A Post Quantum Vector Commitment Scheme with Efficient Insertions and Deletions

Vir Pathak[1][0009−0008−4597−514X], Sushmita Ruj[2][0000−0002−8698−6709], and

[1] Department of Computer Science
Stony Brook University, USA
vir.pathak@stonybrook.edu
[2] School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
sushmita.ruj@unsw.edu.au

Abstract. Vector Commitments (VCs) are a cryptographic primitive that allows users to commit to a tuple and later provide proofs about entries of the committed tuple. Literature on VCs provides many secure constructions with properties attractive for decentralized applications. These include fast commitment/proof update algorithms after a value in the committed vector changes, algorithms to aggregate position proofs, generating proofs for subvectors, commitment hiding, and many more. However, these works assume a static setting, where only vector values change and the length of the committed vector never changes. We initiate the study of designing VCs which support efficient updates to commitments and proofs after a change in vector length occurs (i.e. an insertion or a deletion of a vector value at an *arbitrary* position). In order to construct such a primitive, we formalize a VC framework implicitly considered in VC based verifiable decentralized storage constructions. Parties control a subset of positions in the committed vector. A proof for an inserted vector value at a given position may only be generated via participation from enough honest parties. We call such primitives *Multiparty Dynamic Vector Commitments*. Our work provides formal definitions for multiparty dynamic VC functionality and security. In particular, standard VC binding does not extend to our dynamic setting. This motivates us to define the *dynamic multiparty binding* security notion. We also comment on how multiparty dynamic VCs can be applied to extend existing VC based instantiations of verifiable storage. Finally, we provide a vector commitment construction and show that it possesses multiparty dynamic binding under standard lattice assumptions.

Keywords: Vector Commitment · Dynamic · Lattice-based

1 Introduction

Vector commitments were proposed by Catalano and Fiore [7] and Libert and Yung [14]. We use VCs to generate a commitment $\mathbf{c}$ to a vector $\mathbf{x} \in \mathcal{M}^\ell$ in some space $\mathcal{M}$. Later, a user who knows the value x_i of the ith entry of $\mathbf{x}$ can generate

a proof convincing a verifier that the ith entry of the vector $\mathbf{x}$ committed in $\mathbf{c}$ is indeed x_i. VCs come with a security guarantee known as *position binding*, which informally guarantees that a malicious prover cannot produce two proofs π_i and π_i' convincing a verifier that the ith position in $\mathbf{x}$ is x_i and x_i' respectively (with $x_i \neq x_i'$).

In recent years, VCs have generated a great deal of attention. New functionalities have been proposed, such as subvector proofs [13], aggregating position proofs [25], efficiently updating all proofs after an entry changes [23], cross-aggregating proofs [11], and many more. These properties were introduced as a way to seamlessly instantiate applications such as SNARK construction, stateless blockchain design, or verifiable databases. Our work is motivated by this last use case. Instantiating a verifiable database via VCs has been long been studied in a large body of VC constructions [7] [4]. The main idea is as follows: assume a client has a large file which can be modeled as a vector. The file is to be stored in a storage node. The client will keep a commitment to this file and the storage node will store the actual file contents along with the corresponding position proofs. When the client queries for a certain portion of the file, the node presents it to the client, along with the position proofs or a corresponding subvector proof. To make sure the storage node presented the correct values, the client can run the VC verification algorithm on these values and proofs against the commitment which the client stores. A related notion is verifiable decentralized storage via VC, [4], where multiple storage nodes store different chunks of the file. This corresponds to a large vector being distributed into subvectors for each node in the network to store. A client again stores a commitment to the entire vector. When queried for contents in a node's designated subvector, the node can provide those contents, along with proofs establishing their veracity.

Unfortunately, VC designs for these use cases assume a static model, where information cannot be added or deleted. In all of the aforementioned use cases, we can envision a dynamic situation where new content can be added and old content can be removed from the committed vector. Therefore, we would like to design vector commitment schemes which can work in a such a setting. Namely, we would like to design schemes that can support efficient commitment/proof updates in the case of an insertion/deletion of a value to the committed vector.

Adding *stateless* insertion to most constructions is difficult. To see why, consider the problem of inserting a value at position i and updating the commitment/existing position proofs. Since all values after position i get pushed 1 position over, we must know all values in the vector after position i. Furthermore, it is not clear how to insert securely. In particular, we demonstrate a construction that satisfies standard VC binding (assuming standard worst case lattice problems) and proceed to add insertion/deletion algorithms which compromise the scheme's security.

Our construction works in a new *multiparty* based VC model we propose. Briefly, we assume that various parties will control different subvectors of the committed vector v. We further assume that no pair of parties P_i, P_j will have an entry in common in the subvectors they control. In our multiparty model, we

propose a threshold version of standard VC binding (in addition to modifications needed to capture binding in a dynamic setting). For security reasons, it seems the multiparty model is necessary for our construction. Parties generate partial proofs which can be reconstructed to generate a full position proof. In the Appendix (see 6), we show why the analogous single party version of our scheme is insecure. In this way, we are able to show a separation between standard binding VCs and VCs which are binding in a dynamic setting.

Apart from the security reasons, multi-party setting is required in many applications. For example, in decentralized verifiable database protocols parties store parts of the database and insert or delete content from the database. Such cases require parties to communicate about values getting inserted or deleted, thus maintaining the updated state of the database. Therefore requiring participation from multiple nodes in the protocol for a valid insertion is not a huge cost. We have worked in a multiparty model that provides a reasonable security claim for our construction. However, we do understand that for some applications requiring the participation of multiple parties to do insertion is clearly an unsatisfactory solution.

1.1 Our Results

This work presents the following contributions:

1. We introduce multiparty dynamic vector commitments. We provide formal definitions describing the new functionalities.
2. We show that the standard VC binding security notion is not sufficient for this new dynamic setting. Therefore, we introduce *multiparty dynamic binding*, which captures the security notion for VCs in the setting we consider.
3. We design a novel multiparty dynamic VC and prove security under standard lattice assumptions. Our scheme supports updates after insertions/deletions, as well as modifications to existing entries.
4. We discuss how such a VC design can be leveraged in improving existing VC based verifiable storage protocols.

While our VC design allows clients to perform insertion/deletion updates, we note that insertion requires multiple parties to participate due to the partial lattice trapdoor techniques we use. Existing instantiations of verifiable storage via VC cannot handle requests to enlarge or shrink the file via insertions or deletions, as the underlying VCs do not possess such functionality. One exception is the verifiable decentralized storage solution proposed by Campanelli et. al [4], which does allow for insertions and deletions. However, their solution does not allow for insertions or deletions at arbitrary positions in the vector. Furthermore, their VC security is based on groups of unknown order. In contrast, ours is lattice based and therefore enjoys *post quantum* security in the random oracle model. Our construction only requires modifying proofs' indexing for positions affected after insertion/deletion. Nontrivially modifying all position proofs after such an update is costly. Since we are also able to trivially modify an existing entry

by deleting it and subsequently inserting the modified value, our scheme also achieves *maintainability* [23] (i.e. the ability to update *all* position proofs in time sublinear in vector length after an entry change). We surpass the maintainable state-of-the-art [23] [5].

Our scheme's main drawback is that requires the participation of multiple parties for insertion. However, the multiparty protocol only requires a single round of communication. We consider designing VCs supporting updates of entry values, insertions, and deletions without requiring the participation of multiple parties as an important open problem.

Let us briefly clarify on our usage of the word "dynamic". We emphasize that dynamism previously referred to notions only involving updates to commitments/proofs after *updating existing vector values*. That is, to the best of our knowledge, existing dynamic vector commitment schemes like [24] only support fast updates to commitments and proofs following an alteration of one of the existing vector entries. Our notion of dynamism significantly differs from this, since we require fast updates to commitments and proofs after insertions/deletions of vector values at arbitrary positions instead of after modifications of existing entries. To the best of our knowledge, designing VC schemes which support such changes in the vector length have not been explored. Campanelli et. al [4] modify a VC scheme in order to support this functionality for a decentralized storage construction . However, one can only insert/delete at certain spots in the vector. Moreover, our work is done in the lattice setting, while their security is based on computational problems in groups of unknown order, showing that dynamism can be achieved from multiple assumptions.

We note that this notion of "dynamism" appears in the context of proofs of retrievability [22,21] and proofs of data possession in [9,20].

1.2 Technical Overview

A Multiparty Model The decentralized storage framework serves as a blueprint for ours. In our setting, we allow a fixed set of parties $\{P_1, \ldots, P_k\}$ to control values at designated positions in the vector. A party P_i will be responsible for the values at the indices in S_{P_i}, and $S_{P_i} \cap S_{P_j} = \emptyset$ for $i \neq j$. Under this setting, we are able to construct a scheme with secure insertion and deletion functionalities. We note that if a party P_i requests insertion of a value a, at a position j, that results in all entries at or after j getting pushed forward by one. Moreover, S_{P_i} gets updated to $S_{P_i} \cup \{j\}$. Similarly, if P_i requests for a deletion at j, values after j get pushed back by one and S_{P_i} gets updated to $S_{P_i} \setminus \{j\}$. Therefore we must take care to update all parties' index sets, in addition to the commitment and existing position proofs after such an update. Finally, if a party P_i wants to update or delete a value at position j, we must first check that P_i actually controls this position (namely $j \in S_{P_i}$).

When designing a VC scheme, we imagine a global commitment available to all parties. When a party wants to insert or delete an entry, it may require help from a sufficient number of other parties. Consider a VC scheme which initially commits to vectors in $\mathbb{Z}_q^\ell$ for a prime q. Let $\mathbf{x}$ be such a vector and let $\mathbf{c}$ be a

commitment to $\mathbf{x} = (x_1, \ldots, x_\ell)$. The goal of this work is to introduce the notion of multiparty *dynamic* vector commitments. To accomplish this, we define the following algorithms: `Add, Del, AddPfs, DelPfs`. A party P_i invokes the `Add` algorithm in order to insert a new value a at a position j. If a sufficient number of other parties are willing to participate, the algorithm generates a new proof for the inserted value and updates the commitment $\mathbf{c}$ to $\mathbf{c}'$ which is a commitment to $\mathbf{x}' = (x_1, \ldots, x_{i-1}, a, x_i, x_{i+1}, \ldots, x_\ell) \in \mathbb{Z}_q^{\ell+1}$. The `AddPfs` algorithm efficiently updates all position proofs after such an insertion occurs. The `Del` and `DelPfs` algorithms are analogous to `Add` and `AddPfs` respectively in the case of a deletion of an entry, rather than an insertion.

We acknowledge that a multiparty model is not an ideal solution for achieving a VC scheme with efficient insertions and deletions. However we believe it is useful for instantiating decentralized verifiable database solutions. Assume a party in a decentralized network wants the help of other nodes in the network to store a large file. We assume the file can be modeled as a vector v. As with the decentralized verifiable database setting, these nodes will store subvectors of the file. Let us also stipulate that no two nodes can store a common portion of the file. That is, no two nodes can store a common subvector of v. Furthermore, assume that the file can be changed by modifying/inserting/deleting arbitrary entries. Then multiparty dynamic VCs present a natural solution: the party owning the large file initially can generate a commitment to it and distribute subvectors to participating nodes, who store the subvectors and their corresponding position proofs. Any node may propose an insertion/deletion, and others may help generating a proof for the inserted element and updating the commitment/ existing position proofs after these operations. An insertion protocol requiring the participation of multiple nodes is a drawback. However communication between nodes during insertion is inevitable, as all nodes must be in agreement on the positions they control in any real world deployment of the scheme.

Construction Adding efficient `Add, Del, AddPfs`, and `DelPfs` algorithms is seemingly challenging due to the following observation: after an insertion or deletion at some index i, all vector entries at or after/before i get shifted by one position. Therefore, after an insertion/deletion, we need to update all position proofs after i. If a vector has length ℓ, updating all the proofs after an insertion/deletion will take $O(\ell)$ time in the worst case if we just use naive proof update functionalities which already exist. In this work, we show how to improve upon this naive upper bound.

Our scheme builds on top of a scheme due to Wee and Wu [27]. The Wee-Wu scheme can be thought of as follows: it begins by generating ℓ statistically close to random matrices $(A_i)_{i\in\{1,\ldots\ell\}}$ with each $A_i \in \mathbb{Z}_q^{n\times m}$ along with corresponding trapdoors. Using the trapdoors, a user can jointly sample a statistically close to random commitment $\mathbf{c} \in \mathbb{Z}_q^m$ for $\mathbf{x} \in \mathbb{Z}_q^\ell$ along with ℓ "short" position proofs $(\mathbf{v}_1, \ldots \mathbf{v}_\ell)$. The position proofs are sampled in a way relating A_i, $\mathbf{c}$, x_i, and $\mathbf{v}_i$ in a matrix-vector equation for each i. Verification of a position proof amounts to checking $\mathbf{v}_i$ is short and certain conditions hold.

Table 1. Comparison of selected VC schemes. p is short for *poly*, λ is the security parameter, and ℓ is the length of vectors being committed. The scheme due to Campanelli et al. [4] can support updates after insertions/deletions to the committed but only at certain positions. Our construction supports insertion/deletion at arbitrary positions.

Scheme	Security	Ins/Del	$\|pp\|$	$\|\pi\|$	$\|c\|$
Tomescu et. al. [25]	classical	no	$O(\ell)$	$\|\mathbb{G}\|$	$\|\mathbb{G}\|$
Lai et. al. (LMC) [13]	classical	no	$O(\ell^2)$	$\|\mathbb{G}\|$	$\|\mathbb{G}\|$
Wee et. al. [26]	post quantum	no	$O(p(\lambda,\ell))$	$O(p(\lambda,\log\ell))$	$O(p(\lambda,\log^2\ell))$
Campanelli [4]	classical	yes*	$\|\mathbb{G}\|$	$O(1)\|\mathbb{G}\|$	$4\|\mathbb{G}\|+2\|\mathbb{Z}_{2^{2\lambda}}\|$
Our Construction	post quantum	**yes**	$O(p(\lambda,\ell))$	$O(p(\lambda,\log\ell))$	$O(p(\lambda,\log^2\ell))$

If we want to insert a value a at the ith position we can sample a new random verification key A'. Using lattice trapdoor preimage sampling techniques, we can obtain a preimage which can be used as a short proof in the verification equation. In this way, the old commitment c along with the old position proofs do not have to be modified. However the existing verification keys must be reordered to accommodate the insertion. Deleting a value at a position i simply amounts to deleting a verification key A_i and reordering the remaining verification keys.

The key property about the Wee-Wu scheme is that commitments and position proof are not tied to value-position pairs (x_i, i). This is in contrast to many existing vector commitment schemes. For example in Hyperproofs [23], computing a commitment to a vector $\mathbf{x} = (x_1, \ldots, x_\ell) \in \mathbb{Z}_q^\ell$ means computing $g^{\sum_{i=1}^{\ell} x_i \cdot S_i(\tau)}$, where S_i is a multilinear Lagrange basis polynomial. We are given commitments $g^{S_i(\tau)}$ as public parameters. To compute the above value, we must take each $g^{S_i(\tau)}$ and raise it to the value x_i. Then we must combine all of these quantities using group multiplication. Trying to insert a new value x_i' at position i means we must now compute $g^{S_j(\tau)^{x_{j-1}}}$ for all $j \geq i$. In the worst case, this will take $O(\ell)$ group exponentiations and group multiplications. The source of this slowdown was that the process to generate the commitment to x implicitly depends on the position i along with the vector value x_i. This is why we suspect VC schemes based on polynomial commitments are unlikely to support efficient updates after entries are inserted to/deleted from committed vectors.

One issue with our framework is that allowing anyone to generate the new verification key and proof is insecure because this process introduces a lattice trapdoor which can be used to break the binding of the scheme. In this way, we show that standard VC binding does not immediately imply binding in a dynamic setting. In order to address this and present a scheme allowing for reasonably secure insertion and deletion, we introduce our multiparty dynamic framework. We call the corresponding security notion *multiparty dynamic binding*. This notion captures essentially the same concept as standard binding, except now the adversary can make queries for insertions and deletions before presenting a commitment and two conflicting values/proofs which pass verification. Additionally, our scheme operates in a multiparty setting where we assume that an adver-

sary cannot corrupt more than a certain threshold number of parties controlling positions in the vector.

We make use of the recent lattice trapdoor splitting techniques developed by Albrecht et. al [3]. A *partial lattice trapdoor scheme* allows a user to take a standard lattice preimage sampling trapdoor with respect to an input matrix A and "split" it among multiple parties. Each party receives a partial trapdoor. When the parties want to compute a short preimage for some target y with respect to the matrix A, they can use their partial trapdoors to compute a partial preimage. Subsequently, they can recombine all their preimages to get a short, random preimage x for the target y with respect to A. The scheme guarantees that an adversary cannot generate such a preimage if it cannot corrupt at least a threshold number of parties. Our strategy for secure insertion is the following: we sample extra matrices $\{A'_1, \ldots, A'_n\}$ to be used as verification keys when a party wants to insert a value. For each A'_i, we give party P_j a partial trapdoor ptd_{i_j}. If P_j wants to insert some x', it, along with enough other participating parties, generates partial preimages with respect to some insertion key A'_i and target $H(x')$, where H is a random oracle. At verification time, these partial preimages are recombined into a short preimage h such that $A'_i \cdot h = H(x')$ and h is short. Batch insertions/deletions are also possible without increasing the number of rounds. Deletion is straightforward. For insertion, we execute the same process with a sufficient number of the desired insertion matrices. Parties would compute all the partial preimages locally at once.

A Note on Real World Deployment We now consider how deploying a scheme matching our framework would work in a real world decentralized system. While our public parameter size is linear in the initial vector length plus the number of possible insertions, the advantage of our work is that modifying the actual commitment or all of the existing proofs is as cheap as possible during insertions or deletions.

We must treat the set of verification keys and their indices at any given time as part of the *state* of the system. Note that a possible attack vector for an adversary of the system is to convince network participants into believing conflicting states. For example, assume the current state's verification keys are (A_1, A_2, A_3). Then after an insertion at position two, they become (A_1, A'_2, A_2, A_3). If a participant is made aware of the insertion, the position of insertion, the new key A'_2 for the second index, as well as the previous state's verification keys (A_1, A_2, A_3) they will be able to correctly verify a position proof. However, if one of these pieces of information are not known after an insertion, then a participant is prone to incorrectly verifying proofs with respect to the current state of the system. Therefore a real world deployment must modify the underlying consensus algorithm the participants use to agree on the condition of the state. The consensus must now include agreement upon the set of the verification keys as well as their indices.

We must also note that while our VC captures *binding* in a traditional sense, it is not robust. That is, if some party P_1 wants to insert a value x' at some

position and parties $P_2, \ldots, P_k$ are willing to participate, then an incorrect proof for x' can be generated if one of these parties is malicious and deviates from the protocol. Moreover, all parties must know the value x' that P_1 wants to insert in order to generate the correct local preimages. This will incur a communication overhead, although there is only a single round of communication.

1.3 Related Work

Vector commitments were proposed by Catalano and Fiore [7]. Along with formally defining the primitive, they showed how VCs could be applied to applications such as verifiable databases and zero knowledge elementary databases.

An important property is the ability to generate proofs for arbitrary *subvectors* in the committed vector. Subvector openings were first introduced by Lai and Malavolta [13], who proposed a new SNARK design technique with VCs with subvector openings along with PCPs as the underlying mechanisms. This was a direct improvement of the classical "CS Proofs" (Computationally Sound Proofs)[15] paradigm due to Killian [12]. Knowledge of subvector openings along with proof aggregation/disaggregation techniques are instrumental building blocks in the decentralized storage protocol in [4].

Classical constructions geared towards other decentralized applications quickly emerged, such as stateless blockchains [8] and verifiable decentralized storage [4]. In order to efficiently instantiate such applications via VCs, it is critical to build VCs which could support efficient updates after a specific *vector value* was altered. The authors of the Hyperproofs VC scheme [23] defined maintainability, which was the property of being able to update all position proofs in *sublinear* time after a change in one of the committed vector entries. By noticing a tree structure in PST polynomial commitments [18], Hyperproofs achieves maintainability with $O(\log \ell)$ update time, where ℓ is the length of the committed vector. Their technique was generalized and shown to apply for any pairing based VC scheme with various algebraic properties [5]. Surveys on VC appear in [17,19].

Designing lattice based VCs is a topic currently receiving increased interest. Albrecht et al. design a scheme which opens to low degree polynomials as a way to design a lattice based SNARK [2]. However, their scheme relies on a nonstandard knowledge assumption, which was later shown to be insecure in a work by Wee and Wu [26]. De Castro and Peikert design transparent functional commitments [6], which can be efficiently specialized to VCs. Finally, Wee and Wu [27] design a VC whose security reduces to SIS. Our work uses this construction as a building block. We keep the same commit and prove functionalities and augment the original `Setup` procedure to allow for efficient updates after insertions or deletions.

2 Preliminaries

2.1 Notation

We use λ for the security parameter. The letter q denotes a prime and $\mathbb{Z}_q$ denotes the integers modulo q. A capital letter such as A or B will denote a matrix,

whereas a lowercase letter like x or a will denote a value (usually a value in $\mathbb{Z}_q$). A bolded letter (like $\mathbf{x}$) denotes a vector.

Definition 1 (Negligible Function). *A function $f : \mathbb{N} \to \mathbb{R}$ is said to be in the class of negligible functions (written as $f \in negl(\lambda)$) if for all $c \geq 0$ there exists $n_0 \in \mathbb{N}$ such that for all integers $n \geq n_0$, we have $|f(n)| < \frac{1}{n^c}$.*

That is, a negligible function is a function which decreases faster than the inverse of any polynomial.

Definition 2. *Let λ be a parameter. The class $poly(\lambda)$ denotes the functions which are bounded from above by a polynomial in λ.*

Definition 3. *We say a probabilistic algorithm is in the class Probabilistic Polynomial Time (PPT) if its expected running time is bounded from above by a polynomial in its input size.*

Definition 4. *We say that two families of distributions $\{D_1\}_{\lambda\in\mathbb{N}}$ and $\{D_2\}_{\lambda\in\mathbb{N}}$ are statistically close if their statistical distance is bounded by a negligible function in λ.*

For a matrix or a vector, we denote $||\cdot||$ as its infinity norm.

2.2 Lattice Preliminaries

Definition 5 (Lattice). *Let B be a full rank matrix in $\mathbb{R}^{k\times k}$. The lattice $\mathcal{L}(B)$ generated by B is the set of all integer linear combinations of the columns in B. That is, $\mathcal{L}(B) = \{B\mathbf{x} : \mathbf{x} \in \mathbb{Z}^k\}$.*

Definition 6 (SIS Assumption). *Let $n, m \in \mathbb{Z}_{>0}, \beta \in \mathbb{R}_{>0}, q \in \mathbb{Z}_{>0}$ be functions in the security parameter λ. The $SIS_{n,m,q,\beta}$ assumption states that a for any efficient adversary $\mathcal{A}$, the probability of $\mathcal{A}$ winning the following game with a challenger is negligible in λ: (1). The challenger samples $A \leftarrow \mathbb{Z}_q^{n\times m}$ and sends it to $\mathcal{A}$. (2). $\mathcal{A}$ outputs a vector $\mathbf{x} \in \mathbb{Z}_q^m$.*
$\mathcal{A}$ wins if $A\mathbf{x} = 0 \pmod q$ and $||\mathbf{x}|| \leq \beta$.

The Shortest Integer Solution (SIS) problem has been shown [1] to be at least as hard as standard problems in the worst case related to finding short vectors in lattices. We recall the definition of a discrete Gaussian, following Wee and Wu's [27] exposition.

Definition 7 (Discrete Gaussian). *Let $s \in \mathbb{R}_+$ and $n \in \mathbb{Z}_{>0}$. We define the Gaussian function of width s to be*

$$\rho_s(\mathbf{x}) = exp(-\pi||\mathbf{x}||_2^2/s^2).$$

We can define a probability distribution on lattice cosets. For a coset $\mathbf{c} + \mathcal{L}$, we define the mass function to be

$$\rho_s(\mathbf{c} + \mathcal{L}) = \sum_{\mathbf{x}\in\mathcal{L}} \rho(\mathbf{c} + \mathbf{x}).$$

Define the discrete Gaussian distribution on $\mathbb{Z}^n$ *by*

$$D_{\mathbb{Z}^n,s}(\mathbf{x}) = \rho_s(\mathbf{x})/\rho_s(\mathbb{Z}^n).$$

For $s > 0$, $A \in \mathbb{Z}_q^{n\times m}$, and $\mathbf{v} \in \mathbb{Z}_q^n$ we write $A_s^{-1}(\mathbf{v})$ to denote a random variable $\mathbf{x}$, where $\mathbf{x} \leftarrow D_{\mathbb{Z}^m,s}$ conditioned on $A\mathbf{x} = \mathbf{v} \pmod q$. We sample matrices in an analogous way by applying A_s^{-1} to all the columns of the input matrix.

Definition 8 (Gadget Matrix). *Let* $n, q \in \mathbb{Z}_{>0}$. *Let* $m' = n\lfloor \log q \rfloor + 1$. *The corresponding gadget matrix is defined as* $G = I_n \otimes g^T \in \mathbb{Z}_q^{n\times m'}$. *Here,* $g^T = [1, 2, \ldots, 2^{\log q}]$.

Next, we recall the trapdoor and preimage sampling algorithms due to Micciancio and Peikert [16]. We again follow Wee and Wu's exposition.

Theorem 1 (Trapdoor Preimage Sampling). *Let* n, m, q, m' *be as before. There exists efficient algorithms (TrapGen, SamplePre) which work as follows:*

1. *TrapGen*$(1^n, q, m) \to (A, td)$*: Takes matrix dimensions* n *and* m *along with the modulus* q *as input. The algorithm generates an* $n \times m$ *matrix over* $\mathbb{Z}_q$ *along with a trapdoor* $td \in \mathbb{Z}_q^{m\times m'}$.
2. *SamplePre*$(A, td, \mathbf{v}, s) \to x$*: Takes a matrix* $A \in \mathbb{Z}_q^{n\times m}$ *along with its trapdoor* $td \in \mathbb{Z}_q^{m\times m'}$, *an image* $\mathbf{v} \in \mathbb{Z}_q^n$ *and a Gaussian width parameter* s *as input. The algorithm outputs a vector* $\mathbf{u} \in \mathbb{Z}_q^m$.

For $m \geq O(n \log q)$, *these tuple of algorithms have the following properties:*

1. *The output* (A, td) *of the TrapGen algorithm is such that* $A{\cdot}td = G$, $||td|| = 1$, *and the statistical distance between* A *and a uniformly sampled matrix* A' *over* $\mathbb{Z}_q^{n\times m}$ *is less than or equal to* 2^{-n}.
2. *For all* $td \in \mathbb{Z}_q^{m\times m'}$, *all* $\mathbf{v}$ *in the column span of* A, *the vector* $\mathbf{x}$ *is such that* $A\mathbf{x} = \mathbf{v}$.
3. *If* td *is a gadget trapdoor for* A *(e.g. if* (A, td) *is the output of TrapGen), then for all* $s \geq \sqrt{mm'} \cdot ||td|| \cdot \omega(\sqrt{\log n})$, *the statistical distance between*

$$\{\mathbf{x} \leftarrow \mathit{SamplePre}(A, td, v, s)\} \quad \text{and } \{\mathbf{x} \leftarrow A_s^{-1}(\mathbf{v})\}$$

less than or equal to 2^{-n}.

We can extend the SamplePre targets to include matrices $V \in \mathbb{Z}_q^{n\times k}$ *in the following way: We define SamplePre*(A, td, V, s) *to be the matrix whose* i*th column is SamplePre*$(A, td, \mathbf{v_i}, s)$, *where* $\mathbf{v_i}$ *is the* i*th column of* V.

We now proceed to state the $\text{BASIS}_{\text{rand}}$ assumption [27].

Definition 9 (BASIS). *Let* n, m, q, s *be as before chosen with respect to the security parameter* λ. *Let Samp be an efficient sampling algorithm that takes* λ *and* $A \in \mathbb{Z}_q^{n\times m}$ *as input. The algorithm outputs a matrix* $B \in \mathbb{Z}_q^{n'\times m'}$ *along with some auxiliary information aux. We say the* $BASIS_{rand}$ *Assumption holds if all efficient adversaries* $\mathcal{A}$ *win the following game between* $\mathcal{A}$ *and a challenger Chal with negligible probability:*

1. *Chal selects* $A \leftarrow \mathbb{Z}_q^{n \times m}$, *and runs* $(B, aux) \leftarrow Samp(\lambda, A), T \leftarrow B_s^{-1}(G_{n'})$. *He sends* (A, B, T, aux) *to* $\mathcal{A}$.
2. $\mathcal{A}$ *outputs a vector* $\mathbf{x}$.

$\mathcal{A}$ *wins if* $A\mathbf{x} = 0 \pmod q$ *and* $0 < ||\mathbf{x}|| \leq \beta$.

Definition 10 ($\mathbf{BASIS_{rand}}$). *Let* n, m, q, s, β *be as before. Let* $\ell \in \mathbb{Z}_{>0}$. *When Samp is instantiated in the following way, we refer to the corresponding assumption as the* $BASIS_{rand}$ *assumption parameterized by* n, m, q, s, β, *and* ℓ*:*

Given the input matrix A *and the security parameter, the algorithm samples* $i^* \leftarrow \{1, \ldots, \ell\}$. *For all* $i \in \{1, \ldots \ell\}$ *with* $i \neq i^*$, *the algorithm samples* $A_i \leftarrow \mathbb{Z}_q^{(n+1) \times m}$. *Next, it chooses* $\mathbf{a} \leftarrow \mathbb{Z}_q^m$ *and sets* $A_i^* = \begin{bmatrix} \mathbf{a^T} \\ A \end{bmatrix}$.

Finally, it sets

$$B = \begin{bmatrix} A_1 & & & & \vert -G_{n+1} \\ & A_2 & & & \vert -G_{n+1} \\ & & . & & \vert -G_{n+1} \\ & & & . & \vert -G_{n+1} \\ & & & A_\ell & \vert -G_{n+1} \end{bmatrix}$$

with $aux = i^*$.

Wee and Wu provide a tight reduction showing $\text{BASIS}_{\text{rand}}$ is at least as hard as the SIS problem. We conclude this section by defining a partial lattice trapdoor scheme. We follow the exposition due to Albrecht et. al [3].

Definition 11 (Partial Lattice Trapdoor). *A (t, k)-threshold partial lattice trapdoor scheme over* $\mathbb{Z}_q$ *is an extension of a (full) lattice trapdoor scheme* $(TrapGen, SamplePre)$ *over* $\mathbb{Z}_q$ *with the PPT algorithms* $(PTrapGen, PSampPre, Rec)$*:*

1. $ptd_j \leftarrow PTrapGen(A, td, j)$ *: The partial trapdoor generation algorithm inputs a matrix A, its (full) trapdoor td, and an index* $j \in [k]$ *and generates a partial trapdoor* ptd_j *of A for j.*
2. $x_j \leftarrow PSampPre(A, ptd_j, T, y, \sigma)$*: The partial preimage sampling algorithm, given a partial trapdoor* ptd_j *for index j, a set* $T \subseteq [k]$*, a target image y, and a Gaussian parameter* σ*, samples a partial preimage* x_j *for j.*
3. $x \leftarrow Rec((x_j)_{j \in T})$*: The reconstruction algorithm, given a tuple of partial preimages* $(x_j)_{j \in T}$ *reconstructs a preimage x.*

Let $k, t \in \mathbb{N}$ *be such that* $t \leq k$ *and* n, m, m', β *be as before. Let s be a Gaussian width parameter for* $(TrapGen, SamplePre)$ *(i.e. the parameter from point (3) in Theorem 1) and let* $\sigma > 0$ *be the Gaussian width parameter from above. A* (t, k)*-threshold partial lattice trapdoor scheme is said to be* (n, m, s, β, σ)*-correct with respect to* $(TrapGen, SamplePre)$ *if for any t-subset* $T \subseteq_t [k]$ *and vector* $y \in \mathbb{Z}_q^n$, *it holds*

$$Pr\left[\begin{matrix} A\cdot x=y \\ \wedge \|x\|\leq\beta \end{matrix}\middle|\begin{matrix} (A,td)\leftarrow TrapGen(1^\lambda,1^n,1^m) \\ ptd_j\leftarrow PTrapGen(A,td,j)\forall j\in[k] \\ x_j\leftarrow PSampPre(A,ptd_j,T,y,\sigma)\forall j\in T \\ x\leftarrow Rec((x_j)_{j\in T}) \end{matrix}\right]\geq 1-negl(\lambda).$$

That is, the reconstructed preimage x is a correct preimage if it was reconstructed using partial preimages x_j, where all the x_j were obtained by running the $PTrapGen$ algorithm.

Albrecht et. al [3] develop a partial lattice trapdoor scheme over R_q, where R_q is the ring of integers over a cyclotomic field modulo some prime ideal q. However, their scheme naturally translates to one over $\mathbb{Z}_q$. Similar to the methods developed by Gentry et. al [10], the distribution of reconstructed preimages satisfy the following property: for fixed (A, v_j) and pairs (x_j, z_j) of images and partial preimages respectively, we have that the two distributions

1. Sample z_j uniformly at random. Then, use ptd_j to sample short x_j such that $A \cdot x_j = v_j \otimes z_j \pmod q$.
2. independently sample a Gaussian x_j from an ambient lattice. Compute z_j satisfying $A \cdot x_j = v_j \otimes z_j \pmod q$

This guarantee allows Albrecht et al. to construct a threshold signature scheme using GPV signatures [10] as a natural blueprint. In GPV, signing a message m, constitutes sampling a short, random preimage x such that $A \cdot x = H(m) \pmod q$. In the threshold version, the partial signatures are the partial preimages x_j. We use the reconstruction algorithm to obtain the short preimage x from the x_j's. In the context of our scheme, we will keep an extra insertion matrix A'_i. When a party wants to insert some $x' \in \mathbb{Z}_q$ at position i, it will obtain partial preimages $(x_j)_{j\in T}$ from other participating parties. A verifier can use these partial preimages to reconstruct a short h such that $A_i \cdot h = x' \pmod q$. In this way, we can reduce security to forging threshold signatures.

3 Multiparty Dynamic VC

We refer to the setting in Section 1.2. We say a party is *honest* if it generates the correct partial preimages when participating in an insertion and keeps its private parameters secret (see our definition below). Our scheme only requires participation of multiple parties for the insertion of new entries. However, our definition of multiparty dynamic VCs allows participation of other parties when deleting or updating an entry.

Definition 12 (Multiparty Dynamic Vector Commitment). *Let $P = \{P_1, \ldots, P_r\}$ be a set of r parties. We say an r-Party Dynamic Vector Commitment is a tuple of the following algorithms:*

$\mathtt{Setup}(\lambda, \ell) \to (pp, pp_1, \ldots pp_r)$: Given the security parameter λ and the initial length $\ell = poly(\lambda)$ of the vector being committed, output public parameters pp, as well as private parameters $pp_1, \ldots, pp_r$. The private parameters pp_j are sent to party P_j through a secure channel. Note that this inherently requires

the setup process to be trusted. The `Setup` *algorithm also outputs index sets* $S_{P_1}, \ldots S_{P_r}$ *such that* $\bigcup_{i=1}^{r} S_{p_i} = [\ell]$ *and* $S_{P_i} \cap S_{P_j} = \emptyset$ *for* $i \neq j$.

The next three algorithms are ***public****: meaning that anyone can run them and that they may implicitly use public parameters generated at Setup time.*

`Commit`$(\mathbf{x}) \rightarrow (\mathbf{c}, aux)$*: Outputs a commitment* $\mathbf{c}$ *for a vector* $\mathbf{x}$ *of length* ℓ*. Some auxiliary information aux may also be generated.*

`Open`$(x_i, i, aux) \rightarrow \pi_i$*: Possibly using the auxiliary information aux, the opening algorithm generates a proof* π_i *convincing a verifier that* x_i *is the value at the ith position in the committed vector* $\mathbf{x}$*.*

`Verify`$(\mathbf{c}, y, i, \pi_i) \rightarrow 0/1$*: Given a commitment* $\mathbf{c}$*, a value* y*, a position* i*, and a proof* π_i*, a verifier outputs* 1 *or* 0*, respectively indicating an acceptance or rejection of the proof that* y *is the ith value in the committed vector (i.e.,* $y = x_i$*).*

The following algorithms can only be initiated by one of the parties. They may require the joint participation of multiple parties. These protocols may require parties to implicitly perform computations locally using their private parameters.

`Add`$(\mathbf{c}, x', i, P_j, P) \rightarrow (\mathbf{c}', u, pp')$*:At least* $d+1$ *parties (denote the set of parties by* P*) may jointly run this algorithm. Let* $P_j \in P$ *be a chosen party in this set who wants to initiate an insertion. The algorithm updates the commitment* $\mathbf{c}$ *corresponding to vector* $[x_1, \ldots, x_\ell]$ *to a commitment* $\mathbf{c}'$ *corresponding to the vector* $[x_1, \ldots, x_{i-1}, x', x_i, x_{i+1}, \ldots, x_\ell]$*.*

Assume P_j *controls positions in the index set* S_{P_j}*. Then* S_{P_j} *gets updated to* $S_{P_j} \cup \{i\}$*.*

Some update information u *and updated public parameters* pp' *may also be released. Finally, the algorithm outputs a new proof* π_i *that verifies with respect to the updated commitment, the new inserted value, and the ith position.*

`AddPfs`$(\mathbf{c}', i, x', \{\pi_j\}_{j \in [\ell']}, u)$*: Once the commitment to a vector* $\mathbf{x}$ *(currently of length* ℓ'*) has been modified to* $\mathbf{c}'$ *after an insertion, this algorithm modifies all existing position proofs for the other entries using public update information* u*.*

Note that an entry at position $k > i$ *in the vector will now be at position* $k+1$*. Parties whose index sets are affected must locally update them accordingly during such an insertion.*

`Del`$(\mathbf{c}, i, P_j, P) \rightarrow (\mathbf{c}', u, pp')$*: At least* $d+1$ *parties* P *may jointly run this algorithm. Assume position* i *is an index controlled by exactly one of these parties* $P_j \in P$*. The algorithm first checks if* $i \in S_{P_j}$*. If not, it aborts. Otherwise, it updates the commitment* $\mathbf{c}$ *corresponding to vector* $[x_1, \ldots, x_\ell]$ *to a commitment* $\mathbf{c}'$ *corresponding to the vector* $[x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_\ell]$*. Some update information* u *and updated public parameters* pp' *may also be outputted.*

Assume party P_j *controls positions in index set* S_{P_j}*. Then* S_{P_j} *gets updated to* $S_{P_j} \setminus \{i\}$*.*

`DelPfs`$(\mathbf{c}', i, x_i, \{\pi_j\}_{j \in [\ell']}, u)$ *: Once a commitment to a vector* $\mathbf{x}$ *(currently of length* ℓ'*) has been modified to* $\mathbf{c}'$ *after a deletion of the ith entry* x_i*, this algorithm modifies all existing position proofs for the other entries using public update information* u*.*

Note that an entry at position $k > i$ in the vector will now be at position $k-1$. The algorithm updates all other parties' index sets accordingly during such a deletion.

$\texttt{Update}(\mathbf{c}, i, x_i, x_i', P_j, P) \leftarrow (\mathbf{c}', u, pp')$: *At least $d+1$ parties P may jointly run this algorithm. Assume position i is an index controlled by exactly one of these parties $P_j \in P$. The algorithm first checks if $i \in S_{P_j}$. If not, it aborts. Otherwise, it updates the commitment $\mathbf{c}$ corresponding to vector $[x_1, \ldots, x_i, \ldots, x_\ell]$ to a commitment $\mathbf{c}'$ corresponding to the vector $[x_1, \ldots, x_i', \ldots, x_\ell]$. Some update information u and updated public parameters pp' may also be outputted.*

$\texttt{UpdatePfs}(\mathbf{c}', i, x_i', \{\pi_j\}_{j \in [\ell']}, u)$: *Once a commitment to a vector $\mathbf{x}$ (currently of length ℓ') has been modified to $\mathbf{c}'$ after the ith entry x_i gets modified to x_i', this algorithm modifies all existing position proofs for the other entries using public update information u.*

In our definition above, we assume the following:

1. By "jointly", we mean that a set of parties must engage in the multiparty protocol described. We assume that parties can communicate with each other and with the Trusted Setup through secure and authenticated channels.
2. A party P_j can insert in any position it wants to. To insert in position i, it does not have to control position i beforehand. After the insertion occurs, the index sets of the other parties increment (or stay the same) accordingly.
3. On the other hand, we require P_j to control position i if they want to delete the ith value. Once P_j runs `Del`, the other parties check if P_j controls position i in order to determine if the deletion is valid. If it is not, they reject the `Del` update and/or the modified public parameters which P_j returns after the Del operation. Note that all parties keep a copy of the index sets which they control as well as the index sets which other parties control.
4. Recall in the decentralized verifiable database setting, we envision a party initially holding a large file. Using the public parameters already generated, it may generate a commitment to the file and distribute corresponding subvectors to nodes participating in storing it. Therefore, we opted to designate committing as a public operation.

Next, correctness is similar to standard VC correctness, except we modify it to accommodate insertions and deletions of elements into the committed vector.

Definition 13 (Dynamic VC Correctness). *Let ℓ be the length of the initial vector being committed to. Let $pp \leftarrow \texttt{Setup}(1^\lambda, \ell)$ and $\mathbf{c}$ be a commitment to a vector of length ℓ possibly using pp. Possibly after a sequence of updates, insertions, or deletions, assume the resulting commitment changes to $\mathbf{c}'$ and the public parameters change to pp'. Moreover, assume the length of the vector committed to is now some ℓ'.*

Then a position proof for an entry x_i with $i \in \ell'$ can be generated in one of the following ways:

1. *x_i was an entry in the vector initially committed. Possibly, x_i was at position j in the initial vector. A proof π was generated using $\texttt{Open}(x_i, j, aux)$. After*

the possible sequence of insertions or deletions to existing entries, x_i went from being the jth entry in the vector to the ith. Moreover $\mathbf{c}$ got updated to $\mathbf{c}'$ and π got modified to π' after applying the corresponding sequence of algorithms to update it. Then we require $\texttt{Verify}(c', x_i, i, \pi') = 1$ with all but negligible probability in the security parameter.

2. *x_i was an entry inserted after the initial vector was committed. Let $\overline{\mathbf{c}}$ be the commitment to this vector before x_i was inserted. Possibly, x_i was inserted at position j. A proof π was generated by at least $d+1$ parties denoted by P (i.e. using $\texttt{Add}(\overline{\mathbf{c}}, x_i, j, P_k, P)$). After the possible sequence of insertions or deletions to existing entries, x_i went from being the jth entry in the vector to the ith. Moreover $\overline{\mathbf{c}}$ got updated to $\mathbf{c}'$ and π got modified to π' after applying the corresponding sequence of algorithms to update it. Then we require $\texttt{Verify}(\mathbf{c}', x_i, i, \pi') = 1$ with all but negligible probability in the security parameter.*

Security is similar to standard VC security, except now we require infeasibility for a fixed number $d > 0$ or less parties to generate conflicting proofs for inserted values. In what follows, let $\mathcal{C} = \{C_1, \dots C_k\}$ be a set of corrupt parties. Each corrupt party C_i controls positions in the index set S_{C_i} with $S_{C_i} \cap S_{C_j} = \emptyset$ for $i \neq j$.

Definition 14 (d out of r Dynamic Binding). *Let VC_{dyn} be a Multiparty Dyamic Vector Commitment scheme. We say VC_{dyn} is d out of r Dynamic Binding if a polynomially bounded adversary $\mathcal{A}$ wins the following game with negligible probability:*

1. *The adversary outputs a set of corrupt parties $\mathcal{C}$ and the positions each party controls. The challenger will control the rest of the $r-k$ honest parties.*
2. *The challenger first checks $k < d+1$, if $\bigcup_{i=1}^{k} S_{C_i} \subset [\ell]$, and if $S_{C_i} \cap S_{C_j} = \emptyset$ for $i \neq j$. Next, the challenger runs $(pp, pp_1, \dots, pp_r) \leftarrow VC_{dyn}.\texttt{Setup}(1^\lambda, 1^\ell)$. The challenger proceeds to send $(pp, \{pp_i\}_{i=1}^{k})$ to $\mathcal{A}$ (without loss of generality, assume the first k out of r parties are the corrupt ones and assume pp_i is meant for party C_i for $i = 1, \dots, k$).*
3. *$\mathcal{A}$ generates a commitment $\mathbf{c}$ and sends it to the challenger.*
4. *$\mathcal{A}$ can query the challenger for an insertion of a value x_i' at any position i. Alternatively, it can query for a deletion at some position i in the vector (where $i \in S_{C_j}$ for $j = 1, \dots, k$). The challenger will run $\texttt{Add}$ or $\texttt{Del}$ to modify the commitment, obtain updated public parameters pp' as well as update information u, and a new position proof π_i (in the case of an insertion). This will be run simulating any set parties of $\mathcal{A}$'s choosing. Note that the challenger will have to be involved, as $\mathcal{A}$ controls less than $d+1$ positions. The challenger will additionally send $\mathcal{A}$ any communication between the chosen $d+1$ parties when simulating the $\texttt{Add}$ or $\texttt{Del}$ algorithms. Lastly, the challenger sends pp' and u so that $\mathcal{A}$ can run $VC_{dyn}.\texttt{AddPfs}$ or $VC_{dyn}.\texttt{DelPfs}$ depending on if an insertion or deletion was queried. If an insertion was queried, the challenger will also send the new proof π_i.*

5. *$\mathcal{A}$ finally outputs a commitment $\mathbf{c}'$, a position proofs π_j and π'_j corresponding to values x_j, x'_j. $\mathcal{A}$ wins the game if $x_j \neq x'_j$ but $VC_{dyn}.\texttt{Verify}(\mathbf{c}', x_j, j, \pi_j) = VC_{dyn}.\texttt{Verify}(\mathbf{c}', x'_j, j, \pi'_j) = 1$.*

If $\mathcal{A}$ makes at most Q queries, we say $\mathcal{A}$ is a Q-Query dynamic binding adversary.

4 Lattice Based Construction

4.1 Starting Point: The Wee-Wu VC Scheme

We begin by reviewing the Wee-Wu VC design, which our scheme builds on. Let λ be the security parameter. The construction is parameterized by the following values in λ: n, m, q, s_0, s_1, and B. Set $m' = n\lceil \log q \rceil + 1$. The values n and m are matrix dimensions, q is a prime, s_0, s_1 are Gaussian width values, and B is the SIS bound. We now describe the scheme. Assume we are committing to a vector $\mathbf{x} \in \mathbb{Z}_q^\ell$. As of now, this scheme does not support efficient insertions/deletions.

$\texttt{Setup}(\lambda, \ell)$: For $i = 1, \ldots, \ell$, sample

$$(A_i, td_i) \leftarrow TrapGen(1^n, q, m)$$

with $A_i \in \mathbb{Z}_q^{n \times m}$. Next, form the matrix

$$B_\ell = \left[\begin{array}{cccc|c} A_1 & & & & -G \\ & A_2 & & & -G \\ & & . & & -G \\ & & & . & -G \\ & & & A_\ell & -G \end{array}\right] \in \mathbb{Z}_q^{n\ell \times (\ell m + m')}$$

Then, note that we have $B_\ell R = G_{n\ell}$, where $R = \begin{pmatrix} diag(td_1, \ldots td_\ell) \\ 0 \end{pmatrix}$. To obtain a (statistically close to) random trapdoor for B_ℓ, we run

$$T \leftarrow SamplePre(B_\ell, R, G_{n\ell}, s_0).$$

We finish the `Setup` Process by outputting $pp := (A_1, \ldots A_\ell, T)$.
Note that the `Setup` process is *trusted*: if an untrusted party runs `Setup`, they have access to all the trapdoors for the A_i public parameters. This can be used to break scheme's binding.

$\texttt{Commit}(pp, \mathbf{x})$: Use the trapdoor T to run

$$(\mathbf{v_1}, \ldots \mathbf{v}_\ell, \hat{\mathbf{c}}) \leftarrow SamplePre(B_\ell, T, -\mathbf{x} \otimes \mathbf{e}_1, s_1)$$

where $\mathbf{e}_1$ is the first standard basis vector in $\mathbb{Z}_q^m$. We set the commitment to be $\mathbf{c} = G\hat{\mathbf{c}} \in \mathbb{Z}_q^n$. A position proof π_i for index i is simply the vector $\mathbf{v_i} \in \mathbb{Z}_q^m$. In this way, the commitment and proofs are sampled *jointly*. We output the commitment $\mathbf{c} = G\hat{\mathbf{c}}$ and auxiliary information $(\mathbf{v_1}, \ldots, \mathbf{v}_\ell)$.

$\texttt{Open}(x_i, i, aux)$: Output $\pi_i = \mathbf{v}_i \in \mathbb{Z}_q^m$.

$\texttt{Verify}(\mathbf{c}, x_i, i, \pi)$: Parse the proof π as the vector $\mathbf{v}_i \in \mathbb{Z}_q^m$. By construction of how we sampled our commitment/proofs, we must have $A_i \cdot \mathbf{v}_i = \mathbf{c} - x_i e_1$. To verify, we therefore check the following: $||\mathbf{v}_i|| \leq B$ and $A_i \cdot \mathbf{v}_i = \mathbf{c} - x_i \mathbf{e}_1$.

Before presenting our scheme, we present a first attempt in Appendix A. We also show why this attempt is not secure. The reader is encouraged to read this to understand why our construction works.

4.2 Scheme Description

In this section, we present a secure d out of ℓ multiparty dynamic VC scheme. While our scheme supports updates after insertion/deletion, it is not endowed with a natural update mechanism which the original Wee-Wu scheme had due to its homomorphic properties. We also note that the participation of multiple parties are only truly necessary for insertion.

Our protocol makes use of a random oracle $H : \mathbb{Z}_q \to \mathbb{Z}_q^n$. The security of our scheme will reduce to the standard binding of the Wee-Wu scheme or the existential unforgeability of the threshold GPV [10] signature scheme presented by Albrecht et al. in [3]. Assume that the initial vector we commit to is of length ℓ as before. Let $k < d < \ell$ be a threshold. For simplicity, we will assume every party can control at most k positions and controls exactly one of the ℓ positions at the intial commit time (so there are ℓ parties). Therefore, the vector can be stretched to a length of at most $k\ell$. Assuming $PT = (PTrapGen, PSampPre, Rec)$ is d out of ℓ, our construction will be d out of ℓ dynamic binding.

$\texttt{Setup}(1^\lambda, 1^\ell)$: First we describe how to generate the initial public parameters pp. For $i = 1, \ldots, \ell$, sample

$$(A_i, td_i) \leftarrow TrapGen(1^n, q, m)$$

with $A_i \in \mathbb{Z}_q^{n \times m}$. Next, form the matrix

$$B_\ell = \left[\begin{array}{cccc|c} A_1 & & & & -G \\ & A_2 & & & -G \\ & & . & & -G \\ & & & . & -G \\ & & & A_\ell & -G \end{array}\right] \in \mathbb{Z}_q^{n\ell \times (\ell m + m')}$$

Then, note that we have $B_\ell R = G_{n\ell}$, where $R = \begin{pmatrix} diag(td_1, \ldots td_\ell) \\ 0 \end{pmatrix}$. To obtain a (close to) random trapdoor for B_ℓ, we run

$$T \leftarrow SamplePre(B_\ell, R, G_{n\ell}, s_0).$$

Choose new Gaussian width parameters s_1, s_1'.

Next, we describe how to generate the private parameters $pp_1, \ldots pp_\ell$. For $j = 1, \ldots, k\ell$, run $(A_j', td_j') \leftarrow TrapGen(1^n, q, m)$. Next, for $k = 1, \ldots, \ell$, run $ptd_{j_k} \leftarrow PTrapGen(A_j', td_j', k)$. Then for the nth party define $pp_n = (ptd_{1_n}, \ldots, ptd_{k\ell_n})$.

Set $pp := (A_1, \ldots A_\ell, T, A_1', \ldots A_{k\ell}')$. Securely send the mth private parameter pp_m to the mth party for $m = 1 \ldots \ell$.

Denote all the parties by $P_1, \dots P_\ell$. For each P_i, output their initial set S_{P_i} of indices they control (at Setup time recall that this set consists of a single position).

$\texttt{Commit}(pp, \mathbf{x})$: Let $\mathbf{x} = (x_1, \dots, x_\ell) \in \mathbb{Z}_q^\ell$ Compute

$$\mathbf{a} = \mathbf{a}_1 || \mathbf{a}_2 || \dots || \mathbf{a}_\ell = H(x_1) || H(x_2) || \dots || H(x_\ell).$$

Use the trapdoor T and the matrix B_ℓ from $\texttt{Setup}$ to run

$$(\mathbf{v}_1, \dots \mathbf{v}_\ell, \hat{\mathbf{c}}) \leftarrow SamplePre(B_\ell, T, \mathbf{a}, s_1).$$

Output $\mathbf{c} = G\hat{\mathbf{c}}$ as the commitment and $aux = (\mathbf{v}_1, \dots \mathbf{v}_\ell)$.

$\texttt{Open}(x_i, i, aux)$: Output $\pi = \mathbf{v}_i$.

$\texttt{Add}(\mathbf{c}, x', i, p, P)$: This algorithm is invoked by some set of parties P controlling positions in the vector. After this algorithm terminates, position i will hold the value x' and will be controlled by party $p \in P$. Without loss of generality, let $P = \{P_1, \dots P_k\}$ and let $p = P_1$. Their respective private parameters are denoted by $pp_1, \dots, pp_k$.

Leave the commitment $\mathbf{c}$ unchanged.

Take an A'_j in the current set of public parameters. Each party running the algorithm does the following

1. Recall that a party P_i will have a partial trapdoor ptd_{j_i} for the matrix A'_j. Party P_i takes their partial trapdoor and computes a partial preimage by running

$$\mathbf{x}_i \leftarrow PSampPre(A'_j, ptd_{j_i}, \{1, \dots, k\}, c - H(x'), s'_1).$$

2. Publicly post the partial preimage $\mathbf{x}_i$.

Update the public parameters to $pp' = (A_1, A_2, \dots A_{i-1}, A'_j, A_i, \dots A_\ell, A'_1, \dots A'_{j-1}, A'_{j+1}, \dots A'_{k\ell})$. Let $p = P_1$'s index set be S_{P_1}. Update this set to $S_{P_1} \cup \{i\}$. Output the insertion position, p's updated index set, the updated public parameters, and the set of partial preimages. That is, output (i, S_{P_1}, pp', v'), where $v' = \{\mathbf{x}_h\}_{h=1}^k$.

$\texttt{AddPfs}(\mathbf{c}, i, x', \{\pi_j\}_{j \in [\ell']}, (i, pp', v'))$: Output $\mathbf{v}' = (\mathbf{v}_1, \dots \mathbf{v}_{i-1}, v', \mathbf{v}_i, \dots, \mathbf{v}_{\ell'})$. Additionally, use the fact that the insertion occurred at position i to modify all parties' index sets accordingly.

$\texttt{Del}(\mathbf{c}, i, p, P)$: Without loss of generality, assume $p = P_1 \in P$ controls the position i.

Leave the commitment $\mathbf{c}$ unchanged. Set the updated public parameters pp' to $pp' = (A_1, A_2, \dots, A_{i-1}, A_{i+1}, \dots A_\ell, A'_1, \dots A'_{k\ell})$. Take $p = P_1$'s index set S_{P_1} and update it to $S_{P_1} \setminus \{i\}$. Output (pp', S_{P_1})

$\texttt{DelPfs}(\mathbf{c}, i, x_i, \{\pi_j\}_{j \in [\ell']}, (pp')$: Output $\mathbf{v}' = (\mathbf{v}_1, \dots, \mathbf{v}_{i-1}, \mathbf{v}_{i+1}, \dots \mathbf{v}'_\ell)$. Update all other parties' index sets using the fact that the deletion occurred at position i.

$\texttt{Verify}(\mathbf{c}, y, i, \pi_i)$: Parse π as $\mathbf{v_i} \in \mathbb{Z}_q^m$. Check if $||\mathbf{v}_i|| \leq B$ and $A_i \cdot \mathbf{v}_i = \mathbf{c} - H(x_i) \pmod q$.

5 Security

In this section, we prove the security of the scheme presented in the previous section in the random oracle model. Breaking dynamic binding reduces to either breaking standard binding in the Wee-Wu scheme, or forging the threshold-GPV signature [3] in the multi user setting.

We present the security of d out of ℓ Threshold GPV game in Appendix B. Next we present the security of d out of ℓ Threshold GPV with commitments game which is used to prove the security of our scheme.

d out of ℓ Threshold GPV With Commitments: We make a slight modification to the d out of ℓ Threshold GPV security game. The game now is as follows:

1. The adversary $\mathcal{A}$ outputs a set of corrupt parties $\mathcal{C} = \{C_1, \ldots, C_k\}$
2. The challenger first checks $k < d+1$, if $\bigcup_{i=1}^{k} S_{C_i} \subset [\ell]$, and if $S_{C_i} \cap S_{C_j} = \emptyset$ for $i \neq j$.
3. Next, for $i = 1, \ldots, k\ell$, the challenger runs

$$(A_i, td_i) \leftarrow TrapGen(1^n, 1^m, q).$$

   For $j = 1, \ldots, k\ell$
   For $h = 1, \ldots, \ell$, the challenger runs

$$ptd_{j_h} \leftarrow PT.PTrapGen(A_j, td_j, h).$$

   The challenger sends ptd_{j_h} for $h \in \{1, \ldots, k\}$ to $\mathcal{A}$.
4. $\mathcal{A}$ fixes a value $\mathbf{c} \in \mathbb{Z}_q^n$ and sends it to the challenger.
5. $\mathcal{A}$ can make at most Q queries. On the ith query, $\mathcal{A}$ asks for an index j_i, positions T_i, and a value μ_i. It sends (j_i, T_i, μ_i) to the challenger.
6. If j_i has already been queried, the challenger outputs $\perp$. Otherwise, it does the following:
   For $h \in T_i$ run

$$\pi_{i_h} \leftarrow PT.PSampPre(A_{j_i}, ptd_{j_{i_h}}, T_i, \mathbf{c} - H(\mu_i), \sigma)$$

   Send $\{\pi_{i_h}\}_{h \in T_i}$ to $\mathcal{A}$.
7. At the end of the querying, $\mathcal{A}$ outputs an index w, a value x, and a vector v. $\mathcal{A}$ wins if x was not the query it made with respect to A_w, v is short, and $A_w \cdot v = \mathbf{c} - H(x)$.

Lemma 1 (d out of ℓ Threshold GPV with Commitments is Secure).

Proof. The only difference between d out of ℓ GPV and d out of ℓ Threshold GPV with Commitments is that in d out of ℓ Threshold GPV with Commitments, the target which we find a preimage for is $\mathbf{c} - H(x)$ rather than just $H(x)$. Since $\mathbf{c}$ is fixed and $H(x)$ is modeled as a random oracle, the distribution of the target in d out of ℓ Threshold GPV with Commitments is exactly the same as it is in d out of ℓ Threshold GPV.

Theorem 2. *Let $\mathcal{A}$ be a d out of ℓ dynamic binding adversary. Let ϵ be the probability $\mathcal{A}$ wins conditioned on its output being with respect to a verification key that existed since committing time. Then there exists a standard VC binding adversary $\mathcal{B}$ and a d out of ℓ Threshold GPV with Commitments adversary $\mathcal{C}$ such that*

$$Pr[\mathcal{A} \text{ wins the dynamic VC binding game}] \leq$$
$$Pr[\mathcal{B} \text{ wins the standard VC binding game}]+$$
$$Pr[\mathcal{C} \text{ wins the } d \text{ out of } \ell \text{ Threshold GPV with Commitments game}]$$

Proof. We first condition on $\mathcal{A}$'s output being with respect to a verification key which existed since committing time. Then our standard standard binding adversary $\mathcal{B}$ works as follows: it receives $(A_1, \dots A_\ell, T, A'_1, \dots A'_{k\ell})$ from its (standard VC binding) challenger. $\mathcal{B}$ passes the public parameters pp onto the dynamic binding adversary $\mathcal{A}$ and acts as the dynamic binding challenger. $\mathcal{A}$ will output some commitment $\mathbf{c}$ and query for insertions/deletions. $\mathcal{B}$ will generate new verification keys for insertions and perform the necessary reorderings of the public parameters.

$\mathcal{A}$ finally outputs $i, \mathbf{c}', x, x', \mathbf{v}, \mathbf{v}'$. $\mathcal{B}$ outputs the same values, except possibly the position i, as $\mathcal{B}$ must break VC binding with respect to the original public parameters. Since insertion/deletion reshuffles the indices for the (original) public A_i matrices, $\mathcal{B}$, will need to keep track of this as it answers $\mathcal{A}$'s queries and output the original position which i corresponded to at the beginning of the game. Then we clearly see that $\mathcal{B}$ wins its game with the same probability $\mathcal{A}$ wins the dynamic binding game.

Next, we condition on $\mathcal{A}$'s output being with respect to an inserted verification key. We construct an algorithm $\mathcal{C}$ which solves d out of ℓ GPV with Commitments.

In the d out of ℓ with Commitments game, the challenger sends $k\ell$ many instances $A'_1 \dots A'_{k\ell}$ to $\mathcal{C}$. Next, $\mathcal{C}$ plays the role of the dynamic VC binding challenger to $\mathcal{A}$. $\mathcal{C}$ generates $(A_1, \dots A_\ell, T)$ and sends $crs = (A_1, \dots A_\ell, T, A'_1 \dots A'_{k\ell})$ to $\mathcal{A}$. $\mathcal{A}$ responds with some commitment $\mathbf{c}$.

$\mathcal{A}$ and $\mathcal{C}$ engage in the dynamic binding game. For an insertion query with respect to a set of indices T, position i for some value x', $\mathcal{C}$ and $\mathcal{A}$ locally generate partial preimages with respect to one of matrices A'_j left in the public parameters. $\mathcal{A}$ will generate partial preimages for positions in T which it controls, while $\mathcal{C}$ will generate partial preimages for the other positions in T. The set of partial preimages can be reconstructed into a short proof $\mathbf{h}'$ such that $A'_j \cdot \mathbf{h}' = \mathbf{c} - H(x')$. $\mathcal{A}$ gets access to all these partial preimages.

At the end of the querying, $\mathcal{A}$ outputs an index j, a value x_j, and a short proof $\mathbf{h}$ which are supposed to break binding with respect to one of the (new) verification keys A'_j. $\mathcal{C}$ checks that x_j is not the value $\mathcal{A}$ queried to be inserted with respect to the matrix A'_j, aborting if this is the case. Otherwise, $\mathcal{C}$ sends x_j and $\mathbf{h}'$ to the d out of ℓ Threshold GPV challenger, where $\mathbf{h}'$ acts as a forged signature for x_j. Moreover, $\mathcal{C}$ wins its game with the same probability $\mathcal{A}$ wins the dynamic binding game. This completes the proof.

5.1 Parameter Instantiation and Complexity Analysis

Let λ be the security parameter and ℓ be the length of the initial vector to be committed. We can take the same parameters as done in the Wee-Wu scheme. That is,

1. Set $n = \lambda$, $m = O(n \log q)$.
2. Set $s_0 = O(\ell m \log(n\ell))$, $s_1 = O(\ell^{3/2} m^{3/2} \log(n\ell)) = O(\ell^{5/2} m^{5/2} \log^2(n\ell))$, and $s_1' = \omega(\sqrt{\log m})$.
3. Set the SIS bound $B = \sqrt{\ell m + m' \cdot s_1} = O(\ell^3 n^3 \log^2(n\ell) \log^3 q)$.
4. Set $q = B \cdot poly(n)$. In this way, the VC binding reduces to the $\mathrm{SIS}_{n,m,2B,q}$ assumption, while forging a GPV signature (i.e. breaking binding with respect to an inserted verification key) reduces to the $\mathrm{SIS}_{n,m,B,q}$ assumption.

We have $\log q = O(\log \lambda + \log \ell)$. Then the commitment $\mathbf{c}$ to a message of length ℓ is an element in $\mathbb{Z}_q^n$. The commitment size is therefore

$$|\mathbf{c}| = O(n \log q) = O(\lambda(\log q + \log \ell)).$$

A position proof existing since commitment time π is a vector $\mathbf{v} \in \mathbb{Z}_q^m$ whose norm is bounded by B. Therefore we have

$$|\pi| = O(m \log B) = O(\lambda \cdot (\log^2 \lambda + \log^2 \ell)).$$

A verification key is of dimension $m \times n$ with $m = n \log q$. If the initial length is ℓ, and we do r insertions, the size of all the verification keys is

$$O(l + r)(nm \log q) = O(l + r)(n^2 \log^2 q) = O(l + r)(\lambda^2 log^2 q).$$

Next, r insertions constitute $r(d + 1)$ vectors. They contribute

$$r(d + 1)(m) = r(d + 1)(\log q)$$

bits to the size. Totally,

$$|pp| = O((\ell + r)(\lambda^2 log^2 q) + r(d + 1)(\lambda \log q))$$

For our scheme, insertion of Q new elements corresponds to an instantiation of the threshold GPV signature scheme in the Q user setting. Therefore, we can allow Q to take a value of $poly(\lambda)$. That is, we can allow $poly(\lambda)$ many insertions at any given time.

Next we expand upon the complexity of inserting, deleting, and updating entries. A deletion's complexity is independent of the vector size and therefore takes $O(1)$ time.

Insertion is also independent of the vector size. We conduct our analysis in terms of the length of the committed vector. Assume an instantiation of our scheme is d out of ℓ multiparty dynamic. Then running `Add` amounts to each participating party sampling a partial preimage for a public parameter $A_j' \in$

$\mathbb{Z}_q^{n \times m}$. Each party can run the sampling procedure independently. Therefore the insertion complexity is not dependent on the vector size and runs in time $O(1)$.

Note that modifying all existing proofs after an insertion or deletion (i.e. running `AddPfs` or `DelPfs` also runs in $O(1)$ time, since these algorithms do not need to examine existing proofs and we delegate reconstructing a partial proof to verification time. Updating index sets is a task we delegate to the parties. Therefore updating the proofs also runs in $O(1)$ time.

Since a modification is a deletion followed by an insertion, it also runs in $O(1)$ to modify an entry in the committed vector, as well as update all the existing position proofs. Therefore, our scheme presents a new method for achieving *maintainability*, which is the ability to update all existing position proofs after a modification of an entry in sublinear time in the vector length.

6 Conclusion

In this paper, we defined multiparty dynamic vector commitments. A multiparty dynamic VC works in a framework where multiple parties control the entry values of designated positions in a committed vector. If a party controls some position i, they have the power to modify or delete the entry value in that position. Multiparty dynamic VCs also support efficient updates after the length of the committed vector changes via insertion or deletion. By naively trying to extend the Wee and Wu VC scheme, we saw that binding in the static setting does not necessarily imply security in our dynamic setting. This motivated the notion of multiparty dynamic binding, which captures security for dynamic VCs. We proceeded to present a multiparty dynamic VC using lattice trapdoor splitting techniques. However, our scheme does not natural, homomorphic updates after modifications to existing entries in the committed vector. We believe an important further step is to design dynamic VCs which support natural updates after modifications to entries and do not require other parties to participate during insertions. Another important open problem is to design a multiparty dynamic VC where the public parameters' size remains small even after many insertions.

Acknowledgement

The authors would like to thank the anonymous reviewers whose comments and suggestions were very helpful to improve the quality of the paper.

References

1. Ajtai, M.: Generating hard instances of lattice problems. In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. pp. 99–108 (1996)
2. Albrecht, M.R., Cini, V., Lai, R.W., Malavolta, G., Thyagarajan, S.A.: Lattice-based snarks: Publicly verifiable, preprocessing, and recursively composable. In: Annual International Cryptology Conference. pp. 102–132. Springer (2022)

3. Albrecht, M.R., Lai, R.W.F., Lapiha, O., Woo, I.K.Y.: Partial lattice trapdoors: How to split lattice trapdoors, literally. In: Hanaoka, G., Yang, B. (eds.) Advances in Cryptology - ASIACRYPT 2025 - 31st International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, VIC, Australia, December 8-12, 2025, Proceedings, Part III. Lecture Notes in Computer Science, vol. 16247, pp. 265–296. Springer (2025). https://doi.org/10.1007/978-981-95-5099-9_9, https://doi.org/10.1007/978-981-95-5099-9_9
4. Campanelli, M., Fiore, D., Greco, N., Kolonelos, D., Nizzardo, L.: Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In: Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part II 26. pp. 3–35. Springer (2020)
5. Campanelli, M., Nitulescu, A., Ràfols, C., Zacharakis, A., Zapico, A.: Linear-map vector commitments and their practical applications. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 189–219. Springer (2022)
6. de Castro, L., Peikert, C.: Functional commitments for all functions, with transparent setup and from sis. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 287–320. Springer (2023)
7. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16. pp. 55–72. Springer (2013)
8. Chepurnoy, A., Papamanthou, C., Srinivasan, S., Zhang, Y.: Edrax: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive (2018)
9. Erway, C.C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. ACM Trans. Inf. Syst. Secur. **17**(4), 15:1–15:29 (2015). https://doi.org/10.1145/2699909, https://doi.org/10.1145/2699909
10. Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: Proceedings of the fortieth annual ACM symposium on Theory of computing. pp. 197–206 (2008)
11. Gorbunov, S., Reyzin, L., Wee, H., Zhang, Z.: Pointproofs: Aggregating proofs for multiple vector commitments. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 2007–2023 (2020)
12. Kilian, J.: A note on efficient zero-knowledge proofs and arguments. In: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing. pp. 723–732 (1992)
13. Lai, R.W., Malavolta, G.: Subvector commitments with application to succinct arguments. In: Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I 39. pp. 530–560. Springer (2019)
14. Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In: Micciancio, D. (ed.) Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings. Lecture Notes in Computer Science, vol. 5978, pp. 499–517. Springer (2010). https://doi.org/10.1007/978-3-642-11799-2_30, https://doi.org/10.1007/978-3-642-11799-2_30
15. Micali, S.: CS proofs (extended abstracts). In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, November 20-22, 1994.

pp. 436–453. IEEE Computer Society (1994). https://doi.org/10.1109/SFCS.1994.365746, https://doi.org/10.1109/SFCS.1994.365746
16. Micciancio, D., Peikert, C.: Trapdoors for lattices: Simpler, tighter, faster, smaller. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 700–718. Springer (2012)
17. Nitulescu, A.: Sok: Vector commitments, https://www.di.ens.fr/~nitulesc/files/vc-sok.pdf
18. Papamanthou, C., Shi, E., Tamassia, R.: Signatures of correct computation. In: Theory of Cryptography Conference. pp. 222–242. Springer (2013)
19. Pathak, V.N., Ruj, S., van der Meyden, R.: Vector commitment design, analysis, and applications: A survey. ACM Computing Surveys (To Appear) (2026), https://eprint.iacr.org/2025/667
20. Sengupta, B., Ruj, S.: Publicly verifiable secure cloud storage for dynamic data using secure network coding. In: Chen, X., Wang, X., Huang, X. (eds.) Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016. pp. 107–118. ACM (2016). https://doi.org/10.1145/2897845.2897915, https://doi.org/10.1145/2897845.2897915
21. Sengupta, B., Ruj, S.: Efficient proofs of retrievability with public verifiability for dynamic cloud storage. IEEE Trans. Cloud Comput. **8**(1), 138–151 (2020). https://doi.org/10.1109/TCC.2017.2767584, https://doi.org/10.1109/TCC.2017.2767584
22. Shi, E., Stefanov, E., Papamanthou, C.: Practical dynamic proofs of retrievability. In: Sadeghi, A., Gligor, V.D., Yung, M. (eds.) 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. pp. 325–336. ACM (2013). https://doi.org/10.1145/2508859.2516669, https://doi.org/10.1145/2508859.2516669
23. Srinivasan, S., Chepurnoy, A., Papamanthou, C., Tomescu, A., Zhang, Y.: Hyperproofs: Aggregating and maintaining proofs in vector commitments. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3001–3018 (2022)
24. Tas, E.N., Boneh, D.: Vector commitments with efficient updates. In: Bonneau, J., Weinberg, S.M. (eds.) 5th Conference on Advances in Financial Technologies, AFT 2023, Princeton, NJ, USA, October 23-25, 2023. LIPIcs, vol. 282, pp. 29:1–29:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.AFT.2023.29, https://doi.org/10.4230/LIPIcs.AFT.2023.29
25. Tomescu, A., Abraham, I., Buterin, V., Drake, J., Feist, D., Khovratovich, D.: Aggregatable subvector commitments for stateless cryptocurrencies. In: International Conference on Security and Cryptography for Networks. pp. 45–64. Springer (2020)
26. Wee, H., Wu, D.J.: Lattice-based functional commitments: Fast verification and cryptanalysis. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 201–235. Springer (2023)
27. Wee, H., Wu, D.J.: Succinct vector, polynomial, and functional commitments from lattices. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 385–416. Springer (2023)

Appendix A: A first attempt

The crucial property of the Wee-Wu scheme which we use in our construction is that a position proof for index i is not dependent on i itself, unlike a number

of other VC schemes. Rather, it is only dependent on its verification key A_i. Therefore, insertion/deletion of a value and position proof corresponds to an insertion/deletion of a verification key. In this way, we will show that no modifications to the prior commitment or position proofs are necessary when inserting or deleting values.

We will now add (single-party) insertion and deletion mechanisms to Wee-Wu. Unfortunately, we will see that these modifications in their current state compromise the security of the scheme. However, this exercise motivates the ultimate multiparty construction we propose. Assume we commit to an initial vector of length ℓ. After deletions/insertions of this value, this length can change.

$\texttt{Setup}(\lambda, \ell)$: Same as in the Wee-Wu scheme. That is, for $i = 1, \ldots, \ell$, sample

$$(A_i, td_i) \leftarrow TrapGen(1^n, q, m)$$

with $A_i \in \mathbb{Z}_q^{n \times m}$. Next, form the matrix

$$B_\ell = \left[\begin{array}{cccc|c} A_1 & & & & -G \\ & A_2 & & & -G \\ & & . & & -G \\ & & & . & -G \\ & & & A_\ell & -G \end{array}\right] \in \mathbb{Z}_q^{n\ell \times (\ell m + m')}$$

Then, note that we have $B_\ell R = G_{n\ell}$, where $R = \begin{pmatrix} diag(td_1, \ldots td_\ell) \\ 0 \end{pmatrix}$. To obtain a (close to) random trapdoor for B_ℓ, we run

$$T \leftarrow SamplePre(B_\ell, R, G_{n\ell}, s_0).$$

Additionally, choose a new Gaussian width parameter s_1'.

Output $pp := (A_1, \ldots A_\ell, T)$.

$\texttt{Commit}(pp, \mathbf{x})$: Same as before. That is, use the trapdoor T to run

$$(\mathbf{v}_1, \ldots \mathbf{v}_\ell, \hat{\mathbf{c}}) \leftarrow SamplePre(B_\ell, T, -\mathbf{x} \otimes \mathbf{e}_1, s_1)$$

where $\mathbf{e}_1$ is the first standard basis vector in $\mathbb{Z}_q^n$. Output the commitment $\mathbf{c} = G\hat{\mathbf{c}}$ and auxiliary information $(\mathbf{v}_1, \ldots, \mathbf{v}_\ell)$.

$\texttt{Open}(x_i, i, aux)$: Output $\pi_i = \mathbf{v}_i \in \mathbb{Z}_q^m$.

$\texttt{Verify}(pp, \mathbf{c}, i, x_i, \pi)$: Check if $||\mathbf{v}_i|| \leq B$ and if $A_i \cdot \mathbf{v}_i = \mathbf{c} - x_i \mathbf{e}_1 \pmod q$.

$\texttt{Add}(i, \mathbf{c}, x')$: Keep the commitment $\mathbf{c}$ unchanged.

$\texttt{AddPf}(pp, \mathbf{c}, i, x', \{\pi_j\}_{j=1,\ldots,\ell})$: Run

$$(A'_i, td'_i) \leftarrow TrapGen(1^n, q, m).$$

Next, run

$$\mathbf{v}' \leftarrow SamplePre(A'_i, td'_i, \mathbf{c} - x'\mathbf{e}_1, s'_1)$$

where $\mathbf{c}$ is the commitment generated to the initial vector before any insertions/deletions. Take the (ordered) verification keys $(A_1, \ldots, A_\ell)$, and modify them to be $(A_1, \ldots, A_{i-1}, A'_i, A_i, \ldots, A_\ell)$. Output $\mathbf{v}'$, which is the new verification proof for x' with respect to position i. The previous verification proofs for position $j > i$ verifies for position $j + 1$, as the jth verification key has now become the $j + 1$th verification key.

$\texttt{Del}(i, c)$: Keep the commitment $\mathbf{c}$ unchanged

$\texttt{DelPf}(i, \mathbf{c}, \{\pi_j\}_{j=1,\ldots,\ell})$: Rearrange the verification keys to be $(A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_\ell)$.

This scheme is correct, but it is not secure because the user generating the new verification key also now has access to the corresponding trapdoor. They can use this trapdoor to submit false proofs. Therefore, extending Wee-Wu in this way does not guarantee binding, which means that standard VC binding does not extend to the insertion/deletion setting. A trivial method to circumvent this issue is to make a user send their desired insertion value x' to a trusted party and request it to sample a random matrix and a random short preimage for x' with respect to the random matrix sampled. The preimage would serve as the new proof with respect to the new verification key. The trusted party would present this new proof along with the verification key, as well as reorder the public parameters.

The construction we present takes the above approach, except we use a partial lattice trapdoor scheme to get rid of the trusted party and decentralize the process of generating a random short preimage for an inserted value.

Appendix B: d out of ℓ Threshold GPV

We first define a variant of SIS and prove its hardness. It is essentially a multi-user version of forging threshold GPV signatures [3] (where an adversary only gets to ask a single signing query). This variant will be used in the security analysis of our dynamic option for insertions/deletions.

Security Game: The security game takes place between an adversary $\mathcal{A}$ and a challenger. Let PT be a d out of ℓ partial lattice trapdoor scheme. The game is as follows:

1. The adversary $\mathcal{A}$ outputs a set of corrupt parties $\mathcal{C} = \{C_1, \ldots, C_k\}$

2. The challenger first checks $k < d + 1$, if $\bigcup_{i=1}^{k} S_{C_i} \subset [\ell]$, and if $S_{C_i} \cap S_{C_j} = \emptyset$ for $i \neq j$.
3. Next, for $i = 1, \ldots, k\ell$, the challenger runs

$$(A_i, td_i) \leftarrow TrapGen(1^n, 1^m, q).$$

For $j = 1, \ldots, k\ell$
For $h = 1, \ldots, \ell$, the challenger runs

$$ptd_{j_h} \leftarrow PT.PTrapGen(A_j, td_j, h).$$

The challenger sends ptd_{j_h} for $h \in \{1, \ldots, k\}$ to $\mathcal{A}$.
4. $\mathcal{A}$ can make at most Q queries. On the ith query, $\mathcal{A}$ asks for an index j_i, positions T_i, and a value μ_i. It sends (j_i, T_i, μ_i) to the challenger.
5. If j_i has already been queried, the challenger outputs $\perp$. Otherwise, it does the following:
For $h \in T_i$ run

$$\pi_{i_h} \leftarrow PT.PSampPre(A_{j_i}, ptd_{j_{i_h}}, T_i, H(\mu_i), \sigma)$$

Send $\{\pi_{i_h}\}_{h \in T_i}$ to $\mathcal{A}$
6. At the end of the querying, $\mathcal{A}$ outputs an index w, a value x, and a vector v. $\mathcal{A}$ wins if x was not the query it made with respect to A_w, v is short, and $A_w \cdot v = H(x)$.

Lemma 2 (d out of ℓ Threshold GPV is Secure). *Let $\ell = poly(\lambda)$ and let $\mathcal{A}$ be an adversary for d out of ℓ Threshold GPV. Then $\mathcal{A}$'s winning probability is negligible.*

Proof. Let $\epsilon_{threshold}$ denote the advantage a PPT adversary has in breaking the existential unforgery of a threshold GPV signature with respect to some public key. Then under standard lattice assumptions, we know that $\epsilon_{threshold}$ is negligible. Next, let $\epsilon_{d/\ell}$ denote the winning probability for a PPT adversary for the d out of ℓ threshold GPV game. Note that the d out of ℓ GPV game is exactly the same as the game of trying to forge a threshold GPV signature [2] with respect to one of $k\ell = poly(\lambda)$ public keys. Then by a standard complexity leveraging argument, we know

$$\epsilon_{d/\ell} \leq (k\ell) \cdot \epsilon_{threshold}.$$

Since $\epsilon_{threshold}$ is negligible in λ and $k\ell$ is polynomial in λ, we must have that $\epsilon_{d/\ell}$ is negligible in λ.