresponding to the elements of I, and $S_I := g^{\prod_{i \in [n] \setminus I} e_i} = g^{e_{[n] \setminus I}} = U_n^{1/e_I}$ (which is a generalization of the former S_i), where, we recall, the e_i's are defined from the crs.

$\mathsf{VC.Setup}(1^\lambda, \ell, n) \to \mathsf{crs}$ generates a hidden order group $\mathbb{G} \leftarrow \mathsf{Ggen}(1^\lambda)$ and samples a generator $g \leftarrow_\$ \mathbb{G}$. It also determines a deterministic collision resistant function $\mathsf{PrimeGen}$ that maps integers to primes.
Returns $\mathsf{crs} = (\mathbb{G}, g, \mathsf{PrimeGen})$

$\mathsf{VC.Specialize}(\mathsf{crs}, n) \to \mathsf{crs}_n$ computes n primes of $(\ell + 1)$ bits $e_1, \ldots, e_n$, $e_i \leftarrow \mathsf{PrimeGen}(i)$ for each $i \in [n]$, and $U_n = g^{e_{[n]}}$ and returns $\mathsf{crs}_n \leftarrow (\mathsf{crs}, U_n)$. One can think of U_n as an accumulator to the set $[n]$.

$\mathsf{VC.Com}(\mathsf{crs}, \vec{v}) \to (C, \mathsf{aux})$ Computes for each $i \in [n]$, $S_i \leftarrow g^{e_{[n] \setminus \{i\}}}$ and then $C \leftarrow S_1^{v_1} \ldots S_n^{v_n}$ and $\mathsf{aux} \leftarrow (v_1, \ldots, v_n)$.

$\mathsf{VC.Open}(\mathsf{crs}, I, \vec{y}, \mathsf{aux}) \to \pi_I$ Computes for each $j \in [n] \setminus I$, $S_j^{1/e_I} \leftarrow g^{e_{[n] \setminus (I \cup \{j\})}}$ and $S_I \leftarrow g^{e_{[n] \setminus I}}$ and then

$$\Lambda_I \leftarrow \prod_{j=1, j \notin I}^{n} \left(S_j^{1/e_I}\right)^{y_j} = \left(\prod_{j=1, j \notin I}^{n} S_j^{y_j}\right)^{1/e_I}$$

Returns $\pi_I := (S_I, \Lambda_I)$

$\mathsf{VC.Ver}(\mathsf{crs}, C, I, \vec{y}, \pi_I) \to b$ Parse $\pi_I := (S_I, \Lambda_I)$, and compute $S_i = S_I^{e_{I \setminus \{i\}}} = U_n^{1/e_i}$ for every $i \in I$. Return 1 (accept) if both the following checks hold, and 0 (reject) otherwise:

$$S_I^{e_I} = U_n \ \wedge \ C = \Lambda_I^{e_I} \prod_{i \in I} S_i^{y_i}$$

The correctness of the above construction holds essentially the same as the one of the SVC of [CF13, LM19] with the addition of the S_I elements of the openings, whose correctness can be seen by inspection (and is the same as for RSA accumulators).

Incremental Aggregation. Let us now show that the SVC above has incremental aggregation. Note that our algorithms also implicitly show that the RSA-based SVC of [LM19] is incrementally aggregatable.

$\mathsf{VC.Disagg}(\mathsf{crs}, I, \vec{v}_I, \pi_I, K) \to \pi_K$ Parse $\pi_I := (S_I, \Lambda_I)$. First compute S_K from S_I, $S_K \leftarrow S_I^{e_{I \setminus K}}$, and then, for every $j \in I \setminus K$, $\chi_j = S_K^{1/e_j}$, e.g., by computing $\chi_j \leftarrow S_I^{e_{I \setminus (K \cup \{j\})}}$.
Return $\pi_K := (S_K, \Lambda_K)$ where

$$\Lambda_K \leftarrow \Lambda_I^{e_{I \setminus K}} \cdot \prod_{j \in I \setminus K} \chi_j^{v_j}$$

$\mathsf{VC.Agg}(\mathsf{crs}, (I, \vec{v}_I, \pi_I), (J, \vec{v}_J, \pi_J)) \to \pi_K$ Parse $\pi_I := (S_I, \Lambda_I)$ and similarly π_J. Also, let $K = I \cup J$, and assume for simplicity that $I \cap J = \emptyset$ (if this is not the case, one could simply disaggregate π_I (or π_J) to $\pi_{I \setminus J}$ (or $\pi_{J \setminus I}$)).
First, compute $S_K \leftarrow \mathbf{ShamirTrick}(S_I, S_J, e_I, e_J)$. Next, compute $\phi_j \leftarrow S_K^{e_{J \setminus \{j\}}} = S_I^{1/e_j}$ for every $j \in J$, and similarly $\psi_i \leftarrow S_K^{e_{I \setminus \{i\}}} = S_J^{1/e_i}$ for every $i \in I$. Then compute

$$\rho_I \leftarrow \frac{\Lambda_I}{\prod_{j \in J} \phi_j^{v_j}} \qquad \text{and} \qquad \sigma_J \leftarrow \frac{\Lambda_J}{\prod_{i \in I} \psi_i^{v_i}}$$

Return $\pi_K := (S_K, \Lambda_K)$ where $\Lambda_K \leftarrow \mathbf{ShamirTrick}(\rho_I, \sigma_J, e_I, e_J)$.

Aggregation Correctness. It follows from the correctness of Shamir's trick and by construction. The details are in the full version

Security. For the security of the above SVC scheme we observe that the difference with the corresponding [LM19] lies in the generation of S_i's. In [LM19] they are generated in the trusted setup phase, thus they are considered "well-formed" in the security proof. In our case, the S_i's are reconstructed during verification time from the S_I that comes in the opening π_I which can (possibly) be generated in an adversarial way. However, in the verification it is checked that $S_I^{e_I} = U$, where $U = g^{e_{[n]}}$ is computed in the trusted setup. So under the Low Order assumption we get that S_I has the correct form, $S_I = g^{e_{[n]}/e_I} = g^{e_{[n] \setminus I}}$, with overwhelming probability. Except for this change, the rest reduces to the position binding of the [LM19] SVC. The proof of the theorem is in the full version.

Theorem 4.3 (Position-Binding). *Let* Ggen *be the generator of hidden order groups where the Low Order assumption holds and the [LM19] SVC is position binding. Then the SVC scheme defined above is position binding.*

As showed in [LM19], their SVC is position binding under the strong Distinct-Prime-Product Root assumption in the standard model. We conclude that the above SVC is position binding in hidden order groups where the Low Order and the Strong Distinct-Prime-Product Root assumptions hold.

4.3 Comparison with Related Work

We compare our two SVC schemes with the recent scheme proposed by Boneh et al. [BBF19] and the one by Lai and Malavolta [LM19], which extends [CF13] to support subvector openings.[13] We present a detailed comparison in Table 1, considering to work with vectors of length N of ℓ-bit elements and security parameter λ. In particular we consider an instantiation of our first SVC with $k = 1$ (and thus $n = N \cdot \ell$). A detailed efficiency analysis of our schemes is in the full version.

SETUP MODEL. [BBF19] works with a fully universal CRS, whereas our schemes have both a universal CRS with deterministic specialization, which however, in comparison to [CF13, LM19], outputs *constant-size* parameters instead of linear.

AGGREGATION. The VC of [BBF19] supports aggregation only on openings created by $\mathsf{VC.Open}$ (i.e., it is one-hop) and does not have disaggregatable proofs (unless in a different model where one works linearly in the length of the vector or knows the full vector). In contrast, we show the first schemes that satisfy incremental aggregation (also, our second one immediately yields a method for the incremental aggregation of [LM19]). As we mention later, incremental aggregation can be very useful to precompute openings for a certain number of vector blocks allowing for interesting time-space tradeoffs that can speedup the running time of $\mathsf{VC.Open}$.

EFFICIENCY. From the table, one can see that our first SVC has: slightly worse commitments size than all the other schemes, computational asymptotic performances similar to [BBF19], and opening size slightly better than [BBF19]. Our second SVC is the most efficient among the schemes with constant-size parameters; in particular, it has faster asymptotics than our first SVC and [BBF19] for having a smaller logarithmic factor (e.g., $\log(N - m)$ vs. $\log(\ell N)$), which is due to the avoidance of using one prime per bit of the vector. In some cases, [CF13, LM19] is slightly better, but this is essentially a benefit of the linear-size parameters, namely the improvement is due to having the S_i's elements already precomputed.

When considering applications in which a user creates the commitment to a vector and (at some later points in time) is requested to produce openings for various subvectors, *our incremental aggregation property leads to use preprocessing to achieve more favorable time and memory costs*. In a nutshell, the idea of preprocessing is that one can precompute and store information that allows to speedup the generation of openings, in particular by making opening time less dependent on the total length of the vector. Our method in Section 3.2 works generically for any SVC that has incremental aggregation. A similar preprocessing solution can also be designed for the SVC of [BBF19] by using its one-hop aggregation; we provide a detailed description of the method in the full version. The preprocessing for [BBF19] however has no flexibility in choosing how much auxiliary storage can be used, and one must store (a portion of) a non-membership witness *for every bit* of the vector.

Even in the simplest case of $B = 1$ (shown in Table 1) both our SVCs save a factor ℓ in storage, which concretely turns into $3\times$ less storage.

[13] We refer to [BBF19] to see how these schemes compare with Merkle trees.

Furthermore we support flexible choices of B thus allowing to tune the amount of auxiliary storage. For instance, we can choose $B = \sqrt{N}$ so as to get $2\sqrt{N}|\mathbb{G}|$ bits of storage, and opening time about $O(\ell m \log n(\sqrt{n} + \log m))$ and $O(m(\sqrt{n} + \log^2 m))$ in the first and second scheme respectively. Our flexibility may also allow one to choose the buckets size B and their distribution according to applications-dependent heuristics; investigating its benefit may be an interesting direction for future work.

Metric	**Our First SVC**	**Our Second SVC**	**[BBF19]**	**[CF13, LM19]**
	Setup			
VC.Setup	$O(1)$	$O(1)$	$O(1)$	$O(1)$
$\lvert\mathsf{crs}\rvert$	$3\ \lvert\mathbb{G}\rvert$	$1\ \lvert\mathbb{G}\rvert$	$1\ \lvert\mathbb{G}\rvert$	$1\ \lvert\mathbb{G}\rvert$
VC.Specialize	$O(\ell \cdot N \cdot \log(\ell N))\ \mathbb{G}$	$O(\ell \cdot N)\ \mathbb{G}$	—	$O(\ell \cdot N \cdot \log N)\ \mathbb{G}$
$\lvert\mathsf{crs}_N\rvert$	$1\ \lvert\mathbb{G}\rvert$	$1\ \lvert\mathbb{G}\rvert$	—	$N\ \lvert\mathbb{G}\rvert$
	Commit a vector $\vec{v} \in (\{0,1\}^\ell)^N$			
VC.Com	$O(\ell \cdot N \cdot \log(\ell N))\ \mathbb{G}$	$O(\ell \cdot N \cdot \log N)\ \mathbb{G}$	$O(\ell \cdot N \cdot \log(\ell N))\ \mathbb{G}$	$O(\ell \cdot N)\ \mathbb{G}$
$\lvert C\rvert$	$4\ \lvert\mathbb{G}\rvert + 2\ \lvert\mathbb{Z}_{2^{2\lambda}}\rvert$	$1\ \lvert\mathbb{G}\rvert$	$1\ \lvert\mathbb{G}\rvert$	$1\ \lvert\mathbb{G}\rvert$
	Opening and Verification for $\vec{v}_I$ **with** $\lvert I\rvert = m$			
VC.Open	$O(\ell \cdot (N-m) \cdot \log(\ell N))\ \mathbb{G}$	$O(\ell \cdot (N-m) \cdot \log(N-m))\ \mathbb{G}$	$O(\ell \cdot (N-m) \cdot \log(\ell N))\ \mathbb{G}$	$O(\ell \cdot (N-m) \cdot m \log m)\ \mathbb{G}$
$\lvert\pi_I\rvert$	$3\ \lvert\mathbb{G}\rvert$	$2\ \lvert\mathbb{G}\rvert$	$5\ \lvert\mathbb{G}\rvert + 1\ \lvert\mathbb{Z}_{2^{2\lambda}}\rvert$	$1\ \lvert\mathbb{G}\rvert$
VC.Ver	$O(\ell \cdot m \cdot \log(\ell N))\ \mathbb{Z}_{2^{2\lambda}} + O(\lambda)\ \mathbb{G}$	$O(\ell \cdot m \log m)\ \lvert\mathbb{G}\rvert$	$O(m \cdot \ell \cdot \log(\ell N))\ \mathbb{Z}_{2^{2\lambda}} + O(\lambda)\ \mathbb{G}$	$O(\ell \cdot m)\ \mathbb{G}$
	Commitment and Opening with Precomputation			
VC.Com	$O(\ell \cdot N \cdot \log(\ell \cdot N) \cdot \log(N))\ \mathbb{G}$	$O(\ell \cdot N \log^2(N))\ \mathbb{G}$	$O(\ell \cdot N \cdot \log(\ell \cdot N) \cdot \log(N))\ \mathbb{G}$	$O(\ell \cdot N \log^2(N))$
$\lvert\mathsf{aux}\rvert$	$2N\ \lvert\mathbb{G}\rvert$	$2N\ \lvert\mathbb{G}\rvert$	$2N\ \lvert\mathbb{G}\rvert + O(\ell \cdot N \log(\ell N))$	$2N\ \lvert\mathbb{G}\rvert$
VC.Open	$O(m \cdot \ell \cdot \log(m) \log(\ell N))\ \mathbb{G}$	$O(m \cdot \ell \cdot \log^2 m)\ \mathbb{G}$	$O(m \cdot \ell \cdot \log(m) \log(\ell N))\ \mathbb{G}$	$O(m \cdot \ell \cdot \log^2(m))\ \mathbb{G}$
Aggregation	Incremental	Incremental	One-hop	Incremental
Disaggregation	Yes	Yes	No	Yes

Table 1. Comparison between the SVC's of [BBF19], [LM19] and this work; our contributions are highlighted in gray. We consider committing to a vector $\vec{v} \in (\{0,1\}^\ell)^N$ of length N, and opening and verifying for a set I of m positions. By '$O(x)\ \mathbb{G}$' we mean $O(x)$ group operations in $\mathbb{G}$; $|\mathbb{G}|$ denotes the bit length of an element of $\mathbb{G}$. An alternative algorithm for $\mathsf{VC.Open}$ in [LM19] costs $O(\ell \cdot (N-m) \cdot \log(N-m))$. Our precomputation is for $B = 1$.

4.4 Experimental Evaluation

We have implemented in Rust our first SVC scheme of section 4.1 (with and without preprocessing) and the recent SVC of [BBF19] (referred as BBF in what follows). Here we discuss an experimental evaluation of these schemes. [14] Below is a summary of the comparison, details of the experiments are in the full version.

- Our SVC construction is faster in opening and verification than BBF (up to $2.5\times$ and $2.3\times$ faster respectively), but at the cost of a slower commitment stage (up to $6\times$ slower). These differences tend to flatten for larger vectors and opening sizes.
- Our SVC construction with preprocessing allows for extremely fast opening times compared to non-preprocessing constructions. Namely, it can reduce the running time by several orders of magnitude for various choices of vector and opening sizes, allowing to obtain practical opening times—of the order of seconds—that would be impossible without preprocessing—of the order of hundred of seconds. In a file of 1 Mibit

[14] We did not include BBF with precomputation in our experimental evaluation because this scheme has worse performances than our preprocessing construction in terms of both required storage and running time. We elaborate on this in the full version.

(2^{20} bits), preprocessing reduces the time to open 2048 bits from one hour to less than 5 seconds! This efficient opening, however, comes at the cost of a one-time preprocessing (during commitment) and higher storage requirements. We discuss how to mitigate these space requirements by trading for opening time and/or communication complexity later in this section. We stress that it is thanks to the incremental aggregation property of our construction that allows these tradeoffs (they are not possible in BBF with preprocessing).

– Although our SVC construction with preprocessing has an expensive commitment stage, this tends to be amortized throughout very few openings[15], as few as 30 (see full version for more details). These effects are particularly significant over a higher number of openings: over 1000 openings our SVC construction with preprocessing has an amortized cost of less than 6 seconds, while our SVC construction and BBF have amortized openings above 90 seconds.

Time/Storage Tradeoffs Our construction allows for some tradeoffs between running times and storage by selecting larger precomputed chunks or by committing to hashed blocks of the file. See the full version for a detailed discussion.

5 Verifiable Decentralized Storage

In this section we introduce verifiable decentralized storage (VDS). We recall that in VDS there are two types of parties (called nodes): the generic *client nodes* and the more specialized *storage nodes* (a storage node can also act as a client node). We refer the reader to Section 1.2 for a discussion on the motivation and requirements of VDS.

5.1 Syntax

Here we introduce the syntax of VDS. A VDS scheme is defined by a collection of algorithms that are to be executed by either storage nodes or client nodes. The only exception is the Bootstrap algorithm that is used to bootstrap the entire system and is assumed to be executed by a trusted party, or to be implemented in a distributed fashion (which is easy if it is public coin).

The syntax of VDS reflects its goal: guaranteeing data integrity in a highly dynamic and decentralized setting (the file can change and expand/shrink often and no single node stores it all). In VDS we create both parameters and an initial commitment for an empty file at the beginning (through the probabilistic Bootstrap algorithm, which requires a trusted execution). From then on this commitment is changed through incremental updates (of arbitrary size). Updating is divided in two parts. A node can carry out an update and "push" it to all the other nodes, i.e. providing auxiliary information (that we call "update hint") other nodes can use to update their local cer-

[15] Amortized opening time roughly represents how computationally expensive a scheme is "in total" throughout all its operations. *Amortized opening time for m openings* is the cost of one commitment plus the cost of m openings, all averaged over the m openings.

tificates (if affected by the change) and a new digest[16]. These operations are done respectively through $\mathsf{StrgNode.PushUpdate}$ and $\mathsf{StrgNode.ApplyUpdate}$. Opening and verifying are where VC (with incremental aggregation) and VDS share the same mechanism. To respond to a query, a storage node can produce (possibly partial) proofs of opening via the $\mathsf{StrgNode.Retrieve}$ algorithm. If these proofs need to be aggregated, any node can use algorithm $\mathsf{AggregateCertificates}$. Anyone can verify a proof through $\mathsf{ClntNode.VerRetrieve}$.

Some more details about our notation follow. In VDS we model the files to be stored as vectors in some message space $\mathcal{M}$ (e.g., $\mathcal{M} = \{0, 1\}$ or $\{0, 1\}^\ell$), i.e., $\mathsf{F} = (\mathsf{F}_1, \ldots, \mathsf{F}_N)$. Given a file F, we define a *portion* of it as a pair (I, F_I) where F_I is essentially the I-subvector of F. We denote input (resp. output) states by st (resp. st'). Update operations op are modifications, additions or deletions, i.e. $\mathsf{op} \in \{\mathsf{mod}, \mathsf{add}, \mathsf{del}\}$, and $\varDelta$ denotes the update description , e.g., which positions to change and the new values. We denote by $\varUpsilon_\varDelta$ the *update hint* that whoever is producing the update can share with other nodes to generate a new digest from the changes. The output bit b marks acceptance/rejection. For a query Q, we mark by π_Q a certificate vouching for a response F_Q.

Definition 5.1 (Verifiable Decentralized Storage). *Algorithm to bootstrap the system:*

$\mathsf{Bootstrap}(1^\lambda) \to (\mathsf{pp}, \delta_0, \mathsf{st}_0)$ *which outputs a digest and storage node's local state for an empty file. All the algorithms below implicitly take the public parameters* pp *as input.*

The algorithms for storage nodes are:

$\mathsf{StrgNode.AddStorage}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, Q, \mathsf{F}_Q, \pi_Q) \to (\mathsf{st}', J, \mathsf{F}_J)$ *by which a storage node can extend its storage from* (I, F_I) *to* $(J, \mathsf{F}_J) := (I, \mathsf{F}_I) \cup (Q, \mathsf{F}_Q)$. *Note: this allows anyone holding a valid certificate for a file portion* F_Q *to become a storage node of such portion.*

$\mathsf{StrgNode.RmvStorage}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, K) \to (\mathsf{st}', J, \mathsf{F}_J)$ *by which a storage node can shrink its local storage to* (J, F_J).

$\mathsf{StrgNode.PushUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \varDelta) \to (\delta', n', \mathsf{st}', J, \mathsf{F}'_J, \varUpsilon_\varDelta)$ *which allows a storage node to perform an update on* (I, F_I) *generating a corresponding new digest, length and local view, along with hint* $\varUpsilon_\varDelta$ *others can use to update their own digests/local view.*

$\mathsf{StrgNode.ApplyUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \varDelta, \varUpsilon_\varDelta) \to (b, \delta', n', \mathsf{st}', J, \mathsf{F}'_J)$ *which allows a storage node to incorporate changes in a file pushed by another node.*

$\mathsf{StrgNode.Retrieve}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, Q) \to (\mathsf{F}_Q, \pi_Q)$ *which allows a storage node to respond to a query and to create a certificate vouching for the correctness of the returned blocks.*

The algorithms for clients nodes are:

$\mathsf{ClntNode.ApplyUpdate}(\delta, \mathsf{op}, \varDelta, \varUpsilon_\varDelta) \to (b, \delta')$ *which updates a digest by hint* $\varUpsilon_\varDelta$.

[16] One can also see this update hint as a certificate to check that a new digest is consistent with some changes. This issue does not arise in our context at all but the $\mathsf{Bootstrap}$ algorithms are deterministic.

$\mathsf{ClntNode.VerRetrieve}(\delta, Q, \mathsf{F}_Q, \pi_Q) \to b$ *which verifies a response to a query.*

$\mathsf{AggregateCertificates}(\delta, (I, \mathsf{F}_I, \pi_I), (J, \mathsf{F}_J, \pi_J)) \to \pi_K$ *which aggregates two certificates π_I and π_J into a single certificate π_K (with $K := I \cup J$). In a running VDS, any node can aggregate two (or more) incoming certified data blocks into a single certified data block.*

Remark 5.1 (On CreateFrom). For completeness, our VDS syntax also includes the functionalities (StrgNode.CreateFrom, ClntNode.GetCreate) that allow a storage node to initialize storage (and corresponding digest) for a new file that is a subset of an existing one, and a client node to verify such resulting digest. Although this feature can be interesting in some application scenarios, we still see it as an extra feature that may or may not be satisfied by a VDS construction. We refer to the full version for more discussion and a detailed description of this functionality.

5.2 Correctness and Efficiency of VDS

Intuitively, we say that a VDS scheme is *efficient* if running VDS has a "small" overhead in terms of the storage required by all the nodes and the bandwidth to transmit certificates. More formally, a VDS scheme is said efficient if there is a fixed polynomial $p(\cdot)$ such that $p(\lambda, \log n)$ (with λ the security parameter and n the length of the file) is a bound for all certificates and advices generated by the VDS algorithms as well as for digests δ and the local state st of storage nodes. Note that combining this bound with the requirement that all algorithms are polynomial time in their input, we also get that no VDS algorithm can run linearly in the size of the file (except in the trivial case that the file is processed in one shot, e.g., in the first StrgNode.AddStorage).

Efficiency essentially models that running VDS is cost-effective for all the nodes in the sense that it does not require them to store significantly more data than they would have to store without. Notice that by requiring certificates to have a fixed size implies that they do not grow with aggregation.

For correctness, intuitively speaking, we want that for any (valid) evolution of the system in which the VDS algorithms are honestly executed we get that any storage node storing a portion of a file F can successfully convince a client holding a digest of F about retrieval of any portion of F. And such (intuitive notion of) correctness is also preserved when updates, aggregations, or creations of new files are done.

Turning this intuition into a formal correctness definition turned out to be nontrivial. This is due to the distributed nature of this primitive and the fact that there could be many possible ways in which, at the time of answering a retrieval query, a storage node may have reached its state starting from the empty node state. The basic idea of our definition is that an empty node is "valid", and then any "valid" storage node that runs StrgNode.PushUpdate "transfers" such validity to both itself and to other nodes that apply such update. A bit more precisely, we model "validity" as the ability to correctly certify retrievals of any subsets of the stored portion. A formal correctness definition follows. To begin with, we define the notion of validity for the view of a storage node.

Definition 5.2 (Validity of storage node's view). *Let* pp *be public parameters as generated by* Bootstrap. *We say that a local view* $(\delta, n, \mathsf{st}, I, \mathsf{F}_I)$ *of a storage node* *is* valid *if* $\forall Q \subseteq I$*:* $\mathsf{ClntNode.VerRetrieve}(\delta, Q, \mathsf{F}_Q, \pi_Q) = 1$*, where* $(\mathsf{F}_Q, \pi_Q) \leftarrow \mathsf{StrgNode.Retrieve}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, Q)$

Remark 5.2. By Definition 5.2 the output of a bootstrapping algorithm $(\mathsf{pp}, \delta_0, \mathsf{st}_0) \leftarrow \mathsf{Bootstrap}(1^\lambda)$ is always such that $(\mathsf{pp}, \delta_0, 0, \mathsf{st}_0, \emptyset, \emptyset)$ is valid. This provides a "base case" for Definition 5.4.

Second, we define the notion of admissible update, which intuitively models when a given update can be meaningfully processed, locally, by a storage node.

Definition 5.3 (Admissible Update). *An update* $(\mathsf{op}, \varDelta)$ *is* admissible *for* (n, I, F_I) *if:*

- *for* $\mathsf{op} = \mathsf{mod}$*,* $K \subseteq I$ *and* $|\mathsf{F}'_K| = |K|$*, where* $\varDelta := (K, \mathsf{F}'_K)$*.*
- *for* $\mathsf{op} = \mathsf{add}$*,* $K \cap I = \emptyset$ *and* $|\mathsf{F}'_K| = |K|$ *and* $K = \{n+1, n+2, \ldots, n+|K|\}$*, where* $\varDelta := (K, \mathsf{F}'_K)$*.*
- *for* $\mathsf{op} = \mathsf{del}$*,* $K \subseteq I$ *and* $K = \{n - |K| + 1, \ldots, n\}$*, where* $\varDelta := K$*.*

In words, the above definition formalizes that: to push a modification at positions K, the storage node must store those positions; to push an addition, the new positions K must extend the currently stored length of the file; to push a deletion of position K, the storage node must store data of the positions to be deleted and those positions must also be the last $|K|$ positions of the currently stored file (i.e., the file length is reduced).

Definition 5.4 (Correctness of VDS). *A VDS scheme* VDS *is* correct *if for all honestly generated parameters* $(\mathsf{pp}, \delta_0, \mathsf{st}_0) \leftarrow \mathsf{Bootstrap}(1^\lambda)$ *and any storage node's local view* $(\delta, n, \mathsf{st}, I, \mathsf{F}_I)$ *that is valid, the following conditions hold.*
UPDATE CORRECTNESS. *For any update* $(\mathsf{op}, \varDelta)$ *that is admissible for* (n, I, F_I) *and for any* $(\delta', n', \mathsf{st}', J, \mathsf{F}'_J, \varUpsilon_\varDelta) \leftarrow \mathsf{StrgNode.PushUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \varDelta)$*:*

1. $(\mathsf{pp}, \delta', n', \mathsf{st}', J, \mathsf{F}'_J)$ *is valid;*
2. *for any valid* $(\delta, n, \mathsf{st}_s, I_s, \mathsf{F}_{I_s})$*, if* $(b_s, \delta'_s, n', \mathsf{st}'_s, I'_s, \mathsf{F}'_s) \leftarrow \mathsf{StrgNode.ApplyUpdate}(\delta, n, \mathsf{st}_s, I_s, \mathsf{F}_{I_s}, \mathsf{op}, \varDelta, \varUpsilon_\varDelta)$ *then we have:* $b_s = 1$*,* $\delta'_s = \delta'$*,* $n'_s = n'$*, and* $(\delta'_s, n'_s, \mathsf{st}'_s, I'_s, \mathsf{F}'_s)$ *is valid;*
3. *if* $(b_c, \delta'_c) \leftarrow \mathsf{ClntNode.ApplyUpdate}(\delta, \mathsf{op}, \varDelta, \varUpsilon_\varDelta)$*, then* $\delta'_c = \delta'$ *and* $b_c = 1$*.*

ADD-STORAGE CORRECTNESS. *For any* (Q, F_Q, π_Q) *such that*
$\mathsf{ClntNode.VerRetrieve}(\delta, Q, \mathsf{F}_Q, \pi_Q) = 1$*, if* $(\mathsf{st}', J, \mathsf{F}_J) \leftarrow \mathsf{StrgNode.AddStorage}(\delta, \mathsf{st}, I, \mathsf{F}, Q, \mathsf{F}_Q, \pi_Q)$ *then* $(\delta, n, \mathsf{st}', J, \mathsf{F}_J)$ *is valid.*
REMOVE-STORAGE CORRECTNESS. *For any* $K \subseteq I$*,*
if $(\mathsf{st}', J, \mathsf{F}_J) \leftarrow \mathsf{StrgNode.RmvStorage}(\delta, \mathsf{st}, I, \mathsf{F}, K)$ *then* $(\delta, n, \mathsf{st}', J, \mathsf{F}_J)$ *is valid.*
CREATE CORRECTNESS. *For any* $J \subseteq I$*, if* $(\delta', n', \mathsf{st}', J, \mathsf{F}_J, \varUpsilon_J)$ *is output of*
$\mathsf{StrgNode.CreateFrom}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, J)$ *and* $(b, \delta'') \leftarrow \mathsf{ClntNode.GetCreate}(\delta, J, \varUpsilon_J)$*, then* $b = 1$*,* $n' = |J|$*,* $\delta'' = \delta'$ *and* $(\mathsf{pp}, \delta', n', \mathsf{st}', J, \mathsf{F}_J)$ *is valid.*
AGGREGATE CORRECTNESS. *For any pair of triples* (I, F_I, π_I) *and* (J, F_J, π_J) *such that*
$\mathsf{ClntNode.VerRetrieve}(\delta, I, \mathsf{F}_I, \pi_I) = 1$ *and* $\mathsf{ClntNode.VerRetrieve}(\delta, J, \mathsf{F}_J, \pi_J) = 1$*,*
if $\pi_K \leftarrow \mathsf{AggregateCertificates}((I, \mathsf{F}_I, \pi_I), (J, \mathsf{F}_J, \pi_J))$ *and* $(K, \mathsf{F}_K) := (I, \mathsf{F}_I) \cup (J, \mathsf{F}_J)$*, then*
$\mathsf{ClntNode.VerRetrieve}(\delta, K, \mathsf{F}_K, \pi_K) = 1$.

Remark 5.3 (Relation with Updatable VCs). Our notion of VDS is very close to the notion of updatable VCs [CF13] extended to support subvector openings and incremental aggregation. On a syntactical level, in comparison to updatable VCs, our VDS notion makes more evident the decentralized nature of the primitive, which is reflected in the definition of our algorithms where for example it is clear that no one ever needs to store/know the entire file. One major difference is that in VDS the public parameters must necessarily be *short* since no node can run linearly in the size of the file (nor it can afford such storage), whereas in VCs this may not be necessarily the case. Another difference is that in updatable VCs [CF13] updates can be received without any hint, which is instead the case in VDS. Finally, it is interesting to note that, as of today, there exists no VC scheme that is updatable, incrementally aggregatable and with subvector openings, that enjoys short parameters and has the required short verification time. So, in a way, our two VDS realizations show how to bypass this barrier of updatable VC by moving to a slightly different (and practically motivated) model.

5.3 Security of VDS

In this section we discuss the security definition of VDS schemes. For lack of space a formal definition is in the full version. Intuitively speaking, we require that a malicious storage node (or a coalition of them) cannot convince a client of a false data block in a retrieval query. To formalize this, we let the adversary fully choose a *history* of the VDS system that starts from the empty state and consists of a sequence of steps, where each step is either an update (addition, deletion, modification) or a creation (from an existing file) and is accompanied by an advice. A client's digest δ is updated following such history and using the adversarial advices, and similarly one gets a file F corresponding to such digest. At this point, the adversary's goal is to provide a tuple $(Q, \pi_Q, \mathsf{F}^*_Q)$ that is accepted by a client with digest δ but where $\mathsf{F}^*_Q \neq \mathsf{F}_Q$.

VDS PROOF OF STORAGE. As an additional security mechanism we consider the possibility to ensure a client that a given file is stored by the network at a certain point of time without having to retrieve it. To this end, we extend the VDS notion to provide a *proof of storage* mechanism in the form of a proof of retrievability (PoR) [JK07] or a proof of data possession (PDP) [ABC$^+$07]. Our proof of storage model for VDS is such that proofs are publicly verifiable given the file's digest. Also, in order to support the decentralized and open nature of DSNs, the entire proof mechanism should not use any secret, and proofs should be generatable in a distributed fashion (this is a main distinguishing feature compared to existing PoRs/PDPs) while staying compact. The formalization of this property is in the full version.

5.4 Realizing VDS

We show two realizations of VDS in hidden-order groups, summarized below.

Theorem 5.1 (VDS_1)**.** *Under the strong RSA assumption in a hidden-order group $\mathbb{G}$, there exists a VDS scheme VDS_1 in which, for a file F: a digest δ_F is $2|\mathbb{G}| + \log|\mathsf{F}|$ bits-long; a storage node holding (I, F_I) keeps a state st_I of $2|\mathbb{G}|$ bits, answers retrieval of* *portion Q with a certificate of $2|\mathbb{G}|$ bits in time $O(\ell \cdot (|I| - |Q|) \log |\mathsf{F}|)$, and pushes an update Δ in time $O(\ell \cdot |I| \log |\mathsf{F}|)$ for $\mathsf{op} = \mathsf{mod}$, $O(\ell \cdot |\Delta| \log |\mathsf{F}|)$ for $\mathsf{op} = \mathsf{add}$, and $O(\ell \cdot (|I| - |\Delta|) \log |\mathsf{F}|)$ for $\mathsf{op} = \mathsf{del}$; a client verifies a query for positions in Q (resp. an update Δ) in time $O(\ell \cdot |Q| \log |F|)$ (resp. $O(\ell \cdot |\Delta| \log |F|)$).*

Theorem 5.2 (VDS_2)**.** *Under the strong distinct-prime-product root and the Low Order assumptions in a hidden-order group $\mathbb{G}$, there exists a VDS scheme VDS_2 in which, for a file F: a digest δ_F is $2|\mathbb{G}| + \log |\mathsf{F}|$ bits-long; a storage node holding (I, F_I) keeps a state st_I of $2|\mathbb{G}|$ bits, answers retrieval of portion Q with a certificate of $2|\mathbb{G}|$ bits in time $O(\ell \cdot (|I| - |Q|) \log(|I| - |Q|))$, and pushes an update Δ in time $O(\ell \cdot |\Delta| \log |\Delta|)$ for $\mathsf{op} = \mathsf{mod}, \mathsf{add}$, and $O(\ell \cdot (|I| + |\Delta| \log |\Delta|))$ for $\mathsf{op} = \mathsf{del}$; a client verifies a query for positions in Q (resp. an update Δ) in time $O(\ell \cdot |Q| \log |Q|)$ (resp. $O(\ell \cdot |\Delta| \log |\Delta|)$).*

In terms of assumptions, VDS_1 is based on a weaker assumption than VDS_2 (although the assumptions are equivalent when $\mathbb{G}$ is instantiated with RSA groups).

In terms of performances, as one can see, VDS_1 and VDS_2 do similarly, with VDS_2 being slightly more efficient. In VDS_1 the complexity of all operations includes a factor $\alpha = \log |\mathsf{F}|$, whereas in VDS_2 operations are affected by a factor logarithmic only in the number of positions involved in the given operation (e.g., how many are updated), which is typically much smaller than the entire file. Also, VDS_2 has the interesting feature that storage nodes can add and modify values in time which depends only on the update size but not on the size of the local storage.

Finally, VDS_1 has the additional feature of being compatible with our succinct arguments of knowledge, which enable the $\mathsf{StrgNode.CreateFrom}$ functionality and compact Proofs of Data Possession (see next section for an intuition and the full version for the details).

The main ideas of the two constructions are described in the following paragraphs; full constructions are in the full version.

Our First VDS Construction Our first VDS VDS_1 is obtained by extending the techniques used for our SVC of Section 4.1.

Let us assume for a moment that a digest for file F is a commitment to F. Then, a storage node holding a portion (I, F_I) keeps as local state $\mathsf{st}_I = \pi_I = (\Gamma_I, \Delta_I)$, and this clearly enables it to certify retrieval queries for any portion $Q \subseteq I$ by using disaggregation in order to create π_Q from π_I. Moreover, such certificates of retrieval queries can be arbitrarily aggregated over the network.

In order to support updates, the main obstacle is that our commitment cannot be publicly updated without knowing the entire vector due to the presence of the AoK of union of Acc_0 and Acc_1. To solve this, we exploit the fact that in the VDS security model the digest provided by the adversary must be compatible with the claimed history of changes. So we can remove the AoK. Then, updating the digest boils down to updating the two RSA accumulators $(\mathsf{Acc}_0, \mathsf{Acc}_1)$ appropriately. For instance, changing the i-th bit from 0 to 1 requires to remove p_i from Acc_0 (i.e., $\mathsf{Acc}_0' = \mathsf{Acc}_0^{1/p_i}$ computable through π_I) and adding it to Acc_1 (i.e., $\mathsf{Acc}_1' = \mathsf{Acc}_1^{p_i}$). This can be performed by a storage node holding positions in the set I such that $i \in I$, and verified by anyone having previous and new digest. As we show in the full description of the scheme, by using similar ideas other storage nodes holding other positions, say J, can also update their local state st_J accordingly.

Finally, in this VDS we take advantage of our efficient AoK protocols to support two additional features. The first one is a compact proof of data possession by which the storage node can convince a verifier that it stores a certain subset of positions without sending the data at those positions. The second one is what we call "CreateFrom": a storage node holding a prefix F' of F can publish a new digest $\delta_{\mathsf{F}'}$ corresponding to F' as a new file, and to convince any client about its correctness without the need for them to know neither F' nor F.

Our Second VDS Construction Our second scheme VDS_2 is obtained by modifying our second SVC scheme from Section 4.2 and makes key use of its aggregation/disaggregation properties.

As in our first VDS scheme, a storage node holding (I, F_I) keeps an opening π_I as local state, and uses our disaggregation and aggregation methods to certify retrieval queries for $Q \subset I$.

Let us now turn to how we can support updates. Let us consider an update on a subset K of the vector. First, the commitment is updatable as $C' \leftarrow C \cdot \prod_{i \in K} S_i^{\mathsf{F}'_i - \mathsf{F}_i}$. To update the opening proof, which we recall is $\pi_I := (S_I, \Lambda_I)$, we note that the Λ_I-part is updatable without the need of hint as $\Lambda'_I \leftarrow \Lambda_I \cdot \left(\prod_{j \in K \setminus I} S_j^{1/\prod_{i \in I} e_i}\right)^{\mathsf{F}'_j - \mathsf{F}_j}$. This part works as in [CF13] with some additional techniques that let a node do this in time $O(|I| + |K| \log |K|)$ and without having to store all the S_j values. The S_I-part resembles an RSA accumulator witness as observed in section 4.2, and thus we can use techniques similar to those of our first VDS construction to update it. That is, upon update on K, S_K is sufficient for any node to update S_I (more details are in the full version).

A remaining problem is that the SVC scheme works with a specialized CRS, $U_n = g^{e_{[n]}}$, which depends on the vector's length. In the SVC schemes, this CRS is generated (deterministically) only once, but in VDS the vector's length evolves according to the updates, i.e., for each addition or deletion U_n should also be updated. To solve this problem, in our VDS_2 scheme we make U_n part of the digest together with C, and each node is responsible to verifiably update U_n. Technically, U_n is an RSA accumulator to the vector positions, and thus it can be updated by using techniques similar to our first scheme.

Acknowledgements We thank Ben Fisch for valuable clarifications about the notions of Proof of Retrievable Commitment and Proof of Replication, and Justin Drake for pointing out the need (due to the attack discussed in [BBF18]) of using a hash function mapping into $\mathsf{Primes}(2\lambda)$ in the Fiat-Shamir transformation when making our succinct arguments of knowledge non-interactive.

Research leading to these results has been partially supported by the Spanish Government under projects SCUM (ref. RTI2018-102043-B-I00), CRYPTOEPIC (refs. ERC2018-092822, EUR2019-103816), and SECURITAS (ref. RED2018-102321-T), by the Madrid Regional Government under project BLOQUES (ref. S2018/TCS-4339), and by research gifts from Protocol Labs.

References

ABC$^+$07. G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. Song. Provable data possession at untrusted stores. In P. Ning, S. De Capitani di Vimercati, and P. F. Syverson, editors, *ACM CCS 2007*, pages 598–609. ACM Press, October 2007.

BBF18. D. Boneh, B. Bünz, and B. Fisch. A Survey of Two Verifiable Delay Functions. Cryptology ePrint Archive, Report 2018/712, 2018. `https://eprint.iacr.org/2018/712`.

BBF19. D. Boneh, B. Bünz, and B. Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, Heidelberg, August 2019.

Bd94. J. C. Benaloh and M. de Mare. One-Way Accumulators: A Decentralized Alternative to Digital Sinatures (Extended Abstract). In T. Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg, May 1994.

BGR12. K. Brogle, S. Goldberg, and L. Reyzin. Sequential Aggregate Signatures with Lazy Verification from Trapdoor Permutations - (Extended Abstract). In X. Wang and K. Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 644–662. Springer, Heidelberg, December 2012.

BH01. J. Buchmann and S. Hamdy. A Survey on {IQ} Cryptography, 2001.

BP97. N. Bari and B. Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 480–494. Springer, Heidelberg, May 1997.

CF13. D. Catalano and D. Fiore. Vector Commitments and Their Applications. In K. Kurosawa and G. Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 55–72. Springer, Heidelberg, February / March 2013.

CL02. J. Camenisch and A. Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, August 2002.

CMS99. C. Cachin, S. Micali, and M. Stadler. Computationally Private Information Retrieval with Polylogarithmic Communication. In J. Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg, May 1999.

CS99. R. Cramer and V. Shoup. Signature Schemes Based on the Strong RSA Assumption. In J. Motiwalla and G. Tsudik, editors, *ACM CCS 99*, pages 46–51. ACM Press, November 1999.

CSWH01. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. *Freenet: A Distributed Anonymous Information Storage and Retrieval System*, pages 46–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

DG20. S. Dobson and S. D. Galbraith. Trustless Groups of Unknown Order with Hyperelliptic Curves. Cryptology ePrint Archive, Report 2020/196, 2020. `https://eprint.iacr.org/2020/196`.

DK02. I. Damgård and M. Koprowski. Generic Lower Bounds for Root Extraction and Signature Schemes in General Groups. In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 256–271. Springer, Heidelberg, April / May 2002.

Fis18. B. Fisch. PoReps: Proofs of Space on Useful Data. Cryptology ePrint Archive, Report 2018/678, 2018. `https://eprint.iacr.org/2018/678`.

GHR99. R. Gennaro, S. Halevi, and T. Rabin. Secure Hash-and-Sign Signatures Without the Random Oracle. In J. Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 123–139. Springer, Heidelberg, May 1999.

GKM$^+$18. J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. Updatable and Universal Common Reference Strings with Applications to zk-SNARKs. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 698–728. Springer, Heidelberg, August 2018.

GRWZ20. S. Gorbunov, L. Reyzin, H. Wee, and Z. Zhang. Pointproofs: Aggregating Proofs for Multiple Vector Commitments. Cryptology ePrint Archive, Report 2020/419, 2020. `https://eprint.iacr.org/2020/419`.

JK07. A. Juels and B. S. Kaliski Jr. Pors: proofs of retrievability for large files. In P. Ning, S. De Capitani di Vimercati, and P. F. Syverson, editors, *ACM CCS 2007*, pages 584–597. ACM Press, October 2007.

KZG10. A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010.

Lab17. P. Labs. Filecoin: A Decentralized Storage Network, 2017. `https://filecoin.io/filecoin.pdf`.

Lip12. H. Lipmaa. Secure Accumulators from Euclidean Rings without Trusted Setup. In F. Bao, P. Samarati, and J. Zhou, editors, *ACNS 12*, volume 7341 of *LNCS*, pages 224–240. Springer, Heidelberg, June 2012.

LM19. R. W. F. Lai and G. Malavolta. Subvector Commitments with Application to Succinct Arguments. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 530–560. Springer, Heidelberg, August 2019.

LMRS04. A. Lysyanskaya, S. Micali, L. Reyzin, and H. Shacham. Sequential Aggregate Signatures from Trapdoor Permutations. In C. Cachin and J. Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 74–90. Springer, Heidelberg, May 2004.

LRY16. B. Libert, S. C. Ramanna, and M. Yung. Functional Commitment Schemes: From Polynomial Commitments to Pairing-Based Accumulators from Simple Assumptions. In I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi, editors, *ICALP 2016*, volume 55 of *LIPIcs*, pages 30:1–30:14. Schloss Dagstuhl, July 2016.

LY10. B. Libert and M. Yung. Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs. In D. Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 499–517. Springer, Heidelberg, February 2010.

Mer88. R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In C. Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, August 1988.

OWB19. A. Ozdemir, R. S. Wahby, and D. Boneh. Scaling Verifiable Computation Using Efficient Set Accumulators. Cryptology ePrint Archive, Report 2019/1494, 2019. `https://eprint.iacr.org/2019/1494`.

Sha83. A. Shamir. On the Generation of Cryptographically Strong Pseudorandom Sequences. *ACM Trans. Comput. Syst.*, 1(1):38–44, 1983.

TAB$^+$20. A. Tomescu, I. Abraham, V. Buterin, J. Drake, D. Feist, and D. Khovratovich. Aggregatable Subvector Commitments for Stateless Cryptocurrencies. Cryptology ePrint Archive, Report 2020/527, 2020. `https://eprint.iacr.org/2020/527`.

Tam03. R. Tamassia. Authenticated Data Structures. In G. Di Battista and U. Zwick, editors, *Algorithms - ESA 2003*, pages 2–5, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

Wes18. B. Wesolowski. Efficient verifiable delay functions. Cryptology ePrint Archive, Report 2018/623, 2018. `https://eprint.iacr.org/2018/623`.