Vector Commitment Techniques and Applications to Verifiable Decentralized Storage

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

[1] IMDEA Software Institute, Madrid, Spain
{matteo.campanelli,dario.fiore,dimitris.kolonelos}@imdea.org
[2] Protocol Labs
{nicola,luca}@protocol.ai

Abstract. Vector commitments with subvector openings (SVC) [Lai-Malavolta and 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 propose a new SVC construction in groups of unknown order that, similarly to that of Boneh et al. has constant-size public parameters, commitments and openings, but in addition enjoys new features. First, our SVC has incremental aggregation: one can merge openings in a succinct way an unbounded number of times. Thanks to incremental aggregation we obtain: faster generation of openings via preprocessing, and a method to generate openings in a distributed way. Second, we propose efficient arguments of knowledge of subvector openings for our SVC, which immediately yields a keyless proof of storage with compact proofs.
Finally, we introduce and contruct *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.

1 Introduction

Commitment schemes are one of the most fundamental cryptographic primitives. They can be seen as the digital equivalent of a sealed envelop: committing to a message m is akin to putting m in the envelop; opening the commitment is like opening the envelop and revealing the value inside. Two are the basic properties of commitments. *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 $\boldsymbol{v}$ of length n and to later open the commitment at any position $i \in [n]$. In other words, one can convince a verifier that v_i is the i-th committed value. 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 also proposed two constructions 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.

Very recently, two works [BBF19, LM19] proposed new constructions of vector commitments that enjoy a new property called *subvector openings* (also called *batch openings* in [BBF19]). A vector commitment with subvector openings (called SVC, for short) allows one to open a commitment at a collection of positions $I = \{i_1, \dots, i_m\}$ 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 to have several applications. Lai and Malavolta [LM19] show how to use SVC to replace Merkle trees in the PCP-based construction of CS proofs [Kil92, Mic94]; Boneh et al. [BBF19] show a similar result for IOPs [BCS16]; Fisch [Fis18] show how to use SVC to reduce the communication complexity of keyless Proofs of Retrievability constructions.

1.1 Our Contribution

We continue the study of VCs with subvector openings; our results follow.

A New VC with Incremental Aggregation. Our first result is a new SVC construction with constant-size public parameters and constant-size subvector openings. Our scheme works in hidden-order groups [DK02] (that can be instantiated using classical RSA groups or class groups [BH01]). The computational complexity of its algorithms is asymptotically similar to the scheme of Boneh et al. [BBF19] but in addition it enjoys a new property that we call *incremental aggregation*. In a nutshell, aggregation means that different subvector openings can be merged together into a single *short* opening which can be created *without* knowing the entire committed vector. Moreover, aggregation is incremental if aggregated proofs can be further aggregated. Although the VC of Boneh et al. [BBF19] has aggregation, it can be performed only once. In contrast, we show the first VC scheme where openings can be aggregated an unbounded number of times. Furthermore, our scheme supports *disaggregation* of openings, that is from an opening for a collection of positions I one can create an opening for any subset of positions in $K \subset I$ (and thus openings for any sets that partition I).

We show two main applications of incrementally aggregatable openings.

First, it can improve the efficiency of the VC scheme itself. In many applications of VCs (such as all the aforementioned ones to PCP/IOP-based succinct arguments and to proofs of retrievability), one party commits to a vector that she later opens at various (unpredictable) positions. Although generating an opening in all existing VCs takes at least time $O(|\boldsymbol{v}|)$ (i.e., the same as committing), the VC of Boneh et al. [BBF19] supports an alternative method that, by using an auxiliary information precomputed at commitment time in time $O(|\boldsymbol{v}| \log |\boldsymbol{v}|)$, can generate an opening for m positions in time roughly $O(m \log |\boldsymbol{v}|)$. This is possible thanks to one-hop aggregation: one precomputes an opening for every bit of the vector; at opening time, we aggregate the ones for the opened bits.

In our case, thanks to incremental aggregation, *we can instead store an already aggregated opening for every chunk of, e.g., B bits*, and then further disaggregate and aggregate for the bits corresponding to the m opened positions. In applications where one works with vectors of blocks of bits (as it is virtually always the case, with block size typically the output of a collision resistant hash, 2λ), we can store one opening per block—an auxiliary information already 3 times less than in [BBF19]—and never disaggregate, which makes opening very fast. Furthermore we can further reduce auxiliary storage by aggregating larger chunks – e.g., using $O(\sqrt{|\boldsymbol{v}|})$ storage to get $O(m\sqrt{|\boldsymbol{v}|} \log |\boldsymbol{v}|)$ opening time – thus obtaining flexible tradeoffs not possible in [BBF19]. We implemented our solution and show experimentally how this method makes opening practical in many cases (with savings of one or more orders of magnitude).

As a second application, incremental aggregation can generate openings in a distributed fashion. Namely, 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. In section 1.2 we further elaborate on how this property helps in decentralized storage networks.

Efficient Arguments of Knowledge of Subvector Opening. Our second result is the proposal of efficient arguments of knowledge (AoK) with constant-size proofs for our new VC. The first AoK can prove knowledge of the subvector that opens a commitment at a public set of positions, and it extends to proving that two commitments share a common subvector. The second AoK is similar except that the subvector one proves knowledge of is also committed; essentially one can create two vector commitments C and C' together with a short proof that C' is a commitment to a subvector of the vector committed in C.

An immediate application of our first 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, a property useful in open systems where the client is a set of distrustful parties (e.g., verifiers in a blockchain) and the server may even be one of these clients. A classical keyless PoS is based on Merkle trees and random spot-checks [JK07], recently generalized to work with vector commitments [Fis18]. A drawback of this construction is that proofs grow with the number of spot-checks (and the size of the tree) and become undesirably large in some applications, e.g., if need to be stored in a blockchain. With our AoK we can obtain openings of fixed size, as short as 2KB, which is 40x shorter than those based on Merkle trees in a representative setting without relying on SNARKs (that would be unfeasible in terms of time and memory)[3].

From Updatable VCs to Verifiable Decentralized Storage. In their seminal work on VCs, Catalano and Fiore [CF13] also defined updatable VCs. This means that if one changes the i-th value of a vector from v_i to v'_i it is possible to update: a commitment C to $\boldsymbol{v}$ into a commitment C' to $\boldsymbol{v}'$, a valid opening for C (at any position) into a valid opening for C'. And importantly, these updates can be done without knowing the entire vector and in time that depends only on the number of modified positions. As an application, in [CF13] it is shown how updatable VCs can be used to realize verifiable databases (VDB) [BGV11], a primitive that enables a client to outsource a database to an untrusted server in such a way that the client can retrieve (and update) a DB record and be assured that it has not been tampered with by the server.

In this work we study how to extend this model to a scenario where storage is distributed across different nodes of a decentralized network. This problem is motivated by the emerging trend of *decentralized storage networks* (DSNs), a decentralized and open alternative to traditional cloud storage and hosting services. Filecoin (which is built on top of IPFS), Storj, Dat, Freenet and general-purpose blockchains like Ethereum[4] are some emerging projects in this space.

Our contribution is to put forward a new cryptographic primitive called *verifiable decentralized storage* (VDS) that can be used to obtain data integrity guarantees in DSNs. We propose a definition of VDS and a construction obtained by extending the techniques of our VC scheme; in particular, both incremental aggregation and the arguments of knowledge are key ingredients for building a cost-effective VDS solution.

In the following section we elaborate on the VDS problem: we begin by discussing the requirements imposed by DSNs, and then give a description of our VDS primitive and realization.

---

[3] We provide further details in Section 4

[4] `https://filecoin.io`, `https://storj.io`, `https://datproject.org`, `https://freenetproject.org`, `https://www.ethereum.org`

1.2 Verifiable Decentralized Storage

Decentralized Storage Networks. *Openness* and *decentralization* are the main characteristics of DSNs: 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. Abstracting from the details of each system, a DSN consists of participants called *nodes* that can be either a storage provider (aka *storage node*) or a *client node*. Akin to centralized cloud storage, a client can outsource the storage of large data; the key difference of DSN however is that storage is provided by, and distributed across, a collection of nodes that can enter and leave the system at their wish. Also, DSNs can have some reward mechanism to economically incentivize storage nodes.

The openness and the presence of economic incentives raise a number of security questions that need to be solved in order to make these systems viable. 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. In what follows we elaborate on the problem and the desired requirements, and then on our solution.

The Problem of Verifiable Decentralized Storage. 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, be it 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, e.g., logarithmic or constant in the size of 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 that can be generated by the node who holds 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 in the most extreme case, e.g., the file is a blockchain), 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 to well-behaving storage nodes[5].
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 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., IPFS, after executing a P2P protocol to discover the storage nodes holding the desired data blocks, one gets such blocks from these nodes. In other 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. Notably, data aggregation and propagation may follow different strategies.[6] To accommodate flexible 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.[7]

Defining VDS. We define VDS as a collection of algorithms that capture all the properties above; these are the algorithms that can be executed by clients and storage nodes to maintain the system. A client for a file F is anyone who holds a digest δ_F with which it can: verify retrieval queries, verify and apply updates of F (that result in forks of δ_F into some other $\delta_{F'}$). A storage node for some blocks $F_I = \{F_i\}_{i \in I}$ of a file F is anyone that in addition to F_I stores the digest δ_F and a local state st_{F_I} with which it can: answer and certify retrieval queries for any subset of F_I; push and certify updates of F that involve blocks in F_I; verify and apply updates of F from other nodes. Finally, any node can aggregate retrieval certificates for different blocks of the same file.

In our VDS notion, an update of F can be: (i) a modification of some blocks, (ii) appending new blocks, or (iii) deleting some blocks (from the end). In all cases, an update of F results into a file F' and a new digest $\delta_{F'}$. An additional type of update considered by our VDS notion is what we call "CreateFrom". In it, a storage node holding a prefix F' of a file F can publish a new digest $\delta_{F'}$ corresponding to F' as a new file *and* to convince any client about its correctness *without the need for the client to know neither F nor F'*.[8] As a potential use case for this feature, consider a network that is supposed to store the entire editing history of some data (e.g., one or more files of a Git project); namely the i-th block of the VDS file contains the data value after the i-th edit

[5] 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]).

[6] E.g., in Freenet data is sent back along the same route the query came through, with the goal of providing anonymity between who requests and who delivers data.

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

[8] This can be seen as a deletion that can be performed without holding the blocks to be deleted and is more efficient to verify when the prefix F' is much smaller than F.

(e.g., the i-th Git commit). Then "CreateFrom" can be used to verifiably create a digest of any past version of the data (e.g., of a fork at any point in the past).

In terms of efficiency, in VDS the digests and every certificate (for both retrieval queries or modifications) are required to be of size at most $O(\log |F|)$; similarly, the storage node's local state st_{F_I} has size at most $O(|F_I| + \log |F|)$.

The main security property of a VDS scheme intuitively requires that no efficient adversary can create a certificate for falsified data blocks (or updates) that passes verification. As an extra security property, we also consider the possibility that anyone holding a digest δ_F can check if the DSN is storing correctly F without having to retrieve it. Namely, we let VDS provide a Proof of Storage mechanism, which we define similarly to Proof of Retrievability [JK07] and Proof of Data Possession [ABC+07]. Similarly to the case of data retrieval queries, the creation of these proofs of storage must be possible while preserving the aforementioned properties of locality and no-central-coordination.

Constructing VDS. We propose a construction of VDS in hidden-order groups. Our construction is obtained by extending the techniques of our new vector commitment scheme in order to handle updates and to ensure that all such update operations can be performed locally. In particular we show crucial use of the new properties of our construction: subvector openings, incremental aggregation and disaggregation, and arguments of knowledge for sub-vector commitments.

By abstracting the ideas in our construction, other VDS constructions can be obtained using Merkle trees or RSA accumulators.[9] Compared to a Merkle-tree based solution, we can achieve succinct certificates for every operation as well as to (efficiently) support compact proofs of storage without expensive SNARKs[10]. Compared to RSA Accumulators, our scheme takes advantage of our AoK thanks to which it supports CreateFrom updates and compact proofs of storage.

2 Preliminaries

In this section we describe notation and definitions used throughout the paper.

Notation. We denote the security parameter by λ and the set of all polynomial functions by $\mathsf{poly}(\lambda)$. A function $\epsilon(\lambda)$ is said *negligible* – denoted $\epsilon(\lambda) \in \mathsf{negl}(\lambda)$ – if it vanishes faster than the inverse of any polynomial. An algorithm $\mathcal{A}$ is said PPT if it is modeled as a probabilistic Turing machine that runs in time $\mathsf{poly}(\lambda)$. We denote by $y \leftarrow \mathcal{A}(x)$ the process of running $\mathcal{A}$ on input x and assigning the output to y. For a set S, $|S|$ denotes its cardinality, and $x \leftarrow\!\!\$\, S$ denotes selecting x uniformly at random over S. For a positive integer $n \in \mathbb{N}$ we let $[n] := \{1, \ldots, n\}$. We denote vectors $\boldsymbol{v}$ in bold, and for $\boldsymbol{v} \in \mathcal{M}^n$ v_i is its entry at position i. We let $\mathsf{Primes}(\lambda)$ be the set of all prime integers less than 2^λ.

2.1 Groups of Unknown Order and Computational Assumptions

Our constructions use a group $\mathbb{G}$ of unknown (aka hidden) order, in which the adaptive root assumption [Wes18] and the Strong RSA assumption [BP97] (defined below) 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 $\frac{1}{\mathsf{ord}_{min}}, \frac{1}{\mathsf{ord}_{max}}, \frac{1}{\mathsf{ord}_{max} - \mathsf{ord}_{min}} \in \mathsf{negl}(\lambda)$.

[9] In fact, a similar idea from RSA accumulators was discussed in [BBF19].

[10] In Merkle trees certificates depend logarithmically on the file size and linearly on the number of blocks (since they are not aggregatable).

Definition 2.1 (Adaptive Root Assumption [Wes18]). *We say that the* adaptive root assumption *holds for* Ggen *if for any PPT adversary* $(\mathcal{A}_1, \mathcal{A}_2)$*:*

$$\Pr\left[\begin{array}{c} u^\ell = w \\ \wedge w \neq 1 \end{array} : \begin{array}{l} \mathbb{G} \leftarrow \mathsf{Ggen}(\lambda) \\ (w, \mathsf{state}) \leftarrow \mathcal{A}_1(\mathbb{G}) \\ \ell \leftarrow \mathsf{Primes}(\lambda) \\ u \leftarrow \mathcal{A}_2(\ell, \mathsf{state}) \end{array}\right] = \mathsf{negl}(\lambda)$$

Definition 2.2 (Strong-RSA Assumption [BP97]). *We say that the* strong RSA assumption *holds for* Ggen *if for any PPT adversary* $\mathcal{A}$*:*

$$\Pr\left[u^e = g \;:\; \begin{array}{l} \mathbb{G} \leftarrow \mathsf{Ggen}(\lambda) \\ g \leftarrow \mathbb{G} \\ (u, e) \leftarrow \mathcal{A}(\mathbb{G}, g) \end{array}\right] = \mathsf{negl}(\lambda)$$

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]. The reason why we cannot directly use the RSA group is that the order of $-1 \in \mathbb{Z}_N^*$ is known, and thus the adaptive root assumption does not hold. In the quotient group, $\{-1, 1\}$ is the identity element; hence, knowing the order of -1 does not help in finding a root for a non-identity element and thus solving the adaptive root assumption.

Shamir's Trick. Informally speaking, Shamir's trick [Sha83] is a way to compute an xy-root of a group element g given an x-root and a y-root of it in groups of unknown order, when x and y are co-prime. That is, given $\rho_x = g^{\frac{1}{x}}$, $\rho_y = g^{\frac{1}{y}}$, x and y, one can compute a, b st $ax + by = 1$ using the extended gcd algorithm. Then $g^{\frac{1}{xy}} = g^{\frac{ax+by}{xy}} = g^{\frac{a}{y} + \frac{b}{x}} = \rho_y^a \cdot \rho_x^b$. More formally, we recall the following algorithm:

ShamirTrick(ρ_x, ρ_y, x, y)

---

if $\rho_x^x \neq \rho_y^y$ **then return** $\perp$

Use the extended Euclidean Algorithm to compute a, b, d s.t. $ax + by = d = \gcd(x, y)$

if $d \neq 1$ **then return** $\perp$

return $\rho_x^b \rho_y^a$

2.2 Arguments of Knowledge

Let $R : \mathcal{X} \times \mathcal{W} \to \{0, 1\}$ be an NP relation for a language $\mathcal{L} = \{x : \exists w \text{ s.t. } R(x, w) = 1\}$. An argument system for R is a triple of algorithms $(\mathsf{Setup}, \mathsf{P}, \mathsf{V})$ such that: $\mathsf{Setup}(1^\lambda)$ takes as input a security parameter λ and outputs a common reference string crs; the prover $\mathsf{P}(\mathsf{crs}, x, w)$ takes as input the crs, the statement x and witness w; the verifier $\mathsf{V}(\mathsf{crs}, x)$ takes in the crs, the statement x, and after interacting with the prover outputs 0 (reject) or 1 (accept). An execution between the prover and verifier is denoted with $\langle \mathsf{P}(\mathsf{crs}, x, w), \mathsf{V}(\mathsf{crs}, x)\rangle = b$, where $b \in \{0, 1\}$ is the output of the verifier. If V uses only public randomness, we say that the protocol is public coin.

Definition 2.3 (Completeness). *We say that an argument system* $(\mathsf{Setup}, \mathsf{P}, \mathsf{V})$ *for a relation* $R : \mathcal{X} \times \mathcal{W} \to \{0, 1\}$ *is complete if, for all* $(x, w) \in \mathcal{X} \times \mathcal{W}$ *such that* $R(x, w) = 1$ *we have*

$$\Pr\left[\langle \mathsf{P}(\mathsf{crs}, x, w), \mathsf{V}(\mathsf{crs}, x)\rangle = 1 : \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)\right] = 1.$$

Consider an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ modeled as a pair of algorithms such that $\mathcal{A}_0(\mathsf{crs}) \to (x, \mathsf{state})$ (i.e. outputs an instance $x \in \mathcal{X}$ after $\mathsf{crs} \leftarrow \mathsf{Setup}(\lambda)$ is run) and $\mathcal{A}_1(\mathsf{crs}, x, \mathsf{state})$ interacts with a honest verifier. We want an argument of knowledge to satisfy the following properties:

Soundness. We say that an argument $(\mathsf{Setup}, \mathsf{P}, \mathsf{V})$ is sound if for all PPT adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ we have

$$\Pr\left[\begin{array}{c} \langle \mathcal{A}_1(\mathsf{crs}, x, \mathsf{state}), \mathsf{V}(\mathsf{crs}, x)\rangle = 1 \\ \text{and } \nexists w : R(x, w) = 1 \end{array} \middle| \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(\lambda) \\ (x, \mathsf{state}) \leftarrow \mathcal{A}_0(\mathsf{crs}) \end{array}\right] \in \mathsf{negl}(\lambda).$$

Knowledge Extractability. We say that $(\mathsf{Setup}, \mathsf{P}, \mathsf{V})$ is an *argument of knowledge* if for all polynomial time adversaries $\mathcal{A}_1$ there exists an extractor $\mathcal{E}$ running in polynomial time such that, for all adversaries $\mathcal{A}_0$ it holds

$$\Pr\left[\begin{array}{c} \langle \mathcal{A}_1(\mathsf{crs}, x, \mathsf{state}), \mathsf{V}(\mathsf{crs}, x)\rangle = 1 \\ \text{and } (x, w') \notin \mathcal{R} \end{array} \middle| \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Setup}(\lambda) \\ (x, \mathsf{state}) \leftarrow \mathcal{A}_0(\mathsf{crs}) \\ w' \leftarrow \mathcal{E}(\mathsf{crs}, x, \mathsf{state}) \end{array}\right] \in \mathsf{negl}(\lambda).$$

Succinctness. Finally we informally recall the notion of succinct arguments, which requires the communication and verifier's running time in a protocol execution to be independent of the witness length.

Succinct Arguments of Knowledge for Hidden Order Groups. We recall two succinct AoK protocols for the exponentiation relation in groups of unknown order that have been recently proposed by Boneh et. al. [BBF19]. Both protocols work for a hidden order group $\mathbb{G}$ generated by Ggen in which the adaptive root assumption holds. Also, they are public-coin protocols that can be made non-interactive in the random oracle model using the Fiat-Shamir [FS87] heuristic and its generalization to multi-round protocols [BCS16].

1. Protocol PoE: is an argument system for the following relation:

   $$R_{\mathsf{PoE}} = \left\{((u, w, x) \in \mathbb{G}^2 \times \mathbb{Z}, \varnothing) \;:\; u^x = w \in \mathbb{G} \quad\right\}$$

   PoE is a sound argument system under the adaptive root assumption for Ggen. It is neither zero-knowledge nor knowledge sound. Its main feature is *succinctness*, as the verifier can get convinced about $u^x = w$ without having to execute the exponentiation herself. Moreover the information sent by the prover is only 1 group element.[11]

2. Protocol PoKE^*: is an argument of knowledge for the following relation, parametrized by a generator $g \in \mathbb{G}$:

   $$R_{\mathsf{PoKE}^*} = \left\{(w, x) \in \mathbb{G} \times \mathbb{Z} \;:\; g^x = w \in \mathbb{G} \quad\right\}$$

   PoKE^* is an argument of knowledge that in [BBF19] is proven secure in the generic group model for hidden order groups [DK02]. This protocol is also succinct consisting of only 1 group element and 1 field element in $\mathbb{Z}_{2^\lambda}$.

[11] Technically, this protocol is not succinct as there is no witness and the verifier must read and process the exponent x; however, verification is still more efficient than running the full exponentiation.

3. Protocol $\mathsf{PoKE2}$: is an argument of knowledge for the following relation, parametrized by a generator $g \in \mathbb{G}$:

$$R_{\mathsf{PoKE2}} = \left\{ ((w, u) \in \mathbb{G}^2, x \in \mathbb{Z}) \ : \ u^x = w \in \mathbb{G} \quad \right\}$$

$\mathsf{PoKE2}$ is similar to PoKE^* but it is secure for arbitrary bases u chosen by the adversary, instead of bases randomly sampled a priori as in PoKE^*. Similarly, it is an argument of knowledge in the generic group model for hidden order groups and is also succinct, with a proof consisting of 2 group elements and 1 element of $\mathbb{Z}_{2^\lambda}$.

2.3 Vector Commitments with Subvector Openings

A vector commitment (VC) [LY10, CF13] is a primitive that allows one to commit to a vector $\boldsymbol{v}$ of length n in such a way that it can later open the commitment at any position $i \in [n]$. For security, a VC should be *position binding* in the sense that it is not possible to open a commitment to two different values at the same position. Also, what makes VC interesting is *conciseness*, which requires commitment and openings to be of fixed size, independent of the vector's length.

In our work we consider a generalization of vector commitments proposed by Lai and Malavolta [LM19] that is called *VCs with subvector openings*,[12] which is in turn a specialization of the notion of *functional vector commitments* by Libert et al. [LRY16]. In a nutshell, a functional VC is like a VC with the additional possibility of opening the commitment to a function of the committed vector, i.e., $f(\boldsymbol{v})$. Subvector openings are a specific class of functions in which one can open the commitment to an ordered collection of positions (with a short proof).

In this section we recall this generalization of vector commitments with subvector openings (that for brevity we call SVC). It is easy to see that the original notion of Catalano and Fiore [CF13] is a special case when the opened subvector includes one position only.

We begin by recalling the notion of subvectors from [LM19].

Definition 2.4 (Subvectors [LM19]). *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. For a vector $\boldsymbol{v} \in \mathcal{M}^n$, the I-subvector of $\boldsymbol{v}$ is $\boldsymbol{v}_I := (v_{i_1}, \ldots, v_{i_{|I|}})$.*

Let $I, J \subseteq [n]$ be two sets, and let $\boldsymbol{v}_I, \boldsymbol{v}_J$ be two subvectors of some vector $\boldsymbol{v} \in \mathcal{M}^n$. The *ordered union* of $\boldsymbol{v}_I$ and $\boldsymbol{v}_J$ is the subvector $\boldsymbol{v}_{I \cup J} := (v_{k_1}, \ldots, v_{k_m})$, where $I \cup J = \{k_1, \ldots, k_m\}$ is the ordered sets union of I and J.

Definition 2.5 (Vector Commitments with Subvector Openings). *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 and satisfy* correctness*,* position binding *and* conciseness *defined below.*

$\mathsf{VC.Setup}(1^\lambda, \mathcal{M}) \to \mathsf{crs}$ *Given the security parameter λ, and description of a message space $\mathcal{M}$ for the vector components, the probabilistic setup algorithm outputs a common reference string* crs.

$\mathsf{VC.Com}(\mathsf{crs}, \boldsymbol{v}) \to (C, \mathsf{aux})$ *On input* crs *and a vector $\boldsymbol{v} \in \mathcal{M}^n$, the committing algorithm outputs a commitment C and an auxiliary information* aux.

$\mathsf{VC.Open}(\mathsf{crs}, I, \boldsymbol{y}, \mathsf{aux}) \to \pi_I$ *On input the CRS* crs*, a vector $\boldsymbol{y} \in \mathcal{M}^m$, an ordered index set $I \subset \mathbb{N}$ and auxiliary information* aux*, the opening algorithm outputs a proof π_I that $\boldsymbol{y}$ is the I-subvector of the committed message.*

[12] This is also called VCs with batchable openings in an independent work by Boneh et al. [BBF19].

$\mathsf{VC.Ver}(\mathsf{crs}, C, I, \boldsymbol{y}, \pi_I) \to b \in \{0,1\}$ *On input the CRS* crs*, a commitment* C*, an ordered set of indices* $I \subset \mathbb{N}$*, a vector* $\boldsymbol{y} \in \mathcal{M}^m$ *and a proof* π_I*, the verification algorithm accepts (i.e., it outputs 1) only if* π_I *is a valid proof that* C *was created to a vector* $\boldsymbol{v} = (v_1, \ldots, v_n)$ *such that* $\boldsymbol{y} = \boldsymbol{v}_I$*.*

Correctness. *A SVC scheme* VC *is (perfectly)* correct *if for all* $\lambda \in \mathbb{N}$*, any vector length* n *any ordered set of indices* $I \subseteq [n]$*, and any* $\boldsymbol{v} \in \mathcal{M}^n$*, we have:*

$$\Pr\left[\mathsf{VC.Ver}(\mathsf{crs}, C, I, \boldsymbol{v}_I, \pi_I) = 1 \quad : \quad \begin{array}{r} \mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M}) \\ (C, \mathsf{aux}) \leftarrow \mathsf{VC.Com}(\mathsf{crs}, \boldsymbol{v}) \\ \pi_I \leftarrow \mathsf{VC.Open}(\mathsf{crs}, I, \boldsymbol{v}_I, \mathsf{aux}) \end{array}\right] = 1$$

Position Binding. *A SVC scheme* VC *satisfies position binding if for all PPT adversaries* $\mathcal{A}$ *we have:*

$$\Pr\left[\begin{array}{c} \mathsf{VC.Ver}(\mathsf{crs}, C, I, \boldsymbol{y}, \pi) = 1 \\ \wedge\ \boldsymbol{y} \neq \boldsymbol{y}' \wedge \\ \mathsf{VC.Ver}(\mathsf{crs}, C, I, \boldsymbol{y}', \pi') = 1 \end{array} \ : \ \begin{array}{r} \mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M}) \\ (C, I, \boldsymbol{y}, \pi, \boldsymbol{y}', \pi') \leftarrow \mathcal{A}(\mathsf{crs}) \end{array}\right] \in \mathsf{negl}(\lambda)$$

Conciseness. *A vector commitment is* concise *if there is a fixed polynomial* $p(\lambda)$ *in the security parameter such that the size of the commitment* C *and the outputs of* VC.Open *are both bounded by* $p(\lambda)$*, i.e., they are independent of* n*.*

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 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 VC.Com, VC.Open and 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 VC.Specialize deterministic, it can be executed by anyone, and it allows to re-use the same crs for multiple vectors lengths.

Definition 2.6 (VCs with Specializable CRS). *A VC scheme* VC *has a specializable CRS if there exists a DPT algorithm* $\mathsf{VC.Specialize}(\mathsf{crs}, n)$ *that, on input a (universal) CRS* crs *generated by* $\mathsf{VC.Setup}(1^\lambda, \mathcal{M})$ *and an integer* $n = \mathsf{poly}(\lambda)$*, produces a specialized CRS* crs_n *such that the algorithms* VC.Com*,* VC.Open *and* VC.Ver *can be defined in terms of algorithms* $\mathsf{VC.Com}^\star$*,* $\mathsf{VC.Open}^\star$ *and* $\mathsf{VC.Ver}^\star$ *as follows:*

- $\mathsf{VC.Com}(\mathsf{crs}, \boldsymbol{v})$ *sets* $n := |\boldsymbol{v}|$*, runs* $\mathsf{crs}_n \leftarrow \mathsf{VC.Specialize}(\mathsf{crs}, n)$ *and* $(C^\star, \mathsf{aux}^\star) \leftarrow \mathsf{VC.Com}^\star(\mathsf{crs}_n, \boldsymbol{v})$*, and returns* $C := (C^\star, n)$ *and* $\mathsf{aux} := (\mathsf{aux}^\star, n)$*.*
- $\mathsf{VC.Open}(\mathsf{crs}, I, \boldsymbol{y}, \mathsf{aux})$ *parses* $\mathsf{aux} := (\mathsf{aux}^\star, n)$*, runs* $\mathsf{crs}_n \leftarrow \mathsf{VC.Specialize}(\mathsf{crs}, n)$ *and returns* $\pi_I \leftarrow \mathsf{VC.Open}^\star(\mathsf{crs}_n, I, \boldsymbol{y}, \mathsf{aux}^\star)$*.*

– $\mathsf{VC.Ver}(\mathsf{crs}, C, I, \boldsymbol{y}, \pi_I)$ *parses* $C := (C^\star, n)$*, runs* $\mathsf{crs}_n \leftarrow \mathsf{VC.Specialize}(\mathsf{crs}, n)$ *and returns* $\mathsf{VC.Ver}^\star(\mathsf{crs}_n, C^\star, I, \boldsymbol{y}, \pi$

Basically, for a VC with specializable CRS it is sufficient to describe the algorithms $\mathsf{VC.Setup}$, $\mathsf{VC.Specialize}, \mathsf{VC.Com}^\star, \mathsf{VC.Open}^\star$ and $\mathsf{VC.Ver}^\star$. Furthermore, a concrete advantage is that when working on multiple commitments, openings and verifications that involve the same length n, one can execute $\mathsf{crs}_n \leftarrow \mathsf{VC.Specialize}(\mathsf{crs}, n)$ only once.

3 Our Vector Commitment Construction

In this section we give our first technical contribution, which is a new construction of a vector commitment with subvector openings that has short parameters (i.e., independent of the vector's length).

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 $\boldsymbol{v} = (v_1, \ldots, v_n) \in \{0,1\}^n$. In order to commit to this vector we produce two accumulator, 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 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 $\boldsymbol{v}$ 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.

3.1 Succinct AoK Protocols for Union of RSA Accumulators

Let $\mathbb{G}$ be a an 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

$$R_{\mathsf{PoProd}_2} = \left\{ ((Y, C), (a, b)) \in \mathbb{G}^2 \times \mathbb{Z}^3 \ : \ Y = g_1^a g_2^b \wedge C = g_3^{a \cdot b} \quad \right\}$$

Our protocol (described in Fig. 1) 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 $Y = A \cdot B$.

Fig. 1. PoProd_2 protocol

$\mathsf{Setup}(1^\lambda)$: 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 $r_a, r_b \in [\ell] \ \wedge \ Q_Y^\ell g_1^{r_a} g_2^{r_b} = Y \ \wedge \ Q_C^\ell g_3^{r_c} = C$

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

Proof For ease of exposition we show a security proof for a slight variant of the protocol PoProd_2. Then, towards the end of this proof we show that security of this variant implies security for our protocol. We let PoProd_2' be the same protocol as PoProd_2 with only difference that the prover computes also $r_c \leftarrow r_a \cdot r_b \pmod \ell$ and sends r_c in the proof, and the verifier V checks in the verification if $r_c = r_a \cdot r_b \pmod \ell$.

Let $\mathcal{A}' = (\mathcal{A}'_0, \mathcal{A}'_1)$ be an adversary of the Knowledge Extractability of PoProd_2' such that: $((Y, C), \mathsf{state}) \leftarrow \mathcal{A}'_0(\mathsf{crs})$, $\mathcal{A}'_1(\mathsf{crs}, (Y, C), \mathsf{state})$ executes with $\mathsf{V}(\mathsf{crs}, (Y, C))$ the protocol PoProd_2' and the verifier accepts with a non-negligible probability ϵ. We will construct an extractor $\mathcal{E}'$ that having access to the internal state of $\mathcal{A}'_1$ and on input $(\mathsf{crs}, (Y, C), \mathsf{state})$, outputs a witness (a, b) of $R_{\mathsf{PoProd}_2'}$ with overwhelming probability and runs in (expected) polynomial time.

To prove knowledge extractability of PoProd_2' we rely on the knowledge extractability of the protocol PoKRep from [BBF19], which is indeed implicit in our protocol. More precisely, given a PoProd_2' execution between $\mathcal{A}'$ and V, $(\ell, Q_Y, Q_C, r_a, r_b, r_c)$, $\mathcal{E}'$ constructs an adversary $\mathcal{A}_Y = (\mathcal{A}_{Y,0}, \mathcal{A}_{Y,1})$ of PoKRep Knowledge Extractability and, by using the input and internal state of $\mathcal{A}'_1$, simulates an execution between $\mathcal{A}_Y$ and V: $\mathcal{A}_{Y,0}$ outputs $(\mathsf{crs}_Y, Y, \mathsf{state}) := ((\mathbb{G}, g_1, g_2), Y, \mathsf{state})$, $\mathcal{A}_{Y,1}$ outputs (Q_Y, r_a, r_b). It is obvious that if the initial execution is accepted by V so is the PoKRep execution. From Knowledge Extractability of PoKRep we know that there exists an extractor $\mathcal{E}_Y$ corresponding to $\mathcal{A}_{Y,1}$ that outputs (a, b) such that $g_1^a g_2^b = Y$. Additionally, it is implicit from the extraction that $a = r_a \pmod \ell$ and $b = r_b \pmod \ell$ (for more details we refer to the Knowledge Extractability proof of PoKRep in [BBF19]). So, $\mathcal{E}'$ uses $\mathcal{E}_Y$ and gets (a, b). Similarly, it simulates PoKE^* for $g_3^c = C$, uses the extractor $\mathcal{E}_c$ and gets c.

As one can see, the expected running time of $\mathcal{E}'$ is the (expected) time to obtain a successful execution of the protocol plus the running time of the 2 extractors: $\frac{1}{\epsilon} + t_{\mathcal{E}_Y} + t_{\mathcal{E}_c} = \mathsf{poly}(\lambda)$.

Now what is left to prove to conclude our theorem is to show that the extracted a, b, c are such that $a \cdot b = c$ with all but negligible probability. To this end, we observe that we could run $\mathcal{E}'$ a second time using a different random challenge ℓ'; by using again $\mathcal{E}_Y, \mathcal{E}_c$ (after simulating the corresponding PoKRep and PoKE^* executions) we would get a', b', c' such that $g_1^{a'} g_2^{b'} = Y = g_1^a g_2^b$, $g_3^{c'} = C = g_3^c$. We argue that $a = a'$, $b = b'$ and $c = c'$ holds over the integers with overwhelming probability under the assumption that computing a multiple of the order of the group $\mathbb{G}$ is hard (such assumption is in turn implied by the adaptive root assumption). If such event does not hold one can make a straightforward reduction to this problem. Therefore, we proceed by assuming that from the two executions we have $a = a'$, $b = b'$, and $c = c'$ over the integers. Moreover, since both executions are accepted we have $r'_c = r'_a \cdot r'_b \pmod{\ell'} \Rightarrow c' = a' \cdot b' \pmod{\ell'} \Rightarrow c = a \cdot b \pmod{\ell'}$, but ℓ' was sampled uniformly at random from $\mathsf{Primes}(\lambda)$ after a, b, c were determined. So $a \cdot b = c$ over the integers, unless with a negligible probability $\leq \frac{\#\{\text{factors of } ab-c\}}{|\mathsf{Primes}(\lambda)|} \leq \frac{\mathsf{poly}(\lambda)}{|\mathsf{Primes}(\lambda)|} = \mathsf{negl}(\lambda)$.

Finally, it is trivial to reduce the Knowledge Extractability of PoProd_2 to Knowledge Extractability of PoProd_2'. Given a generic adversary $\mathcal{A}$ that wins the Knowledge Extractability experiment of protocol PoProd_2 with a non-negligible probability, we can construct a generic adversary $\mathcal{A}'$ breaking Knowledge Extractability of PoProd_2' with a non-negligible probability. $\mathcal{A}'$ runs the $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$ algorithm and sends crs to $\mathcal{A}$. The adversary $\mathcal{A}$ outputs $((Y, C), \mathsf{state}) \leftarrow \mathcal{A}_0(\mathsf{crs})$ and sends it to $\mathcal{A}'_0$, which outputs as it is. Then $\mathcal{A}'_1$ interacts with V in the protocol PoProd_2' (as a prover) and at the same time with $\mathcal{A}_1$ in PoProd_2 (as a verifier). After receiving ℓ from V it forwards it to $\mathcal{A}_1$. $\mathcal{A}_1$ answers with $\pi := ((Q_Y, Q_C), r_a, r_b)$. $\mathcal{A}'_1$ computes $r_c \leftarrow r_a r_b \mod \ell$ and sends $\pi' := ((Q_Y, Q_C), r_a, r_b, r_c)$ to V. The verifier V accepts π' with the same probability that a verifier of PoProd_2 would accept π since $r_c = r_a r_b \mod \ell$ in both cases. Thus, we conclude that $\mathcal{A}'$ breaks Knowledge Extractability of PoProd_2 with a non-negligible probability, which contradicts what we proved in the above. □

In Appendix A we 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 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., $\mathsf{H}_{\mathsf{prime}} : \{0,1\}^* \to \mathsf{Primes}(\lambda)$. A simple way to achieve such a function is to apply a standard hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ to an input $\boldsymbol{y}$ together with a counter i, and if $p_{y,i} = \mathsf{H}(\boldsymbol{y}, i)$ is prime then output $p_{y,i}$, otherwise continue to $\mathsf{H}(\boldsymbol{y}, i+1)$ 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. We do not insist, though, in the previous or any other specific instantiation of $\mathsf{H}_{\mathsf{prime}}$ in this work. For more discussion on hash-to-prime functions we refer to [GHR99, CMS99, CS99, BBF19].

3.2 Our 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. To describe our scheme we use the notation below:

– Our message space is $\mathcal{M} = \{0,1\}^k$. Then for a vector $\boldsymbol{v} \in \mathcal{M}^n$, 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_{ij} 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.
– $\mathsf{PartndPrimeProd}(I, \boldsymbol{y}) \rightarrow ((a_{I1}, b_{I1}), \ldots, (a_{Ik}, b_{Ik}))$: given a set of indices $I = \{i_1, \ldots, i_m\} \subseteq [n]$ and a vector $\boldsymbol{y} \in \mathcal{M}^m$, this function computes

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

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., $\mathsf{PartndPrimeProd}([n], \boldsymbol{v})$ outputs a_j and b_j.
– $\mathsf{PrimeProd}(I) \rightarrow u_I$: 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 $\boldsymbol{y}$, it always holds $a_{Ij} \cdot b_{Ij} = u_I$.

Our Construction. Below we describe our SVC scheme.

$\mathsf{VC.Setup}(1^\lambda, \{0,1\}^k) \rightarrow \mathsf{crs}$ 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 $\mathsf{PrimeGen}$ that maps integers to primes.
Returns $\mathsf{crs} = (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen})$

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

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

1. Compute $((a_1, b_1), \ldots, (a_k, b_k)) \leftarrow \mathsf{PartndPrimeProd}([n], \boldsymbol{v})$; 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 $\{i \in [n] : v_{ij} = 0\}$ and $\{i \in [n] : v_{ij} = 1\}$ respectively.

2. For all $j \in [k]$, compute $C_j = A_j \cdot B_j \in \mathbb{G}$ and a proof $\pi_{\mathsf{prod}}^{(j)} \leftarrow \mathsf{PoProd}_2.\mathsf{P}(\mathsf{crs}, (C_j, U_n), (a_j, b_j))$. 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 $C^\star := \left(\{A_1, B_1, \ldots, A_k, B_k\}, \left\{ \pi_{\mathsf{prod}}^{(1)}, ..., \pi_{\mathsf{prod}}^{(k)} \right\} \right)$ and $\mathsf{aux}^\star := \boldsymbol{v}$.

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

– let $J = [n] \setminus I$ and compute $((a_{J1}, b_{J1}), \dots, (a_{Jk}, b_{Jk})) \leftarrow \mathsf{PartndPrimeProd}(J, \boldsymbol{v}_J)$;
– for all $j \in [k]$ compute

$$\mathit{\Gamma}_{Ij} := g_0^{a_{Jj}} \text{ and } \mathit{\Delta}_{Ij} = g_1^{b_{Jj}}$$

Notice that $a_{Jj} = a_j / a_{Ij}$ and $b_{Jj} = b_j / b_{Ij}$. Also $\mathit{\Gamma}_{Ij}$ is a membership witness for the set $\{i_l \in I : y_{lj} = 0\}$ in the accumulator A_j, and similarly for $\mathit{\Delta}_{Ij}$.
Return $\pi_I := \{\pi_{I1}, \dots, \pi_{Ik}\} \leftarrow \{(\mathit{\Gamma}_{I1}, \mathit{\Delta}_{I1}), \dots, (\mathit{\Gamma}_{Ik}, \mathit{\Delta}_{Ik})\}$

$\mathsf{VC.Ver}^\star(\mathsf{crs}_n, C^\star, I, \boldsymbol{y}, \pi_I) \to b$ computes $((a_{I1}, b_{I1}), \dots, (a_{Ik}, b_{Ik}))$ using $\mathsf{PartndPrimeProd}(I, \boldsymbol{y})$, and then returns $b \leftarrow b_{acc} \wedge b_{prod}$ where:

$$b_{acc} \leftarrow \bigwedge_{j=1}^{k} \left(\mathit{\Gamma}_{Ij}^{a_{Ij}} = A_j \wedge \mathit{\Delta}_{Ij}^{b_{Ij}} = B_j \right) \tag{1}$$

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

Remark 3.1. For more efficient verification, in the $\mathsf{VC.Open}^\star$ algorithm can be included $2k$ proofs of exponentiation PoE. In this way the verifier doesn't have to perform all the exponentiations in $\mathsf{VC.Ver}^\star$. As noted in [BBF19], although asymptotically the verification cost is the same, the operations are in $\mathbb{Z}_\lambda$ 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). Its security, i.e., position binding, can be reduced to the Strong RSA and Adaptive root assumptions in the hidden order group $\mathbb{G}$ used in the construction and to the knowledge extractability of PoProd_2.

Theorem 3.2 (Position-Binding). *Let* Ggen *be the generator of hidden order groups where the Strong RSA and Adaptive Root 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.*

Proof To prove the theorem we use a hybrid argument. We start by defining the game G_0 as the actual position binding game of Definition 2.5, and our goal is to prove that for any PPT $\mathcal{A}$, $\Pr[G_0 = 1] \in \mathsf{negl}(\lambda)$.

Game G_0:

$G_0 = \mathsf{PosBind}_{\mathsf{VC}}^{\mathcal{A}}(\lambda)$

$\mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M})$
$(C, I, \boldsymbol{y}, \pi, \boldsymbol{y}', \pi') \leftarrow \mathcal{A}(\mathsf{crs})$
$b \leftarrow \mathsf{VC.Ver}(\mathsf{crs}, C, I, \boldsymbol{y}, \pi) = 1 \wedge \boldsymbol{y} \neq \boldsymbol{y}' \wedge \mathsf{VC.Ver}(\mathsf{crs}, C, I, \boldsymbol{y}', \pi') = 1$
return b

Lemma 3.1. *For any PPT $\mathcal{A}$ in game G_0 there exists an algorithm $\mathcal{E}$ and an experiment G_1 such that*

$$\Pr[G_0 = 1] \leq \Pr[G_1 = 1] + \mathsf{negl}(\lambda)$$

Proof By construction of $\mathsf{VC.Com}$, the commitment C returned by the adversary $\mathcal{A}$ in game G_0 contains k proofs of PoProd_2, and by construction of $\mathsf{VC.Ver}$ if G_0 returns 1 all these proofs verify. It is not hard to argue that for any adversary $\mathcal{A}$ playing in game G_0 there is an extractor $\mathcal{E}$ that outputs the k witnesses $\{a_j, b_j\}_{j\in[k]}$.

Game G_1**:** is the same as G_0 except that we also execute $\mathcal{E}$, which outputs $\{a_j, b_j\}_{j\in[k]}$, and we additionally check that $U_n = g^{a_j b_j}$ for all $j \in [k]$. Below is a detailed description of G_1 in which we "open the box" of the VC algorithms.

G_1

$\mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M}); \mathsf{bad}_1 \leftarrow \mathsf{false}$

$(\{A_j, B_j \pi^{(j)}_{\mathsf{prod}}\}_{j\in[k]}, n), I, \boldsymbol{y}, \{\Gamma_{Ij}, \Delta_{Ij}\}_{j\in[k]}, \boldsymbol{y}', \{\Gamma'_{Ij}, \Delta'_{Ij}\}_{j\in[k]}) \leftarrow \mathcal{A}(\mathsf{crs})$

$\{a_j, b_j\}_{j\in[k]} \leftarrow \mathcal{E}(\mathsf{crs})$

$u_n \leftarrow \mathsf{PrimeProd}(n); U_n \leftarrow g^{u_n}$

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

$$b_{wit} \leftarrow \bigwedge_{j=1}^{k} A_j \cdot B_j = g_0^{a_j} g_j^{b_j} \wedge U_n = g^{a_j \cdot b_j}$$

if $b_{prod} = 1 \wedge b_{wit} = 0$ **then** $\mathsf{bad}_1 \leftarrow \mathsf{true}$

$\{a_{Ij}, b_{Ij}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y})$; $\{a'_{Ij}, b'_{Ij}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y}')$

$$b \leftarrow b_{prod} \wedge \bigwedge_{j=1}^{k} \left({\Gamma_{Ij}}^{a_{Ij}} = A_j \wedge {\Delta_{Ij}}^{b_{Ij}} = B_j\right) \wedge \boldsymbol{y} \neq \boldsymbol{y}' \wedge$$

$$\bigwedge_{j=1}^{k} \left({\Gamma'_{Ij}}^{a'_{Ij}} = A_j \wedge {\Delta'_{Ij}}^{b'_{Ij}} = B_j\right)$$

if $\mathsf{bad}_1 = \mathsf{true}$ **then** $b \leftarrow 0$

return b

Clearly, the games G_0 and G_1 are identical except if the flag bad_1 is raised true, i.e., $\Pr[G_0 = 1] - \Pr[G_1 = 1] \leq \Pr[\mathsf{bad}_1 = \mathsf{true}]$. However, the event in which bad_1 is set true is the event in which one of the witnesses returned by the extractor is not correct. By the knowledge extractability of PoProd_2 we immediately get that $\Pr[\mathsf{bad}_1 = \mathsf{true}] \in \mathsf{negl}(\lambda)$. □

Game G_2: is the same as G_1 except that G_2 outputs 0 if there is an index j such that $U_n = g^{a_j \cdot b_j}$ but $u_n \neq a_j \cdot b_j$. Precisely, if this happens a flag bad_2 is set true and the outcome of the experiment is 0. See below for the detailed description of G_2.

G_2

$\mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M}); \mathsf{bad}_1, \mathsf{bad}_2 \leftarrow \mathsf{false}$

$(\{A_j, B_j \pi_{\mathsf{prod}}^{(j)}\}_{j\in[k]}, n), I, \boldsymbol{y}, \{\Gamma_{Ij}, \Delta_{Ij}\}_{j\in[k]}, \boldsymbol{y}', \{\Gamma'_{Ij}, \Delta'_{Ij}\}_{j\in[k]}) \leftarrow \mathcal{A}(\mathsf{crs})$

$\{a_j, b_j\}_{j\in[k]} \leftarrow \mathcal{E}(\mathsf{crs})$

$u_n \leftarrow \mathsf{PrimeProd}(n); U_n \leftarrow g^{u_n}$

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

$$b_{wit} \leftarrow \bigwedge_{j=1}^{k} A_j \cdot B_j = g_0^{a_j} g_j^{b_j} \wedge U_n = g^{a_j \cdot b_j}$$

if $b_{prod} = 1 \wedge b_{wit} = 0$ **then** $\mathsf{bad}_1 \leftarrow \mathsf{true}$

$$b_{col} \leftarrow \bigwedge_{j=1}^{k} u_n = a_j \cdot b_j$$

if $b_{prod} = 1 \wedge b_{col} = 0$ **then** $\mathsf{bad}_2 \leftarrow \mathsf{true}$

$\{a_{Ij}, b_{Ij}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y})$; $\{a'_{Ij}, b'_{Ij}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y}')$

$$b \leftarrow b_{prod} \wedge \bigwedge_{j=1}^{k} \left({\Gamma_{Ij}}^{a_{Ij}} = A_j \wedge {\Delta_{Ij}}^{b_{Ij}} = B_j\right) \wedge \boldsymbol{y} \neq \boldsymbol{y}' \wedge$$
$$\bigwedge_{j=1}^{k} \left({\Gamma'_{Ij}}^{a'_{Ij}} = A_j \wedge {\Delta'_{Ij}}^{b'_{Ij}} = B_j\right)$$

if $\mathsf{bad}_1 = \mathsf{true} \vee \mathsf{bad}_2 = \mathsf{true}$ **then** $b \leftarrow 0$

return b

Lemma 3.2. *If the adaptive root assumption holds for* Ggen*, then* $\Pr[G_1 = 1] - \Pr[G_2 = 1] \leq \mathsf{negl}(\lambda)$.

Proof Clearly, G_1 and G_2 proceed identically except if bad_2 is set true. We claim that $\Pr[\mathsf{bad}_2 = \mathsf{true}]$ is negligible for any $\mathcal{A}, \mathcal{E}$ running in G_2. If this event happens, one indeed obtains an integer $v = u_n - a_j \cdot b_j$ such that $g^v = 1 \in \mathbb{G}$, i.e., v is a multiple of the group order, and this implies an algorithm that efficiently solve the adaptive root assumption. A formal reduction is straightforward and is omitted. □

Game G_3: is an experiment that can be seen as a simplification of G_2.

G_3

$\mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M})$

$(\boldsymbol{v}, \{A_j, B_j\}_{j\in[k]}, I, \boldsymbol{y}, \{\Gamma_{Ij}, \Delta_{Ij}\}_{j\in[k]}, \boldsymbol{y}', \{\Gamma'_{Ij}, \Delta'_{Ij}\}_{j\in[k]}) \leftarrow \mathcal{A}'(\mathsf{crs})$

$\{a_j, b_j\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}([n], \boldsymbol{v})$

$\{a_{Ij}, b_{Ij}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y})$; $\{a'_{Ij}, b'_{Ij}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y}')$

$$b \leftarrow \bigwedge_{j=1}^{k} (A_j \cdot B_j = g_0^{a_j} \cdot g_1^{b_j}) \bigwedge_{j=1}^{k} \left({\Gamma_{Ij}}^{a_{Ij}} = A_j \wedge {\Delta_{Ij}}^{b_{Ij}} = B_j\right) \wedge \boldsymbol{y} \neq \boldsymbol{y}' \wedge$$
$$\bigwedge_{j=1}^{k} \left({\Gamma'_{Ij}}^{a'_{Ij}} = A_j \wedge {\Delta'_{Ij}}^{b'_{Ij}} = B_j\right)$$

return b

First, we show the following lemma that relates the probability of winning in G_3 with that of winning in G_2.

Lemma 3.3. *For any $(\mathcal{A}, \mathcal{E})$ running in G_2 there is an $\mathcal{A}'$ running in G_3 such that* $\Pr[G_2 = 1] = \Pr[G_3 = 1]$.

Proof We build $\mathcal{A}'$ from $(\mathcal{A}, \mathcal{E})$ as follows. On input crs, $\mathcal{A}'$ executes $(\{A_j, B_j, \pi_{\mathsf{prod}}^{(j)}\}_{j\in[k]}, n), I, \boldsymbol{y}, \{\Gamma_{Ij}, \Delta_{Ij}\}_{j\in[k]}, \boldsymbol{y}', \{\Gamma'_{Ij}, \Delta'_{Ij}\}_{j\in[k]}) \leftarrow \mathcal{A}(\mathsf{crs})$ and $\{a_j, b_j\}_{j\in[k]} \leftarrow \mathcal{E}(\mathsf{crs})$. Next, $\mathcal{A}'$ reconstructs a vector $\boldsymbol{v} \in (\{0,1\}^k)^n$ from the set $\{a_j, b_j\}_{j\in[k]}$. This can be done by setting $v_{ij} = 0$ if $p_i \mid a_j$ and $v_{ij} = 1$ if $p_i \mid b_j$, where $p_i \leftarrow \mathsf{PrimeGen}(i)$ (in case both or neither cases occur, abort). Finally, $\mathcal{A}'$ runs all the checks as in game G_2, and if G_2 would output 1, then $\mathcal{A}'$ outputs $(\boldsymbol{v}, \{A_j, B_j\}_{j\in[k]}, I, \boldsymbol{y}, \{\Gamma_{Ij}, \Delta_{Ij}\}_{j\in[k]}, \boldsymbol{y}', \{\Gamma'_{Ij}, \Delta'_{Ij}\}_{j\in[k]})$, otherwise $\mathcal{A}'$ aborts.

To claim that $\Pr[G_2 = 1] = \Pr[G_3 = 1]$, we observe that whenever G_2 returns 1 it is the case that $a_j \cdot b_j = u_n = \prod_{i=1}^n p_i$ for all $j \in [k]$; therefore $\mathcal{A}'$ never aborts. □

Game G_4: this is the same as game G_3 except that the game outputs 0 if during any computation of lines 3 and 4 it happens that $\mathsf{PrimeGen}(i) = \mathsf{PrimeGen}(i')$ for distinct $i \neq i'$. It is straightforward to show that the probability of this event is bounded by the probability of finding collisions in $\mathsf{PrimeGen}$, i.e., that under the collision resistance of $\mathsf{PrimeGen}$ it holds $\Pr[G_3] - \Pr[G_4] \in \mathsf{negl}(\lambda)$.

To conclude the proof of our Theorem, we prove that any PPT adversary can win in G_4 with only negligible probability assuming that the strong RSA assumption holds in $\mathbb{G}$.

Lemma 3.4. *If the* strong RSA assumption *holds for* Ggen*, then for every PPT adversary $\mathcal{A}'$ running in game G_4 we have that $Pr[G_4 = 1] \in \mathsf{negl}(\lambda)$.*

Proof For the proof, we rely on the following lemma that defines a computational problem that we prove it is implied by the Strong RSA assumption.

Lemma 3.5. *Let* Ggen *be a hidden order group generation algorithm where the* strong RSA assumption *holds and* $\mathsf{PrimeGen}$ *a deterministic collision resistant function that maps integers to primes. Then for any PPT adversary $\mathcal{A}$ and any $n = \mathsf{poly}(\lambda)$, the probability below is negligible:*

$$\Pr\left[\begin{array}{c} u^p = g_0^a \cdot g_1^b \\ \wedge (p \nmid a \vee p \nmid b) \\ \wedge u \in \mathbb{G} \wedge (a,b) \in \mathbb{Z}^2 \wedge p \in S \end{array} : \begin{array}{l} \mathbb{G} \leftarrow \mathsf{Ggen}(\lambda) \\ g_0, g_1 \leftarrow \mathbb{G} \\ S = \{p_i \leftarrow \mathsf{PrimeGen}(i)\}_{i=1}^n \\ (u,p,a,b) \leftarrow \mathcal{A}(\mathbb{G}, g_0, g_1, S) \end{array}\right] \in \mathsf{negl}(\lambda)$$

We proceed assuming that the lemma holds; its proof is deferred to the end.

Suppose by contradiction the existence of a PPT adversary $\mathcal{A}'$ such that $\Pr[G_4] = \epsilon$ with ϵ non-negligible. Below we show how to construct an adversary $\mathcal{B}$ that uses $\mathcal{A}'$ in order to solve the problem of Lemma 3.5 with probability ϵ.

- $\mathcal{B}(\mathbb{G}, g_0, g_1)$ samples a random $g \leftarrow_\$ \mathbb{G}$, then determines a $\mathsf{PrimeGen}$ as in $\mathsf{VC.Setup}$, sets $\mathsf{crs} \leftarrow (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen})$, and finally runs $\mathcal{A}$ on input crs.
- $\mathcal{A}(\mathsf{crs})$ responds with a tuple $(\boldsymbol{v}, \{A_j, B_j\}_{j\in[k]}, I, \boldsymbol{y}, \pi, \boldsymbol{y}', \pi')$.

– $\mathcal{B}$ computes $\{a_j, b_j\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}([n], \boldsymbol{v})$,
$\{a_{Ij}, b_{Ij}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y})$ and
$\{a'_{Ij}, b'_{Ij}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y}')$ as in game G_3.
– If $\mathcal{A}'$ wins the game then we have that all the following conditions holds:

$$\boldsymbol{y} \neq \boldsymbol{y}', \bigwedge_{j=1}^{k} \left({\Gamma_{Ij}}^{a_{Ij}} = A_j \wedge {\Delta_{Ij}}^{b_{Ij}} = B_j \right) = 1, \bigwedge_{j=1}^{k} \left({\Gamma'_{Ij}}^{a'_{Ij}} = A_j \wedge {\Delta'_{Ij}}^{b'_{Ij}} = B_j \right) = 1$$

,

$$\bigwedge_{j=1}^{k} (A_j \cdot B_j = g_0^{a_j} \cdot g_1^{b_j}).$$

From $\boldsymbol{y} \neq \boldsymbol{y}'$ we get that there is at least one pair of indices $l \in [m]$ and $j \in [k]$ such that $y_{lj} \neq y'_{lj}$. Say wlog that $y_{lj} = 0$ and $y'_{lj} = 1$. Also, if we parse $I = \{i_1, \ldots, i_m\}$, we let $i = i_l \in [m]$. So we fix these indices i and j, and let $p_i = \mathsf{PrimeGen}(i)$ be the corresponding prime.
Notice that by construction of $\mathsf{PartndPrimeProd}$ (and since we assumed no collision occurs in $\mathsf{PrimeGen}$) we have that either $p_i \nmid a_j$ or $p_i \nmid b_j$ holds. Additionally, by our assumption that $y_{lj} = 0$ and $y'_{lj} = 1$, the following holds: $p_i \mid a_{Ij}$, $p_i \nmid b_{Ij}$, $p_i \nmid a'_{Ij}$, $p_i \mid b'_{Ij}$.
From the other condition on the validity of the proofs, $\mathcal{B}$ can compute two group elements $\hat{\Gamma}, \hat{\Delta}$ such that $\hat{\Gamma}^{p_i} = A_j$ and $\hat{\Delta}^{p_i} = B_j$.
Combining this with the condition $A_j \cdot B_j = g_0^{a_j} \cdot g_1^{b_j}$, we have that $(\hat{\Gamma} \cdot \hat{\Delta})^{p_i} = g_0^{a_j} \cdot g_1^{b_j}$.
– $\mathcal{B}$ sets $w = \hat{\Gamma} \cdot \hat{\Delta}$ and outputs the tuple (w, p_i, a_j, b_j).

From all the above observations, if $\mathcal{A}'$ makes game G_4 return 1, then the tuple returned by $\mathcal{B}$ is a suitable solution for the problem of Lemma 3.5, which in turn reduces to the Strong RSA assumption. □

By combining all the lemmas we have that any PPT adversary has at most negligible probability of breaking the position binding of our SVC scheme. □

Proof [Proof of Lemma 3.5] Suppose that for a PPT adversary $\mathcal{A}$ the above probability is a non-negligible value ϵ. We will construct an adversary $\mathcal{B}$ that breaks strong RSA assumption with a non-negligible probability. $\mathcal{B}$ takes as input $(\mathbb{G}, g)$. We denote as $G_{\mathcal{A}}$ the game defined in lemma (parametrized by an adversary $\mathcal{A}$). We define two different reductions:

Reduction 1. In reduction 1 the adversary $\mathcal{B}$ breaks strong RSA assumption only in case where the adversary $\mathcal{A}$ outputs a tuple (u, p, a, b) such that $p \mid a$ (and thus from assumption $p \nmid b$) and fails otherwise. $\mathcal{B}$ proceeds as follows.

$\mathcal{B}(\mathbb{G}, g)$ samples $\gamma \leftarrow_{\$} [1, 2^{\lambda}\mathsf{ord}_{max}]$, where ord_{max} is the upper bound of the order of $\mathbb{G}$ outputted by $\mathsf{Ggen}(1^{\lambda})$ (see section 2.1), and sets $g_0 \leftarrow g^{\gamma}, g_1 \leftarrow g$. $\mathcal{B}$ runs $\mathcal{A}$ on input $(\mathbb{G}, g_0, g_1)$. γ is sampled from a large enough domain so that g^{γ} is statistically close to a uniformly distributed g_0 from $\mathbb{G}$ hence g_0, g_1 are indistinguishable to two uniformly random elements of $\mathbb{G}$. $\mathcal{A}(\mathbb{G}, g_0, g_1, S)$ responds with a tuple (u, p, a, b) and sends it to $\mathcal{B}$. We condition our analysis on the event $p \mid a$, meaning that $\mathcal{B}$ stops in case $p \nmid a$.

Assume that $u^p = g_0^a \cdot g_1^b \wedge (p \mid a \wedge p \nmid b) \wedge u \in \mathbb{G} \wedge (a, b) \in \mathbb{Z}^2 \wedge p \in S$ then we will show that $\mathcal{B}$ can break the strong RSA assumption. We argue that $p \mid a$ leads to $\gcd(p, \gamma a + b) = 1$.

Let $\gcd(p, \gamma a + b) \neq 1$, meaning that $\gcd(p, \gamma a + b) = p$, then $p \mid \gamma a + b \Rightarrow \gamma a + b = 0 \pmod p$. However, $p \mid a \Rightarrow a = 0 \pmod p$. From the two previous facts we infer that $b = 0 \pmod p \Rightarrow p \mid b$, hence $p \mid a \wedge p \mid b$, which is a contradiction. Therefore, assuming that $\gcd(p, \gamma a + b) = 1$, $\mathcal{B}$ uses the extended Euclidean algorithm to compute (α, β) such that $\alpha p + \beta(a\gamma + b) = 1$. We know that $u^p = g_0^a g_1^b = g^{a\gamma+b} \Rightarrow u = g^{\frac{a\gamma+b}{p}}$ hence it follows that $g^{1/p} = g^{\frac{\alpha p + \beta(a\gamma+b)=1}{p}} = g^{\alpha + \beta \frac{a\gamma+b}{p}} = g^\alpha \cdot u^\beta$. Finally, $\mathcal{B}$ outputs $(g^\alpha \cdot u^\beta, p)$ which is a valid strong-RSA solution.

Reduction 2. In reduction 2 the adversary $\mathcal{B}$ breaks strong RSA assumption only in case where the adversary $\mathcal{A}$ outputs a tuple (u, p, a, b) such that $p \nmid a$ and fails otherwise.

$\mathcal{B}(\mathbb{G}, g)$ samples $\gamma \leftarrow_\$ [1, 2^\lambda \mathsf{ord}_{max}]$, where ord_{max} is the upper bound of the order of $\mathbb{G}$ outputted by $\mathsf{Ggen}(1^\lambda)$ (see section 2.1), defines $S := \{p_i \leftarrow \mathsf{PrimeGen}(i)\}_{i=1}^n$ and $\mathsf{prod} \leftarrow \prod_{i=1}^n p_i$ and sets $g_0 \leftarrow g, g_1 \leftarrow g^{\gamma \cdot \mathsf{prod}}$. $\mathcal{B}$ sends $(\mathbb{G}, g_0, g_1)$ to $\mathcal{A}$. γ is sampled from a large enough domain so that g^γ is statistically close to a uniformly distributed g_1 from $\mathbb{G}$ hence g_0, g_1 are indistinguishable to two uniformly random elements of $\mathbb{G}$. $\mathcal{A}(\mathbb{G}, g_0, g_1, S)$ responds with a tuple (u, p, a, b) and sends it to $\mathcal{B}$. We condition our analysis on the event $p \nmid a$, meaning that $\mathcal{B}$ stops in case $p \mid a$.

Assume that $u^p = g_0^a \cdot g_1^b \wedge p \nmid a \wedge u \in \mathbb{G} \wedge (a, b) \in \mathbb{Z}^2 \wedge p \in S$ then we will show that $\mathcal{B}$ can break the strong RSA assumption. We argue that $\gcd(p, a + b\gamma\mathsf{prod}) = 1$. Let $\gcd(p, a + b\gamma\mathsf{prod}) \neq 1$, meaning that $\gcd(p, a + b\gamma\mathsf{prod}) = p$, then $p \mid a + b\gamma\mathsf{prod} \Rightarrow a + b\gamma\mathsf{prod} = 0 \pmod p$. However, prod includes p ($p \in S$) we know that $p \mid b\gamma\mathsf{prod} \Rightarrow b\gamma\mathsf{prod} = 0 \pmod p$. From the two previous facts we infer that $a = 0 \pmod p \Rightarrow p \mid a$ which is a contradiction. $\mathcal{B}$ uses the extended Euclidean algorithm to compute (α, β) such that $\alpha p + \beta(a + b\gamma\mathsf{prod}) = 1$. We know that $u^p = g_0^a g_1^b = g^{a + b\gamma\mathsf{prod}} \Rightarrow u = g^{\frac{a+b\gamma\mathsf{prod}}{p}}$ hence it follows that $g^{1/p} = g^{\frac{\alpha p + \beta(a+b\gamma\mathsf{prod})=1}{p}} = g^{\alpha + \beta\frac{a+b\gamma\mathsf{prod}}{p}} = g^\alpha \cdot u^\beta$. Finally, $\mathcal{B}$ outputs $(g^\alpha \cdot u^\beta, p)$ which is a valid strong-RSA solution.

To conclude the proof, notice that:

$$\begin{aligned}\Pr[G_\mathcal{A} = 1] &= \Pr[G_\mathcal{A} = 1 | p \mid a] \Pr[p \mid a] + \Pr[G_\mathcal{A} = 1 | p \nmid a] \Pr[p \nmid a] \\ &\leq \Pr[G_\mathcal{A} = 1 | p \mid a] + \Pr[G_\mathcal{A} = 1 | p \nmid a]\end{aligned}$$

The reductions 1 and 2 described above show that under the strong RSA assumption $\Pr[G_\mathcal{A} = 1 | p \mid a]$ and $\Pr[G_\mathcal{A} = 1 | p \nmid a]$ respectively are negligible. Hence, we have that $\Pr[G_\mathcal{A} = 1] \in \mathsf{negl}(\lambda)$, which concludes the proof. □

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. 3.1. 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 2.1 for a discussion on possible instantiations of $\mathbb{G}$.

We note that by using the specific PoProd_2 protocol given in Sec. 3.1 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.

3.3 Incrementally Aggregatable Subvector Openings

In this section we define an aggregation property for SVC schemes and we show that our SVC construction is aggregatable. 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 3.1 (Aggregatable Subvector Openings). *A vector commitment scheme* VC *with subvector openings is called* aggregatable *if there exists algorithms* VC.Agg*,* VC.Disagg *working as follows:*

$\mathsf{VC.Agg}(\mathsf{crs}, (I, \boldsymbol{v}_I, \pi_I), (J, \boldsymbol{v}_J, \pi_J)) \to \pi_K$ *takes as input two triples* $(I, \boldsymbol{v}_I, \pi_I), (J, \boldsymbol{v}_J, \pi_J)$ *where* I *and* J *are sets of indices,* $\boldsymbol{v}_I \in \mathcal{M}^{|I|}$ *and* $\boldsymbol{v}_J \in \mathcal{M}^{|J|}$ *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* $K = I \cup J$*.*

$\mathsf{VC.Disagg}(\mathsf{crs}, I, \boldsymbol{v}_I, \pi_I, K) \to \pi_K$ *takes as input a triple* $(I, \boldsymbol{v}_I, \pi_I)$ *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 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* $(I, \boldsymbol{v}_I, \pi_I)$ *s.t.* $\mathsf{VC.Ver}(\mathsf{crs}, C, I, \boldsymbol{v}_I, \pi_I) = 1$*, the following two properties hold:*

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

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

where $K = I \cup J$ *and* $\boldsymbol{v}_K$ *is the ordered union* $\boldsymbol{v}_{I \cup J}$ *of* $\boldsymbol{v}_I$ *and* $\boldsymbol{v}_J$*;*

2. *for any subset of indices* $K \subset I$*,*

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

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

Aggregation Conciseness. *There exists a fixed polynomial* $p(\cdot)$ *in the security parameter such that all openings produced by* VC.Agg *and* 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.

3.4 Incremental Aggregation of Our Construction

Here we show that our SVC scheme of Section 3.2 is aggregatable.

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

$$\Gamma_{Kj} \leftarrow \Gamma_{Ij}^{a_{Lj}}, \quad \Delta_{Kj} \leftarrow \Delta_{Ij}^{b_{Lj}}$$

and return $\pi_K := \{\pi_{K1}, \dots, \pi_{Kk}\} := \{(\Gamma_{K1}, \Delta_{K1}), \dots, (\Gamma_{Kk}, \Delta_{Kk})\}$

$\mathsf{VC.Agg}(\mathsf{crs}, (I, \boldsymbol{v}_I, \pi_I), (J, \boldsymbol{v}_J, \pi_J)) \to \pi_K := \{(\Gamma_{K1}, \Delta_{K1}), \dots, (\Gamma_{Kk}, \Delta_{Kk})\}$**.**

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

$$\Gamma_{Kj} \leftarrow \mathbf{ShamirTrick}(\Gamma_{I'j}, \Gamma_{Jj}, a_{I'j}, a_{Jj}), \quad \Delta_{Kj} \leftarrow \mathbf{ShamirTrick}(\Delta_{I'j}, \Delta_{Jj}, b_{I'j}, b_{Jj}).$$

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 $\{a_{Kj}, b_{Kj}\}_{j\in[k]} \leftarrow \mathsf{PartndPrimeProd}(K, \boldsymbol{v}_K)$, then $a_{Ij} = a_{Lj} \cdot a_{Kj}$, and thus $A_j = \Gamma_{Ij}^{a_{Ij}} = \Gamma_{Ij}^{a_{Lj} \cdot a_{Kj}} = \Gamma_{Kj}^{a_{Kj}}$ (and similarly for Δ_{Kj}).

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_{Jj} (resp. $b_{I'j}$ and b_{Jj}) are coprime unless a collision occurs in $\mathsf{PrimeGen}$.

3.5 Efficiency of our VC scheme

We summarize the efficiency of our construction in terms of both the computational cost of the algorithms and the communication (CRS, commitment and openings size). For this analysis we consider an instantiation of $\mathsf{PrimeGen}$ with a deterministic function that maps every integer in $[n]$ into a unique prime number, which can be of $\log n$ bits.

Our scheme is presented in order to support vectors of length n of k-bits-long strings. We summarize efficiency in terms of k and n. However, we note that k is actually only a parameter and our scheme can work with any setting of vectors $\boldsymbol{v}$ of length N of ℓ-bits long strings. In this case, it is sufficient to fix an arbitrary k that divides ℓ and to spread each $v_i \in \{0,1\}^\ell$ over ℓ/k positions.

In terms of computation, $\mathsf{VC.Setup}$ generates the group description and samples 3 generators, while $\mathsf{VC.Specialize}$ computes one exponentiation in $\mathbb{G}$ with an $(n \log n)$-long integer. The CRS consists of 3 elements of $\mathbb{G}$, and the specialized CRS (for any n) is one group element. Committing to a vector $\boldsymbol{v} \in (\{0,1\}^k)^n$ requires about k exponentiations with an $(n \log n)$-long integer each. A commitment consists of $4k$ elements of $\mathbb{G}$ and $2k$ integers in $\mathbb{Z}_{2^\lambda}$. Creating an opening for a set I of m positions has about the same cost of committing, and the opening consists of $2k$ group elements (resp. $4k$ elements when using the PoE to make verification more efficient, see Remark 3.1). Verifying an opening for set I requires about k exponentiations with $(m \cdot \log n)$-bit integers (resp. $4k$ exponentiations with λ-bit integers, $2k$ multiplications in $\mathbb{G}$ and $O(km \log(n))$ multiplications in $\mathbb{Z}_{2^\lambda}$, when using PoE) to check equation (1), plus $5k$ exponentiations with λ-bit integers and $3k$ multiplications in $\mathbb{G}$ to verify PoProd_2 proofs in equation (2).

3.6 Comparison with Related Work

We compare our VC construction 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 scheme with $k = 1$ (and thus $n = N \cdot \ell$).

Setup Model. [BBF19] works with a fully universal CRS, whereas our scheme has a universal CRS with deterministic specialization, which however, in comparison to [LM19], outputs *constant-size* parameters instead of linear.

Aggregation. Our scheme is the only one that has *incremental aggregation*; the VC of [BBF19] supports aggregation only on openings created by VC.Open (i.e., it is one-hop). Neither [BBF19] nor [LM19] has disaggregatable proofs (unless in a different model where one works linearly in the length of the vector or knows the full vector). 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 VC.Open.

Efficiency. From the table, one can see that our VC has: slightly worse commitments size than [BBF19, LM19], computational asymptotic performances similar to [BBF19], and opening size slightly better than [BBF19]. However, 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 construction can use preprocessing to achieve more favorable time and memory costs*. The idea is that in such a setting 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. Notably, in comparison to a similar solution for [BBF19] ours can save a factor ℓ in storage.

The idea is the following. In our VC one can precompute and store an aggregated opening (two group elements) for each sequence of B ℓ-bit-long substrings of the vector. Upon opening, the user has to disaggregate some openings and then aggregate for the requested positions. This way, our opening time for m positions goes from, roughly, $O((n-m)\log n)$ down to $O(mB \log m \log n)$, which is way more efficient. This is possible by storing $2n/(B\ell)$ group elements. Interestingly, our incremental aggregation/disaggregation property allows us to choose B in a flexible way allowing for different time-space tradeoffs. For instance we can choose $B = \sqrt{n}$ so as to get $O(\sqrt{n})$ storage and about $O(m\sqrt{n} \log n)$ opening time, or can even choose B according to applications-dependent heuristics. Such flexibility is not possible in the VC of [BBF19] where 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) our solution saves a factor ℓ in storage, which concretely turns into $3\times$ less storage. A detailed analysis of this precomputation technique is provided in the next section.

3.7 Detailed Efficiency Comparison for Committing and Opening with Precomputation

We provide a detailed efficiency comparison between the VC of [BBF19] and our construction in the scenario where one a user commits to a vector and then must generate openings for various

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

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

Table 1. Comparison between the VCs of [BBF19], [LM19] and this work. We consider committing to a vector $\boldsymbol{v} \in (\{0,1\}^\ell)^N$ 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 VC.Open in [LM19] costs $O(\ell \cdot (N-m) \cdot \log(N-m))$.

subvectors. This is for example the use case when the VC is used for proofs of retrievability and IOPs [BBF19]. In such a case, both the VC of [BBF19] and ours allow the user to precompute and store some advice information that can speedup the opening creation. In particular, in both constructions various storage-computation tradeoffs are possible. In what follows we investigate what the opener should store so that computation of openings is practical.

As before, we consider a vector $\boldsymbol{v}$ of N blocks of ℓ bits each and consider the instantiation of our construction with $k = 1$ and $n = N\ell$. Similarly we consider an instantiation of [BBF19] VC for binary vectors of length $n = N\ell$.

Boneh-Bunz-Fisch VC. Let us recall that in [BBF19] a commitment to $\boldsymbol{v} \in \{0,1\}^n$ is $Acc = g^b$ with $b = \prod_{j \in [n], v_j = 1} p_j$.

When asked for opening of some positions in the set I, the vector owner has to provide a batched membership proof for all $\{p_j : j = (i-1)\ell + l, i \in I, l \in [\ell], v_j = 1\}$ and a batched non-membership proof for all $\{p_j : j = (i-1)\ell + l, i \in I, l \in [\ell], v_j = 0\}$.

For the membership proof, in the commitment phase one can precompute $\{W_i = g^{b/b_i} : i \in [N]\}$ where $b_i = \prod_{l \in [\ell], v_{il} = 1} p_{(i-1)\ell + l}$, which can be done in time $O(N \log N \cdot \ell \log(\ell N))$ and adds N elements of $\mathbb{G}$ to the advice information. Next, in the opening phase, in order to compute a membership witness for a set of positions I one can use the aggregation property to compute a witness W_I from all W_i with $i \in I$.

For the non-membership proof, there are instead two options:

1. Compute the batch-nonmembership witness from scratch

2. Precompute and store (unbatched) non-membership witnesses for all 0's of the vector and then aggregate the necessary ones to provide the opening asked.

We argue that an intermediate solution of precomputing a fraction of non-membership witnesses and computing the rest from scratch does not provide any benefit since even if a single non-membership witness needs to be computed, it requires the whole vector and computing the corresponding product of primes. So, in the end the intermediate solution will be more costly than both the above ones.

1. Compute non-membership witness from scratch To compute a non-membership witness one needs the product b of all the primes in the accumulator (i.e., all primes that correspond to 1's in $\boldsymbol{v}$). There are in turn two possible ways to deal with this:

- Precompute and store b, which requires $O(\log(N\ell)\cdot N\cdot\ell)$ computation and $|b| = O(N\cdot\ell\cdot\log(N\ell))$ bits of storage.
- Compute b online from all p_i's, which requires $O(\log(N\ell)\cdot N\cdot\ell)$ computing power.

The computations needed to obtain a single non-membership witness is proportional to the size of b, which is $O(\ell \cdot N \cdot \log(\ell N))$ $\mathbb{G}$. Hence, virtually there is no big improvement in the opening time by precomputing b, since the group exponentiations are more costly (although concretely it saves the online computation of it). Furthermore, keeping $|b| = O(N \cdot \ell \cdot \log(N\ell))$ bits of storage may get impractical for big N.

2. Precompute non-membership witnesses and then aggregate. The idea is similar to the aggregation technique mentioned above for membership witnesses. However, a crucial difference is that, as stated in the [BBF19], for non-membership witnesses one has only *one-hop* aggregation. This means one must precompute and store non-membership witnesses for each block of the vector that. However these non-membership witnesses have size proportional to the number of bits of the each block (plus one group element).

This technique requires storage of $O(N)$ group elements plus $O(N\cdot\ell)$ field elemetns on average. Precisely, the size non-membeship witness for each block is $|\mathbb{G}| + \log(N\ell) \times \#\{\text{0-bits in the block}\}$, hence the total siez of non-membership witnesses is $N|\mathbb{G}| + N\ell\log(N\ell)$ in the worst case and $N|\mathbb{G}| + N\ell\log(N\ell)/2$ in an average case where half of the bits of the vector are 0. To conclude, with the VC of [BBF19], one would need, on average, to precompute and store $2N|\mathbb{G}| + N\ell\log(N\ell)/2$ bits.

This work's VC. Here we show that with our VC construction one could save on storage space and is also possible to achieve flexible solutions that offerent different time-memory tradeoffs. This is possible thanks to the incremental aggregation and disaggregation properties of our scheme.

The idea is the following. Assume one wants to commit to a vector $\boldsymbol{v}$ of N positions of ℓ bits each. Let B be an integer parameter such that $B \mid N$. In the commitment phase, one can create openings for $N' = N/B$ subvectors of $\boldsymbol{v}$ that covers the all vector (e.g., B contiguous positions[14]). Let $\pi_{I_1}, \ldots, \pi_{I_{N'}}$ be such openings; these elements are stored as advice information. The storage space needed is $\frac{2N}{B}|\mathbb{G}|$ bits of memory.

[14] One could actually choose many different ways to group subvectors in precomputation; the best way could be application-dependent, e.g., based on the expected structure of the positions to be opened one could optimize the grouping so as to minimize the need of unaggregating openings.

Next, in the opening phase, in order to compute the opening for a subvector $\boldsymbol{v}_I$ of m positions, one should: (i) fetch the subset of openings π_{I_j} such that, for some S, $I \subseteq \cup_{j \in S} I_j$, (ii) possibly unaggregate some of them and then aggregate in order to compute π_I. The computation time for this operation, is in the worst case:

$$m \log(m) \cdot \ell \cdot \log(N\ell) + (mB - m)\ell \log(N\ell) \ \mathbb{G}$$

This computing time is obtained by considering the application of pairwise aggregation in a tree fashion, which in turn requires running the extended gcd algorithm on integers of growing size.

To give a very general example of the above process, assume one has stored $\pi_{\{1,2\}}$ and $\pi_{\{3,4,5\}}$ and is asked for $\pi_{\{2,3\}}$, then she has to compute first π_2 and π_3 by unaggregating $\pi_{\{1,2\}}$ and $\pi_{\{3,4,5\}}$ respectively, and then aggregate them to $\pi_{\{2,3\}}$. Below are two more examples in picture:

	$\boldsymbol{v}_1$				$\boldsymbol{v}_2$				$\boldsymbol{v}_3$				$\boldsymbol{v}_4$				$\boldsymbol{v}_5$			
$\underline{B=2}$:	1	0	0	0	1	1	1	0	1	1	1	1	1	0	1	0	0	0	0	0
	$\Gamma_{\{1,2\}}, \Delta_{\{1,2\}}$								$\Gamma_{\{3,4\}}, \Delta_{\{3,4\}}$								$\Gamma_{\{5\}}, \Delta_{\{5\}}$			

$2 \cdot |\mathbb{G}| \cdot n/2$ bits in opening advice

	$\boldsymbol{v}_1$				$\boldsymbol{v}_2$				$\boldsymbol{v}_3$				$\boldsymbol{v}_4$				$\boldsymbol{v}_5$			
$\underline{B=n}$:	1	0	0	0	1	1	1	0	1	1	1	1	1	0	1	0	0	0	0	0
	$\Gamma_{\{1,2,3,4,5\}}, \Delta_{\{1,2,3,4,5\}}$																			

$2 \cdot |\mathbb{G}| \cdot 1$ bits in opening advice

To conclude, even if we consider the case $B = 1$, our approach requires much less storage than in [BBF19]: $2N|\mathbb{G}|$ vs.$2N|\mathbb{G}| + N\ell \log(N\ell)/2$ bits, while the computing time for an opening of m blocks requires at least 50% more time for the non-membership witnesses in [BBF19] (which leads to 25% more time in the average case).

4 Arguments of Knowledge for our SVC

We propose three Arguments of Knowledge (AoK) related to our vector commitment scheme presented in the previous section. More specifically, the first AoK allows one to prove knowledge of an opening of a subvector. The second AoK, is a direct outcome of the first and allows one to prove that two given commitments share a common subvector. Finally, the third protocol allows one to commit to a prefix-subvector of a vector and prove the knowledge of it succinctly.

Similarly to section 3.1, our protocols build on the techniques for succinct proofs in groups of unknown order from [BBF19]. Furthermore, these arguments of knowledge are not zero knowledge and they serve efficiency purposes. Interestingly, one can prove knowledge of a portion of a vector committed *without having to send the actual vector values.* The proofs are constant-size which leads to an improvement of communication complexity linear in the size of the opening.

4.1 Building block: A Stronger Proof of Product

Before proceeding to describing the main protocols, we introduce another one that is used as building block. This is an argument of knowledge, called PoProd^*, for the relation R_{PoProd^*} described

Fig. 2. PoProd^* protocol

$\mathsf{Setup}(1^\lambda)$: run $\mathbb{G} \leftarrow\$ \mathsf{Ggen}(1^\lambda)$, $g \leftarrow\$ \mathbb{G}$, set $\mathsf{crs}^* := (\mathbb{G}, g)$.
Prover's input: $(\mathsf{crs}^*, (A, B, C, \Gamma, \Delta), (a, b))$. Verifier's input: $(\mathsf{crs}^*, (A, B, C, \Gamma, \Delta))$.

$\mathsf{V} \to \mathsf{P}$: $h \leftarrow\$ \mathbb{G}$
$\mathsf{P} \to \mathsf{V}$: $z := (z_a, z_b)$ computed as $z_a \leftarrow h^a, z_b \leftarrow h^b$
$\mathsf{V} \to \mathsf{P}$: $\ell \leftarrow\$ \mathsf{Primes}(\lambda)$ and $\alpha \leftarrow\$ [0, 2^\lambda)$
$\mathsf{P} \to \mathsf{V}$: $\pi := ((Q_A, Q_B, Q_C), r_a, r_b)$ computed as follows
- $(q_a, q_b, q_{ab}) \leftarrow (\lfloor a/\ell \rfloor, \lfloor b/\ell \rfloor, \lfloor ab/\ell \rfloor)$
- $(r_a, r_b) \leftarrow (a \mod \ell, b \mod \ell)$
- $(Q_A, Q_B, Q_C) := (\Gamma^{q_a} h^{\alpha q_a}, \Delta^{q_b} h^{\alpha q_b}, g^{q_{ab}})$

$\mathsf{V}(\mathsf{crs}, (A, B, C), z_a, z_b, \ell, \alpha, \pi)$:
- Compute $r_c \leftarrow r_a \cdot r_b \mod \ell$
- Output 1 iff $r_a, r_b \in [\ell] \ \wedge\ Q_A^\ell \Gamma^{r_a} h^{\alpha r_a} = A z_a^\alpha \ \wedge\ Q_B^\ell \Delta^{r_b} h^{\alpha r_b} = B z_b^\alpha \ \wedge\ Q_C^\ell g^{r_c} = C$

below, which uses a common reference string consisting of a hidden order group $\mathbb{G} \leftarrow \mathsf{Ggen}(1^\lambda)$ and a random generator $g \in \mathbb{G}$:

$$R_{\mathsf{PoProd}^*} = \left\{((A, B, C, \Gamma, \Delta), (a, b)) \in \mathbb{G}^5 \times \mathbb{Z}^3 \ :\ A = \Gamma^a \wedge B = \Delta^b \wedge C = g^{a \cdot b} \quad \right\}$$

The relation R_{PoProd^*} is similar to R_{PoProd} defined in Section 3.1 with the difference that now the first two bases Γ and Δ are not part of the common reference string, but part of the statement instead. As argued in [BBF19] the PoKE^* protocol is not secure anymore for adversarially chosen bases, therefore we cannot use PoProd protocol which assumes knowledge extractability of PoKE^*. To deal with this problem, we thus modify the protocol by using the protocol $\mathsf{PoKE2}$, which is secure for arbitrary bases. This comes with some cost: in our PoProd^* a proof consists of 5 group elements and 2 field elements, that is 2 group elements more comparing to proofs of PoProd. The protocol is in Fig. 2.

Theorem 4.1. *The* PoProd^* *protocol in Fig. 2 is an argument of knowledge for* R_{PoProd^*} *in the generic group model.*

The proof of the theorem above is similar to the proof of Theorem 3.1, except that we use the extractor $\mathcal{E}_{\mathsf{PoKE2}}$ of the protocol $\mathsf{PoKE2}$ from [BBF19] in order to extract integers a and b and $\mathcal{E}_{\mathsf{PoKE}^*}$ in order to extract the exponent of C.

4.2 A Succinct AoK of Opening for our VC Construction

We show an argument of knowledge of an I-opening with respect to a commitment C to a vector, where I is a set of positions. We emphasize that the goal of this protocol is not to keep the opening secret (i.e., the protocol is not zero knowledge, also our vector commitment scheme is not hiding). The goal is to reduce the communication complexity of an opening by proving knowledge of the subvector at positions I without having to actually send the values $\boldsymbol{v}_I$. Even though the argument of knowledge itself adds an overhead it is independent of the number of the positions. Hence, the protocol makes more sense for large sets of positions I as for a small number of positions the overhead of the AoK would exceed the size of the opening values.

Let $\mathsf{VC} = (\mathsf{VC.Setup}, \mathsf{VC.Specialize}, \mathsf{VC.Com}, \mathsf{VC.Open}, \mathsf{VC.Ver})$ be our SVC scheme from Section 3.2, and let us define the following relation

$$R_{\mathsf{PoKOpen}} = \{(\,(C, I), (\boldsymbol{y}, \pi_I)\,) : \mathsf{VC.Ver}(\mathsf{crs}, C, I, \boldsymbol{y}, \pi_I) = 1\}$$

that is parametrized by a CRS $\mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M})$, and where the statement consists of a commitment C and a set of indices $I \subseteq [n]$, and the witness consists of a vector $\boldsymbol{y} \in \mathcal{M}^{|I|}$ and an opening π_I.

For simplicity we present a protocol $\mathsf{PoKOpen}$ for the case when $k = 1$ in our VC (see section 3.2); extension to larger k is immediate. The idea of our protocol is that, given a commitment $C := ((A, B), \pi_{\mathsf{prod}})$ and a set of indices I, the prover, holding $\pi_I := (\Gamma_I, \Delta_I)$, first sends π_I to the verifier and then provides an AoK of (a_I, b_I) such that $\Gamma_I^{a_I} = A \wedge \Delta_I^{b_I} = B \wedge g^{a_I \cdot b_I} = U_I$, where $U_I \leftarrow g^{u_I}$ with $u_I \leftarrow \mathsf{PrimeProd}(I)$. This can be proven by using the PoProd^* protocol presented above. Finally the verifier should also verify the π_{prod} proof as in the normal verification of an opening algorithm.

$\mathsf{PoKOpen}$ protocol

Prover's input: $(\mathsf{crs}, (C, I), (\boldsymbol{y}, \pi_I)$. Verifier's input: $(\mathsf{crs}, (C, I))$.

$\underline{\mathsf{V}}$ Compute $u_I \leftarrow \mathsf{PrimeProd}(I)$ and then $U_I \leftarrow g^{u_I}$.
Similarly compute $u_n \leftarrow \mathsf{PrimeProd}([n])$ and then $U_n \leftarrow g^{u_n}$

$\underline{\mathsf{P}}$: Parse $\mathsf{crs} := (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen}, U_n)$, $C := (\{A, B\}, \pi_{\mathsf{prod}})$, $\pi_I := (\Gamma_I, \Delta_I)$. Compute $(a_I, b_I) \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y})$ and then $u_I \leftarrow \mathsf{PrimeProd}(I)$ and $U_I \leftarrow g^{u_I}$.

$\underline{\mathsf{P} \rightarrow \mathsf{V}}$: (Γ_I, Δ_I)

Finally a PoProd^* protocol (with an additional check of the commitment) between $\mathsf{P}((\mathbb{G}, g), (A, B, U_I, \Gamma_I, \Delta_I), (a_I, b_I))$ and $\mathsf{V}((\mathbb{G}, g), (A, B, U_I, \Gamma_I, \Delta_I))$ is executed:

$\underline{\mathsf{V} \rightarrow \mathsf{P}}$: $h \leftarrow_\$ \mathbb{G}$

$\underline{\mathsf{P} \rightarrow \mathsf{V}}$: $z := (z_a, z_b)$ computed as $z_a \leftarrow h^{a_I}, z_b \leftarrow h^{b_I}$

$\underline{\mathsf{V} \rightarrow \mathsf{P}}$: $\ell \leftarrow_\$ \mathsf{Primes}(\lambda)$ and $\alpha \leftarrow_\$ [0, 2^\lambda)$

$\underline{\mathsf{P} \rightarrow \mathsf{V}}$: $\pi := ((Q_A, Q_B, Q_C), r_a, r_b)$ computed as follows

- $(q_a, q_b, q_{ab}) \leftarrow (\lfloor a_I/\ell \rfloor, \lfloor b_I/\ell \rfloor, \lfloor a_I b_I/\ell \rfloor)$
- $(r_a, r_b) \leftarrow (a_I \mod \ell, b_I \mod \ell)$
- $(Q_A, Q_B, Q_C) := \left(\Gamma_I^{q_a} h^{\alpha q_a}, \Delta_I^{q_b} h^{\alpha q_b}, g^{q_{ab}}\right)$

$\underline{\mathsf{V}:}$ Parse $\mathsf{crs} := (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen}, U_n)$ and $C := (\{A, B\}, \pi_{\mathsf{prod}})$.

- Compute $r_c \leftarrow r_a \cdot r_b \mod \ell$
- Output 1 iff $r_a, r_b \in [\ell] \;\wedge\; Q_A^\ell \Gamma_I^{r_a} h^{\alpha r_a} = A z_a^\alpha \;\wedge\; Q_B^\ell \Delta_I^{r_b} h^{\alpha r_b} = B z_b^\alpha \;\wedge\; Q_C^\ell g^{r_c} = U_I \;\wedge$ $\mathsf{PoProd}_2.\mathsf{V}(\mathsf{crs}, (A \cdot B, U_n), \pi_{\mathsf{prod}})$

We state the following theorem.

Theorem 4.2. *If PoProd^* is a succinct argument of knowledge for R_{PoProd^*}, then protocol $\mathsf{PoKOpen}$ is a succinct argument of knowledge for relation R_{PoKOpen} with respect to algorithm $\mathsf{VC.Ver}$ of our construction of Section 3.2.*

Proof Let $\mathcal{A}$ be an adversary of the Knowledge Extractability of $\mathsf{PoKOpen}$ such that: $((C, I), \mathsf{state}) \leftarrow \mathcal{A}_0(\mathsf{pp})$, $\mathcal{A}_1(\mathsf{pp}, (C, I), \mathsf{state})$ executes with $\mathsf{V}(\mathsf{pp}, (C, I))$ the protocol $\mathsf{PoKOpen}$ and the verifier accepts with a non-negligible probability ϵ. We will construct an extractor $\mathcal{E}$ that having access to the internal state of $\mathcal{A}_1$ and on input $(\mathsf{pp}, (C, I), \mathsf{state})$, outputs a witness $(\boldsymbol{y}, \pi_I)$ of R_{PoKOpen} with overwhelming probability and runs in (expected) polynomial time.

To prove knowledge extractability of $\mathsf{PoKOpen}$ we rely on the knowledge extractability of PoProd^*. More precisely, given a $\mathsf{PoKOpen}$ execution between $\mathcal{A}$ and V, $(\mathit{\Gamma}_I, \mathit{\Delta}_I, \pi_{\mathsf{PoProd}'})$, $\mathcal{E}$ constructs an adversary $\mathcal{A}' = (\mathcal{A}'_0, \mathcal{A}'_1)$ of PoProd^* Knowledge Extractability and, by using the input and internal state of $\mathcal{A}_1$, simulates an execution between $\mathcal{A}'$ and V. Thus algorithm $\mathcal{A}'_0$ outputs $(((\mathbb{G}, g), (A, B, U_I, \mathit{\Gamma}_I, \mathit{\Delta}_I)), \mathsf{state})$; $\mathcal{A}'_1$ outputs $(z_a, z_b, (Q_A, Q_B, Q_C), r_a, r_b)$. It is obvious that if the initial execution is accepted by V so is the PoProd^* execution. From Knowledge Extractability of PoProd we know that there exists an extractor $\mathcal{E}'$ corresponding to $\mathcal{A}'_1$ that outputs (a_I, b_I) such that $A = \mathit{\Gamma}_I^{a_I} \wedge B = \mathit{\Delta}_I^{b_I} \wedge U_I = g^{a_I \cdot b_I}$. Since U_I is also computed from V it holds that $U_I = g^{u_I}$, unless with a negligible probability that $\mathcal{A}'$ can find an $x \neq u_I$ such that $g^x = U_I = g^{u_I}$ (which implies finding a multiple of the order of $\mathbb{G}$). Therefore $g^{u_I} = U_I = g^{a_I \cdot b_I}$ and using the same argument we know that $u_I = a_I \cdot b_I$ (unless with negligible probability).

So, $\mathcal{E}$ uses $\mathcal{E}'$ and gets a (a_I, b_I) such that $A = \mathit{\Gamma}_I^{a_I} \wedge B = \mathit{\Delta}_I^{b_I} \wedge U_I = g^{a_I \cdot b_I}$. Then computes $u_I \leftarrow \mathsf{PrimeProd}(I)$ and works as follows: for each $i \in I$ computes $p_i \leftarrow \mathsf{PrimeGen}(i)$ and if $p_i \mid a_I$ then sets $y_i = 0$, otherwise if $p_i \mid a_I$ then sets $y_i = 1$. It is clear that p_i divides exactly one of a_I, b_I since $a_I \cdot b_I = u_I = \prod_{i \in I} p_i := \prod_{i \in I} \mathsf{PrimeGen}(i)$ (unless with a negligible probability that a collision happened in $\mathsf{PrimeGen}$). Finally sets the subvector $\boldsymbol{y} = (y_i)_{i \in I}$ and $\pi_I = (\mathit{\Gamma}_I, \mathit{\Delta}_I)$. As stated above $\mathit{\Gamma}_I^{a_I} = A \wedge \mathit{\Delta}_I^{b_I} = B$ and also since V verifies the $\mathsf{PoKOpen}$ protocol it holds that $\mathsf{PoProd}_2.\mathsf{V}(\mathsf{pp}, (A \cdot B, U_n), \pi_{\mathsf{prod}})$ which means that $\mathsf{VC.Ver}(\mathsf{pp}, C, I, \boldsymbol{y}, \pi_I) = 1$.

As one can see, the expected running time of $\mathcal{E}$ is the (expected) time to obtain a successful execution of the protocol plus the running time to obtain $\boldsymbol{y}$ plus the running time of $\mathcal{E}'$. To obtain $\boldsymbol{y}$ it will need to make $|I|$ divisibility checks which takes time $\tilde{O}(|I|)$ plus $|I|$ calls of $\mathsf{PrimeGen}$, which takes $\mathsf{poly}(\lambda)$ time. So overall the expected time is $\frac{1}{\epsilon} + t_{\mathcal{E}'} + \tilde{O}(|I|) + \mathsf{poly}(\lambda) = \mathsf{poly}(\lambda)$. □

Non-interactive $\mathsf{PoKOpen}$**.** A non-interactive version of the protocol $\mathsf{PoKOpen}$ after applying the generalized Fiat-Shamir transform [BCS16] is shortly presented below:

$\mathsf{PoKOpen.P}(\mathsf{crs}, (C, I), (\boldsymbol{y}, \pi_I)) \to \pi$**:** Parse $\mathsf{crs} := (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen}, U_n)$, $C := (\{A, B\}, \pi_{\mathsf{prod}})$, $\pi_I := (\mathit{\Gamma}_I, \mathit{\Delta}_I)$. Compute $(a_I, b_I) \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y})$ and then $u_I \leftarrow \mathsf{PrimeProd}(I)$ and $U_I \leftarrow g^{u_I}$. Finally compute a proof $\pi_{\mathsf{PoProd}^*} \leftarrow \mathsf{PoProd}^*.\mathsf{P}((\mathbb{G}, g), (A, B, U_I, \mathit{\Gamma}_I, \mathit{\Delta}_I), (a_I, b_I))$.
Return $\pi \leftarrow (\mathit{\Gamma}_I, \mathit{\Delta}_I, \pi_{\mathsf{PoProd}^*})$

$\mathsf{PoKOpen.V}(\mathsf{crs}, (C, I), \pi_{\mathsf{PoProd}^*}) \to b$**:** Parse $\mathsf{crs} := (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen}, U_n)$, $C := (\{A, B\}, \pi_{\mathsf{prod}})$ and $\pi := (\mathit{\Gamma}_I, \mathit{\Delta}_I, \pi_{\mathsf{PoProd}^*})$. Compute $u_I \leftarrow \mathsf{PrimeProd}(I)$ and then $U_I \leftarrow g^{u_I}$.
Return 1 if both $\mathsf{PoProd}_2.\mathsf{V}(\mathsf{crs}, (A \cdot B, U_n), \pi_{\mathsf{prod}})$ and
$\mathsf{PoProd}^*.\mathsf{V}((\mathbb{G}, g), (A, B, \mathit{\Gamma}_I, \mathit{\Delta}_I, U_I), \pi_{\mathsf{PoProd}^*})$ output 1, and 0 otherwise.

Remark 4.1 (Achieving sub-linear verification time). For ease of exposition we presented the case of $k = 1$ in the above. For the case of arbitrary k one should prove knowledge of (a_{Ij}, b_{Ij}) such that $\bigwedge_{j=1}^k \mathit{\Gamma}_{Ij}^{a_{Ij}} = A_j \bigwedge_{j=1}^k \mathit{\Delta}_{Ij}^{b_{Ij}} = B \wedge g^{a_{Ij} \cdot b_{Ij}} = U_I$, where $U_I \leftarrow g^{u_I}$ and $u_I \leftarrow \mathsf{PrimeProd}(I)$. Using the same technique as above the size of the AoK is $O(k)$ (as is the commitment and the opening proof). However, since the U_I is the same for each j, the verification is done in $O(|I|/k + \lambda \cdot k)$ time. Interestingly, if $k = \sqrt{|I|}$ the verification time gets $O(\sqrt{|I|})$, which is sublinear in the size of the opening. Essentially, in cases where the opening queries are (approximately) fixed, one can trade a larger commitment size $O(\sqrt{|I|})$ in order to achieve an argument of knowledge of subvectors that has sublinear size and sublinear verification time $O(\sqrt{|I|})$.

Applications to Compact Proofs of Storage. We observe that the protocol $\mathsf{PoKOpen}$ for our VC immediately implies a *keyless* proof of storage, or more precisely a proof of retrievable commitment (PoRC) [Fis18] with non-black-box extraction. In a nutshell, a PoRC is a proof of retrievability [JK07] of a committed file. In [Fis18] Fisch defines PoRC and proposes a construction based on vector commitments – called VC-PoRC – which abstracts away a classical proof of retrievability based on Merkle trees. A bit more in detail, in the VC-PoRC scheme the prover uses a VC to commit to a file (seen as a vector of blocks); then at every audit the verifier chooses a challenge by picking a set of λ_{pos} randomly chosen positions $I = \{i \leftarrow\!\!\$ \, [n]\}$, and the prover responds by sending the subvector $\boldsymbol{v}_I$ and an opening π_I. Here λ_{pos} is a statistical parameter that governs the probability of catching an adversary that deletes (or corrupts) a fraction of the file. For example, if the file is first encoded using an erasure code with constant rate μ (i.e., one where a μ-fraction of blocks suffices to decode and such that the encoded file has size roughly $\mu^{-1} \cdot |F|$), then an erasing adversary has probability at most $\mu^{\lambda_{\mathsf{pos}}}$ of passing an audit.

Our PoRC scheme is obtained by modifying the VC-PoRC of [Fis18] in such a way that the VC opening is replaced by a $\mathsf{PoKOpen}$ AoK. This change saves the cost of sending the λ_{pos} vector values, which gives us *proofs of fixed size*, 7 elements of $\mathbb{G}$ and 2 values of $\mathbb{Z}_{2^\lambda}$. As drawback, our scheme is not black-box extractable; strictly speaking, this means it is not a PoR in the sense of [JK07] since the extractor does not exist in the real world.[15]

We note that another solution with fixed-size proofs can be achieved by using a SNARK to prove knowledge of the VC openings so that the VC-PoRC verifier would accept. For the Merkle tree VC, this means proving knowledge of λ_{pos} Merkle tree openings, which amounts to proving correctness of about $\lambda_{\mathsf{pos}} \log n$ hash computations. On a file of 2^{20} bits with 128 spot-checks, this solution would reduce proof size from 80KB to less than 1KB. But its concrete proving costs are high (more than 20 minutes and hundreds of GB of RAM).

In contrast we can estimate our AoK to be generated in less than 20 seconds and of size roughly 2KB.

Since our PoRC scheme is a straightforward modification of Fisch's VC-PoRC construction, a complete description is omitted. We stress that our technical contribution here is the design of the AoK.

Finally, we note that we can apply the observation of the previous remark in order to also achieve verification time sub-linear in the size $|I|$ of the challenged subvector at the expense of slightly larger commitments (of size $\sqrt{|I|}$).

4.3 An AoK for commitments with common subvector

We note that a simple AND composition of two $\mathsf{PoKOpen}$ arguments of knowledge on two different vector commitments can serve as a protocol proving knowledge of a common subvector of the two vectors committed. More specifically given two vector commitments, C_1, C_2 on two different vector $\boldsymbol{v}_1, \boldsymbol{v}_2$ respectively, one can prove knowledge of a common subvector $\boldsymbol{v}_I$ with a succinct (constant sized) argument without having to send the actual subvector. The two commitments should share the same CRS $\mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M})$ though they can have distinct specialized CRSs crs_{n_1} and

[15] The notion of PoR with non-black-box extractability is close to that of robust proof of data possession [ABC+07, ABC+11].

crs_{n_2} respectively (i.e., $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ may have different length). The underlying relation is:

$$R_{\mathsf{PoKComSub}} = \{(\,(C_1, C_2, I), (\boldsymbol{v}_I, \pi_{I,1}, \pi_{I,2})\,) : \mathsf{VC.Ver}^{\star}(\mathsf{crs}_{n_1}, C_1, I, \boldsymbol{v}_I, \pi_{I,1}) = 1 \\ \wedge\ \mathsf{VC.Ver}^{\star}(\mathsf{crs}_{n_2}, C_2, I, \boldsymbol{v}_I, \pi_{I,2}) = 1\}$$

As mentioned above, it is straightforward to show that an AND composition of PoKOpen on different vector commitments C_1 and C_2 is a protocol for the above relation. That is the prover, holding $\pi_{I,1} := (\Gamma_{I,1}, \Delta_{I,1})$ and $\pi_{I,2} := (\Gamma_{I,2}, \Delta_{I,2})$, first sends $\pi_{I,1}, \pi_{I,2}$ to the verifier and then provides an argument of knowledge of (a_I, b_I) such that $\Gamma_{I,1}^{a_I} = A_1 \wedge \Delta_{I,1}^{b_I} = B_1 \wedge g^{a_I \cdot b_I} = U_I \wedge \Gamma_{I,2}^{a_I} = A_2 \wedge \Delta_{I,2}^{b_I} = B_2$, where $U_I \leftarrow g^{u_I}$ and $u_I \leftarrow \mathsf{PrimeProd}(I)$.

4.4 A Succinct AoK for Commitment on Subvector

Here we present a protocol which succinctly proves that a commitment C' opens to an I-subvector $\boldsymbol{v}_I$ of the opening $\boldsymbol{v}$ of another commitment C. Since C' is a vector commitment $\boldsymbol{v}_I$ should be a normal vector instead of a general subvector, i.e. I should be a set of consecutive positions starting from 1, $I = \{1, \dots, n'\}$ for some $n' \in \mathbb{N}$. We note though that both commitments should share the same crs (but not the same specialized CRS). Below is the relation of the AoK that is parametrized by the two specialized CRSs $\mathsf{crs}_n \leftarrow \mathsf{VC.Specialize}(\mathsf{crs}, n)$ and $\mathsf{crs}_{n'} \leftarrow \mathsf{VC.Specialize}(\mathsf{crs}, n')$ where $\mathsf{crs} \leftarrow \mathsf{VC.Setup}(1^\lambda, \mathcal{M})$ is common.

$$R_{\mathsf{PoKSubV}} = \{(\,(C, C', I), (\boldsymbol{v}_I, \pi_I, \pi'_I)\,) : \mathsf{VC.Ver}^{\star}(\mathsf{crs}_n, C, I, \boldsymbol{v}_I, \pi_I) = 1 \\ \wedge\ \mathsf{VC.Ver}^{\star}(\mathsf{crs}_{n'}, C', I, \boldsymbol{v}_I, \pi'_I) = 1 \wedge |\boldsymbol{v}_I| = n'\}$$

The idea of our protocol is that since the opening $\boldsymbol{v}_I$ is the I-subvector of $\boldsymbol{v}$ one can provide a succinct proof of knowledge of the opening at these positions using the PoKOpen protocol presented above. However this is not enough as one should bind the opening proof with C'. This concretely can happen if one embeds a proof of product for the two components, A' and B', of C' inside the proof of opening. More specifically the prover provides an opening proof $\pi_I := (\Gamma_I, \Delta_I)$ then computes $(a_I, b_I) \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{v}_I)$ and proves that $g_0^{a_I} = A' \wedge g_1^{b_I} = B' \wedge U_{n'} = g^{a_I \cdot b_I} \wedge \Gamma_I^{a_I} = A \wedge \Delta_I^{b_I} = B$. Notice that the last three equalities correspond to the proof of opening protocol and the first three to the proof of product. So a conjunction of PoKOpen and PoProd protocol is sufficient. Lastly g, g_0, g_1 and $U_{n'}$ are part of $\mathsf{crs}_{n'}$ and (A, B), (A', B') part of the C and C' commitments respectively.

We state the following theorem for the security of the protocol above.

Theorem 4.3. *If* PoProd* *and* PoKOpen *are succinct arguments of knowledge for* R_{PoProd^*} *and* R_{PoKOpen}*, then protocol* PoKSubV *in Fig. 3 is a succinct argument of knowledge for relation* R_{PoKSubV} *with respect to algorithm* VC.Ver *of our construction of Section 3.2.*

The intuition of the proof is that one proves knowledge of an opening I for C, namely that $\boldsymbol{v}_I$ is an I-subvector of C, where $(a_I, b_I) \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{v}_I)$, with a normal proof of subvector opening. This is equivalent to $\mathsf{VC.Ver}^{\star}(\mathsf{crs}_n, C, I, \boldsymbol{v}_I, \pi_I) = 1$. Then in the same proof proves that the accumulators of C' are composed by the same (a_I, b_I) which results to proving that C' commits to $\boldsymbol{v}_I$. The last point is equivalent to $\mathsf{VC.Ver}^{\star}(\mathsf{crs}_{n'}, C', I, \boldsymbol{v}_I, \pi'_I) = 1 \wedge |\boldsymbol{v}_I| = n'$.

Fig. 3. PoKSubV protocol

Prover input: $((\mathsf{crs}_n, \mathsf{crs}_{n'}), (C, C', I), (\boldsymbol{v}_I, \pi_I))$. Verifier input: $((\mathsf{crs}_n, \mathsf{crs}_{n'}), (C, C', I))$.

$\mathsf{P} \rightarrow \mathsf{V}$: $\pi_I := (\Gamma_I, \Delta_I)$

A conjuction of PoProd^* and $\mathsf{PoKOpen}$ protocols between $\mathsf{P}(\mathsf{crs}_n, \mathsf{crs}_{n'}, (C, C', I), (\boldsymbol{v}_I, \pi_I))$ and $\mathsf{V}(\mathsf{crs}_n, \mathsf{crs}_{n'}, (C, C', I))$ is executed:

$\mathsf{V} \rightarrow \mathsf{P}$: $h \leftarrow\$ \mathbb{G}$
$\mathsf{P} \rightarrow \mathsf{V}$: $z := (z_a, z_b)$ computed as $z_a \leftarrow h^{a_I}, z_b \leftarrow h^{b_I}$
$\mathsf{V} \rightarrow \mathsf{P}$: $\ell \leftarrow\$ \mathsf{Primes}(\lambda)$ and $\alpha \leftarrow\$ [0, 2^{\lambda})$
$\mathsf{P} \rightarrow \mathsf{V}$: $\pi := ((Q_A, Q_B, Q'_A, Q'_B, Q_C), r_a, r_b)$ computed as follows
- $(q_a, q_b, q_{ab}) \leftarrow (\lfloor a_I/\ell \rfloor, \lfloor b_I/\ell \rfloor, \lfloor a_I b_I/\ell \rfloor)$
- $(r_a, r_b) \leftarrow (a_I \mod \ell, b_I \mod \ell)$
- $(Q_{A'}, Q_{B'}, Q_A, Q_B, Q_C) := (g_0^{q_a}, g_1^{q_b}, \Gamma_I^{q_a} h^{\alpha q_a}, \Delta_I^{q_b} h^{\alpha q_b}, g^{q_{ab}})$

V: Parse $\mathsf{crs}_n := (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen}, U_n)$, $\mathsf{crs}_{n'} := (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen}, U_{n'})$ and $C := (\{A, B\}, \pi_{\mathsf{prod}})$.
- Compute $r_c \leftarrow r_a \cdot r_b \mod \ell$
- Output 1 iff $r_a, r_b \in [\ell] \wedge Q_{A'}^{\ell} g_0^{r_a} = A' \wedge Q_{B'}^{\ell} g_1^{r_b} = B' \wedge Q_A^{\ell} \Gamma_I^{r_a} h^{\alpha r_a} = A z_a^{\alpha} \wedge Q_B^{\ell} \Delta_I^{r_b} h^{\alpha r_b} = B z_b^{\alpha} \wedge Q_C^{\ell} g^{r_c} = U_{n'} \wedge \mathsf{PoProd}_2.\mathsf{V}(\mathsf{pp}, (A \cdot B, U_n), \pi_{\mathsf{prod}})$

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). The main goal of client nodes is to retrieve some blocks (i.e., a portion) of a given file. The role of a storage node is instead to store a portion of a file (or more files) and to answer to the retrieval queries of clients that are relevant to the portion it stores. In terms of security, VDS guarantees that malicious storage nodes cannot send to the clients blocks of the file that have been tampered with.

We refer the reader to Section 1.2 for a discussion on the motivation and requirements of VDS.

In Table 2 we summarize the main roles/capabilities of VDS nodes.

All Participating Nodes	
	Storage Nodes
Store current digest.	**Store** a portion of the file.
Can retrieve blocks of the file and verify responses.	**Can answer and certify** retrievals of subportions.
Can aggregate proofs they received.	**Can produce and publish updates** to their view.
Can update the digest following updates from other nodes	**Can apply updates** from other nodes efficiently.

Table 2. Roles in a decentralized verifiable database.

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 $\mathsf{Bootstrap}$

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). We start by describing these algorithms and then define their correctness properties.

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.

Definition 5.1 (Verifiable Decentralized Storage).

Algorithm to bootstrap the system:

$\mathsf{Bootstrap}(1^\lambda) \to (\mathsf{pp}, \delta_0, \mathsf{st}_0)$ *Given the security parameter λ, the probabilistic bootstrap algorithm outputs public parameters* pp*, initial digest δ_0 and state* st_0*. δ_0 and* st_0 *correspond to the digest and storage node's local state respectively for an empty file.*
All the algorithms below implicitly take as input the public parameters pp.

The algorithms for storage nodes are:

$\mathsf{StrgNode.AddStorage}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, Q, \mathsf{F}_Q, \pi_Q) \to (\mathsf{st}', J, \mathsf{F}_J)$ *This algorithm allows a storage node to add more blocks of a given file* F *to its local storage. Its first inputs are the local view of the storage node that is defined by a digest δ, a length n, a state* st*, and a file portion (I, F_I). Then it takes as input a file subportion (Q, F_Q) together with a valid retrieval certificate π_Q. The output is an updated view of the storage node, that is a new state* st' *and file portion $(J, \mathsf{F}_J) := (I, \mathsf{F}_I) \cup (Q, \mathsf{F}_Q)$. Note that this algorithm can be used to enable anyone who holds a valid retrieval certificate for a file portion F_Q to become a storage node of such portion.*

$\mathsf{StrgNode.RmvStorage}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, K) \to (\mathsf{st}', J, \mathsf{F}_J)$ *This algorithm allows a storage node to remove blocks of a given file* F *from its local storage. Its first inputs are the local view of the storage node that is defined by a digest δ, a length n, a state* st*, and a file portion (I, F_I). Then it takes as input a set of positions $K \subseteq I$, and the output is an updated view of the storage node, that is a new state* st' *and file portion $(J, \mathsf{F}_J) := (I, \mathsf{F}_I) \setminus (K, \cdot)$.*

$\mathsf{StrgNode.CreateFrom}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, J) \to (\delta', n', \mathsf{st}', J, \mathsf{F}_J, \Upsilon_J)$ *This algorithm allows a storage node for a file subportion F_I to create a new file containing only a subset F_J of F_I along with the corresponding digest δ' and length n' and a certificate vouching for the correctness of this operation. The algorithm takes as input the local view of the storage node, i.e., digest δ, length n, local state* st *and file portion (I, F_I), and a set of indices $J \subseteq I$. The algorithm returns a new digest δ', length n', a local state* st'*, a file portion (J, F_J) and an advice Υ. This advice can be used by a client holding only the former digest δ to obtain the new digest δ', by using the* $\mathsf{ClntNode.GetCreate}$ *algorithm described below.*

$\mathsf{StrgNode.PushUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \Delta) \to (\delta', n', \mathsf{st}', J, \mathsf{F}'_J, \Upsilon_\Delta)$ *This algorithm allow a node storing a file subportion F_I to perform an update on the file and to generate a corresponding digest, length and local view, along with a proof vouching for the correctness of the update. The inputs include the local view of the storage node, i.e., digest δ, length n, local state* st *and file portion (I, F_I), an update operation $\mathsf{op} \in \{\mathsf{mod}, \mathsf{add}, \mathsf{del}\}$ and an update description Δ. The outputs are a new digest δ' and length n', a new local state* st'*, an updated file portion (J, F'_J) and an update hint Υ_Δ. If $\mathsf{op} = \mathsf{mod}$, then Δ contains a file portion (K, F'_K) such that $K \subseteq I$ and F'_K represents the new content to be written in positions K. If $\mathsf{op} = \mathsf{add}$, it is also $\Delta = (K, \mathsf{F}'_K)$ except that K is a set of new (sequential) positions $K \cap I = \emptyset$ that start from $n+1$ (and end to $n+|K|$). If $\mathsf{op} = \mathsf{del}$, then Δ only contains a set of positions $K \subseteq I$, which are the ones to be deleted (and are ought to* *be the $|K|$ last sequential positions). The proof Υ_Δ can be used by client nodes holding δ in order to check the validity of the new digest δ', and by other storage nodes, holding additionally the length n, in order to check the validity of the changes and to update their local views accordingly.*

$\mathsf{StrgNode.ApplyUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \Delta, \Upsilon_\Delta) \to (b, \delta', n', \mathsf{st}', J, \mathsf{F}'_J)$ *This algorithm allows a storage node to incorporate changes in a file pushed by another node. The inputs include the local view of the storage node, i.e., digest δ, length n, local state* st *and file portion (I, F_I), an update operation* $\mathsf{op} \in \{\mathsf{mod}, \mathsf{add}, \mathsf{del}\}$, *an update description Δ and an update hint Υ_Δ. The algorithm returns a bit b (to accept/reject the update) and (if $b = 1$) a new digest δ', a new length n', a new (local) state* st' *and an updated file subportion (J, F'_J). If* $\mathsf{op} \in \{\mathsf{mod}, \mathsf{add}\}$ *we have that $J = I$, i.e., the node keeps storing the same indices; if* $\mathsf{op} = \mathsf{del}$ *then J is I minus the deleted indices.*

$\mathsf{StrgNode.Retrieve}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, Q) \to (\mathsf{F}_Q, \pi_Q)$ *This algorithm allows a storage node to answer a retrieval query for blocks with indices in Q and to create a certificate vouching for the correctness of the returned blocks. The inputs include the local view of the storage node, i.e., digest δ, length n local state* st *and file portion (I, F_I), and a set of indices Q. The output is a file portion F_Q and a retrieval certificate π_Q.*

The algorithms for clients nodes are:

$\mathsf{ClntNode.GetCreate}(\delta, J, \Upsilon_J) \to (b, \delta')$ *On input a digest δ, a set of indices J and a creation advice Υ_J, this algorithm returns a bit b (to accept/reject) and (if $b = 1$) a new digest δ' that corresponds to a file F' that is the prefix with indices J of the file represented by digest δ.*

$\mathsf{ClntNode.ApplyUpdate}(\delta, \mathsf{op}, \Delta, \Upsilon_\Delta) \to (b, \delta')$ *On input a digest δ, an update operation* op *such that* $\mathsf{op} \in \{\mathsf{mod}, \mathsf{add}, \mathsf{del}\}$, *an update description Δ and an update hint Υ_Δ, it returns a bit b (to accept/reject update) and (if $b = 1$) a new digest δ'.*

$\mathsf{ClntNode.VerRetrieve}(\delta, Q, \mathsf{F}_Q, \pi_Q) \to b$ *On input a digest δ, a file portion (Q, F_Q) and a certificate π_Q, this algorithm accepts (i.e. it outputs 1) only if π_Q is a valid proof that δ corresponds to a file F with length n of which F_Q is the portion corresponding to indices Q.*

$\mathsf{AggregateCertificates}(\delta, (I, \mathsf{F}_I, \pi_I), (J, \mathsf{F}_J, \pi_J)) \to \pi_K$ *On input a digest δ and two certificated retrieval outputs (I, F_I, π_I) and (J, F_J, π_J), this algorithm aggregates their certificates into a single certificate π_K (with $K := I \cup J$). In a running VDS system, this algorithm can be used by any node to aggregate two (or more) incoming certified data blocks into a single certified data block.*

Remark 5.1. Our notion of VDS is close to the notion of updatable VCs [CF13] and it is not hard to see that a simpler version of VDS does imply updatable VCs. On the other hand, the VDS notion makes more clear 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.

5.2 Correctness and Efficiency

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 for storage nodes.

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 then 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 $\mathsf{StrgNode.PushUpdate}$ "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 definition correctness 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* $\mathsf{Bootstrap}$. *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* (op, Δ) *is* admissible *for* (n, I, F_I) *if:*

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

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* (op, Δ) *that is admissible for* (n, I, F_I) *and for any tuple* $(\delta', n', \mathsf{st}', J, \mathsf{F}'_J, \Upsilon_\Delta) \leftarrow \mathsf{StrgNode.PushUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \Delta)$*:*

1. $(\mathsf{pp}, \delta', n', \mathsf{st}', J, \mathsf{F}'_J)$ *is valid;*

2. *for valid* $(\delta, n, \mathsf{st}_s, I_s, \mathsf{F}_{I_s})$, $(b_s, \delta'_s, n', \mathsf{st}'_s, I'_s, \mathsf{F}'_s) \leftarrow \mathsf{StrgNode.ApplyUpdate}(\delta, n, \mathsf{st}_s, I_s, \mathsf{F}_{I_s}, \mathsf{op}, \Delta, \Upsilon_\Delta)$ *implies* $b_s = 1$, $\delta'_s = \delta'$, $n'_s = n'$, *and* $(\delta'_s, n'_s, \mathsf{st}'_s, I'_s, \mathsf{F}'_s)$ *is valid;*
3. *if* $(b_c, \delta'_c) \leftarrow \mathsf{ClntNode.ApplyUpdate}(\delta, \mathsf{op}, \Delta, \Upsilon_\Delta)$, *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* $(\delta', n', \mathsf{st}', J, \mathsf{F}_J, \Upsilon_J)$ *is output of* $\mathsf{StrgNode.CreateFrom}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, J)$ *and* $(b, \delta'') \leftarrow \mathsf{ClntNode.GetCreate}(\delta, J, \Upsilon_J)$, *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* $\mathsf{ClntNode.VerRetrieve}(\delta, K, \mathsf{F}_K, \pi_K) = 1$.

5.3 Security

In this section we define the security of VDS schemes. 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 $\mathsf{F}^*_Q \neq \mathsf{F}_Q$.

Definition 5.5 (History for Decentralized Storage). *Let* VDS *be a verifiable decentralized storage scheme. A history for* VDS *is a sequence* $\mathcal{H} = (\mathsf{op}^i, \Delta^i, \Upsilon^i_\Delta)_{i \in [\ell]}$ *of tuples, where* op^i *is either in* $\{\mathsf{mod}, \mathsf{add}, \mathsf{del}\}$ *(i.e., it is an update of the file), or* $\mathsf{op}^i = \mathsf{cfrom}$ *(i.e., it is the creation of a new file related to the current one), in which case* Δ^i *is a set of indices. In order to define valid histories we define the function* $\mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H})$ *as follows*

$\mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H})$

```
F_0 ← ∅; b ← 1
for i ∈ [ℓ]
   F_i ← FileChange(F_{i−1}, op^i, Δ^i)
   if op^i ∈ {mod, add, del} then
      (b_i, δ_i) ← ClntNode.ApplyUpdate(δ_{i−1}, op^i, Δ^i, Υ^i_Δ)
   elseif op^i = cfrom then
      (b_i, δ_i) ← ClntNode.GetCreate(δ_{i−1}, Δ^i, Υ^i_Δ)
   endif
   b ← b ∧ b_i
endfor
return (b, δ_ℓ, F_ℓ)
```

$\mathsf{FileChange}(\mathsf{F}, \mathsf{op}, \Delta)$

if $\mathsf{op} \in \{\mathsf{mod}, \mathsf{add}\}$ *parse* $\Delta = (K, \mathsf{F}'_K)$
$\quad \forall i \in K : \mathsf{F}^*_i \leftarrow \mathsf{F}'_i; \forall i \in [|\mathsf{F}|] \setminus K : \mathsf{F}^*_i \leftarrow \mathsf{F}_i,$
elseif $\mathsf{op} = \mathsf{del}$ *parse* $\Delta = K$
$\quad \forall i \in [|\mathsf{F}|] \setminus K : \mathsf{F}^*_i \leftarrow \mathsf{F}_i,$
elseif $\mathsf{op} = \mathsf{cfrom}$ *parse* $\Delta = K$
$\quad \forall i \in K : \mathsf{F}^*_i \leftarrow \mathsf{F}_i,$
endif return F^*

We say that a history $\mathcal{H}$ *is valid w.r.t. public parameters* pp *and initial digest* δ_0 *and state* st_0 *if* $\mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H})$ *returns bit* $b = 1$*.*

Definition 5.6 (Security for Verifiable Decentralized Storage). *Consider the experiment* $\mathsf{VDS\text{-}Security}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda)$ *below. Then we say that a VDS scheme* VDS *is secure if for all PPT* $\mathcal{A}$ *we have* $\Pr[\mathsf{VDS\text{-}Security}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda) = 1] \in \mathsf{negl}(\lambda)$*.*

$\mathsf{VDS\text{-}Security}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda)$

$(\mathsf{pp}, \delta_0, \mathsf{st}_0) \leftarrow \mathsf{Bootstrap}(1^\lambda)$
$(\mathcal{H}, Q, \mathsf{F}^*_Q, \pi^*) \leftarrow \mathcal{A}(\mathsf{pp}, \delta_0, \mathsf{st}_0)$
$(b, \delta, \mathsf{F}) \leftarrow \mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H})$
$b \leftarrow b \wedge \mathsf{F}^*_Q \neq \mathsf{F}_Q \wedge$
$\quad \mathsf{ClntNode.VerRetrieve}(\mathsf{pp}, \delta, Q, \mathsf{F}^*_Q, \pi^*)$
return b

5.4 A VDS Construction in Hidden-Order Groups

We describe a construction of VDS in hidden-order groups. We build it by extending the techniques used to construct our VC scheme from Section 3.2. In particular, we start from a modified version of our VC that achieves a weaker position binding property (in which the adversary reveals the full vector, yet its goal is to find two distinct openings for the same position) and then show how to make this scheme dynamic (i.e., to change vector values or its length) and fully distributed (i.e., updates can be performed without knowing the entire vector).

Preliminaries. We begin by describing the simplified version of our VC, considering the case of $k = 1$, which fits best our VDS construction, regarding efficiency and communication complexity. For convenience of the reader we describe again shortly the algorithms and functions (and variations of them) from sections 3.2 and 3.4 that are used in the scheme (for more details we refer to the corresponding section):

- $\mathsf{PrimeGen}$, a deterministic collision resistant function that maps integers to primes.
- $\mathsf{PartndPrimeProd}(I, \boldsymbol{y}) \rightarrow (a_I, b_I)$: given a set of indices $I = \{i_1, \ldots, i_m\} \subseteq [n]$ and a vector $\boldsymbol{y} \in \mathcal{M}^m$, the function computes $(a_I, b_I) := \left(\prod_{l=1:y_l=0}^{m} p_{i_l}, \prod_{l=1:y_l=1}^{m} p_{i_l}\right)$, where $p_i \leftarrow \mathsf{PrimeGen}(i)$ for all $i \in \mathbb{N}$.

$\mathsf{VC.Setup}(1^\lambda, \{0,1\}^k) \rightarrow \mathsf{crs} := (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen})$.

$\mathsf{VC.Com}'(\mathsf{crs}, \boldsymbol{v}) \rightarrow C$ compute $(a, b) \leftarrow \mathsf{PartndPrimeProd}([n], \boldsymbol{v})$, where $n \leftarrow |\boldsymbol{v}|$; next compute $A = g_0^a$ and $B = g_1^b$. Return $C := (C^\star, n) := ((A, B), |\boldsymbol{v}|)$.

$\mathsf{VC.Ver'}(\mathsf{crs}, C, I, \boldsymbol{y}, \pi_I) \to b$ compute $(a_I, b_I) \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{y})$, and then parse $\pi_I := (\Gamma_I, \Delta_I)$ and return $b \leftarrow (\Gamma_I^{a_I} = A) \wedge (\Delta_I^{b_I} = B)$.

$\mathsf{VC.Disagg'}(\mathsf{crs}, I, \boldsymbol{v}_I, \pi_I, K) \to \pi_K$ let $L := I \setminus K$, and $\boldsymbol{v}_L$ be the subvector of $\boldsymbol{v}_I$ at positions in L. Then compute $a_L, b_L \leftarrow \mathsf{PartndPrimeProd}(L, \boldsymbol{v}_L)$ parse $\pi_I := (\Gamma_I, \Delta_I)$ and set $(\Gamma_K, \Delta_K) \leftarrow (\Gamma_I^{a_L}, \Delta_I^{b_L})$. Return $\pi_K \leftarrow (\Gamma_K, \Delta_K)$.

$\mathsf{VC.Agg'}(\mathsf{crs}, (I, \boldsymbol{v}_I, \pi_I), (J, \boldsymbol{v}_J, \pi_J)) \to \pi_K$:

1. Let $L := I \cap J$. If $L \neq \emptyset$, set $I' := I \setminus L$ and compute $\pi_{I'} \leftarrow \mathsf{VC.Disagg}(\mathsf{crs}, I, \boldsymbol{v}_I, \pi_I, I')$; otherwise let $\pi_{I'} = \pi_I$.
2. Compute $(a_{I'}, b_{I'}) \leftarrow \mathsf{PartndPrimeProd}(I, \boldsymbol{v}_{I'})$ and $\{a_J, b_J\} \leftarrow \mathsf{PartndPrimeProd}(J, \boldsymbol{v}_J)$.
3. Parse $\pi_{I'} := (\Gamma_{I'}, \Delta_{I'})$, $\pi_J := (\Gamma_J, \Delta_J)$ and compute $\Gamma_K \leftarrow \mathbf{ShamirTrick}(\Gamma_{I'}, \Gamma_J, a_{I'}, a_J)$ and $\Delta_K \leftarrow \mathbf{ShamirTrick}(\Delta_{I'}, \Delta_J, b_{I'}, b_J)$
4. Return $\pi_K \leftarrow (\Gamma_K, \Delta_K)$

Finally, let $\mathsf{PoKSubV'}$ be the same protocol as in section 4 but adjusted according to the above algorithms. That is the CRS of is simply crs instead of the two specialized CRSs. Furthermore, since C is not accompanied with PoProd_2 the verifier does not have to check the validity of it. The rest of the protocol remains the same and the underlying relation is:

$$R_{\mathsf{PoKSubV'}} = \{((C, C', I), (\boldsymbol{v}_I, \pi_I, \pi'_I)) : \mathsf{VC.Ver'}(\mathsf{crs}, C, I, \boldsymbol{v}_I, \pi_I) = 1 \\ \wedge \mathsf{VC.Ver'}(\mathsf{crs}, C', I, \boldsymbol{v}_I, \pi'_I) = 1 \wedge |\boldsymbol{v}_I| = n'\}$$

We note that for simplicity in the following we abuse the notation for Shamir's trick by writing e.g. $(\Gamma'_I, \Delta'_I) \leftarrow \big(\mathbf{ShamirTrick}(\Gamma_I, \Gamma_K, \mathsf{F}_I, \mathsf{F}_K)^{a'_K}\,,\ \mathbf{ShamirTrick}(\Delta_I, \Delta_K, \mathsf{F}_I, \mathsf{F}_K)^{b'_K}\big)$ instead of the more precise $(\Gamma'_I, \Delta'_I) \leftarrow \big(\mathbf{ShamirTrick}(\Gamma_I, \Gamma_K, a_I, a_K)^{a'_K}\,,\ \mathbf{ShamirTrick}(\Delta_I, \Delta_K, b_I, b_K)^{b'_K}\big)$.

Our VDS Scheme. The algorithms of the VDS scheme are the following:[16]

$\mathsf{Bootstrap}(1^\lambda) \to (\mathsf{pp}, \delta_0, n_0, \mathsf{st}_0)$ Execute $\mathsf{VC.Setup}(1^\lambda, \{0,1\}^k)$ and get $\mathsf{pp} := (\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen})$. Set $n_0 \leftarrow 0$, $\delta_0 \leftarrow ((g_0, g_1), n_0)$ and $\mathsf{st}_0 \leftarrow (g_0, g_1)$.

The algorithms for storage nodes are:

$\mathsf{StrgNode.AddStorage}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, Q, \mathsf{F}_Q, \pi_Q) \to (\mathsf{st}', J, \mathsf{F}_J)$ If $I = \emptyset$ then set $\mathsf{st}' \leftarrow \pi_Q$, otherwise $\mathsf{st} := \pi_I$. Then compute $\mathsf{st}' \leftarrow \mathsf{VC.Agg'}(\mathsf{pp}, (I, \mathsf{F}_I, \pi_I), (Q, \mathsf{F}_Q, \pi_Q))$. The computation of J and F_J is straightforward: $(J, \mathsf{F}_J) \leftarrow (I \cup Q, \mathsf{F}_I \cup \mathsf{F}_Q)$.

$\mathsf{StrgNode.RmvStorage}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, K) \to (\mathsf{st}', J, \mathsf{F}_J)$ Compute $J \leftarrow I \setminus K$ and the corresponding F_J. Then $\pi_J \leftarrow \mathsf{VC.Disagg'}(\mathsf{pp}, I, \mathsf{F}_I, \pi_I, J)$ and set $\mathsf{st}' \leftarrow \pi_J$.

$\mathsf{StrgNode.CreateFrom}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, J) \to (\delta', n', \mathsf{st}', J, \mathsf{F}_J, \Upsilon_J)$ The new digest δ' of F_J is computed with the commitment algorithm $\delta' \leftarrow \mathsf{VC.Com'}(\mathsf{pp}, \mathsf{F}_J)$. The new length gets $n' \leftarrow |J|$. The previous local state is $\mathsf{st} = \pi_I$ and the new local state gets $\mathsf{st}' \leftarrow \mathsf{VC.Disagg}(\mathsf{pp}, I, \mathsf{F}_I, \pi_I, J)$. Finally, for Υ_J it computes an argument of knowledge of subvector (see section 4), $\pi_{\mathsf{PoKSubV'}} \leftarrow \mathsf{PoKSubV'.P}(\mathsf{pp}, (\delta, \delta', J), (\boldsymbol{v}_J, \pi_I))$ and sets $\Upsilon_J \leftarrow (\delta', \pi_{\mathsf{PoKSubV'}})$.

$\mathsf{StrgNode.PushUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \Delta) \to (\delta', n', \mathsf{st}', J, \mathsf{F}'_J, \Upsilon_\Delta)$ The algorithm works according to the type of update operation op:

[16] Since the scheme has several parts in common with the above VC algorithms, we use those algorithms as shorthands in the description.

– $\mathsf{op} = \mathsf{mod}$: parse $\Delta := (K, \mathsf{F}'_K)$ and $\mathsf{st} := \pi_I$. Execute $\pi_K \leftarrow \mathsf{VC.Disagg}'(\mathsf{pp}, I, \mathsf{F}_I, \pi_I, K)$ and parse $\pi_K := (\Gamma_K, \Delta_K)$. Then compute $(a'_K, b'_K) \leftarrow \mathsf{PartndPrimeProd}(K, \mathsf{F}'_K)$ and set $\delta' \leftarrow ((\Gamma_K^{a'_K}, \Delta_K^{b'_K}), n)$ (i.e., $n' = n$ remains the same). st' is the new opening of I, $\pi'_I \leftarrow \pi_I$, which is the same so the local state does not change $\mathsf{st}' \leftarrow \mathsf{st}$. Since it is a modification operation $(J, \mathsf{F}'_J) \leftarrow (I, \mathsf{F}'_I)$, where F'_I is simply the modified file $\mathsf{F}'_I = (\mathsf{F}_I \setminus \mathsf{F}_K) \cup \mathsf{F}'_K$. Finally, set $\Upsilon_\Delta \leftarrow (\mathsf{F}_K, \pi_K)$.
– $\mathsf{op} = \mathsf{add}$: parse $\Delta := (K, \mathsf{F}'_K)$, $\mathsf{st} := \pi_I$, and the old digest $\delta := ((A, B), n)$. Then compute $(a'_K, b'_K) \leftarrow \mathsf{PartndPrimeProd}(K, \mathsf{F}'_K)$ and the new digest gets $\delta' \leftarrow ((A^{a'_K}, B^{b'_K}), n')$ where $n' \leftarrow n + |K|$. The new state refers to the new file subportion $(J, \mathsf{F}'_J) \leftarrow (I \cup K, \mathsf{F}_I \cup \mathsf{F}_K)$, $\mathsf{st}' := \pi'_J$, and is the same as the old one $\mathsf{st}' \leftarrow \mathsf{st}$ since $\pi_I = \pi'_J$. Finally, set $\Upsilon_\Delta \leftarrow \varnothing$.
– $\mathsf{op} = \mathsf{del}$: parse $\Delta := K$ and $\mathsf{st} := \pi_I$. Execute $\pi_K \leftarrow \mathsf{VC.Disagg}'(\mathsf{pp}, I, \mathsf{F}_I, \pi_I, K)$ and parse $\pi_K := (\Gamma_K, \Delta_K)$. Then the new digest is $\delta' \leftarrow ((\Gamma_K, \Delta_K), n')$ where $n' \leftarrow n - |K|$. The new state refers to the new file subportion $(J, \mathsf{F}'_J) \leftarrow (I \setminus K, \mathsf{F}_I \setminus \mathsf{F}_K))$ and is the same as the old one $\mathsf{st}' \leftarrow \mathsf{st}$ since $\pi_I = \pi'_J$. Finally set $\Upsilon_\Delta \leftarrow (\mathsf{F}_K, \pi_K)$.

$\mathsf{StrgNode.ApplyUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \Delta, \Upsilon_\Delta) \to (b, \delta', n', \mathsf{st}', J, \mathsf{F}'_J)$ Again, it works according to the type of update operation op:
– $\mathsf{op} = \mathsf{mod}$: parse $\Delta := (K, \mathsf{F}'_K)$, $\mathsf{st} := \pi_I$ and $\Upsilon_\Delta := (\mathsf{F}_K, \pi_K)$. Let $b \leftarrow \mathsf{VC.Ver}'(\mathsf{pp}, \delta, K, \mathsf{F}_K, \pi_K)$. Then, if $b = 1$ parse $\pi_K := (\Gamma_K, \Delta_K)$, compute $(a'_K, b'_K) \leftarrow \mathsf{PartndPrimeProd}(K, \mathsf{F}'_K)$ and set $\delta' \leftarrow ((\Gamma_K^{a'_K}, \Delta_K^{b'_K}), n')$ where $n' \leftarrow n$. It is clear that in the case of a modify operation $(J, \mathsf{F}'_J) \leftarrow (I, \mathsf{F}'_I)$, where F'_I is simply the modified file $\mathsf{F}'_I = (\mathsf{F}_I \setminus \mathsf{F}_K) \cup \mathsf{F}'_K$. For the new local state st' that we discern three cases:
  - $I \cap K = \emptyset$: then compute
    $(\Gamma'_I, \Delta'_I) \leftarrow \left(\mathbf{ShamirTrick}(\Gamma_I, \Gamma_K, \mathsf{F}_I, \mathsf{F}_K)^{a'_K}, \mathbf{ShamirTrick}(\Delta_I, \Delta_K, \mathsf{F}_I, \mathsf{F}_K)^{b'_K}\right)$ and set $\mathsf{st}' \leftarrow \pi'_I := (\Gamma'_I, \Delta'_I)$.
  - $I \cap K = K$: compute $(\Gamma'_I, \Delta'_I) \leftarrow (\Gamma_I, \Delta_I)$ and set $\mathsf{st}' \leftarrow \pi'_I := (\Gamma'_I, \Delta'_I)$.
  - For the case where neither $I \cap K = \emptyset$ nor $I \cap K = K$, i.e. $I \cap K = L \notin \{K, \emptyset\}$ we partition K as $K = L \cup \bar{L}$ and apply two sequential updates to π_I, one with L' (s.t. $I \cap \bar{L} = \emptyset$) and one with L (s.t. $I \cap L = L$). That is, compute $(a'_{\bar{L}}, b'_{\bar{L}}) \leftarrow \mathsf{PartndPrimeProd}(\bar{L}, \mathsf{F}'_{\bar{L}})$ and then
    $(\Gamma'_I, \Delta'_I) \leftarrow \left(\mathbf{ShamirTrick}(\Gamma_I, \Gamma_{\bar{L}}, \mathsf{F}_I, \mathsf{F}_{\bar{L}})^{a'_{\bar{L}}}, \mathbf{ShamirTrick}(\Delta_I, \Delta_{\bar{L}}, \mathsf{F}_I, \mathsf{F}_{\bar{L}})^{b'_{\bar{L}}}\right)$. Then let $(\Gamma''_I, \Delta''_I) \leftarrow (\Gamma'_I, \Delta'_I)$. Finally, set $\mathsf{st}' \leftarrow (\Gamma''_I, \Delta''_I)$. Essentially, since the case of $I \cap L = L$ doesn't cause any change to the state, computationally it is as a single update.
– $\mathsf{op} = \mathsf{add}$: parse $\Delta := (K, \mathsf{F}'_K)$, $\mathsf{st} := \pi_I$ and the old digest as $\delta := ((A, B), n)$. Set $b = 1$ iff $K = \{n+1, \ldots, n+|K|\}$. Then if $b = 1$ compute $(a'_K, b'_K) \leftarrow \mathsf{PartndPrimeProd}(K, \mathsf{F}'_K)$ and the new digest becomes $\delta' \leftarrow ((A^{a'_K}, B^{b'_K}), n')$ where $n' \leftarrow n + |K|$. For the new local state, first parse the old one $\mathsf{st} := \pi_I := (\Gamma_I, \Delta_I)$ and the new one gets $\mathsf{st}' \leftarrow \pi'_I$ where $\pi'_I \leftarrow (\Gamma_I^{a'_K}, \Delta_I^{b'_K})$. Finally set $(J, \mathsf{F}'_J) \leftarrow (I, \mathsf{F}_I)$, i.e., the file remains unchanged.
– $\mathsf{op} = \mathsf{del}$: parse $\Delta := K$, $\mathsf{st} := \pi_I$, and $\Upsilon_\Delta := (\mathsf{F}_K, \pi_K)$. Set $b = 1$ iff $K = \{n - |K| + 1, \ldots, n\} \wedge \mathsf{VC.Ver}'(\mathsf{pp}, \delta, K, \mathsf{F}_K, \pi_K) = 1$. Then if $b = 1$ sets $\delta' \leftarrow ((\Gamma_K, \Delta_K), n')$ where $n' \leftarrow n - |K|$. For the new local state, similarly to the modify operation, we discern three cases. If $I \cap K = \emptyset$ then $(\Gamma'_I, \Delta'_I) \leftarrow (\mathbf{ShamirTrick}(\Gamma_I, \Gamma_K, \mathsf{F}_I, \mathsf{F}_K), \mathbf{ShamirTrick}(\Delta_I, \Delta_K, \mathsf{F}_I, \mathsf{F}_K))$ and set $\mathsf{st}' \leftarrow \pi_K := (\Gamma'_I, \Delta'_I)$; else if $I \cap K = K$ $\mathsf{st}' = \mathsf{st}$, else if $I \cap K = L$ then (let $\bar{L} = K \setminus L$) $(\Gamma'_I, \Delta'_I) \leftarrow (\mathbf{ShamirTrick}(\Gamma_I, \Gamma_{\bar{L}}, \mathsf{F}_I, \mathsf{F}_{\bar{L}}), \mathbf{ShamirTrick}(\Delta_I, \Delta_{\bar{L}}, \mathsf{F}_I, \mathsf{F}_{\bar{L}}))$ and set $\mathsf{st}' \leftarrow \pi_I := (\Gamma'_I, \Delta'_I)$ (similarly to the $\mathsf{op} = \mathsf{mod}$ case). Finally $(J, \mathsf{F}'_J) \leftarrow (I \setminus L, \mathsf{F}_I \setminus \mathsf{F}_L)$.

$\mathsf{StrgNode.Retrieve}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, Q) \to (\mathsf{F}_Q, \pi_Q)$ Compute $\mathsf{F}_Q \subseteq \mathsf{F}_I$ and compute π_Q as $\pi_Q \leftarrow \mathsf{VC.Disagg}'(\mathsf{pp}, I, \mathsf{F}_I, \mathsf{st}, Q)$.

The algorithms for client nodes are:

$\mathsf{ClntNode.GetCreate}(\delta, J, \Upsilon_J) \to (b, \delta')$ Parse $\Upsilon_J := (\delta', \pi_{\mathsf{PoKSubV}'})$, set $n' = |J|$ and output $b \leftarrow \mathsf{PoKSubV}'.\mathsf{V}(\mathsf{pp}, (\delta, \delta', J), \pi_J) \wedge J = \{1, \ldots, |J|\}$ and δ'.

$\mathsf{ClntNode.VerRetrieve}(\delta, Q, \mathsf{F}_Q, \pi_Q) \to b$ Output $b \leftarrow \mathsf{VC.Ver}'(\mathsf{pp}, \delta, Q, \mathsf{F}_Q, \pi_Q)$

$\mathsf{ClntNode.ApplyUpdate}(\delta, \mathsf{op}, \Delta, \Upsilon_\Delta) \to (b, \delta')$ This algorithm is almost identical to the first part of the Storage Node algorithm $\mathsf{StrgNode.ApplyUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \Delta, \Upsilon_\Delta)$. The difference is that it executes only the parts that are related to the output of b and δ.

$\mathsf{AggregateCertificates}(\delta, (I, \mathsf{F}_I, \pi_I), (J, \mathsf{F}_J, \pi_J)) \to \pi_K$
Return $\pi_K \leftarrow \mathsf{VC.Agg}'(\mathsf{pp}, (I, \mathbf{F}_I, \pi_I), (J, \mathbf{F}_J, \pi_J))$.

Correctness. Here we state and prove the correctness of our VDS scheme.

Theorem 5.1. *The* VDS *scheme presented above is a correct verifiable decentralized storage scheme.*

Proof In the following we will always assume that $\mathsf{st} := (\mathsf{st}_1, \mathsf{st}_2)$ and $\delta := (\delta^\star, n) := ((\delta_1, \delta_2), n)$. Furthermore, whenever (a_I, b_I) appear, we assume that they are the outputs of $\mathsf{PartndPrimeProd}(I, \mathsf{F}_I)$, for each set of indices I. Finally for each set of indices I we assume $\pi_I := (\Gamma_I, \Delta_I)$.

First we note that in our construction it is sufficient for a local view $(\mathsf{pp}, \delta, n, \mathsf{st}, I, \mathsf{F}_I)$ of a storage node to be valid that
$\mathsf{ClntNode.VerRetrieve}(\delta, I, \mathsf{StrgNode.Retrieve}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, I)) = 1$ holds. More concretely this translates to $\mathsf{st}_1^{a_I} = \delta_1 \wedge \mathsf{st}_2^{b_I} = \delta_2$ and due to the correctness of disaggregation property $\mathsf{st}_1'^{a_Q} = \delta_1 \wedge \mathsf{st}_2'^{b_Q} = \delta_2$ holds where $\mathsf{st}' \leftarrow \mathsf{StrgNode.Retrieve}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, Q)$ for each $Q \subseteq I$. To put things clear, a local view of a storage node $(\mathsf{pp}, \delta, n, \mathsf{st}, I, \mathsf{F}_I)$ is valid if $\mathsf{st}_1^{a_I} = \delta_1 \wedge \mathsf{st}_2^{b_I} = \delta_2$.

Let $(\mathsf{pp}, \delta, n, \mathsf{st}, I, \mathsf{F}_I)$ be a valid local view of a storage node:
Update Correctness. Let (op, Δ) be an admissible update for (I, F_I, n) and $(\delta', n', \mathsf{st}', J, \mathsf{F}'_J, \Upsilon_\Delta)$ be the output of $\mathsf{StrgNode.PushUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \Delta)$. We discern three cases depending on the type of update:

– $\mathsf{op} = \mathsf{mod}$:

1. According to our construction $\delta^{\star\prime} = (\Gamma_K^{a'_K}, \Delta_K^{b'_K})$, where $(\Gamma_K, \Delta_K) = (\Gamma_I^{a_{I\setminus K}}, \Delta_I^{b_{I\setminus K}}) = (\mathsf{st}_1^{\frac{a_I}{a_K}}, \mathsf{st}_2^{\frac{b_I}{b_K}})$ (due to $\mathsf{VC.Disagg}'$). So $\delta' = (\mathsf{st}_1^{\frac{a_I}{a_K} a'_K}, \mathsf{st}_2^{\frac{b_I}{b_K} b'_K})$. Furthermore $\mathsf{st}' = \mathsf{st}$ and $J = I$, so
$$(\mathsf{st}_1'^{a'_J}, \mathsf{st}_1'^{b'_J}) = (\mathsf{st}_1^{\frac{a_I}{a_K} a'_K}, \mathsf{st}_2^{\frac{b_I}{b_K} b'_K}) = (\delta'_1, \delta'_2)$$
2. Let $(\delta, n, \mathsf{st}_s, I_s, \mathsf{F}_{I_s})$ be valid and $(b_s, \delta'_s, n'_s, \mathsf{st}'_s, J_s, \mathsf{F}'_{J_s})$ be the output of $\mathsf{StrgNode.ApplyUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \Delta, \Upsilon_\Delta)$. $b_s = 1$, $\delta'_s = \delta'$ and $n'_s = n'$ come from inspection. If $I \cap K = \emptyset$ then
$(\mathsf{st}'_{s,1}, \mathsf{st}'_{s,2}) \leftarrow \Big(\mathbf{ShamirTrick}(\mathsf{st}_{s,1}, \Gamma_K, \mathsf{F}_I, \mathsf{F}_K)^{a'_K}, \mathbf{ShamirTrick}(\mathsf{st}_{s,2}, \Delta_K, \mathsf{F}_I, \mathsf{F}_K)^{b'_K}\Big) =$
$= (\mathsf{st}_{s,1}^{\frac{a'_K}{a_K}}, \mathsf{st}_{s,2}^{\frac{b'_K}{b_K}})$ and $(a'_I, b'_I) = (a_I, b_I)$ remains the same. So
$$(\mathsf{st}_{s,1}'^{a'_I}, \mathsf{st}_{s,2}'^{b'_I}) = (\mathsf{st}_{s,1}^{\frac{a'_K}{a_K} a_I}, \mathsf{st}_{s,2}^{\frac{b'_K}{b_K} b_I}) = (\delta_{s,1}, \delta_{s,2})$$

If $I \cap K = K$ then st_s doesn't change and $(a'_I, b'_I) = (\frac{a_I}{a_K} a'_K, \frac{b_I}{b_K} b'_K)$, hence

$$(\mathsf{st}'^{a'_I}_{s,1}, \mathsf{st}'^{b'_I}_{s,2}) = (\delta'_{s,1}, \delta'_{s,2})$$

The validity of $(\mathsf{pp}, \delta'_s, n'_s, \mathsf{st}'_s, J_s, \mathsf{F}'_{J_s})$ in the case of $I \cap K = L \notin \{\emptyset, K\}$ is covered by the above two, since it essentially is a sequence of the two above cases.

3. Let (b_c, δ_c) be the output of $\mathsf{ClntNode.ApplyUpdate}(\delta, \mathsf{op}, \varDelta, \varUpsilon_\varDelta)$. It follows directly from the definition of $\mathsf{ClntNode.ApplyUpdate}$ (and its similarity with $\mathsf{StrgNode.ApplyUpdate}$) that $b_c = b_s = 1$ and $\delta'_c = \delta'_s = \delta'$.

– $\mathsf{op} = \mathsf{add}$:

1. According to our construction $\delta^{\star\prime} = (\delta_1^{a'_K}, \delta_2^{b'_K})$ and $\mathsf{st}' = \mathsf{st}$. Also, $J = I \cup K$ and $(a'_J, b'_J) = (a_I a'_K, b_I b'_K)$ and so

$$(\mathsf{st}'^{a'_J}_1, \mathsf{st}'^{b'_J}_1) = (\mathsf{st}_1^{a_I a'_K}, \mathsf{st}_2^{b_I b'_K}) = (\delta_1^{a'_K}, \delta_2^{b'_K}) = (\delta'_1, \delta'_2)$$

2. Let $(\delta, n, \mathsf{st}_s, I_s, \mathsf{F}_{I_s})$ be valid and $(b_s, \delta'_s, n'_s, \mathsf{st}'_s, J_s, \mathsf{F}'_{J_s})$ be the output of $\mathsf{StrgNode.ApplyUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \varDelta, \varUpsilon_\varDelta)$. $b_s = 1$, $\delta'_s = \delta'$ and $n'_s = n'$ come from inspection. Also $J = I$ so $(a'_J, b'_J) = (a_I, b_I)$. $\mathsf{st}' = (\mathsf{st}_1^{a'_K}, \mathsf{st}_2^{b'_K})$ and $\delta^{\star\prime} = (\delta_1^{a'_K}, \delta_2^{b'_K})$ so

$$(\mathsf{st}'^{a'_J}_1, \mathsf{st}'^{b'_J}_1) = (\mathsf{st}_1^{a'_K a_I}, \mathsf{st}_2^{b'_K b_I}) = (\delta'_1, \delta'_1)$$

3. Let (b_c, δ_c) be the output of $\mathsf{ClntNode.ApplyUpdate}(\delta, \mathsf{op}, \varDelta, \varUpsilon_\varDelta)$. Again correctness comes directly from the definition of $\mathsf{ClntNode.ApplyUpdate}$.

– $\mathsf{op} = \mathsf{del}$:

1. According to our construction $(\delta'_1, \delta'_2) = (\varGamma_K, \varDelta_K) = (\delta_1^{\frac{1}{a_K}}, \delta_2^{\frac{1}{b_K}})$, $\mathsf{st}' = \mathsf{st}$ and $J = I \setminus K$. Furthermore, $(a'_J, b'_J) = (\frac{a_I}{a_K}, \frac{b_I}{b_K})$

$$(\mathsf{st}'^{a'_J}_1, \mathsf{st}'^{b'_J}_1) = (\mathsf{st}_1^{\frac{a_I}{a_K}}, \mathsf{st}_2^{\frac{b_I}{b_K}}) = (\delta_1^{\frac{1}{a_K}}, \delta_2^{\frac{1}{b_K}}) = (\delta'_1, \delta'_2)$$

2. Let $(\delta, n, \mathsf{st}_s, I_s, \mathsf{F}_{I_s})$ be valid and $(b_s, \delta'_s, n'_s, \mathsf{st}'_s, J_s, \mathsf{F}'_{J_s})$ be the output of $\mathsf{StrgNode.ApplyUpdate}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, \mathsf{op}, \varDelta, \varUpsilon_\varDelta)$. $b_s = 1$, $\delta'_s = \delta'$ and $n'_s = n'$ come from inspection. Also let $L = I \cap K$ then $J = I \setminus L$ and if $\bar{L} = K \setminus L$ then

$$(\mathsf{st}'_1, \mathsf{st}'_2) \leftarrow (\mathbf{ShamirTrick}(\mathsf{st}_1, \varGamma_{\bar{L}}, \mathsf{F}_I, \mathsf{F}_{\bar{L}}), \mathbf{ShamirTrick}(\mathsf{st}_2, \varDelta_{\bar{L}}, \mathsf{F}_I, \mathsf{F}_{\bar{L}})) = (\mathsf{st}_1^{\frac{1}{a_{\bar{L}}}}, \mathsf{st}_2^{\frac{1}{b_{\bar{L}}}})$$

$$(\mathsf{st}'^{a'_J}_1, \mathsf{st}'^{b'_J}_1) = (\mathsf{st}_1^{\frac{a_J}{a_{\bar{L}}}}, \mathsf{st}_2^{\frac{b_J}{b_{\bar{L}}}}) = (\mathsf{st}_1^{\frac{a_I/a_L}{a_K/a_L}}, \mathsf{st}_1^{\frac{b_I/b_L}{b_K/b_L}}) = (\delta_1^{\frac{1}{a_K}}, \delta_2^{\frac{1}{b_K}}) = (\delta'_1, \delta'_2)$$

3. Let (b_c, δ_c) be the output of $\mathsf{ClntNode.ApplyUpdate}(\delta, \mathsf{op}, \varDelta, \varUpsilon_\varDelta)$. $b_c = b_s = 1$ and $\delta'_c = \delta'_s = \delta'$ from inspection.

Add Storage Correctness. It comes directly from aggregation correctness of $\mathsf{VC.Agg}'$ (see section 3.4).

Remove Storage Correctness. It comes directly from disaggregation correctness of $\mathsf{VC.Disagg}'$ (see section 3.4).

Create Correctness. Let $J \subseteq I$ and $(\delta', n', \mathsf{st}', J, \mathsf{F}_J, \varUpsilon_J)$ be the output of $\mathsf{StrgNode.CreateFrom}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, J)$ and (b, δ'') the output of $\mathsf{ClntNode.GetCreate}(\delta, J, \varUpsilon_J)$, then

$n' = |J|$ comes from inspection of $\mathsf{StrgNode.CreateFrom}$, $\delta'' = \delta'$ comes from inspection of $\mathsf{ClntNode.GetCreate}$ algorithm and validity of $(\mathsf{pp}, \delta', n', \mathsf{st}', J, \mathsf{F}_J)$ comes from correctness of $\mathsf{VC.Com}'$ and $\mathsf{VC.Agg}$. Finally, $b = 1$ comes from correctness of $\mathsf{PoKSubV}'$ protocol.
Aggregate Correctness. It comes directly from aggregation correctness of $\mathsf{VC.Agg}'$ (see section 3.4).

□

Security. Below we state and prove the security of our VDS scheme.

Theorem 5.2 (Security). *Let $\mathbb{G} \leftarrow \mathsf{Ggen}(1^\lambda)$ be a hidden order group where the strong RSA assumption holds, then the* VDS *scheme presented above is a secure Verifiable Decentralized Storage scheme in the generic group model.*

Proof First we observe that in our scheme, for every valid history $\mathcal{H}$, with $\mathsf{Bootstrap}(1^\lambda) \rightarrow (\mathsf{pp}, \delta_0, \mathsf{st}_0) := ((\mathbb{G}, g, g_0, g_1, \mathsf{PrimeGen}), ((g_0, g_1), 0), (g_0, g_1))$, the digest that arises is the same as a commitment of the file with $\mathsf{VC.Com}'$. Concretely, let $(b, \delta, \mathsf{F}) \leftarrow \mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H})$ then if $b = 1$ it holds that $\delta = \mathsf{VC.Com}'(\mathsf{pp}, \mathsf{F})$ or $\delta^\star = (\delta_1, \delta_2) = (g_0^a, g_1^b)$, where $(a, b) \leftarrow \mathsf{PartndPrimeProd}([|\mathsf{F}|], \mathsf{F})$. Particularly this is central to our construction and one can validate that it holds by inspecting all the algorithms that alter the digest.

To prove the theorem we use a hybrid argument. We start by defining the game G_0 as the actual VDS security game of Definition 5.6, and our goal is to prove that for any PPT $\mathcal{A}$, $\Pr[G_0 = 1] \in \mathsf{negl}(\lambda)$.

Game G_0:

$G_0 = \mathsf{VDS\text{-}Security}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda)$

$(\mathsf{pp}, \delta_0, \mathsf{st}_0) \leftarrow \mathsf{Bootstrap}(1^\lambda)$
$(\mathcal{H}, Q, \mathsf{F}^*_Q, \pi^*) \leftarrow \mathcal{A}(\mathsf{pp}, \delta_0, \mathsf{st}_0)$
$(b, \delta, \mathsf{F}) \leftarrow \mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H})$
$b \leftarrow b \wedge \mathsf{F}^*_Q \neq \mathsf{F}_Q \wedge$
  $\mathsf{ClntNode.VerRetrieve}(\mathsf{pp}, \delta, Q, \mathsf{F}^*_Q, \pi^*)$
return b

$\mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H})$

$\mathsf{F}_0 \leftarrow \emptyset; b \leftarrow 1$
for $i \in [\ell]$
  $\mathsf{F}_i \leftarrow \mathsf{FileChange}(\mathsf{F}_{i-1}, \mathsf{op}^i, \Delta^i)$
  if $\mathsf{op}^i \in \{\mathsf{mod}, \mathsf{add}, \mathsf{del}\}$ **then**
    $(b_i, \delta_i) \leftarrow \mathsf{ClntNode.ApplyUpdate}(\delta_{i-1}, \mathsf{op}^i, \Delta^i, \Upsilon^i_\Delta)$
  elseif $\mathsf{op}^i = \mathsf{cfrom}$ **then**
    $(b_i, \delta_i) \leftarrow \mathsf{ClntNode.GetCreate}(\delta_{i-1}, \Delta^i, \Upsilon^i_\Delta)$
  endif
  $b \leftarrow b \wedge b_i$
endfor
return $(b, \delta_\ell, \mathsf{F}_\ell)$

Recall that $\mathcal{H} = (\mathsf{op}^i, \Delta^i, \Upsilon^i_\Delta)_{i \in [\ell]}$ where:

- for $\mathsf{op}^i = \mathsf{mod}$: $\Delta^i := (K^i, \mathsf{F}^i_{K^i})$, $\Upsilon^i_\Delta := (\mathsf{F}^{i-1}_{K^i}, \pi^{i-1}_{K^i})$ and $\mathsf{ClntNode.ApplyUpdate}(\delta^{i-1}, \mathsf{op}^i, \Delta^i, \Upsilon^i_\Delta)$ outputs $b^i = 1$ if $\mathsf{VC.Ver}'(\mathsf{pp}, \delta^{i-1}, K^i, \mathsf{F}^{i-1}_{K^i}, \pi^{i-1}_{K^i}) = 1$ or $(\Gamma^{a_{K^i}}_{K^i} = \delta^{i-1}_1) \wedge (\Delta^{b_{K^i}}_{K^i} = \delta^{i-1}_2)$.
- for $\mathsf{op}^i = \mathsf{add}$: $\Delta^i := (K, \mathsf{F}^i_{K^i})$, $\Upsilon^i_\Delta := \varnothing$ and $\mathsf{ClntNode.ApplyUpdate}(\delta^{i-1}, \mathsf{op}^i, \Delta^i, \Upsilon^i_\Delta)$ outputs $b^i = 1$ if $K^i = \{n^{i-1} + 1, \ldots, n^{i-1} + |K^i|\}$.
- $\mathsf{op}^i = \mathsf{del}$: $\Delta^i := K^i$, $\Upsilon^i_\Delta := (\mathsf{F}^{i-1}_{K^i}, \pi^{i-1}_{K^i})$ and $\mathsf{ClntNode.ApplyUpdate}(\delta^{i-1}, \mathsf{op}^i, \Delta^i, \Upsilon^i_\Delta)$ outputs $b^i = 1$ if $(K^i = \{n^{i-1} - |K^i| + 1, \ldots, n^{i-1}\}) \wedge \mathsf{VC.Ver}'(\mathsf{pp}, \delta^{i-1}, K^{i-1}, \mathsf{F}^{i-1}_{K^i}, \pi^{i-1}_{K^i}))$ or $(K^i = \{n^{i-1} - |K^i| + 1, \ldots, n^{i-1}\} \wedge (\Gamma^{a_{K^i}}_{K^i} = \delta^{i-1}_1) \wedge (\Delta^{b_{K^i}}_{K^i} = \delta^{i-1}_2)$.
- $\mathsf{op}^i = \mathsf{cfrom}$: $\Delta^i := K^i$, $\Upsilon^i_\Delta := (\delta^i, \pi^i_{\mathsf{PoKSubV}'})$ and $\mathsf{ClntNode.GetCreate}(\delta^{i-1}, \Delta^i, \Upsilon^i_\Delta)$ outputs $b^i = 1$ if $\mathsf{PoKSubV}'.\mathsf{V}(\mathsf{pp}, (\delta^{i-1}, \delta^i, |K^i|, K^i), \pi^i_{K^i}) = 1$.

Game G_i: define G_i be the same as G_{i-1} except for the update i:

- if $\mathsf{op}^i = \mathsf{mod}$: $\Delta^i := (K^i, \mathsf{F}^i_{K^i})$, $\Upsilon^i_\Delta := (\mathsf{F}^{i-1}_{K^i}, \pi^{i-1}_{K^i})$ but in the i-th step of $\mathsf{EvalHistory}$ b^i is instead output of:
$$b^i \leftarrow (a_{K^i}|a) \wedge (b_{K^i}|b)$$
where $(a, b) \leftarrow \mathsf{PartndPrimeProd}([|\mathsf{F}^{i-1}|], \mathsf{F}^{i-1})$
In case $b^i = 0$ aborts (abort_i). Otherwise δ^i is computed normally from $\mathsf{ClntNode.ApplyUpdate}(\delta^{i-1}, \mathsf{op}^i, \Delta^i, \Upsilon^i_\Delta)$.
- for $\mathsf{op}^i = \mathsf{add}$: $\Delta^i := (K, \mathsf{F}^i_{K^i})$, $\Upsilon^i_\Delta := \varnothing$ and everything is the same as in G_{i-1}. I.e. (b^i, δ^i) is the output of $\mathsf{ClntNode.ApplyUpdate}(\delta^{i-1}, \mathsf{op}^i, \Delta^i, \Upsilon^i_\Delta)$.
- $\mathsf{op}^i = \mathsf{del}$: $\Delta^i := K^i$, $\Upsilon^i_\Delta := (\mathsf{F}^{i-1}_{K^i}, \pi^{i-1}_{K^i})$. Similarly to the mod case b^i is the output of:
$$b^i \leftarrow (a_{K^i}|a) \wedge (b_{K^i}|b) \wedge (K^i = \{n^{i-1} - |K^i| + 1, \ldots, n^{i-1}\})$$
where $(a, b) \leftarrow \mathsf{PartndPrimeProd}([|\mathsf{F}^{i-1}|], \mathsf{F}^{i-1})$
In case $b^i = 0$ aborts (abort_i). Otherwise δ^i is computed normally from $\mathsf{ClntNode.ApplyUpdate}(\delta^{i-1}, \mathsf{op}^i, \Delta^i, \Upsilon^i_\Delta)$.
- $\mathsf{op}^i = \mathsf{cfrom}$: $\Delta^i := K^i$, $\Upsilon^i_\Delta := (\delta^i, \pi^i_{\mathsf{PoKSubV}'})$ but in the i-th step of $\mathsf{EvalHistory}$ b^i is instead:
$$b^i \leftarrow (\mathsf{F}^{i-1}_{K^i} \subseteq \mathsf{F}^{i-1}) \wedge \delta^i = \mathsf{VC.Com}'(\mathsf{pp}, \mathsf{F}^{i-1}_{K^i}) \wedge J = \{1, \ldots, |J|\}$$
In case $b^i = 0$ aborts (abort_i).

Lemma 5.1. *Let $\mathsf{op}^i = \mathsf{mod}$ then if the strong RSA assumption holds for Ggen,* $\Pr[G_{i-1} = 1] \leq \Pr[G_i = 1] + \mathsf{negl}(\lambda)$.

Proof It is straightforward that the only difference between G_{i-1} and G_i is in the computation of b^i inside the $\mathsf{EvalHistory}$. That is in $G_{i-1} : b^i = (\Gamma^{a_{K^i}}_{K^i} = \delta^{i-1}_1) \wedge (\Delta^{b_{K^i}}_{K^i} = \delta^{i-1}_2)$ and in $G_i : b^i = (a_{K^i}|a) \wedge (b_{K^i}|b)$. Since $\mathsf{abort}_1, \mathsf{abort}_2, \ldots, \mathsf{abort}_{i-2}$ have not happen, from correctness of the VDS scheme it comes that $(\delta^{i-1}_1, \delta^{i-1}_2) = (g^a_0, g^b_1)$, where $(a, b) \leftarrow \mathsf{PartndPrimeProd}([|\mathsf{F}^{i-1}|], \mathsf{F}^{i-1})$.

$|\Pr[G_{i-1} = 1] - \Pr[G_i = 1]| = Pr[\mathsf{abort}_i] = Pr[b^i = 0] = Pr[(a_{K^i}|a) \wedge (b_{K^i}|b)]$. But since abort_{i-1} didn't happen $(\Gamma^{a_{K^i}}_{K^i} = g^a_0) \wedge (\Delta^{b_{K^i}}_{K^i} = g^b_1)$. Therefore it is straightforward to abort_i to the strong RSA assumption, i.e. $Pr[\mathsf{abort}_i] = \mathsf{negl}(\lambda)$. □

Lemma 5.2. *Let* $\mathsf{op}^i = \mathsf{del}$ *then if the strong RSA assumption holds for* Ggen*,* $\Pr[G_{i-1} = 1] \leq \Pr[G_i = 1] + \mathsf{negl}(\lambda)$*.*

Proof The same as the above case of $\mathsf{op}^i = \mathsf{mod}$ holds. □

Lemma 5.3. *Let* $\mathsf{op}^i = \mathsf{add}$ *then* $\Pr[G_{i-1} = 1] = \Pr[G_i = 1]$*.*

Proof G_{i-1} and G_i are identical. □

Lemma 5.4. *Let* $\mathsf{op}^i = \mathsf{cfrom}$ *then for any PPT* $\mathcal{A}$ *in* G_i *there exists an algorithm* $\mathcal{E}$ *such that* $\Pr[G_{i-1} = 1] \leq \Pr[G_i = 1] + \mathsf{negl}(\lambda)$ *of the strong RSA assumption holds.*

Proof Let $\mathcal{E}$ be the extractor of $\mathsf{PoKSubV}'$ protocol that corresponds to $\mathcal{A}$. Since $\mathsf{PoKSubV}'$ is knowledge sound, $\mathcal{E}$ outputs $(\mathbf{F}^{i-1}_{K^i}, \pi_{K^i}, \pi'_{K^i})$ such that $\mathsf{VC.Ver}'(\mathsf{pp}, \delta^{i-1}, K^i, \mathbf{F}^{i-1}_{K^i}, \pi_{K^i}) = 1 \wedge \mathsf{VC.Ver}'(\mathsf{pp}, \delta^i, K^i, \mathbf{F}^{i-1}_{K^i}, \pi'_{K^i}) = 1 \wedge |\mathbf{F}^{i-1}_{K^i}| = n'$, where $\delta^i = (\delta^{\star i}, n^i)$. Since $\mathsf{abort}_1, \mathsf{abort}_2, \ldots, \mathsf{abort}_{i-2}$ have not happen, from correctness of the VDS scheme it comes that $\delta^{i-1} = \mathsf{VC.Com}'(\mathsf{pp}, \mathsf{F}^{i-1})$. From the first verification equation above we get that under strong RSA assumption $\mathsf{F}^{i-1}_{K_i} \subseteq \mathsf{F}^{i-1}$. From the second verification equation above we get that $\mathsf{F}^{i-1}_{K_i}$ is an opening of δ^i. From the third equation above we get that δ^i is a digest for a file of size $|\mathsf{F}^{i-1}_{K_i}|$. From the last two points we get that $\delta^i = \mathsf{VC.Com}'(\mathsf{pp}, \mathsf{F}^{i-1}_{K^i})$.

So $\Pr[G_{i-1} = 1] \leq \Pr[G_i = 1] + \mathsf{negl}(\lambda)$. □

We conclude that in any case $\Pr[G_{i-1} = 1] \leq \Pr[G_i = 1] + \mathsf{negl}(\lambda)$. Since $|\mathcal{H}| = \ell = \mathsf{poly}(\lambda)$ with a hybrid argument we get that $\Pr[G_0 = 1] \leq \Pr[G_\ell = 1] + \mathsf{negl}(\lambda)$. But clearly $G_\ell = 0$ always (since no abort has happened), and thus $\Pr[\mathsf{VDS\text{-}Security}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda) = 1] = P[G_0 = 1] = \mathsf{negl}(\lambda)$. □

5.5 VDS Proof of Storage

For a VDS scheme we additionally 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. Finally, a main distinguishing feature compared to existing PoRs/PDPs is that proofs are generated in a distributed fashion by a collection of storage nodes and remain compact regardless of the number of nodes involved in the their generation.

Below we begin by defining the syntax and correctness of proof of storage for a VDS scheme; these are defined the same for modeling both retrievability and data possession. The difference between the two is only in the security notion.

A VDS scheme VDS as in Definition 5.1 admits proofs of storage if there exist algorithms ($\mathsf{StrgNode.PoS\text{-}Challenge}, \mathsf{StrgNode.PoS\text{-}Prove}, \mathsf{ClntNode.PoS\text{-}Ver}$) that work as follows.

$\mathsf{StrgNode.PoS\text{-}Challenge}(\delta) \rightarrow r$ This is a probabilistic algorithm that, given a file's digest δ, outputs a challenge r.

$\mathsf{StrgNode.PoS\text{-}Prove}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, r) \to \pi_r$ This algorithm allows a storage node to (partially) answer a PoS challenge r. The inputs include the local view of the storage node, i.e., digest δ, length n local state st and file portion (I, F_I), and a challenge $r \in \mathcal{C}$. The output is a proof π_r.

$\mathsf{StrgNode.PoS\text{-}Aggregate}(\delta, r, \pi_{r,1}, \pi_{r,2}) \to (b, \pi_r)$ On input a digest δ, a challenge $r \in \mathcal{C}$ and two partial proofs $\pi_{r,1}, \pi_{r,2}$, this algorithm outputs an aggregated proof π_r and a bit b such that $b = 1$ iff π_r is a "complete" proof for challenge r (i.e., it can be verified).

$\mathsf{ClntNode.PoS\text{-}Ver}(\delta, r, \pi_r) \to b$ On input a digest δ, a challenge $r \in \mathcal{C}$ and a "complete" proof π_r, this algorithm accepts (outputs 1) or rejects (outputs 0).

Definition 5.7 (Correctness of VDS PoS). *A VDS scheme* VDS *has a correct PoS mechanism if* VDS *is* correct *and if for all honestly generated parameters* $(\mathsf{pp}, \delta_0, \mathsf{st}_0) \leftarrow \mathsf{Bootstrap}(1^\lambda)$*, any file* F *of length* n *and any set of* ℓ *valid storage node's local views* $(\delta, n, \mathsf{st}_j, I_j, \mathsf{F}_{I_j})$ *such that* $\bigcup_{j=1}^{\ell}(I_j, \mathsf{F}_{I_j}) = ([n], \mathsf{F})$*, the following holds:*

if $r \leftarrow\$ \mathsf{StrgNode.PoS\text{-}Challenge}(\delta)$*,* $\pi_{r,j} \leftarrow \mathsf{StrgNode.PoS\text{-}Prove}(\delta, n, \mathsf{st}_j, I_j, \mathsf{F}_{I_j}, r)$ *for all* $j \in [\ell]$*, and* π_r *is obtained by aggregating* $\{\pi_{r,j}\}_{j\in[\ell]}$ *in an arbitrary order using repeated usage of* $\mathsf{StrgNode.PoS\text{-}Aggregate}$ *until getting* $b = 1$*, then* $\mathsf{ClntNode.PoS\text{-}Ver}(\delta, r, \pi_r) = 1$*.*

PoS Security. Here we define two security properties for the above PoS mechanism: retrievability and data possession. Similarly to [JK07, ABC+07], the idea is to ask that from any adversary, controlling all storage nodes, who creates a proof π_r that is accepted with sufficiently high probability it is possible to extract the entire file. In the retrievability case, this is formalized through requiring the existence of an extractor that extracts the file by interacting multiple times with such prover (via rewinding). In the data possession case, it is the same except that the extractor is non-black-box, i.e., we assume that for any adversary there is an extractor; in other words, the extractor is a cryptographic one that does not exist in the real world, and for this reason the data possession notion is weaker than retrievability.

We build our definitions inspired to the one of Proof of Retrievable commitment (PoRC) soundness in [Fis18]. To this end, we define the following two experiments.

$\mathsf{VDSPoSAdm}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda)$	$\mathsf{VDSPoSExtr}^{\mathcal{A},\mathcal{E}}_{\mathsf{VDS}}(\lambda)$
$(\mathsf{pp}, \delta_0, \mathsf{st}_0) \leftarrow \mathsf{Bootstrap}(1^\lambda)$	$(\mathsf{pp}, \delta_0, \mathsf{st}_0) \leftarrow \mathsf{Bootstrap}(1^\lambda)$
$(\mathcal{H}^*, \alpha^*) \leftarrow \mathcal{A}_1(\mathsf{pp}, \delta_0, \mathsf{st}_0)$	$(\mathcal{H}^*, \alpha^*) \leftarrow \mathcal{A}_1(\mathsf{pp}, \delta_0, \mathsf{st}_0)$
$(b^*, \delta^*, \mathsf{F}^*) \leftarrow \mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H}^*)$	$(b^*, \delta^*, \mathsf{F}^*) \leftarrow \mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H}^*)$
if $b^* = 0$ abort,	$\widehat{\mathsf{F}} \leftarrow \mathcal{E}^{\mathcal{A}_2(\mathsf{pp},\delta_0,\mathsf{st}_0,\delta^*,\alpha^*,\cdot)}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \delta^*)$
else $r \leftarrow\$ \mathsf{StrgNode.PoS\text{-}Challenge}(\delta^*)$;	**return** 1 iff $\widehat{\mathsf{F}} \neq \bot \wedge \mathsf{F}^* \not\equiv_\delta \widehat{\mathsf{F}}$
$\pi^*_r \leftarrow \mathcal{A}_2(\mathsf{pp}, \delta_0, \mathsf{st}_0, \delta^*, \alpha^*, r)$	
return $\mathsf{ClntNode.PoS\text{-}Ver}(\delta^*, r, \pi^*_r)$	

Above, given two files $\mathsf{F} \in \mathcal{M}^n$ and $\mathsf{F}' \in \mathcal{M}^{n'}$ and a parameter $\mu \in [0, 1]$ we say that F agrees on a μ-fraction with F', denoted $\mathsf{F} \equiv_\mu \mathsf{F}'$, if and only if $n = n'$ and $|\{i \in [n] : \mathsf{F}_i = \mathsf{F}'_i\}| \geq \mu \cdot n$.

The experiment $\mathsf{VDSPoSAdm}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda)$ is parametrized by a two-stage adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and models the interaction between an adversarial prover that creates a (valid) VDS history which results into a digest δ^* and then replies to one honestly generated challenge. This experiment is used to formalize the notion of ϵ-admissible adversaries, which in brief are adversaries that in this game answer successfully to the challenge with probability at least ϵ. The second experiment $\mathsf{VDSPoSExtr}^{\mathcal{A},\mathcal{E}}_{\mathsf{VDS}}(\lambda)$ is again parametrized by a two-stage adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, and additionally by an extractor $\mathcal{E}$ having oracle access to $\mathcal{A}_2$. The goal of the extractor is to return a file $\widehat{\mathsf{F}}$ which agrees on a μ-fraction of indices with the file F^* implicitly returned by $\mathcal{A}_1$.

Definition 5.8 (Admissible VDS PoS Adversary). *A VDS adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ is ϵ-admissible if and only if the experiment $\mathsf{VDSPoRAdm}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda)$ does not abort with probability $1 - \mathsf{negl}(\lambda)$ and $\Pr[\mathsf{VDSPoRAdm}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda, \mathsf{F}) = 1] \geq \epsilon$.*

Definition 5.9 (Retrievability for VDS). *A VDS scheme VDS is (μ, ϵ)-retrievable if it is secure and for some $\lambda_{\epsilon,\mu} \in O(\log \epsilon / \log \mu)$ and every $\lambda > \lambda_{\epsilon,\mu}$ there exists an extractor $\mathcal{E}$ that runs in time $\mathsf{poly}(\lambda, n, 1/\epsilon)$ such that for any adversary $\mathcal{A}$ which is ϵ-admissible we have $\Pr[\mathsf{VDSPoSExtr}^{\mathcal{A},\mathcal{E}}_{\mathsf{VDS}}(\lambda) = 1] \in \mathsf{negl}(\lambda)$.*

Definition 5.10 (Data Possession for VDS). *A VDS scheme VDS has ϵ-data-possession if it is secure and for some $\lambda_{\epsilon,\mu} \in O(\log \epsilon / \log \mu)$ and every $\lambda > \lambda_{\epsilon,\mu}$ and every adversary $\mathcal{A}$ which is ϵ-admissible there is an extractor $\mathcal{E}$ that runs in time $\mathsf{poly}(\lambda, n, 1/\epsilon)$ such that $\Pr[\mathsf{VDSPoSExtr}^{\mathcal{A},\mathcal{E}}_{\mathsf{VDS}}(\lambda) = 1] \in \mathsf{negl}(\lambda)$.*

Parallel Proof of Storage. We extend our PoS notion for VDS to a setting where one can simultaneously check storage of k different files of the same length with a single challenge. The syntactical change we do is to assume that one can generate a challenge by only knowing the length of the files. Informally, the parallel version of retrievability (resp. data possession) is a parallel repetition of the protocol, and then from any adversary that answers successfully for all files it is possible to extract files so that each is consistent with at least a μ-fraction of the original one.

The parallel security experiments are as follows.

$\mathsf{VDSPoS\text{-}Par\text{-}Adm}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda)$

$(\mathsf{pp}, \delta_0, \mathsf{st}_0) \leftarrow \mathsf{Bootstrap}(1^\lambda)$
$\{(\mathcal{H}^*_i, \alpha^*_i)\}_{i=1}^k \leftarrow \mathcal{A}_1(\mathsf{pp}, \delta_0, \mathsf{st}_0)$
$\{(b^*_i, \delta^*_i, \mathsf{F}^*_i) \leftarrow \mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H}^*_i)\}_{i=1}^k$
if $\exists i \in [k] : b^*_i = 0 \vee \neg(\wedge_{i \in [k-1]} |\mathsf{F}^*_i| = |\mathsf{F}^*_{i+1}|)$ abort,
else $r \leftarrow\$ \mathsf{StrgNode.PoS\text{-}Challenge}(|\mathsf{F}^*_1|)$;
$\{\pi^*_{r,i}\}_{i=1}^k \leftarrow \mathcal{A}_2(\mathsf{pp}, \delta_0, \mathsf{st}_0, \{\delta^*_i, \alpha^*_i\}_{i=1}^k, r)$
return 1 iff $\mathsf{ClntNode.PoS\text{-}Ver}(\delta^*_i, r, \pi^*_{r,i})\ \forall i \in [k]$

$\mathsf{VDSPoS\text{-}Par\text{-}Extr}^{\mathcal{A},\mathcal{E}}_{\mathsf{VDS}}(\lambda)$

$(\mathsf{pp}, \delta_0, \mathsf{st}_0) \leftarrow \mathsf{Bootstrap}(1^\lambda)$
$\{(\mathcal{H}^*_i, \alpha^*_i)\}_{i=1}^k \leftarrow \mathcal{A}_1(\mathsf{pp}, \delta_0, \mathsf{st}_0)$
$\{(b^*_i, \delta^*_i, \mathsf{F}^*_i) \leftarrow \mathsf{EvalHistory}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \mathcal{H}^*_i)\}_{i=1}^k$
$\{\widehat{\mathsf{F}}_i\}_{i=1}^k \leftarrow \mathcal{E}^{\mathcal{A}_2(\mathsf{pp}, \delta_0, \mathsf{st}_0, \delta^*_i, \alpha^*_i, \cdot)}(\mathsf{pp}, \delta_0, \mathsf{st}_0, \{\delta^*_i\}_{i=1}^k)$
return 1 iff $\forall i \in [k] : \widehat{\mathsf{F}}_i \neq \bot \wedge \exists j \in [k] : \mathsf{F}^*_j \not\equiv_\delta \widehat{\mathsf{F}}_j$

Definition 5.11 (Admissible VDS PoS Parallel Adversary). *A VDS adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ is parallel ϵ-admissible if and only if the experiment $\mathsf{VDSPoS\text{-}Par\text{-}Adm}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda)$ does not abort with probability $1 - \mathsf{negl}(\lambda)$ and*
$\Pr[\mathsf{VDSPoS\text{-}Par\text{-}Adm}^{\mathcal{A}}_{\mathsf{VDS}}(\lambda, \mathsf{F}) = 1] \geq \epsilon$.

Definition 5.12 (Parallel Retrievability for VDS). *A VDS scheme* VDS *is parallel* (μ, ϵ)*-retrievable if it is secure and for some* $\lambda_{\epsilon,\mu} \in O(\log \epsilon / \log \mu)$ *and every* $\lambda > \lambda_{\epsilon,\mu}$ *there exists an extractor* $\mathcal{E}$ *that runs in time* $\mathsf{poly}(\lambda, n, 1/\epsilon)$ *such that for any adversary* $\mathcal{A}$ *which is parallel* ϵ*-admissible we have*
$\Pr[\mathsf{VDSPoS\text{-}Par\text{-}Extr}_{\mathsf{VDS}}^{\mathcal{A},\mathcal{E}}(\lambda) = 1] \in \mathsf{negl}(\lambda)$.

Definition 5.13 (Parallel Data Possession for VDS). *A VDS scheme* VDS *has parallel* ϵ*-data-possession if it is secure and for some* $\lambda_{\epsilon,\mu} \in O(\log \epsilon / \log \mu)$ *and every* $\lambda > \lambda_{\epsilon,\mu}$ *and every adversary* $\mathcal{A}$ *which is* ϵ*-admissible there is an extractor* $\mathcal{E}$ *that runs in time* $\mathsf{poly}(\lambda, n, 1/\epsilon)$ *such that* $\Pr[\mathsf{VDSPoS\text{-}Par\text{-}Extr}_{\mathsf{VDS}}^{\mathcal{A},\mathcal{E}}(\lambda) = 1] \in \mathsf{negl}(\lambda)$.

With the following theorem we show that it is enough to prove security in the (nonparallel) setting. The idea of the proof is that one can construct an extractor for the parallel game by running k extractors of the nonparallel game. The analysis of this reduction is rather simple and is therefore omitted.

Theorem 5.3. *A VDS scheme that has* (μ, ϵ)*-retrievability (resp. data possession) also achieves parallel* (μ, ϵ)*-retrievability (resp. data possession).*

5.6 Proof of Storage for our VDS

In this section we show that our VDS scheme from Section 5.4 admits both a PoR and a PDP mechanism.

Retrievability. In the case of PoR we can describe the algorithms generically from the VDS algorithms. Namely, any VDS always admits a PoR.

$\mathsf{StrgNode.PoS\text{-}Challenge}(n) \to r$ samples λ_{pos} integers $r_1, \ldots, r_{\lambda_{\mathsf{pos}}} \leftarrow_{\$} [n]$ and define $r = \{r_1, \ldots, r_{\lambda_{\mathsf{pos}}}\}$.

$\mathsf{StrgNode.PoS\text{-}Prove}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, r) \to \pi_r$ Parse $r := \{r_1, \ldots, r_{\lambda_{\mathsf{pos}}}\}$ and let $Q := I \cap r$, compute $(\mathsf{F}_Q, \pi_Q) \leftarrow \mathsf{StrgNode.Retrieve}(\delta, n, \mathsf{st}, I, \mathsf{F}_I, Q)$ and return $\pi_{r,Q} := (Q, \mathsf{F}_Q, \pi_Q, Q)$.

$\mathsf{StrgNode.PoS\text{-}Aggregate}(\delta, r, \pi_{r,1}, \pi_{r,2}) \to (b, \pi_r)$ Parse $\pi_{r,1} := (Q_1, \mathsf{F}_{Q_1}, \pi_{Q_1})$ and $\pi_{r,2} := (Q_2, \mathsf{F}_{Q_2}, \pi_{Q_2})$.
If $\exists i \in \{1, 2\}$ such that $Q_i = r$ set $b := 1$ and $\pi_r := \pi_{r,i}$.
Otherwise, compute $(Q, \mathsf{F}_Q) := (Q_1, \mathsf{F}_{Q_1}) \cup (Q_2, \mathsf{F}_{Q_2})$ and
$\pi_Q \leftarrow \mathsf{AggregateCertificates}(\delta, (Q_1, \mathsf{F}_{Q_1}, \pi_{Q_1}), (Q_2, \mathsf{F}_{Q_2}, \pi_{Q_2}))$, and set $\pi_r := (Q, \mathsf{F}_Q, \pi_Q)$. If $Q = r$, set $b := 1$, otherwise set $b := 0$.
Return (b, π_r)

$\mathsf{ClntNode.PoS\text{-}Ver}(\delta, r, \pi_r) \to b$ parse $\pi_r := (Q, \mathsf{F}_Q, \pi_Q)$ and return 1 iff $Q = r$ and $\mathsf{ClntNode.VerRetrieve}(\delta, Q, \mathsf{F}_Q, \pi_Q) = 1$.

Correctness is easy by inspection and by the correctness of VDS.

For security we state the following theorem. The proof is omitted since it is almost identical to the proof of the VC-PoRC construction in [Fis18]; the only difference is that instead of reducing to the position binding of the VC we reduce to the security of the VDS scheme.[17]

[17] For this we also observe that Fisch's proof could go through even assuming a weaker notion of position binding in which the adversary declares the whole committed vector in addition to the two discording openings for one position.

Theorem 5.4. *If the VDS scheme* VDS *from Section 5.4 is secure then its extension with the PoS algorithms described above is a* (μ, ϵ)*-retrievable VDS for any* $\epsilon > 0$ *such that* $\epsilon - \mu^{\lambda_{\mathsf{pos}}}$ *is non-negligible in* λ.

Data Possession. The PDP for our VDS is almost the same as the PoR described above except that the last step of aggregation "compacts" the proof by generating an AoK of opening (see Section 4). More precisely, let $\mathsf{PoKOpen}'$ be the same as protocol $\mathsf{PoKOpen}$ but adjusted for the simpler version of our VC scheme given in Section 5.4. Namely, the one where the commitment is (A, B) and the verification is the $\mathsf{VC.Ver}'$ algorithm. So, the relation proven by $\mathsf{PoKOpen}'$ is:

$$R_{\mathsf{PoKOpen}'} = \{(\,(C, I), (\boldsymbol{y}, \pi_I)\,) : \mathsf{VC.Ver}'(\mathsf{pp}, C, I, \boldsymbol{y}, \pi_I) = 1\}.$$

Then, the PDP aggregation algorithm works as follows.

$\mathsf{StrgNode.PoS\text{-}Aggregate}(\delta, r, \pi_{r,1}, \pi_{r,2}) \to (b, \pi_r)$ Parse $\pi_{r,1} := (Q_1, \mathsf{F}_{Q_1}, \pi_{Q_1})$ and $\pi_{r,2} := (Q_2, \mathsf{F}_{Q_2}, \pi_{Q_2})$.
If $\exists i \in \{1, 2\}$ such that $Q_i = r$ set $b := 1$ and $(Q, \mathsf{F}_Q, \pi_Q) := \pi_{r,i}$.
Otherwise, compute $(Q, \mathsf{F}_Q) := (Q_1, \mathsf{F}_{Q_1}) \cup (Q_2, \mathsf{F}_{Q_2})$ and
$\pi_Q \leftarrow \mathsf{AggregateCertificates}(\delta, (Q_1, \mathsf{F}_{Q_1}, \pi_{Q_1}), (Q_2, \mathsf{F}_{Q_2}, \pi_{Q_2}))$.
If $Q \neq r$, set $\pi_r := (Q, \mathsf{F}_Q, \pi_Q)$ and return $(0, \pi_r)$. Otherwise, proceed to compute an AoK of opening, i.e., compute $\pi_r \leftarrow \mathsf{PoKOpen}'.\mathsf{P}((\delta, Q), (\mathsf{F}_Q, \pi_Q))$, and then return $(1, \pi_r)$

$\mathsf{ClntNode.PoS\text{-}Ver}(\delta, r, \pi_r) \to b$ return $\mathsf{PoKOpen}'.\mathsf{V}((\delta, r), \pi_r)$.

Correctness is easy by inspection and by the correctness of VDS.

For security we state the following theorem. The proof is essentially the same as the one for retrievability except that in this case we define a non-black-box extractor which is build from the extractor for $\mathsf{PoKOpen}'$.

Theorem 5.5. *If* $\mathsf{PoKOpen}'$ *is a secure AoK for relation* $R_{\mathsf{PoKOpen}'}$ *and the VDS scheme* VDS *from Section 5.4 is secure, then its extension with the PoS algorithms described above satisfies* (μ, ϵ)*-data possession for any* $\epsilon > 0$ *such that* $\epsilon - \mu^{\lambda_{\mathsf{pos}}}$ *is non-negligible in* λ.

Parallel PDP. We observe that in the case of executing the PDP protocol in parallel for k different digests, our construction has an interesting efficiency property. While verifying one PDP takes time $O(\lambda_{\mathsf{pos}})$ due to the computation of the group element $U_r := g^{u_r}$ with $u_r := \mathsf{PrimeProd}(r)$, in the case of verifying k PDPs with the same challenge the element U_r can be reused. This yields a total verification time $O(\lambda_{\mathsf{pos}} + k)$ instead of $O(k \cdot \lambda_{\mathsf{pos}})$.

6 Experimental Evaluation

We have implemented in Rust our VC scheme (with and without preprocessing) and the recent VC of [BBF19] (referred as BBF in what follows). Here we discuss an experimental evaluation of these three schemes. [18] Below is a summary of the comparison, details of the experiments are in Appendix C.

[18] 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 Appendix C

– Our VC construction is faster in opening and verification than BBF (up to 2.5× and 2.3× faster respectively), but at the cost of a slower commitment stage (up to 6× slower). These differences tend to flatten for larger vectors and opening sizes.
– Our VC 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 (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 VC construction with preprocessing has an expensive commitment stage, this tends to be amortized throughout very few openings[19], as few as 30 (see Figure 6 in Appendix C). These effects are particularly significant over a higher number of openings: over 1000 openings our VC construction with preprocessing has an amortized cost of less than 6 seconds, while our VC construction and BBF have amortized openings above 90 seconds.

Mitigating Space Requirements for Preprocessing Construction The experiments illustrated so far show the benefit of using preprocessing to speedup opening time. This comes at the cost of storing an auxiliary information—N openings—which, in spite of being much smaller than in BBF, can still be quite large. Here we discuss two ways to mitigate this storage cost, which can be used either separately or together.

– **Hashing in blocks.** Let us recall that by selecting a block size $\ell = 2\lambda$ (e.g., 256) one can combine our VC with a collision-resistant hash function and support larger vectors at virtually the same cost. Concretely, given a vector $\boldsymbol{v}$ of N blocks, each of ℓ_H bits, one can obtain a vector $\boldsymbol{v}' \in (\{0,1\}^\ell)^N$ by hashing each ℓ_H-bits block into a ℓ-bits one. The downside of this approach is that subvector openings with respect to the original vector $\boldsymbol{v}$ are less fine grained. On the good side, though, one gets that the efficiency of a VC for a vector of size $N \cdot \ell_H$ is virtually the same[20] as the one for a VC for a vector of size $N 2\lambda$. For example, by selecting $\ell_H = 2$ Kibit our timings for a vector of 262 144 bits would work for one of 1 Mibit. This would yield a committing/preprocessing time of roughly 10 minutes. These advantages also translate opening times: for example, if we expect openings of roughly $M = 2^{11}$ bits we can expect a virtually instantaneous opening time (as we just need to look up a cached precomputed proof). A larger opening size such as $M = 2^{14}$ (resp. $M = 2^{17}$) bits, would yield a running time of roughly 4 seconds (resp. 70 seconds).
– **Selecting larger precomputed chunks.** Another possibility to reduce storage is to precompute less openings by storing more aggregated openings, namely instead of an opening for every chunk of ℓ bits, store one opening for every chunk of $B \cdot \ell$ bits. This technique requires a bit more computation in order to compute disaggregations—about $m(B-1)$ $\mathbb{G}$ operations in the worst

[19] 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.
[20] This is because the cost of hashing is negligible compared to group operations.

case for m positions (cf. Section 3.7)—but opens the way to various tradeoffs to be explored. For instance, one could use certain application-dependent heuristics to choose which positions to precompute aggregated. As an example in the VC application to proofs of space and replication [Fis18] one opens a set of randomly chosen positions, and for each of them, also a set of predetermined positions.[21]

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.

ABC+11. G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song. Remote Data Checking Using Provable Data Possession. *ACM Trans. Inf. Syst. Secur.*, 14(1):12:1–12:34, June 2011.

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 *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

BCS16. E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive Oracle Proofs. In M. Hirt and A. D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Heidelberg, October / November 2016.

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.

BGV11. S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable Delegation of Computation over Large Datasets. In P. Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 111–131. Springer, Heidelberg, August 2011.

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.

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`.

FS87. A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In A. M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[21] There, each vector entry is the node of a DAG and one opens a set of randomly chosen nodes and for each of them a given number of parents.

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.

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.

Kil92. J. Kilian. A Note on Efficient Zero-Knowledge Proofs and Arguments (Extended Abstract). In *24th ACM STOC*, pages 723–732. ACM Press, May 1992.

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. Lai and G. Malavolta. Subvector commitments with application to succinct arguments. In *Annual International Cryptology Conference*, pages 530–560. Springer, 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.

Mic94. S. Micali. CS Proofs (Extended Abstracts). In *35th FOCS*, pages 436–453. IEEE Computer Society Press, November 1994.

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

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`.

A PoProd protocol for Union of RSA Accumulators

Let $\mathbb{G}$ be a an hidden order group as generated by Ggen, and let $g_1, g_2, g_3 \in \mathbb{G}$ be three honestly sampled random generators. A more straightforward succinct argument of knowledge for the union of RSA Accumulators is for the following relation

$$R_{\mathsf{PoProd}} = \left\{ ((A, B, C), (a, b)) \in \mathbb{G}^3 \times \mathbb{Z}^3 \;:\; A = g_1^a \wedge B = g_2^b \wedge C = g_3^{a \cdot b} \quad \right\}$$

Our protocol PoProd is described below.

PoProd protocol

$\mathsf{Setup}(1^\lambda)$: 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: $(\mathsf{crs}, (A, B, C), (a, b))$. Verifier's input: $(\mathsf{crs}, (A, B, C))$.

$\mathsf{V} \to \mathsf{P}$: $\ell \leftarrow_\$ \mathsf{Primes}(\lambda)$
$\mathsf{P} \to \mathsf{V}$: $\pi := ((Q_A, Q_B, Q_C), r_a, r_b)$ computed as follows
- $(q_a, q_b, q_c) \leftarrow (\lfloor a/\ell \rfloor, \lfloor b/\ell \rfloor, \lfloor ab/\ell \rfloor)$
- $(r_a, r_b) \leftarrow (a \mod \ell, b \mod \ell)$
- $(Q_A, Q_B, Q_C) := (g_1^{q_a}, g_2^{q_b}, g_3^{q_c})$

$\mathsf{V}(\mathsf{crs}, (A, B, C), \ell, \pi)$:
- Compute $r_c \leftarrow r_a \cdot r_b \mod \ell$
- Output 1 iff $r_a, r_b \in [\ell] \;\wedge\; Q_A^\ell g_1^{r_a} = A \;\wedge\; Q_B^\ell g_2^{r_b} = B \;\wedge\; Q_C^\ell g_3^{r_c} = C$

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 PoKE^* protocol from [BBF19]. The latter is proven in the generic group model for hidden order groups (where also the adaptive root assumption holds).

Theorem A.1. *The* PoProd *protocol is an argument of knowledge for* R_{PoProd} *in the generic group model.*

The proof is quite similar to the one of theorem 3.1 only instead of using the extractor if PoKRep protocol we use the extractors of two PoKE^* protocols (one for $g_1^a = A$ and one for $g_2^b = B$).

B Succinct Arguments of Knowledge for VDS

All the protocols below are for simplicity presented for the case of $k = 1$.

AoK of correct change

$$R_{\mathsf{PoKChange}} = \left\{ \begin{array}{r} (\,(C, C', I), (\pi_I, \boldsymbol{v}_I, \boldsymbol{v}'_I)\,) : \mathsf{VC.VerUpdate}(\mathsf{crs}, C, (I, \pi_I, \boldsymbol{v}_I, \boldsymbol{v}'_I)) = 1 \\ \wedge C' = \mathsf{VC.ComUpdate}(\mathsf{crs}, C, (I, \pi_I, \boldsymbol{v}_I, \boldsymbol{v}'_I)) \end{array} \right\}$$

In case of an update the new commitment is normally $C' := (A', B') = (\Gamma_I^{b_I}, \Delta_I^{a_I})$. Therefore the prover first sends the proof $\pi_I := (\Gamma_I, \Delta_I)$ to the verifier. Then provides knowledge of the opening of positions I with respect to C and further that $\Gamma_I^{b_I} = A' \wedge \Delta_I^{a_I} = B'$. Putting all together the prover proves knowledge of (a_I, b_I) such that $\Gamma_I^{a_I} = A \wedge \Delta_I^{b_I} = B \wedge g^{a_I \cdot b_I} = U_I \wedge \Gamma_I^{b_I} = A' \wedge \Delta_I^{a_I} = B'$, where $U_I \leftarrow g^{u_I}$ and $u_I \leftarrow \mathsf{PrimeProd}(I)$.

AoK of correct add

$$R_{\mathsf{PoKAdd}} = \left\{ \begin{array}{r} (\,(C, C', I), \boldsymbol{v}'_I\,) : \mathsf{VC.VerUpdate}(\mathsf{crs}, C, (I, \varnothing, \varnothing, \boldsymbol{v}'_I)) = 1 \\ \wedge C' = \mathsf{VC.ComUpdate}(\mathsf{crs}, C, (I, \varnothing, \varnothing, \boldsymbol{v}'_I)) \end{array} \right\}$$

The prover provides an argument of knowledge of (a'_I, b'_I) such that $A^{a'_I} = A' \wedge B^{b'_I} = B' \wedge g^{a_I \cdot b_I} = U_I$, where $U_I \leftarrow g^{u_I}$ and $u_I \leftarrow \mathsf{PrimeProd}(I)$. Also, $C := (A, B)$ and $C' = (A', B')$ are part of the statement.

AoK of correct delete

$$R_{\mathsf{PoKDelete}} = \left\{ \begin{array}{r} (\,(C, C', I), (\pi_I, \boldsymbol{v}_I)\,) : \mathsf{VC.VerUpdate}(\mathsf{crs}, C, (I, \pi_I, \boldsymbol{v}_I, \varnothing)) = 1 \\ \wedge C' = \mathsf{VC.ComUpdate}(\mathsf{crs}, C, (I, \pi_I, \boldsymbol{v}_I, \varnothing)) \end{array} \right\}$$

Recall that in case of deletion the new commitment C' is simply the proof π_I of the subvector deleted. So the prover has only to provide an argument of knowledge of opening in the deleted positions I. That is (a_I, b_I) such that $A'^{a_I} = A \wedge B'^{b_I} = B \wedge g^{a_I \cdot b_I} = U_I$, where $U_I \leftarrow g^{u_I}$ and $u_I \leftarrow \mathsf{PrimeProd}(I)$. Also, $C := (A, B)$ and $C' = (A', B')$ are part of the statement.

C Experimental Results

In this section we include complete tables and plots for our benchmarks.

In some of the tables and plots we show only results for openings of size at most 25% of the vector size as this is often the case in practice. We remark that the timings for verification of our VC construction and BBF do not use proofs of knowledge of exponent, thus both timings can in practice be reduced through the use of this technique. Finally, although we show amortized openings (Figure 6) for only openings of size 2048 bits, we stress that different choices of file and opening size show very similar patterns.

We exclude BBF with precomputation from our experiments as its storage requirements and running times dominate those of our construction with preprocessing. In terms of storage it is linear in the number of the bits in the vector. For our choice of security parameters and block size, it would require 3× more memory independently of the size of the vector. In terms of time, the running times of BBF with preprocessing always dominate those of our preprocessing scheme. Concretely opening and verification of each zero bit requires one more group exponentiation. Finally, the lack of incremental aggregation makes this scheme less flexible than ours as it does not allow to choose different tradeoffs in terms of memory/running time.

The Experimental Setting We implemented our VC, its preprocessing variant and BBF[22] in Rust. We executed our experiments on a virtual machine running Debian GNU/Linux with 8 Xeon Gold 6154 cores and 30 GB of RAM.

We measured running times for the commitment stage (including or not a preprocessing), opening and verification for different choices of vector length (N) and subvector openings (m). Vectors have blocks of $\ell = 256$ bits (which is representative of vectors where blocks are hash outputs) and their total size $n = N\ell$ range from 16 kibibit (Kibit) to 1 mebibit (Mibit)[23]. For preprocessing we considered the basic case in which we precompute one proof per block, i.e., a total of n/ℓ proofs is precomputed. We chose m, the opening size to be of 1, 8 or 64 blocks (i.e. 256, 2048 or 16536 bits). On security parameters: our experiments always used an RSA modulus of 2048 bits and primes of 64 bits for accumulation.

[22] `https://github.com/nicola/rust-yinyan`

[23] 1 Kibit = 2^{10} bits; 1 Mibit = 2^{10} Kibit. We choose powers of two for convenience.

n (file size in bits)	**Running Time**
16 384	52s
32 768	1m 56s
64 536	4m 23s
131 072	10m 7s
262 144	24m 5s
524 288	1h 1m
1 048 576	2h 54m

Table 3. Commitment times for our preprocessing construction (block size $\ell = 256$).

n (size in bits)	**This work**	**BBF**
16 384	18s	3s
32 768	37s	8s
64 536	1m 19s	18s
131 072	3m 0s	45s
262 144	7m 22s	2m 29s
524 288	20m 12s	7m 8s
1 048 576	1h 10m	29m 54s

Table 4. Commitment Times (no preprocessing)

n (size in bits)	**This work (precomp.)**	**This work**	**BBF**
16 384	$2 \cdot 10^{-4}$	5.56	5.86
32 768	$2 \cdot 10^{-4}$	11.17	11.69
64 536	$2 \cdot 10^{-4}$	22.44	23.26
131 072	$2 \cdot 10^{-4}$	44.68	45.49
262 144	$2 \cdot 10^{-4}$	88.98	90.72
524 288	$2 \cdot 10^{-4}$	178.94	184.86
1 048 576	$2 \cdot 10^{-4}$	357.50	370.82

Table 5. Opening Times (in s) for openings of 256 bits

n (size in bits)	**This work (precomp.)**	**This work**	**BBF**
16 384	4.27	5.70	7.96
32 768	4.27	11.34	14.75
64 536	4.27	22.84	28.10
131 072	4.27	45.44	54.17
262 144	4.27	91.45	108.69
524 288	4.27	182.29	222.47
1 048 576	4.27	362.50	453.28

Table 6. Opening Times (in s) for openings of 2048 bits

n (size in bits)	**This work (precomp.)**	**This work**	**BBF**
64 536	73.57	25.96	66.16
131 072	73.57	52.42	122.68
262 144	73.57	104.63	238.07
524 288	73.57	210.40	521.89
1 048 576	73.57	423.48	1100.10

Table 7. Opening Times (in s) for openings of 16384 bits

$m \cdot \ell$ (opening in bits)	**This work**	**BBF**
256	3.31	7.72
2048	8.97	13.28
16384	309.82	314.28

Table 8. Verification Times (in ms)

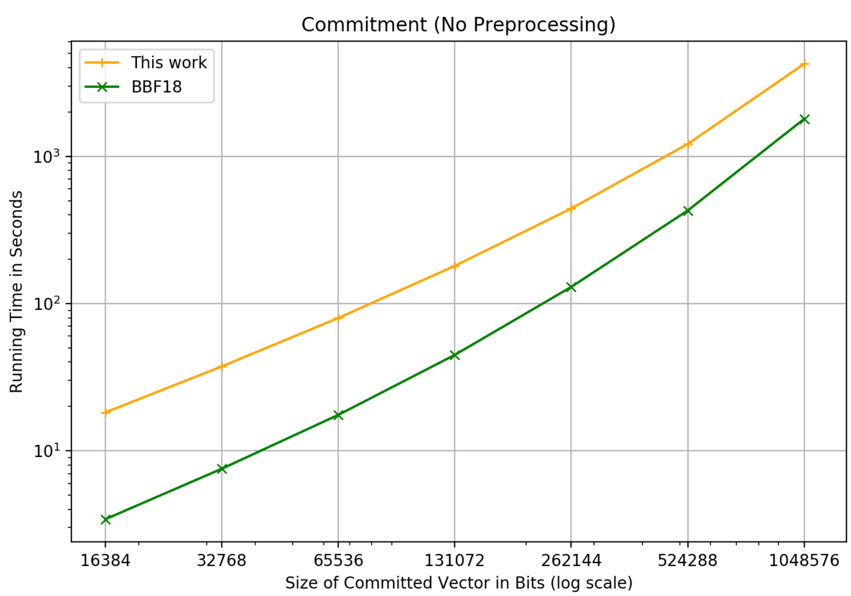

Fig. 4. Commitment Experiments

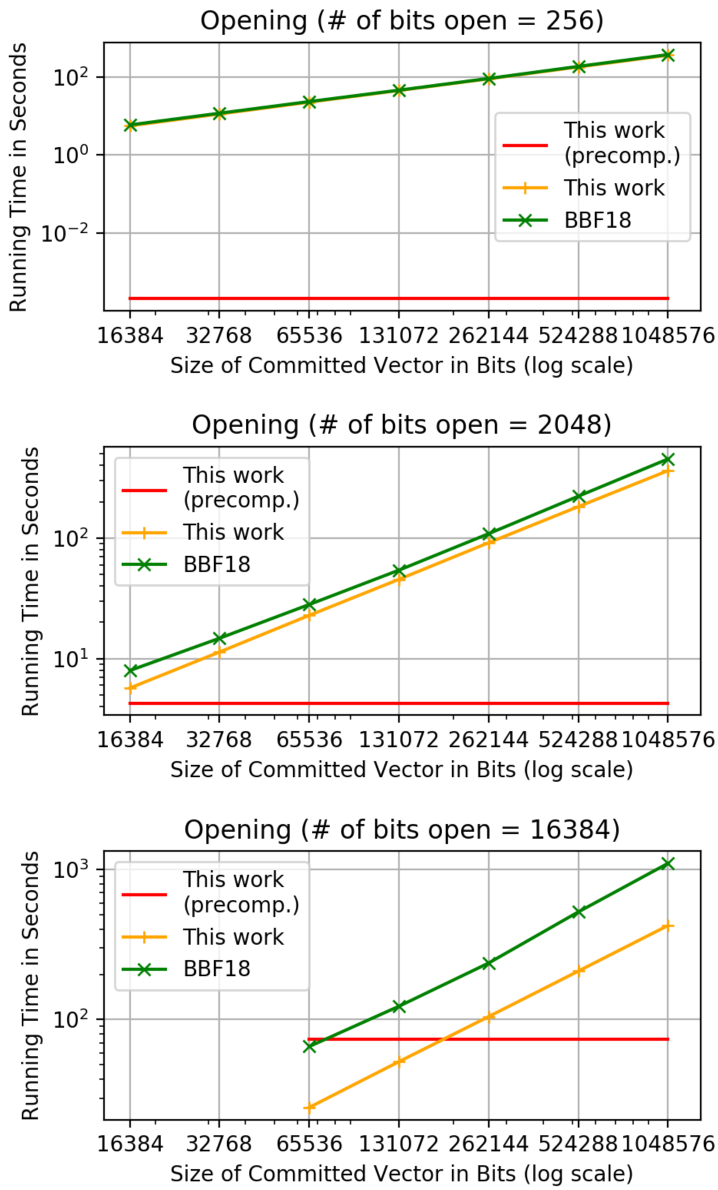

Fig. 5. Opening Experiments

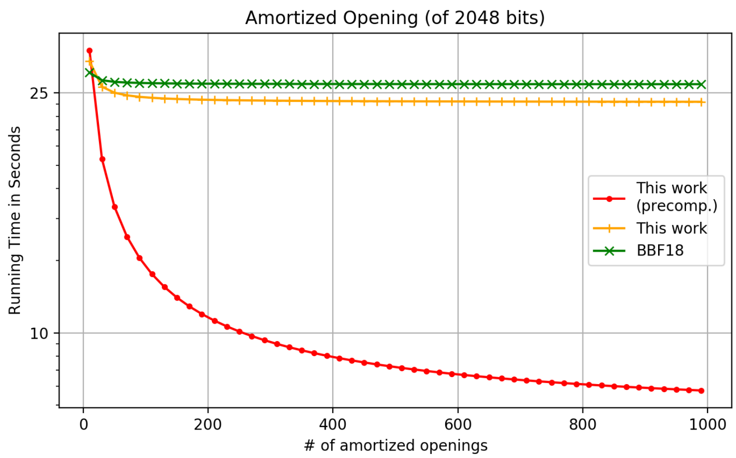

Fig. 6. Amortized Opening Experiments for a file of size 128 Kibib.