Future Generation Computer Systems 115 (2021) 553–567

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Efficient verifiable databases with additional insertion and deletion operations in cloud computing

Qiang Wang [a,b,1], Fucai Zhou [a,*], Jian Xu [a], Zifeng Xu [a]

[a] Software College, Northeastern University, Shenyang, China
[b] College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

ARTICLE INFO

Article history:
Received 13 August 2019
Received in revised form 28 July 2020
Accepted 26 September 2020
Available online 10 October 2020

Keywords:
Vector commitment
Merkle Interval Hash Tree
Publicly verifiable
Verifiable databases
All update operations

ABSTRACT

Verifiable database (VDB) schemes allow the data owner to outsource a large database to the cloud so that any resource-constraint client could later retrieve the database records and check whether the cloud returns valid records or not. Meanwhile, the database can be updated by the data owner. To the best of our knowledge, there is no secure and efficient VDB scheme supporting public verification and all kinds of update operations (i.e., insertion, modification, and deletion) simultaneously. To resolve this problem, we propose an efficient VDB scheme by incorporating vector commitment (VC) with Merkle Interval Hash Tree (MIHT). To enhance the security (i.e., resist the forward automatic update attack), we re-formalize the syntax of VC and present a new construction of VC based on modified Generalized Bilinear Inversion (mGBI) assumption. Our proposed VC scheme is used for guaranteeing the integrity of data. Like the traditional VC schemes, our proposed scheme remains publicly verifiable. MIHT is a new primitive introduced by us, which is mainly used to support all update operations. Security analysis shows that our proposed VDB scheme can achieve real-world security requirements. The detailed performance analyses and simulations show that our proposed schemes are more practical in the real world.

1. Introduction

Enterprises and personal users have been increasingly outsourcing their storage and computing tasks to public clouds such as Amazon EC2 and S3, Google App Engine, and Microsoft Azure because of lower cost, higher reliability, better performance, and faster deployment. In this process, arbitrary resource-limited enterprise or user can enjoy unlimited computational and huge storage resources in a pay per use manner [1]. Despite these tremendous benefits, cloud computing inevitably suffers from some new security challenges. *Verifiability* has been the key road block to this technology transformation because once the storage and computing tasks are outsourced to the cloud, enterprise or user would lose control over them. As a result, the cloud may forge the result of the outsourced computation or tamper storage data to save computational resources or shorten response time [2–5]. Therefore, it is necessary to provide the verifiability property in cloud computing. Besides, the computation and storage resources invested by the users in the verification phase must be as few as possible.

In this paper, we focus on the problem of verifiable database (VDB) especially supporting dynamic updates. This problem was initialized by Benabbas et al. [6] in 2011. It involves three kinds of entities: the data owner, the resource-limited client, the powerful but untrusted cloud. The data owner uploads a large database to the cloud such that any resource-limited client could later retrieve the database records and check whether the records have been tampered with by the cloud with an overwhelming probability. Apart from this, the outsourced database should be only updated by the data owner rather than the malicious cloud. They constructed the first VDB scheme in the dynamic setting from delegating polynomial computations. Since the secret key of the data owner is involved in the process of verification, this scheme does not support public verification. Furthermore, it does not support all update operations (i.e., insertion, deletion, and modification). Following this pioneering work, plenty of works [7–11] have been proposed till now. To support public verification, Catalano and Fiore [7] proposed an elegant framework to build efficient VDB from a new primitive called vector commitment (VC). As pointed out by Chen et al. [9], however, Catalano–Fiore's VDB framework from VC is vulnerable to the forward automatic update (FAU) attack since the update algorithm does not involve any secret knowledge. In other words, arbitrary malicious adversary can update the committed message as casually as possible. To resolve this problem, Chen et al. [9] proposed a new VDB framework

* Corresponding author.
E-mail address: fczhou@mail.neu.edu.cn (F. Zhou).

[1] This work was done while the author was visiting Georgia Institute of Technology.

https://doi.org/10.1016/j.future.2020.09.028
0167-739X/

from VC based on the idea of commitment binding in 2014. The construction is not only publicly verifiable but also secure against the FAU attack. Next year, Chen et al. [11] introduced a new primitive called verifiable database with incremental updates, which can reduce the computation cost when the database undergoes frequent but small modifications. However, all of these schemes mentioned above only support modification operation since the number of data records of the outsourced database mush be *fixed in advance* before outsourcing the database. To support all kinds of update operations simultaneously, Miao et al. [8] incorporated Merkle Sum Hash Tree (MSHT) into [6]. This scheme does not support public verification as well since it is constructed based on [6]. Besides, this scheme is inefficient and cannot be applied to the real world since it is based on the bilinear group of composite order. Very recently, Shen et al. [10] design a VDB scheme supporting all kinds of dynamic update operations and public verification by utilizing the polynomial commitment and index-hash table. This scheme is efficient, but it requires the data owner to send his secret key to the Trusted Third Party (TTP). If the TTP colludes with the malicious cloud, the scheme will be no longer secure. Furthermore, it does not rely on the constant-size cryptographic assumption[2] like other schemes [6,8,9]. From these points, it is pressing need to design a secure and efficient VDB scheme supporting all kinds of update operations and public verification simultaneously.

1.1. Related work

With the development of cloud computing, verifiable computation [12,13] is becoming more and more important. It mainly aims to guarantee the correctness of computation results [4, 14,15] and the integrity of queried results [16–18]. Verifiable database (VDB) scheme is a specific verifiable computation scheme, which enables the data owner to move a large database to the cloud such that any resource-limited client could retrieve the records and check the integrity of queried results. According to the existing approaches, the previous VDB schemes can be classified into two-fold: generic ones and specific ones.

1.1.1. Generic schemes

These generic approaches are mainly based on the generic-purpose techniques for verifiable computation [19–21]. It requires the data owner to compile its database into a program, which can be expressed either as a circuit [19,22] or in RAM-model of computation [23–26]. This kind of approach is tested to be far infeasible for practical applications. Besides, these approaches cannot support efficient updates since the program needs to be re-compiled from scratch when an update occurs. Therefore, these generic approaches do not suit for the dynamic scenario.

1.1.2. Specific schemes

Previous works have also explored custom-built VDB schemes. According to database type, the existing approaches can be roughly categorized into two-fold: SQL-oriented ones and NoSQL-oriented ones. For the SQL database, plenty of works [27–31] are proposed for different kinds of queries such as range, count, and join. Tree-based [32–34] and accumulator-based [35–37] approaches are mainly designed for checking the integrity of outsourced data. It utilizes these authenticated data structures to store the authentication information. However, these kinds of methods are all not suitable for insertion and deletion operations since the data structure will be totally changed when these operations occur. Skip-list-based approaches [30,38,39] are mainly designed for range queries. Besides, some researches incorporate these approaches to support more queries [29–31,40]. For the NoSQL database, Benabbas et al. [6] first proposed an efficient VDB from verifiable computation for the polynomial. Following this work, plenty of works [7,9–11] have been proposed by the researchers. As discussed above, these schemes cannot simultaneously fix the following problems: satisfying public verifiability and high efficiency, resisting forward automatic update attack, supporting all update operations. In this paper, we focus on the problem of VDB for NoSQL database especially for the dynamic scenario.

1.2. Our results and contributions

The main achievement of this paper is a secure and efficient verifiable database scheme with additional insertion and deletion operations in cloud computing. Our results and contributions can be summarized as follows:

1. To resist the forward automatic update attack, we re-formalize the syntax of Vector Commitment (VC) and give a concrete construction based a constant-size cryptographic assumption. Furthermore, our scheme satisfies not only public verifiability but also high efficiency.
2. To support all kinds of update operations, we are the first ones to introduce a novel primitive called Merkle Interval Hash Tree (MIHT), which is mainly used for recording the number of data records in each data block. Besides, we give a concrete construction of MIHT and prove its security formally.
3. To design an efficient VDB scheme supporting additional insertion and deletion operations, we incorporate our proposed VC scheme with MIHT scheme. Our construction is based on the bilinear pairing group of prime order instead of composite order. Besides, our proposed VDB scheme supports public verification. The theoretical analyses and simulations show that our proposed VDB scheme is more efficient.

1.3. Paper organization

The rest of this paper is organized as follows. Section 2 introduces a primitive called verifiable database. Section 3 reviews some knowledge needed beforehand. Section 4 re-formalizes the syntax of vector commitment, proposes a detailed construction, and proves its security. Section 5 introduces a new primitive called Merkle Interval Hash Tree, provides a concrete construction of VDB and proves its security. Section 6 provides a detailed theoretical analysis and simulation. Finally, Section 7 concludes this paper.

2. Verifiable databases

We consider the database with the form of (i, v_i), where i is an index and v_i is the corresponding value at the position i. Verifiable database (VDB) consists of three kinds of entities: the data owner, cloud, and client. The data owner and the client are trusted and follow the prescribed protocol. The cloud is malicious and may tamper with the record that the client queries. In setup phase, the data owner outsources the database to the cloud. Then, the client can issue a query request i to the cloud. Upon receiving the query,

[2] Constant-size cryptographic assumptions (such as computational Diffie–Hellman assumption, subgroup membership assumption in composite order bilinear groups) are not parameterized and independent of any system security parameters or the number of oracle queries an adversary issues. In contrast, non-constant-size assumptions (i.e., q-type assumptions) are parameterized. For example, q-strong Diffie–Hellman assumption is non-constant size since the adversary is given the term $(g, g^{\alpha}, g^{\alpha^2}, \ldots, g^{\alpha^q})$ and q is related to the number of oracle queries.

he returns the corresponding record v_i^* to the client. Finally, the client checks whether v_i^* is the true record at the position i (i.e., $v_i^* = v_i$). VDB guarantees that any malicious attempts to forge the record by the cloud will always be detected with an overwhelming probability when the client retrieves the record. Meanwhile, the data owner can update the record stored in the cloud. To the best of our knowledge, most VDB schemes [6,9,11] do not support additional deletion and insertion operations since they require that the size of the database should be fixed in advance. If an element is inserted or deleted, the size of the outsourced database will be changed. As a result, the protocol needs to be conducted from scratch. Although there exist few schemes [8,10] supporting deletion and insertion operations, they cannot be applied in the real word from the aspect of the security or efficiency as discussed in Section 1. To this end, we design a more secure and efficient VDB scheme, which additionally supports deletion and insertion operations.

Definition 1. A VDB scheme is a tuple of four algorithms Setup, Query, Verify, Update defined as follows:

(PK, SK) ← Setup(1^κ, *DB*): The key generation algorithm is run by the database owner. It takes as input the security parameter κ and the database *DB* and outputs the public key PK and the private key SK.

(v_i, π_{v_i}) ← Query(i, PK): The query algorithm is run by the client. It takes as input the index i and the public key PK and outputs the corresponding result v_i and witness π_{v_i}.

{0 or 1} ← Verify(PK, i, v_i, π_{v_i}): The public verification algorithm is run by any client/database owner. It takes as input the public key PK, the query i, the corresponding result v_i and witness π_{v_i}. It outputs 1 if π_{v_i} is a valid witness and 0 otherwise.

PK′ ← Update(upd, i, v_i', SK, PK): The update algorithm is run by the database owner. It takes as input an update operation upd ∈ {insert, delete, modify}, the index i, the updated value v_i', the private key SK and the public key PK and outputs a new public key PK′ with respect to the latest database. Note that the term $v_i' = null$ when upd = delete.

Due to the limitation of the space, we refer the reader to [6,9] for the definitions of the correctness and security.

3. Preliminaries

In this section, we first give some necessary definitions that are going to be used in the rest of the paper. We use κ to denote the security parameter and $negl(\kappa)$ to denote the negligible function of κ. PPT is an abbreviation of probabilistic polynomial time. If $\mathcal{D}$ is a set, then $j \xleftarrow{\$} \mathcal{D}$ indicates the process of selecting j uniformly at random over $\mathcal{D}$. If q is an integer, we denote the set $\{0, \ldots, q-1\}$ with $[q]$. $a \parallel b$ is the concatenation of two strings a, b. Table A.4 in Appendix gives a quick reference for the notations in the paper.

3.1. Bilinear pairings

Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two cyclic multiplicative groups with a same prime order p and let g be a generator of $\mathcal{G}_1$. Let $e : \mathcal{G}_1 \times \mathcal{G}_1 \to \mathcal{G}_2$ be a bilinear map which satisfies the following properties:

1. Bilinearity. $\forall u, v \in \mathcal{G}_1$ and $a, b \in \mathbb{Z}_p$, $e(u^a, v^b) = e(u, v)^{ab}$.
2. Non-degeneracy. $e(g, g) \neq 1_{\mathcal{G}_2}$.
3. Computability. $\forall u, v \in \mathcal{G}_1$, There exists an efficient algorithm to compute $e(u, v)$.

3.2. Modified generalized bilinear inversion (mGBI) assumption [41]

Given $\mathcal{P} \in \mathcal{G}_1$ and $e(\mathcal{P}, \mathcal{Q}) \in \mathcal{G}_2$, there exists no PPT algorithm that can output $\mathcal{Q}$ except with negligible probability $negl(\kappa)$.

3.3. Algebraic pseudo-random function

We use the algebraic pseudo-random function protocol APRF as a building block in our construction of vector commitment. The APRF is first proposed by Benabbas et al. [6]. The algebraic pseudo-random function is a pseudo-random function (PRF) with a special property called *closed form efficiency*. This property guarantees that some certain algebraic operations on the outputs of pseudo-random function can be computed significantly more efficiently if one possesses the key of the pseudo-random function. For simplicity, we extract the system model from their protocol. The extracted model of APRF is comprised of three algorithms as follows:

(K, params) ← APRF.KeyGen(1^κ, ℓ): Given the security parameter κ and a parameter $\ell \in \mathbb{N}$ that determines the domain size of the pseudo-random function, the key generation algorithm first chooses a group $\mathcal{G}_1$ with the order p and picks up a generator $g \in \mathcal{G}_1$. After that it chooses $(\ell \cdot (s+1)+1)$ random numbers $k_0, k_{1,1}, \ldots, k_{1,s+1}, \ldots, k_{\ell,1}, \ldots, k_{\ell,s+1} \in \mathbb{Z}_p$. It sets params = $((\ell, s), p, g, \mathcal{G}_1)$ and K = $(k_0, k_{1,1}, \ldots, k_{1,s+1}, \ldots, k_{\ell,1}, \ldots, k_{\ell,s+1})$. Finally, it outputs a pair (K, params) $\in \mathcal{K}_\kappa$, where $\mathcal{K}_\kappa$ is the key space for security parameter κ.

r_x ← APRF.F$_K$(params, x): Given the public parameters params and an input $x \in \{0,1\}^\ell$, the random generation algorithm first parses x as a vector $(x_1, \ldots, x_\ell)$ with each $x_i = [x_{i,1}, \ldots, x_{i,s+1}]$ as an $(s+1)$-bit string. After that, it computes and outputs

$$r_x = g^{k_0 k_{1,1}^{x_{1,1}} \cdots k_{1,s+1}^{x_{1,s+1}} \cdots k_{\ell,1}^{x_{\ell,1}} \cdots k_{\ell,s+1}^{x_{\ell,s+1}}} \in Y,$$

where Y is some set determined by params. In the following, we will omit the public parameters params from this algorithm, and write APRF.F$_K(x)$ instead.

y_x ← APRF.CFEval$_{h,z}(x$, K): Let $h : \mathbb{Z}_p^\ell \to \mathbb{Z}_p^l$ and $x_1, \ldots, x_\ell$ be a binary represent of x, where $l = d \cdot \ell$, be defined as:

$$h(x) = h(x_1, \ldots, x_\ell) = \left\{h_0(x_1, \ldots, x_\ell), \ldots, h_{q-1}(x_1, \ldots, x_\ell)\right\},$$

where $h_i(x_1, \ldots, x_\ell) = h_{i_1,\ldots,i_\ell}(x_1, \ldots, x_\ell) = \left(x_1^{i_1} \cdots x_\ell^{i_\ell}\right)$. Let $z = [z_1, \ldots, z_l] = [(i_1, \ldots, i_\ell)]_{i_1,\ldots,i_\ell \le d}$. Given the input the function h, the vector z, the input x and the key K, the closed form evaluation algorithm can compute the polynomial of the form[3]

$$y_x = R(x_1, \ldots, x_\ell) = g^{r(x_1,\ldots,x_\ell)} = \prod_{i_1,\ldots,i_\ell \le d} \text{APRF.F}_K(i_1, \ldots, i_\ell)^{x_1^{i_1} \cdots x_\ell^{i_\ell}} \quad (1)$$

by conducting

$$\text{APRF.CFEval}_{h,z}(x_1, \ldots, x_\ell, K) = g^{k_0 \prod_{j=1}^{\ell}(1+k_{j,1}x_j)\left(1+k_{j,2}x_j^2\right)\cdots\left(1+k_{j,s}x_j^{2^s}\right)} \quad (2)$$

where the APRF.F is initialized with ℓ and $s = \lceil \log d \rceil$.

A secure APRF is required to satisfy the following properties:

1. **Algebraic**: If the range Y of APRF.F$_K(\cdot)$ for every $\kappa \in \mathbb{N}$ and (K, params) $\in \mathcal{K}_\kappa$ forms an abelian group, we say that *PRF* is algebraic.
2. **Pseudo-random**: *PRF* is pseudo-random if for arbitrary adversary $\mathcal{A}$, every polynomial $m(\cdot)$ and all $n \in \mathbb{N}$, the following holds:

$$\left| \Pr\left[A^{\text{APRF.F}_K(\cdot)}\left(1^n, \text{params}\right) = 1\right] - \Pr\left[A^{R(\cdot)}\left(1^n, \text{params}\right) = 1\right] \right| \le negl(\kappa)$$

where (K, params) ← APRF.KeyGen$(1^\kappa, m(\kappa))$, and $R : \{0,1\}^m \to Y$ is a random function.

[3] It is a polynomial of degree d in each variable.

3. **Closed form efficiency**: There exists a more efficient way to compute a "weighted product" of l PRF values than by computing l values separately and then combining them given the private key K.

The security of APRF scheme relies on d-strong decisional Diffie–Hellman assumption [42]. For more detail, we refer the reader to the literature [6]. In our construction of vector commitment, we use the special case of this construction, where $s = 0$ (see Section 4.2). When $s = 0$, for simplicity, we directly use x_i, k_i to present $x_{i,s+1}$ and $k_{i,s+1}$, respectively.

3.4. Digital signature [43]

Digital signature is used to guarantee that the contents of a message have not been altered in transition. The digital signature scheme is a tuple of three algorithms S.KeyGen, S.Sign, S.Verify defined as follows:

(SPK, SSK) $\leftarrow$ S.KeyGen(1^κ). The key generation algorithm takes as input the security parameter κ and outputs a key pair (SPK, SSK).

$\sigma \leftarrow$ S.Sign(SSK, m). The signature algorithm takes as input the private key SSK and the message m and outputs the signature σ.

$\{0, 1\} \leftarrow$ S.Verify(σ, m, SPK). The verification algorithm takes as input the signature σ, the message m and the public key SPK. It outputs 1 if the signature σ is valid and 0 otherwise.

3.5. Merkle hash tree

A Merkle hash Tree (MHT) [44] is a complete binary tree, which can prove that a given value is indeed stored in the leaf node. The MHT is constructed from bottom to top. Each leaf node n_i consists of three parts: the index i, the value v_{n_i}, and the hash value Υ_{n_i}, while the interval node is comprised of two parts: the index i and the hash value Υ_{n_i}. For any leaf node, the hash value is the hash of index and value. For any internal node, the hash value is computed through its index, value and hash values of its two left and right child nodes. After the construction, the prover generates the signature on the hash value of the root node. To prove certain value is stored in some leaf nodes, the prover needs to generate the proof including leaf nodes and all sibling nodes in the path from that leaf node to the root node. The verifier can reconstruct the hash value of the root node through the proof and check the validity of the signature. The security of MHT relies on the collision-resistant property of the hash function.

4. A novel vector commitment

Catalano and Fiore [7] firstly formalized the notion of vector commitment (VC), which allows a committer to commit to an ordered sequence of values $\{m_1, \ldots, m_q\}$ in such a way that it is later possible to open the commitment at some particular position. It is required to satisfy an essential property: *position binding*.[4] Position binding means that anyone cannot be able to open a commitment to two different values at the same position.

Since the algorithm VC.Update does not involve any secret knowledge of the committer (i.e., the private key), arbitrary adversary $\mathcal{A}$ can perform the algorithm VC.Update in the same way as the committer. Hence, it suffers from the forward automatic update attack [9]. That is, the adversary $\mathcal{A}$ firstly retrieves the current record m at the position i. Then, $\mathcal{A}$ stealthily updates the old value m_i to a new value m_i' at the position i by conducting $\mathsf{VC.Update}_{\mathsf{pp}}(C, m_i, i, m_i')$. Finally, the adversary $\mathcal{A}$ outputs the new commitments C' and an update information U'. It is worth noting that each procedure does not require any secret knowledge of the committer in this process. The new commitment C' and the old commitment C are computationally indistinguishable from the view of any third party. As a result, the adversary $\mathcal{A}$ may claim that the updated commitment C' is the original commitment generated by the committer. When this dispute occurs, the judge cannot deduce who is trusted. To fix this problem, we re-formalize syntax by adding a private key VSK in algorithms VC.KeyGen and VC.Update. The biggest difference between our proposed scheme and Catalano and Fiore's VC scheme is that our scheme can resist the forward automatic update attack.

Definition 2 (*Vector Commitment*)**.** The vector commitment scheme is a tuple of six algorithms VC = (VC.KeyGen, VC.Com, VC.Open, VC.Ver, VC.Update, VC.ProofUpdate) defined as follows:

$\{\mathsf{pp}, \mathsf{VSK}\} \leftarrow \mathsf{VC.KeyGen}(1^\kappa, q)$. The key generation algorithm is run by the committer. It takes as input the security parameter κ and the size q of the committed vector (with $q = poly(\kappa)$) and outputs some public parameters pp (which implicitly define the message space $\mathcal{M}$) and a private key VSK.

$(C, \mathsf{aux}) \leftarrow \mathsf{VC.Com}_{\mathsf{pp}}(m_0, \ldots, m_{q-1})$. The committing algorithm is run by the committer. It takes as input the public parameters pp and a sequence of q messages $m_0, \ldots, m_{q-1} \in \mathcal{M}$ and outputs a commitment C and an auxiliary information aux.

$\{m_x, \Lambda_x\} \leftarrow \mathsf{VC.Open}_{\mathsf{pp}}(x, \mathsf{aux}, C)$. The opening algorithm is run by the committer. It takes as input the public parameters pp, the position x, the auxiliary information aux and the commitment C and outputs the xth committed message m_x and a proof Λ_x.

$\{0, 1\} \leftarrow \mathsf{VC.Ver}_{\mathsf{pp}}(C, m, x, \Lambda_x)$. The verification algorithm is run by any client. It takes as input the public parameters pp, the commitment value C, the opened message m, the position x and the proof Λ_x. It outputs 1 if Λ_x is a valid proof that C is a commitment to the sequence $(m_1, \ldots, m_q)$ such that $m = m_x$ and 0 otherwise.

$(C', U) \leftarrow \mathsf{VC.Update}_{\mathsf{pp}}(C, m, m', i, \mathsf{VSK})$. The update algorithm is run by the original committer who wants to update C by changing the ith message m to m'. It takes as input the public parameters pp, the commitment C, the old message m, the new message m', the position i and the private key VSK and outputs a new commitment C' together with an update information U.

$\Lambda_j' \leftarrow \mathsf{VC.ProofUpdate}_{\mathsf{pp}}(\Lambda_j, U)$. The proof update algorithm is run by any client who holds a proof Λ_j for some message at position j. It takes as input the public parameters pp, the proof Λ_j for some message at position j and the update information U and outputs an updated proof Λ_j' for m_j with respect to the new sequence $(m_0, \ldots, m_{j-1}, m', m_{j+1}, \ldots, m_{q-1})$.

4.1. Correctness and security definitions

Intuitively, a vector commitment is correct if whenever anyone follows the prescribed protocol, it will always get the desired result. More formally:

Definition 3 (*Correctness*)**.** Let M_a denote the sequence constructed after a invocations of the VC.Update algorithm (starting from the original sequence M_0 with size q) and likewise for C_a, U_a, aux_a. A vector commitment scheme is correct if the following holds:

$$\Pr\left[\begin{array}{l} \{\mathsf{pp}, \mathsf{VSK}\} \leftarrow \mathsf{VC.KeyGen}(1^\kappa, q); \\ (C_0, \mathsf{aux}_0) \leftarrow \mathsf{VC.Com}_{\mathsf{pp}}(M_0); \\ \text{From } a = 0 \text{ to } \ell = poly(k): \\ \quad j \xleftarrow{\$} \{1, \ldots, q\}; \\ \quad (C_{a+1}, U_{a+1}) \leftarrow \mathsf{VC.Update}_{\mathsf{pp}}(C_a, \\ \qquad\qquad m_a, m_a', j, \mathsf{VSK}); \\ \{m_x, \Lambda_x\} \leftarrow \mathsf{VC.Open}_{\mathsf{pp}}(x, \mathsf{aux}_\ell, C_\ell): \\ \mathsf{VC.Ver}_{\mathsf{pp}}(C_\ell, m_x, x, \Lambda_x) = 1 \end{array}\right] \geq 1 - negl(k)$$

[4] Unlike standard commitments, hiding is not a crucial property in vector commitment.

Intuitively, a vector commitment scheme satisfies position binding if it is infeasible to open a commitment to two different values at the same position i. More formally:

Definition 4 (*Position Binding*)**.** Let VC be a vector commitment scheme, and let $\mathcal{A}(\cdot) = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4)$ be a five-tuple of PPT machine. We define security via the following experiment.

$\mathrm{Exp}^{\mathrm{PB}}_{\mathcal{A}}[VC, \mathcal{M}, \kappa]$:
$\{\mathsf{pp}, \mathsf{VSK}\} \leftarrow \mathsf{VC.KeyGen}(1^\kappa, q)$;
$M_0 \leftarrow \mathcal{A}_0(\mathsf{pp}, \kappa, q)$;
$(C_0, \mathsf{aux}_0) \leftarrow \mathsf{VC.Com}_{\mathsf{pp}}(M_0)$;
From $a = 0$ to $\ell = poly(\kappa)$:
  $x \leftarrow \mathcal{A}_1(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a)$;
  $\Lambda_a \leftarrow \mathsf{VC.Open}_{\mathsf{pp}}(m_x, x, \mathsf{aux}_a, C_a)$;
  $j \leftarrow \mathcal{A}_2(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a, \Lambda_0, \cdots, \Lambda_a)$;
  $(C_{a+1}, U_{a+1}) \leftarrow \mathsf{VC.Update}_{\mathsf{pp}}(C_a, m_j, m'_j, j, \mathsf{VSK})$;
$x^* \leftarrow \mathcal{A}_3(\mathsf{pp}, \{C_a, U_a, \Lambda_a\}_{0\le a\le \ell})$;
$(m'_{x^*}, \Lambda'_{x^*}) \leftarrow \mathcal{A}_4(\mathsf{pp}, x^*, \{C_a, U_a, \Lambda_a\}_{0\le a\le \ell})$;
$b \leftarrow \mathsf{VC.Ver}_{\mathsf{pp}}(C_\ell, m'_{x^*}, x^*, \Lambda'_{x^*})$;
If $m'_{x^*} \neq m_{x^*}$ and $b = 1$:
  output 1;
else
  output 0;

For any $\kappa \in \mathbb{N}$, we define the advantage of arbitrary $\mathcal{A}$ in the above experiment against VC as

$$Adv^{\mathrm{BP}}_{\mathcal{A}}(VC, \mathcal{M}, \kappa) = \Pr[\mathrm{Exp}^{\mathrm{BP}}_{\mathcal{A}}[VC, \mathcal{M}, \kappa] = 1]$$

We say that VC achieves position binding if $Adv^{\mathrm{BP}}_{\mathcal{A}}(VC, \mathcal{M}, \kappa) \le negl(\kappa)$.

Remark 1. The latest auxiliary information aux_a in above experiment can be generated by the updated information U_a and old auxiliary information aux_{a-1}.

Definition 5 (*Concise*)**.** A vector commitment scheme is concise if the size of proof Λ_x is independent of q (i.e., constant).

4.2. A detailed construction

The basic idea behind our construction is the following. The committer uses the committer sequence $\{m_0, \ldots, m_{q-1}\}$ to construct an ℓ-variate polynomial $f(x_1, \ldots, x_\ell)$ such that $f(i_1, \ldots, i_\ell) = m_i$ for $\forall i \in \{0, \ldots, q-1\}$, where $i_1, \ldots, i_\ell$ is the binary representation of i. In our construction, the polynomial $f(x_1, \ldots, x_\ell)$ can be directly generated in a specific way rather than in Lagrange Interpolating way [45]. Apart from this, when an update occurs, our approach does not need to perform the protocol from scratch while the method based on Lagrange Interpolating needs to perform the protocol from scratch. Without loss of generality, let a_i be the ith coefficient of polynomial $f(x_1, \ldots, x_\ell)$. The commitment is a vector C of group elements of the form $g^{a\cdot a_i + r_i}$, where $a \in \mathbb{Z}_p$ and r_i is the ith coefficient of a random polynomial $r(x_1, \ldots, x_\ell)$ of the same degree as $f(x_1, \ldots, x_\ell)$. When the commitment is opened at the position x, the committer returns the committed message $m_x = f(x_1, \ldots, x_\ell)$ and the proof $\Lambda_x = e(g, g^{a\cdot f(x_1,\ldots,x_\ell) + r(x_1,\ldots,x_\ell)})$ to the client. Upon receiving the response, the client verifies whether m_x is the true message committed at the position x by checking whether $\Lambda_x = e(g,g)^{a\cdot m_x} \cdot e(g, g^{r(x_1,\ldots,x_\ell)})$ holds. To improve the efficiency, we take advantage of algebraic pseudo-random function to generate the polynomial $r(x_1, \ldots, x_\ell)$ rather than by choosing a random polynomial. As a result, we can use the closed form efficiency property of algebraic pseudo-random function to compute $g^{r(x_1,\ldots,x_\ell)}$ more efficiently.

The construction of our scheme is detailed as follows:

$\{\mathsf{pp}, \mathsf{VSK}\} \leftarrow \mathsf{VC.KeyGen}(1^\kappa, q)$. Given a security parameter κ, the committer first chooses two groups $\mathcal{G}_1$ and $\mathcal{G}_2$ with the same order $p \in poly(\kappa)$ along with a bilinear map $e : \mathcal{G}_1 \times \mathcal{G}_1 \to \mathcal{G}_2$. Let g be a generator of $\mathcal{G}_1$. After that, it computes $\ell = \lceil \log q \rceil$. For simplicity, we assume that $q = 2^\ell$ in the following. It generates a pair $(\mathsf{K}, \mathsf{params})$ by conducting algorithm $\mathsf{APRF.KeyGen}(1^\kappa, \ell)$. Note that $s = 0$ in params. Next, it picks a random $a \in \mathbb{Z}_p$ and computes $\mathsf{VVK} = e(g,g)^a$. For $\forall i \in \{0, 1, \ldots, q-1\}$, it runs the random generation algorithm $\mathsf{APRF.F_K}(i)$ to product a random number $\mathsf{R}[i]$ and the closet form evaluation algorithm $\mathsf{APRF.CFEval}_{h,z}(i, \mathsf{K})$ to product the value $\mathsf{Y}[i]$, where $\mathsf{Y}[i] = R(i_1, \ldots, i_\ell)$ (see Eq. (1)), and $i_1, \ldots, i_\ell$ is the binary representation of i. It sets:

$$\mathsf{pp} = \left\{ \begin{array}{l} \mathsf{VVK}, \mathsf{params}, (e, g, p, \mathcal{G}_1, \mathcal{G}_2), \\ \Theta = \{(\mathsf{R}[i], \mathsf{Y}[i]) | 0 \le i \le q-1\} \end{array} \right\}$$

and $\mathsf{VSK} = \{\mathsf{K}, a\}$. Finally, it publishes the public parameters pp and keeps the private key VSK secret. The message space is $\mathcal{M} = \mathbb{Z}_p$.[5] Note that the value $\mathsf{Y}[i]$ will never be changed when the key K is fixed.

$(C, \mathsf{aux}) \leftarrow \mathsf{VC.Com}_{\mathsf{pp}}(m_0, \ldots, m_{q-1})$. The committer first parses public parameter pp as $\{\mathsf{VVK}, \mathsf{params}, (e, g, p, \mathcal{G}_1, \mathcal{G}_2), \Theta = \{(\mathsf{R}[i], \mathsf{Y}[i]) | 0 \le i \le q-1\}\}$. After that, it constructs an ℓ-variate polynomial $f(x_1, \ldots, x_\ell)$ as follows[6] :

$$f(x_1, \ldots, x_\ell) = \sum_{i=0}^{q-1} a_i \cdot x_1^{i_1} \cdots x_\ell^{i_\ell} = \sum_{i=0}^{q-1} \left(m_i \cdot \prod_{k=1}^{\ell} \mathsf{pick}_{i_k}(x_k) \right), \quad (3)$$

where

$$\mathsf{pick}_{i_k}(x_k) = \left\{ \begin{array}{ll} x_k & \text{if } i_k = 1 \\ 1 - x_k & \text{if } i_k = 0 \end{array} \right. , \quad (4)$$

and $i_1, \ldots, i_\ell$ is the binary representation of i. Next, it generates the commitment $C = \{C[i] | 0 \le i \le q-1\}$ by computing:

$$C[i] = (g^{a_i})^a \cdot \mathsf{R}[i]. \quad (5)$$

Finally, it outputs the commitment C and auxiliary information $\mathsf{aux} = (m_0, \ldots, m_{q-1})$. It is worth noting that $f(i_1, \ldots, i_\ell) = m_i$.

$\{m_x, \Lambda_x\} \leftarrow \mathsf{VC.Open}_{\mathsf{pp}}(x, \mathsf{aux}, C)$. The committer first parses public parameters pp, aux and x as $\{\mathsf{VVK}, \mathsf{params}, (e, g, p, \mathcal{G}_1, \mathcal{G}_2), \Theta = \{(\mathsf{R}[i], \mathsf{Y}[i]) | 0 \le i \le q-1\}\}$, $(m_0, \ldots, m_{q-1})$ and the binary representation $x_1, \ldots, x_\ell$, respectively. After that, it sets:

$$h(x) = \left\{ h_0(x), \ldots, h_{q-1}(x) \right\} \quad (6)$$

where $h_i(x) = h_{i_1,\ldots,i_\ell}(x_1, \ldots, x_\ell) = \left(x_1^{i_1} \cdots x_\ell^{i_\ell} \right)$. Next, it parses the commitment C as $\{C[i] | 0 \le i \le q-1\}$ and computes the proof Λ_x as follows:

$$\Lambda_x = \prod_{i=0}^{q-1} e(g, C[i]^{h_i(x)}) \quad (7)$$

Finally, it outputs the xth committed message m_x along with the corresponding proof Λ_x.

$\{0, 1\} \leftarrow \mathsf{VC.Ver}_{\mathsf{pp}}(C, m_x, x, \Lambda_x)$. After the client receives the opening m_x and the proof Λ_x, it first parses public parameters pp as

$$\mathsf{pp} = \left\{ \begin{array}{l} \mathsf{VVK}, \mathsf{params}, (e, g, p, \mathcal{G}_1, \mathcal{G}_2), \\ \Theta = \{(\mathsf{R}[i], \mathsf{Y}[i]) | 0 \le i \le q-1\} \end{array} \right\}.$$

[5] The scheme can be easily extended to support arbitrary messages in $\{0,1\}^*$ by using a collision-resistant hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_p$.

[6] To further illustrate it detailly, we take for an instantiation $q = 4$. Without loss of generality, we assume vector $(m_0, m_1, m_2, m_3) = (8, 2, 4, 5)$. it is $f(x_1, x_2) = 8 \cdot (1 - x_1)(1 - x_2) + 2 \cdot (1 - x_1)x_2 + 4 \cdot x_1(1 - x_2) + 5 \cdot x_1 x_2$.

Next, it outputs 1 (valid) or 0 (invalid) by checking whether the following equation holds:

$$\Lambda_x = \mathsf{VVK}^{m_x} \cdot e(g, \mathsf{Y}[x]) \tag{8}$$

$(C', U) \leftarrow \mathsf{VC.Update}_{\mathsf{pp}}(C, m, m', i, \mathsf{VSK})$. The committer first parses the private key VSK, the public parameter pp and the commitment C as $\{\mathsf{K}, a\}$, $\{\mathsf{VVK}, \mathsf{params}, (e, g, p, \mathcal{G}_1, \mathcal{G}_2), \Theta = \{(\mathsf{R}[i], \mathsf{Y}[i]) | 0 \le i \le q-1\}\}$ and $\{C[i] | 0 \le i \le q-1\}$, respectively. After that, it computes $\delta = (m' - m)$ and generates ℓ-variate polynomial $p(x_1, \ldots, x_\ell)$ by computing

$$p(x_1, \ldots, x_\ell) = \sum_{\gamma=0}^{q-1} c_j \cdot {x_1}^{\gamma_1} \cdots {x_\ell}^{\gamma_\ell} = \delta \cdot \prod_{k=1}^{\ell} \mathsf{pick}_{i_k}(x_k), \tag{9}$$

where $\gamma_1, \ldots, \gamma_\ell$ is the binary representation of γ. Afterwards, it generates the updated commitment:

$$C' = \left\{ C[0] \cdot g^{a \cdot c_0}, \ldots, C[q-1] \cdot g^{a \cdot c_{q-1}} \right\}.$$

Finally, it outputs the updated commitment C' and the updated information $U = (i, \delta)$.

$\Lambda_j' \leftarrow \mathsf{VC.ProofUpdate}_{\mathsf{pp}}(\Lambda_j, m', U)$. A client who owns a proof Λ_j, that is a valid with respect to C for some message at position j, can use the update information $U = (i, \delta)$ to produce a new proof Λ_j' for m_j with respect to C'. It parses pp as $\mathsf{pp} = \{\mathsf{VVK}, \mathsf{params}, (e, g, p, \mathcal{G}_1, \mathcal{G}_2), \Theta = \{(\mathsf{R}[i], \mathsf{Y}[i]) | 0 \le i \le q-1\}\}$. Next, it computes the updated proof as below:

$$\Lambda_j' = \Lambda_j \cdot \mathsf{VVK}^{\delta} \tag{10}$$

4.3. Correctness and security analysis

4.3.1. Correctness

If the vector commitment achieves correctness property as defined in Definition 3, algorithm $\mathsf{VC.Ver}_{\mathsf{pp}}$ will output 1. If the algorithm $\mathsf{VC.Ver}_{\mathsf{pp}}$ outputs 1, Eq. (8) must hold. Therefore, we will show the correctness of our construction mainly based on Eq. (8) in the following.

Due to Eq. (2), the right-hand side (RHS) of Eq. (8) can be expressed as below:

$$\mathrm{RHS} = \mathsf{VVK}^{m_x} \cdot e\left(g, \mathsf{APRF.CFEval}_{h,z}(x_1, \ldots, x_\ell, \mathsf{K})\right)$$

Since $\mathsf{VVK} = e(g, g)^a$, we can re-write the above equation as

$$\mathrm{RHS} = e(g, g)^{a \cdot m_x} \cdot e\left(g, \mathsf{APRF.CFEval}_{h,z}(x_1, \ldots, x_\ell, \mathsf{K})\right)$$

According to Eq. (3), the above equation can be expressed as follows:

$$\begin{aligned}\mathrm{RHS} &= e(g, g)^{a \cdot f(x_1, \ldots, x_\ell)} \cdot e\left(g, \mathsf{APRF.CFEval}_{h,z}(x_1, \ldots, x_\ell, \mathsf{K})\right) \\ &= e(g, g)^{a \cdot \sum_{i=0}^{q-1} a_i \cdot {x_1}^{i_1} \cdots {x_\ell}^{i_\ell}} \cdot e\left(g, \mathsf{APRF.CFEval}_{h,z}(x_1, \ldots, x_\ell, \mathsf{K})\right)\end{aligned}$$

Based on Eqs. (1) and (2), the above equation can be re-written as follows:

$$\mathrm{RHS} = e(g, g)^{a \cdot \sum_{i=0}^{q-1} a_i \cdot {x_1}^{i_1} \cdots {x_\ell}^{i_\ell}} \cdot e\left(g, \prod_{i=0}^{q-1} \mathsf{APRF.F}_{\mathsf{K}}(i)^{{x_1}^{i_1} \cdots {x_\ell}^{i_\ell}}\right)$$

Since $\mathsf{APRF.F}_{\mathsf{K}}(i) = \mathsf{R}[i]$, the above equation can be expressed as follows:

$$\mathrm{RHS} = e(g, g)^{a \cdot \sum_{i=0}^{q-1} a_i \cdot {x_1}^{i_1} \cdots {x_\ell}^{i_\ell}} \cdot e\left(g, \prod_{i=0}^{q-1} \mathsf{R}[i]^{{x_1}^{i_1} \cdots {x_\ell}^{i_\ell}}\right)$$

Because $h_i(x) = x_1^{i_1} \cdots x_\ell^{i_\ell}$, the above equation can be re-written as follows:

$$\begin{aligned}\mathrm{RHS} &= e(g, g)^{a \cdot \sum_{i=0}^{q-1} a_i \cdot h_i(x)} \cdot e\left(g, \prod_{i=0}^{q-1} \mathsf{R}[i]^{h_i(x)}\right) \\ &= e\left(g, \prod_{i=0}^{q-1} g^{a \cdot a_i \cdot h_i(x)}\right) \cdot e\left(g, \prod_{i=0}^{q-1} \mathsf{R}[i]^{h_i(x)}\right) \\ &= e\left(g, \prod_{i=0}^{q-1} g^{a \cdot a_i \cdot h_i(x)} \cdot \mathsf{R}[i]^{h_i(x)}\right) \\ &= \prod_{i=0}^{q-1} e\left(g, g^{a \cdot a_i \cdot h_i(x)} \cdot \mathsf{R}[i]^{h_i(x)}\right) \\ &= \prod_{i=0}^{q-1} e\left(g, \left(g^{a \cdot a_i} \cdot \mathsf{R}[i]\right)^{h_i(x)}\right)\end{aligned}$$

Due to Eq. (5), the above equation can be expressed as follows:

$$\mathrm{RHS} = \prod_{i=0}^{q-1} e\left(g, C[i]^{h_i(x)}\right)$$

According to Eq. (7), the above equation can be rewritten as follows:

$$\mathrm{RHS} = \Lambda_x$$

Obviously, if the committer and client are honest and follow all procedures described above, the opening m_x and the proof Λ_x can always pass the client's verification. This completes the proof.

Remark 2. It is not hard to find that we can efficiently generate the polynomial $f(x_1, \ldots, x_\ell)$ based on Eq. (3) rather than in Lagrange Interpolating way [45]. Furthermore, the ith coefficient a_i of the polynomial is equal to ith message m_i.

Remark 3. It is easy to find that the coefficient $c_j \in \{0, \pm 1, \pm\delta\}$ in Eq. (9). The committer only needs to conduct 1 exponentiation operations (i.e., g^δ) and at most q multiplication operations over the group $\mathcal{G}_1$. When the first element m_0 is updated, the committer needs to conduct q multiplication operations over the group $\mathcal{G}_1$. Therefore, our algorithm VC.Update is efficient.

Remark 4. Based on Eq. (8), $\Lambda_j = \mathsf{VVK}^{m_j} \cdot e(g, \mathsf{Y}[j])$. When the jth committed message m_j is updated to m_j', the value of term $\mathsf{Y}[j]$ will not be changed (see algorithm VC.KeyGen). Therefore, the updated proof $\Lambda_j' = \mathsf{VVK}^{m_j'} \cdot e(g, \mathsf{Y}[j]) = \mathsf{VVK}^{m_j + \delta} \cdot e(g, \mathsf{Y}[j]) = \Lambda_j \cdot \mathsf{VVK}^{\delta}$ in algorithm VC.ProofUpdate.

4.3.2. Security

Theorem 1. *If there exists a PPT adversary $\mathcal{A}$ winning the position binding experiment as defined in Definition 4 with non-negligible probability $\mathrm{negl}(\kappa)$, then adversary $\mathcal{A}$ can construct an efficient algorithm $\mathcal{B}$ to break mGBI assumption.*

Proof. The adversary $\mathcal{A}$ outputs a forgery at position $x^* \in [q]$. The forget consists of a claimed opening m'_{x^*} at the position x^* and a corresponding proof Λ'_{x^*}. If $\mathcal{A}$ breaks the position binding, the following must be true: $m'_{x^*} \neq m_{x^*}$ and $\mathsf{VC.Ver}_{\mathsf{pp}}(C_\ell, m'_{x^*}, x^*, \Lambda'_{x^*}) = 1$. Then the adversary $\mathcal{A}$ will leverage this forgery to construct a method $\mathcal{B}$ to solve mGBI problem.

Supposed $\mathcal{A}$ forges the wrong pair $(m'_{x^*}, \Lambda'_{x^*})$ passing the verification, the following equation holds:

$$\Lambda'_{x^*} = \mathsf{VVK}^{m'_{x^*}} \cdot e\left(g, \mathsf{Y}[x^*]\right). \tag{11}$$

The adversary $\mathcal{A}$ sends the index x^* and the message m_{x^*} at this position to the opening oracle and gets the proof Λ_{x^*}. Similarly, for the correct opening m_{x^*} and the proof Λ_{x^*}, we have:

$$\Lambda_{x^*} = \mathsf{VVK}^{m_{x^*}} \cdot e\left(g, \mathsf{Y}[x^*]\right). \tag{12}$$

Due to Eqs. (11) and (12), it is not hard to see that:

$$e(g, g^{a\cdot(m'_{x^*}-m_{x^*})}) = \mathsf{VVK}^{m'_{x^*}-m_{x^*}} = \frac{\Lambda'_{x^*}}{\Lambda_{x^*}}. \tag{13}$$

According to Eq. (7), the term $\prod_{i=0}^{q-1} C[i]^{h_i(x)}$ is first computed in the process of calculating the proof Λ_{x^*}. Therefore, the adversary $\mathcal{A}$ needs to first forge a value V', and then to generate the forged proof Λ'_{x^*} by computing $e(g, V')$. For simplicity, we assume term $V = \prod_{i=0}^{q-1} C[i]^{h_i(x)}$. The Eq. (13) can be expressed as:

$$e(g, g^{a\cdot(m'_{x^*}-m_{x^*})}) = e(g, V'/V) \tag{14}$$

The left hand of Eq. (14) can be evaluated by the adversary $\mathcal{A}$, so $\mathcal{A}$ can get the value of $e(g, g^{a\cdot(m'_{x^*}-m_{x^*})})$. By knowing g and $e(g, g^{a\cdot(m'_{x^*}-m_{x^*})})$, if $\mathcal{A}$ succeeds to forge m'_{x*} and the witness Λ'_{x^*} passing the verification algorithm, $\mathcal{A}$ will provide an efficient method $\mathcal{B}$ to break mGBI assumption.

Theorem 2. *If the size of proof Λ_x is independent of q, then a vector commitment scheme is concise.*

Proof. It is not hard to see that Λ_x is one single element in $\mathcal{G}_2$. Obviously, it is independent of q. Therefore, our proposed scheme is concise. This completes the proof.

5. Verifiable databases with additional insertion and deletion operations

In this section, we first give the building block of our proposed scheme. Next, we present a concrete construction. Finally, we prove its security required in the real world.

5.1. Building block: Merkle interval hash tree

To support insertion and deletion operations, we introduce a new primitive called Merkle interval hash tree (MIHT), which can be regarded as an extension of general Merkle hash tree (MHT) [44]. The main difference between them is that the data block of the internal node in MIHT additionally consists of a value. The value is equal to the sum of the values of left and right child nodes. Our MIHT satisfies an important property called *Max-heap*: the value of each node in MIHT is less than that of its parent node, with the maximum-value element at the root node. Furthermore, given a query x, MIHT scheme needs to return the index i of the leaf node such that $v_{n_0} + \cdots + v_{n_{i-1}} < x \leq v_{n_0} + \cdots + v_{n_i}$. In other words, the value x is exactly located in the interval of the leaf node n_i. The syntax and security definition of MIHT are all similar to that of MHT. Due to the limitation of space, we omit them and directly give our construction as follows:

$\{\mathsf{MPK}, \mathsf{MSK}\} \leftarrow \mathsf{MIHT.KeyGen}(1^\kappa)$: Given a security parameter κ, the data owner first chooses a cryptographic hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_p$. After that, it conducts algorithm $\mathsf{S.KeyGen}(1^\kappa)$ to generate $\{\mathsf{SSK}, \mathsf{SPK}\}$. Finally, it sets the public key $\mathsf{MPK} = \{\mathcal{H}, \mathsf{SPK}\}$ and the private key $\mathsf{MSK} = \mathsf{SSK}$.

$\{\mathsf{auth}(D), \sigma\} \leftarrow \mathsf{MIHT.Setup}(D, \mathsf{MSK})$: Given the set $D = \{(i, v_{n_i}) | i \in \{0, \ldots, q-1\}\}$ and MSK, the data owner first parses MSK as SSK. The construction of MIHT is similar to that of MHT [5,44]. Each node in MIHT is comprised of three items: index, value and hash value. For any leaf node n_i, it sets the hash value Υ_{n_i} of node n_i by computing:

$$\Upsilon_{n_i} = \mathcal{H}(i \parallel v_{n_i}). \tag{15}$$

For any internal node n_p, it computes the value $v_{n_p} = v_{n_l} + v_{n_r}$ and sets the hash value Υ_{n_p} of node n_p by computing

$$\Upsilon_{n_p} = \mathcal{H}\left(p \| \left(v_{n_l} + v_{n_r}\right) \| \mathcal{H}\left(\Upsilon_{n_l} \parallel \Upsilon_{n_r}\right)\right), \tag{16}$$

where the nodes n_l and n_r are the left and right child nodes of the parent node n_p, respectively. For convenience, we denote the hash value of root node by Υ_{n_R}. Next, it generates the signature $\sigma = \mathsf{S.Sign}(\mathsf{SSK}, \Upsilon_{n_R})$ and sets the Merkle interval hash tree as authenticated data structure $\mathsf{auth}(D)$. Finally, it outputs $\mathsf{auth}(D)$ and σ.

$\{I, \Omega_x\} \leftarrow \mathsf{MIHT.Prove}(x, \mathsf{auth}(D), \sigma)$: Given a query x, the cloud defines an array *Auth* and conducts algorithm $I \leftarrow$ *SearchIndex*$(2^\ell - 2, x, 1, \ell + 1, Auth)$ (see Algorithm 1). Next, it sets the proof $\Omega_x = \{v_{n_l}, Auth\}$. Finally, it outputs the index I and the proof Ω_x that the queried value is exactly located in the interval of the leaf node n_l.

$\{0, 1\} \leftarrow \mathsf{MIHT.Verify}(I, \Omega_x, \mathsf{MPK})$: Given the queried result I, the proof Ω_x and the public key MPK, the client re-generates the hash value of root node Υ_{n_R} with the pair $\{I, \Omega_x\}$ in a similar way to MHT. Finally, it conducts $\mathsf{S.Verify}(\sigma, \mathsf{SPK}, \Upsilon_{n_R})$ to check whether the signature σ is valid. If it is valid, it outputs 1; Otherwise, it outputs 0.

$\{\mathsf{auth}(D'), \sigma'\} \leftarrow \mathsf{MIHT.Update}(\mathsf{MSK}, i, v'_{n_i})$: Given a new value v'_{n_i} for leaf node n_i, the data owner updates the new values and hash values from the leaf node n_i to the root node in MIHT. Next, it conducts $\mathsf{S.Sign}(\mathsf{MSK}, \Upsilon'_{n_R})$ to generate the latest signature σ'. Finally, it outputs the latest authenticated data structure $\mathsf{auth}(D')$ and the signature σ'. This whole procedure in MIHT is similar to that in MHT.

5.1.1. Correctness

If the cloud follows the prescribed algorithm $\mathsf{MIHT.Prove}$ and outputs the index I and the proof $\Omega_x = \{v_{n_l}, Auth\}$, the root node re-generated by the client is the same as the original one.

According to Eq. (16), we can get the hash value of the root node as follows:

$$\Upsilon_{n_R} = \mathcal{H}\left(2^L - 2 \| \left(v_{n_l} + v_{n_r}\right) \| \mathcal{H}\left(\Upsilon_{n_l} \parallel \Upsilon_{n_r}\right)\right), \tag{17}$$

where the nodes n_l and n_r are the left and right child nodes of the root node n_R, respectively.

Let us recursively regard the child node as the root up to the leaf node. Then, the term $\Upsilon_{n_l} \parallel \Upsilon_{n_r}$ in Eq. (17) can be expressed as follows:

$$\begin{aligned}
\Upsilon_{n_l} \parallel \Upsilon_{n_r} &= \Upsilon_{n_{2^{L-1}-(2^1+1)}} \parallel \Upsilon_{n_{2^{L-1}-2^1}} \\
&= \mathcal{H}\left(\Upsilon_{n_{2^{L-1}-(2^2+3)}} \parallel \Upsilon_{n_{2^{L-1}-2^2+2}}\right) \parallel \mathcal{H}\left(\Upsilon_{n_{2^{L-1}-(2^2+1)}} \parallel \Upsilon_{n_{2^{L-1}-2^2}}\right) \\
&\cdots \\
&= \mathcal{H}\left(\cdots \mathcal{H}\left(\begin{array}{l} \mathcal{H}\left(\begin{array}{l} \mathcal{H}\left(\Upsilon_{n_{2^{L-1}-(2^i-2^{i-1})}} \parallel \Upsilon_{n_{2^{L-1}-(2^i-2^{i-2}+1)}}\right) \parallel \\ \mathcal{H}\left(\Upsilon_{n_{2^{L-1}-(2^i-2^{i-2}+2)}} \parallel \Upsilon_{n_{2^{L-1}-(2^i-2^{i-2}+3)}}\right) \end{array}\right) \\ \parallel \cdots \parallel \\ \mathcal{H}\left(\begin{array}{l} \mathcal{H}\left(\Upsilon_{n_{2^{L-1}-(2^i-3)}} \parallel \Upsilon_{n_{2^{L-1}-(2^i-2)}}\right) \parallel \\ \mathcal{H}\left(\Upsilon_{n_{2^{L-1}-(2^i-1)}} \parallel \Upsilon_{n_{2^{L-1}-2^i}}\right) \end{array}\right) \end{array}\right) \cdots\right) \\
&\cdots \\
&= \mathcal{H}\left(\cdots \mathcal{H}\left(\begin{array}{l} \mathcal{H}\left(\Upsilon_{n_0} \parallel \Upsilon_{n_1}\right) \parallel \\ \mathcal{H}\left(\Upsilon_{n_2} \parallel \Upsilon_{n_3}\right) \end{array}\right) \right. \\
&\qquad \left. \parallel \cdots \parallel \right. \\
&\qquad \left. \mathcal{H}\left(\begin{array}{l} \mathcal{H}\left(\Upsilon_{n_{2^{L-1}-4}} \parallel \Upsilon_{n_{2^{L-1}-3}}\right) \parallel \\ \mathcal{H}\left(\Upsilon_{n_{2^{L-1}-2}} \parallel \Upsilon_{n_{2^{L-1}-1}}\right) \end{array}\right) \right) \cdots\right)
\end{aligned} \tag{18}$$

Algorithm 1 Search Index

Input: The real index of current node, k; The queried value, v; The level of the current node, l; The height of tree, L; The nodes in the authenticated path, $Auth$;
Output: The index of leaf node that the queried value is exactly located in its interval, I.

```
function SearchIndex(k, v, l, L, Auth)
    mapIndex ← 2^L − k − 1
    if l = L then
        return k
    end if
    right = mapIndex ∗ 2
    left = mapIndex ∗ 2 + 1
    i ← 2^L − left − 1
    j ← 2^L − right − 1
    if v_{n_i} ≤ v then
        // If current node does not have sibling node (i.e., root node), record it.
        // Otherwise, record its sibling node.
        if mapIndex = 1 then
            Auth[L − l] ← (k, v_{n_k}, ϒ_{n_k})
        else
            Auth[L − l] ← (k − 1, v_{n_{k−1}}, ϒ_{n_{k−1}})
        end if
        return SearchIndex(j, v − v_{n_i}, l + 1, L, Auth)
    else
        if mapIndex = 1 then
            Auth[L − l] ← (k, v_k, ϒ_{n_k})
        else
            Auth[L − l] ← (k + 1, v_{k+1}.value, ϒ_{n_{k+1}})
        end if
        return SearchIndex(i, v, l + 1, L, Auth)
    end if
end function
```

The term v_{n_l} in Eq. (17) can be expressed as follows:

$$v_{n_l} = v_{n_0} + \cdots + v_{n_{2^{L-2}-1}}. \tag{19}$$

Similarly, we can get:

$$v_{n_r} = v_{n_{2^{L-2}}} + \cdots + v_{n_{2^{L-1}-1}}. \tag{20}$$

According to Algorithm 1, the array $Auth$ records all sibling nodes in the path from that leaf node to the root node. $Auth[i]$ stores the sibling node at the level $i+2$. For simplicity, we assume that the index of sibling node at the level $i+2$ is j_i. It is not hard to find that:

$$Auth[i][1] = v_{2^L - 2^{L-i-2}*(2^L-1-j_i-1)} + \cdots + v_{2^L-1-2^{L-i-2}*(2^L-1-j_i)}, \tag{21}$$

$$\begin{aligned} Auth[i][2] &= \mathcal{H}\left(\cdots \mathcal{H}\left(\begin{array}{l} \mathcal{H}\left(\begin{array}{l} \mathcal{H}\left(\Upsilon_{n_{2^L-2^{L-i-2}*(2^L-1-j_i-1)}} \parallel \Upsilon_{n_{2^L-2^{L-i-2}*(2^L-1-j_i-1)+1}}\right) \parallel \\ \mathcal{H}\left(\Upsilon_{n_{2^L-2^{L-i-2}*(2^L-1-j_i-1)+2}} \parallel \Upsilon_{n_{n_{2^L-2^{L-i-2}*(2^L-1-j_i-1)+3}}}\right) \end{array}\right) \\ \parallel \cdots \parallel \\ \mathcal{H}\left(\begin{array}{l} \mathcal{H}\left(\Upsilon_{n_{2^L-1-2^{L-i-2}*(2^L-1-j_i)-3}} \parallel \Upsilon_{n_{2^L-1-2^{L-i-2}*(2^L-1-j_i)-2}}\right) \parallel \\ \mathcal{H}\left(\Upsilon_{n_{2^L-1-2^{L-i-2}*(2^L-1-j_i)-1}} \parallel \Upsilon_{n_{2^L-1-2^{L-i-2}*(2^L-1-j_i)}}\right) \end{array}\right) \end{array}\right)\cdots\right) \end{aligned} \tag{22}$$

Based on Eqs. (19), (20), it is not hard to see that:

$$v_{n_R} = v_{n_0} + v_{n_1} + \cdots + v_{n_{2^{L-1}-1}}. \tag{23}$$

According to Eq. (21), the above equation can be express as follows:

$$\begin{aligned} v_{n_R} &= \left(Auth[0][1] + \cdots + \left(\left(Auth[L-2][1] + v_{n_l}\right)\right)\right) \\ &= v_{n_l} + \sum_{i}^{L-2} Auth[i][1] \end{aligned} \tag{24}$$

Based on Eqs. (18) and (22), we can get:

$$\Upsilon_{n_l} \parallel \Upsilon_{n_r} = \mathcal{H}\left(Auth[0][2] \| \cdots \| \left(\mathcal{H}\left(Auth[L-2][2] \parallel \Upsilon_{n_l}\right)\right)\right). \tag{25}$$

Due to Eqs. (24) and (25), it is not hard to see that:

$$\Upsilon_{n_R} = \mathcal{H}\left(\begin{array}{l} 2^L - 2 \parallel \\ \left(\left(Auth[0][1] + \cdots + \left(\left(Auth[L-2][1] + v_{n_l}\right)\right)\right)\right) \parallel \\ \mathcal{H}\left(Auth[0][2] \| \cdots \| \left(\mathcal{H}\left(Auth[L-2][2] \parallel \Upsilon_{n_l}\right)\right)\right) \end{array}\right). \tag{26}$$

Obviously, if all procedures are performed as described above, the proof Ω_x can always pass the client's verification. This completes the proof.

5.1.2. Security

Theorem 3. *If $\mathcal{H}$ is a collision resistant hash function, then our proposed Merkle interval hash tree is secure.*

Proof. The adversary $\mathcal{A}$ outputs a forgery for the value x. The forgery consists of a claimed index I^* and the corresponding proof $\Omega_x^* = \{v_{n_{I^*}}, Auth^*[0], \ldots, Auth^*[L-2]\}$. If $\mathcal{A}$ break the security of Merkle interval hash tree, the adversary $\mathcal{A}$ can leverage this forgery to construct a method $\mathcal{B}$ to break the collision-resistant property of hash function.

Supposed $\mathcal{A}$ produces the forgery passing the verification, the following equation holds:

$$\Upsilon_{n_R} = \mathcal{H}\left(\begin{array}{l} 2^L - 2 \parallel \\ \left(\left(Auth^*[0][1] + \cdots + \left(\left(Auth^*[L-2][1] + v_{n_{I^*}}\right)\right)\right)\right) \parallel \\ \mathcal{H}\left(Auth^*[0][2] \| \cdots \| \left(\mathcal{H}\left(Auth^*[L-2][2] \parallel \Upsilon_{n_{I^*}}\right)\right)\right) \end{array}\right) \tag{27}$$

Similarly, for the correct index I and the proof $\Omega_x = \{v_{n_I}, Auth[0], \ldots, Auth[L-2]\}$, we have:

$$\Upsilon_{n_R} = \mathcal{H}\left(\begin{array}{l} 2^L - 2 \parallel \\ \left(\left(Auth[0][1] + \cdots + \left(\left(Auth[L-2][1] + v_{n_l}\right)\right)\right)\right) \parallel \\ \mathcal{H}\left(Auth[0][2] \| \cdots \| \left(\mathcal{H}\left(Auth[L-2][2] \parallel \Upsilon_{n_l}\right)\right)\right) \end{array}\right), \tag{28}$$

Due to Eqs. (27) and (28), it is not hard to see that:

$$\begin{aligned} &\mathcal{H}\left(\begin{array}{l} 2^L - 2 \parallel \\ \left(\left(Auth[0][1] + \cdots + \left(\left(Auth[L-2][1] + v_{n_l}\right)\right)\right)\right) \parallel \\ \mathcal{H}\left(Auth[0][2] \| \cdots \| \left(\mathcal{H}\left(Auth[L-2][2] \parallel \Upsilon_{n_l}\right)\right)\right) \end{array}\right) = \\ &\mathcal{H}\left(\begin{array}{l} 2^L - 2 \parallel \\ \left(\left(Auth^*[0][1] + \cdots + \left(\left(Auth^*[L-2][1] + v_{n_l^*}\right)\right)\right)\right) \parallel \\ \mathcal{H}\left(Auth^*[0][2] \| \cdots \| \left(\mathcal{H}\left(Auth^*[L-2][2] \parallel \Upsilon_{n_l^*}\right)\right)\right) \end{array}\right) \end{aligned} \tag{29}$$

If adversary $\mathcal{A}$ succeeds to forge I^* and Ω_x^*, $\mathcal{A}$ can leverage this forgery to construct an efficient algorithm $\mathcal{B}$ to break the property

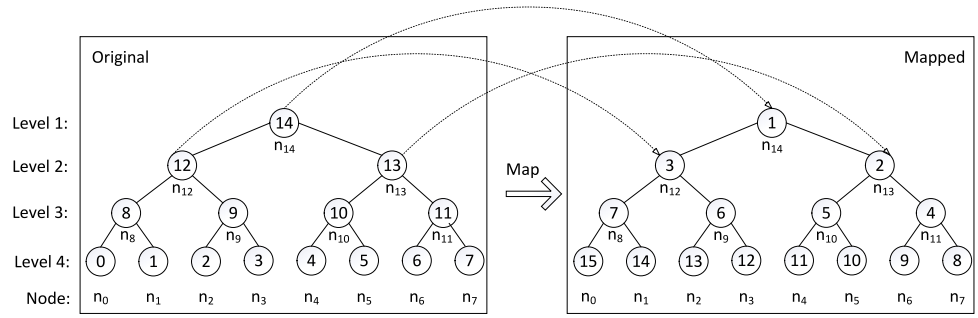

Fig. 1. The relationship of the original index and mapped index in MIHT.

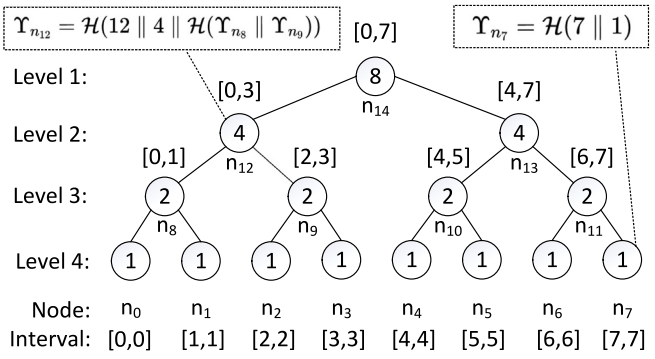

Fig. 2. The MIHT during setup phase.

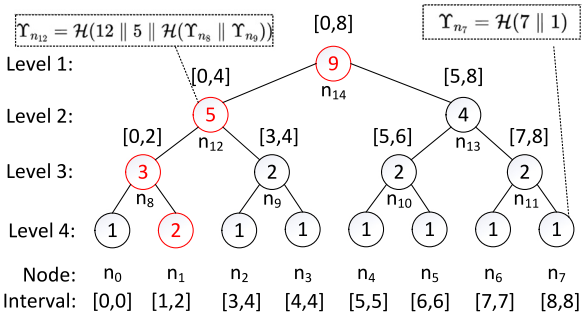

Fig. 3. The MIHT during update phase.

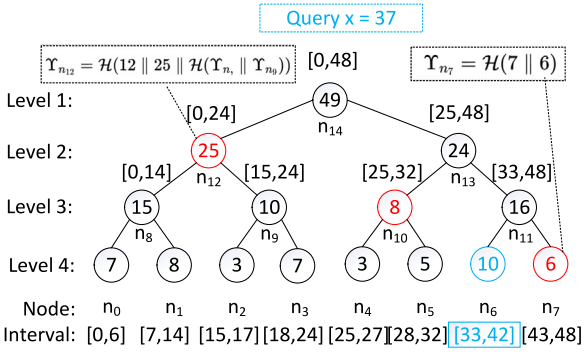

Fig. 4. The MIHT during query phase.

of collision resistance (i.e., $\mathcal{H}(a) = \mathcal{H}(b) \wedge a \neq b$). This completes the proof.

Remark 5. Before giving an illustrative instantiation of MIHT, we first introduce the relationship of the original index and the mapped index in algorithm 1(see line 2). We take as an example MIHT with eight leaf nodes. Each node n_i in MIHT is comprised of a triple $(i, v_{n_i}, \Upsilon_{n_i})$. For convenience, we omit the value and hash value of all nodes in MIHT. Fig. 1 shows the relationship of the original index and mapped index, where the value in the node is the index. It is not hard to find an important property: for any parent node n_p in the mapped MIHT, the index r/l of its right (left) node n_r/n_l is equal to $2p/2p+1$ (see lines 6, 7 in Algorithm 1).

5.1.3. An illustrative instantiation

To further illustrate the construction of MIHT, we also take an example MIHT with eight leaf nodes. Fig. 2 shows the phase of setup, where the value of each leaf node $v_{n_i} = 1$ for $i \in \{0, \ldots, 7\}$. For simplicity, we omit the index and the hash value of each node in the following. For each leaf node n_i, the hash value can be computed as $\Upsilon_{n_i} = \mathcal{H}(i \parallel v_{n_i}) = \mathcal{H}(i \parallel 1)$. For any internal node n_p, the value v_p is the sum of the values in its left and right child nodes. The hash value Υ_{n_p} is the hash of its index, value, and two hash value of its two child nodes. As shown in Fig. 2, $\Upsilon_{n_{12}} = \mathcal{H}(12\|(2+2)\|\mathcal{H}(\Upsilon_{n_8} \parallel \Upsilon_{n_9})) = \mathcal{H}(12\|4\|\mathcal{H}(\Upsilon_{n_8} \parallel \Upsilon_{n_9}))$.

Once receiving the update information (i, v'_{n_i}), the MIHT will be updated by conducting the algorithm MIHT.Update(MSK, i, v'_{n_i}). As shown in Fig. 3, the value of node n_1 is changed from 1 to 2. The hash value of node n_1 will be updated as $\Upsilon_{n_1} = \mathcal{H}(1 \parallel 2)$. Then, the value and hash value of each node (marked in red) on the path from it to the root node are updated. After a sequence of update operations, the latest MIHT is shown in Fig. 4. As can be seen in Fig. 4, the queried value $x = 37$ is exactly located in the interval [33, 42] of the leaf node n_6. Now we will show how Algorithm 1 outputs the index of node n_6 and authenticated path *Auth* (marked in red). In Remark 5, we have introduced the relationship of the original index and the mapped index. For convenience, we still use the original index of each node in Fig. 4. It searches accordingly from top to bottom and from left to right. We set the compared value $cv = x$. It first goes to the left child node n_{12} of root node n_{14} and compares its value $v_{n_{12}}$ with cv. Since $v_{12} < cv$ (i.e., $25 < 37$), the interval of left sub-tree (i.e., $[0, 24]$) does not include x. Therefore, it goes to its sibling node $n_{12+1} = n_{13}$ and sets the compared value $cv = cv - v_{n_{12}} = 37 - 25 = 12$. Since $v_{n_{10}} = 8 < cv$, it goes to node n_{11} and sets $cv = cv - v_{n_{10}} = 12 - 8 = 4$. Since $v_{n_6} = 10 > cv$ and n_6 is the leaf node, it returns the index of node n_6. Trivially, the corresponding proof is $\Omega_x = \{v_{n_6}, n_7, n_{10}, n_{12}\}$. Upon receiving the result and the proof Ω_x, the client uses them to re-construct the hash value of root node and then checks the validity of signature.

Remark 6. To support insertion and deletion in VDB, we incorporate our novel VC scheme with MIHT. In MIHT, the index i of leaf node n_i is corresponding to that of data block in VDB. The value

Table 1
The Relationship of MIHT and VDB.

Index	Leaf	Value	Data Block in VDB
0	n_0	7	$\triangle_0(m_0) = v_0 \| \| v_1 \| \| v_2 \| \| v_3 \| \| v_4 \| \| v_5 \| \| v_6$
1	n_1	8	$\triangle_1(m_1) = v_7 \| \| v_8 \| \| v_9 \| \| v_{10} \| \| v_{11} \| \| v_{12} \| \| v_{13} \| \| v_{14}$
2	n_2	3	$\triangle_2(m_2) = v_{15} \| \| v_{16} \| \| v_{17}$
3	n_3	7	$\triangle_3(m_3) = v_{18} \| \| v_{19} \| \| v_{20} \| \| v_{21} \| \| v_{22} \| \| v_{23} \| \| v_{24}$
4	n_4	3	$\triangle_4(m_4) = v_{25} \| \| v_{26} \| \| v_{27}$
5	n_5	5	$\triangle_5(m_5) = v_{28} \| \| v_{29} \| \| v_{30} \| \| v_{31} \| \| v_{32}$
6	n_6	10	$\triangle_6(m_6) = v_{33} \| \| v_{34} \| \| v_{35} \| \| v_{36} \| \| v_{37} \| \| v_{38} \| \| v_{39} \| \| v_{40} \| \| v_{41} \| \| v_{42}$
7	n_7	6	$\triangle_7(m_7) = v_{43} \| \| v_{44} \| \| v_{45} \| \| v_{46} \| \| v_{47} \| \| v_{48}$

v_{n_i} of leaf node n_i is the number of data record stored in ith data block. When the data owner wants to insert a new record before v_1, the value v_{n_1} of leaf node n_1 will be increased by 1 as shown in Fig. 3. Similarly, the value of leaf node will be reduced by 1 when a deletion operation occurs. When a modification operation occurs, the MIHT remains unchanged since the number of data records (i.e., the value of leaf node) does not be changed.

5.2. A detailed construction

The basic idea behind our construction is the following. To support all update operations (i.e., modification, insertion, deletion) in VDB simultaneously, we incorporate our VC scheme with MIHT scheme. We mainly use our VC scheme to commit to q data blocks of VDB for the integrity of the data, where q is fixed at the setup phase. MIHT are mainly used for recording the number of data in each block. If an insertion or deletion operation occurs in the outsourced database, the value of corresponding nodes will be increased/decreased by 1 as discussed in Remark 6. For any update operation at position i of VDB, data owner needs to first retrieve its index k in MIHT and then open the kth committed block in VC. Now we take for example the MIHT in Fig. 4. Table 1 shows the relationship of MIHT and VDB. Note that the term m_i in Eq. (3) is set to the ith data block $\triangle_i$ of VDB. When the data owner wants to modify v_{37} to v'_{37}, it queries $x = 37$ to MIHT, gets the index $k = 6$, and verifies its validity. If it is valid, it requires the cloud to open the 6-th committed block $m_6 = \triangle_6 = v_{33} \parallel v_{34} \parallel v_{35} \parallel v_{36} \parallel \boldsymbol{v_{37}} \parallel v_{38} \parallel v_{39} \parallel v_{40} \parallel v_{41} \parallel v_{42}$, and check the integrity of result. If it is valid, it retrieves v_{37} and updates the commitment with respect to the new data block $\triangle'_6 = v_{33} \parallel v_{34} \parallel v_{35} \parallel v_{36} \parallel \boldsymbol{v'_{37}} \parallel v_{38} \parallel v_{39} \parallel v_{40} \parallel v_{41} \parallel v_{42}$. To retrieve v_{37}, it first needs to recover the interval of leaf node n_6 by adding the values of nodes n_{12} and n_{10}. Note that nodes n_{12} and n_{10} are contained in the proof Ω_{32} as discussed above. Next, it retrieves v_{37} by finding the $(x - v_{n_{12}} - v_{n_{10}} = 5)$-th record in the block $\triangle_6$. For inserting an element v^* before v_{37}, the committed block $\triangle_6 = v_{33} \parallel v_{34} \parallel v_{35} \parallel v_{36} \parallel v_{37} \parallel v_{38} \parallel v_{39} \parallel v_{40} \parallel v_{41} \parallel v_{42}$ will be updated to $v_{33} \parallel v_{34} \parallel v_{35} \parallel v_{36} \parallel \boldsymbol{v^*} \parallel v_{37} \parallel v_{38} \parallel v_{39} \parallel v_{40} \parallel v_{41} \parallel v_{42}$ in VC. Meanwhile, the value v_{n_6} of leaf node n_6 in MIHT will be increased by 1. Trivially, the update process of the deletion operation is similar to that of insertion operation. The construction of our scheme is detailed as follows:

$(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{Setup}(1^\kappa, DB)$: Given the security parameter κ and a NoSQL database $DB = \{(i, v_i) | i \in [q], v_i \in \mathbb{Z}_p\}$, it first runs $\{\mathsf{pp}, \mathsf{VSK}\} \leftarrow \mathsf{VC.KeyGen}(1^k, q)$. After that, it conducts $(C, \mathsf{aux}) \leftarrow \mathsf{VC.Com}_{\mathsf{pp}}(v_0, \ldots, v_{q-1})$. Next, it performs $\{\mathsf{MPK}, \mathsf{MSK}\} \leftarrow \mathsf{MIHT.KeyGen}(1^\kappa)$ and $\{\mathsf{auth}(D), \sigma\} \leftarrow \mathsf{MIHT.Setup}(D, \mathsf{MSK})$, where $D = \{1\}^q$. Afterwards, it sends DB and $\mathsf{auth}(D)$ to the cloud. Finally, it sets:

$$\mathsf{PK} = \{\mathsf{pp}, C, aux, \mathsf{MPK}, \sigma\}, \mathsf{SK} = \{\mathsf{MSK}, \mathsf{VSK}\}.$$

$(v_x, \pi_{v_x}) \leftarrow \mathsf{Query}(x, \mathsf{PK})$: Upon receiving a query x, it first parses PK as $\{\mathsf{pp}, C, aux, \mathsf{MPK}, \sigma\}$. After that, it runs $\{k, \Omega_x\} \leftarrow \mathsf{MIHT.Prove}(x, \mathsf{auth}(D), \sigma)$. Next, it conducts $\{\triangle_k, \Lambda_k\} \leftarrow \mathsf{VC.Open}_{\mathsf{pp}}(k, \mathsf{aux}, C)$. Afterwards, it recovers v_x by determining its interval according to Ω_x (especially the values of nodes in Ω_x). It sets:

$$\pi_{v_x} = \{k, \Omega_x, \triangle_k, \Lambda_k\},$$

and returns the result v_x and the witness π_{v_x}.

$\{0 \text{ or } 1\} \leftarrow \mathsf{Verify}(\mathsf{PK}, x, v_x, \pi_{v_x})$: Upon receiving the result v_x and the witness π_{v_x}, it first parses PK and π_{v_x} as $\{\mathsf{pp}, C, aux, \mathsf{MPK}, \sigma\}$ and $\{k, \Omega_x, \triangle_k, \Lambda_k\}$, respectively. After that, it runs algorithm $\mathsf{MIHT.Verify}(k, \Omega_x, \mathsf{MPK})$. If the algorithm MIHT.Verify outputs 0, it returns 0 and aborts. Otherwise, it performs $\mathsf{VC.Ver}_{\mathsf{pp}}(C, \triangle_k, k, \Lambda_k)$. If the algorithm $\mathsf{VC.Ver}_{\mathsf{pp}}$ outputs 0, it returns 0 and aborts. Otherwise, it outputs 1.

$\mathsf{PK}' \leftarrow \mathsf{Update}(\mathsf{upd}, x, v'_x, \mathsf{SK}, \mathsf{PK})$: Upon receiving an update upd, the index x, the value v'_x, it first parses SK and PK as $\{\mathsf{MSK}, \mathsf{VSK}\}$ and $\{\mathsf{pp}, C, aux, \mathsf{MPK}, \sigma\}$, respectively. Next, it works as follows:

1. $\mathsf{upd} = \mathsf{modify}$: It sends a query x to the cloud. Upon receiving the result v_x and witness π_{v_x}, it checks whether the result is valid by conducting algorithm $\mathsf{Verify}(\mathsf{PK}, x, v_x, \pi_{v_x})$. If the algorithm Verify outputs 0, it aborts. Otherwise, it performs $(C', U) \leftarrow \mathsf{VC.Update}_{\mathsf{pp}}(C, \triangle_k, \triangle'_k, k, \mathsf{VSK})$, where v_x in the original block $\triangle_k$ is replaced with v'_x, all remaining items in the original block $\triangle_k$ and the update block $\triangle'_k$ are totally identical. Next, it sends the update information U to the cloud. Finally, it sets:
$$\mathsf{PK}' = \{\mathsf{pp}, C', aux, \mathsf{MPK}, \sigma\}.$$
2. $\mathsf{upd} = \mathsf{insert}$: It sends a query x to the cloud. Upon receiving the result v_x and witness π_{v_x}, it checks whether the result is valid by conducting algorithm $\mathsf{Verify}(\mathsf{PK}, x, v_x, \pi_{v_x})$. If the algorithm Verify outputs 0, it aborts. Otherwise, it performs $(C', U) \leftarrow \mathsf{VC.Update}_{\mathsf{pp}}(C, \triangle_k, \triangle'_k, k, \mathsf{VSK})$, where v'_x is inserted before v_x in the original block $\triangle_k$ (i.e., the update block $\triangle'_k$). Next, it runs $\{\mathsf{auth}(D'), \sigma'\} \leftarrow$

Table 2
Comparison of desired properties.

Properties		BVDB [6]	CVDB [9]	MVDB [8]	FVDB [7]	SVDB [10]	WVDB
Model		Amortized	Amortized	Amortized	Amortized	Amortized	Amortized
Order of group		Composite	Prime	Composite	Prime	Prime	Prime
Verification		Private	Public	Private	Public	Public	Public
Constant-size assumption		✓	✓	✓	✓	×	✓
FAU		×	×	×	✓	×	×
TTP		×	×	×	×	✓	×
Update	Insertion	×	×	✓	×	✓	✓
	Modification	✓	✓	✓	✓	✓	✓
	Deletion	×	×	✓	×	✓	✓

Table 3
Comparison of computational cost.

Algorithms		BVDB [6]	CVDB [9]	MVDB [8]	FVDB [7]	WVDB
Setup		$2q \cdot H + 2q \cdot E + q \cdot M + 1 \cdot P$	$\frac{q^2+3q+4}{2} \cdot E + (q-1) \cdot M + 1 \cdot H$	$(4q+1) \cdot E + (q+1) \cdot M + 1 \cdot P + (3\ell - 1) \cdot H$	$\frac{q^2+3q}{2} \cdot E + (q-1) \cdot M$	$(2q-1) \cdot H + 3q \cdot E + q \cdot M + 1 \cdot P$
Query		q·M+2·P	$(q-1) \cdot E + (q-2) \cdot M$	q·M+2·P	$(q-1) \cdot E + (q-2) \cdot M$	$q \cdot M + 1 \cdot P$
Verify		$2 \cdot H + 4 \cdot M + 3 \cdot E + 1 \cdot P$	$2 \cdot M + 1 \cdot E + 1 \cdot H + 4 \cdot P$	$(\ell+3) \cdot H + 4 \cdot M + 3 \cdot E + 1 \cdot P$	$1 \cdot E + 2 \cdot P + 1 \cdot M$	$(\ell+1) \cdot H + 1 \cdot E + 1 \cdot M + 1 \cdot P$
Update	Insertion	–	–	$(\ell+5) \cdot H + 6 \cdot M + 6 \cdot E + 2 \cdot P$	–	$(\ell+1) \cdot H + 1 \cdot E + q \cdot M$
	Modification	$4 \cdot H + 6 \cdot M + 6 \cdot E + 2 \cdot P$	$4 \cdot M + 3 \cdot E + 2 \cdot H + 4 \cdot P$	$(\ell+5) \cdot H + 6 \cdot M + 6 \cdot E + 2 \cdot P$	$1 \cdot M + 1 \cdot E$	$1 \cdot E + q \cdot M$ Worst case
	Deletion	–	–	$(\ell+5) \cdot H + 6 \cdot M + 6 \cdot E + 2 \cdot P$	–	$(\ell+1) \cdot H + 1 \cdot E + q \cdot M$

$\mathsf{MIHT.Update}(\mathsf{MSK}, k, v_{n_k} + 1)$. It sends the update information U to the cloud and sets:

$$\mathsf{PK}' = \left\{\mathsf{pp}, C', aux, \mathsf{MPK}, \sigma'\right\}.$$

3. upd = delete: The update process of the deletion operation is similar to that of the insertion operation except algorithm MIHT.Update. It takes as input $v_{n_k} - 1$ rather than $v_{n_k} + 1$ due to the deletion operation. Meanwhile, the difference between the original block $\triangle_k$ and the update block $\triangle'_k$ is that $\triangle'_k$ does not contain v_x.

Remark 7. When the element v_x stored in the block $\triangle_k$ is modified to v'_x, the number of data stored in the block (i.e., the value of the leaf node n_k) remains same. Hence, the MIHT does not need to be updated. When a new element is inserted/removed, the value of corresponding leaf node will be increased/decreased by 1. Therefore, the MIHT should be updated.

5.3. Security analysis

Theorem 4. *The proposed VDB scheme is correct.*

Proof. If the cloud follows the prescribed algorithm $\mathsf{Query}(x, \mathsf{PK})$ and outputs the result v_x and the witness $\pi_{v_x} = \{k, \Omega_x, \triangle_k, \Lambda_k\}$, the result v_x and the witness π_{v_x} will always pass the verification algorithm $\mathsf{Verify}(\mathsf{PK}, x, v_x, \pi_{v_x})$. If they pass the algorithm Verify, the algorithm Verify will output 1. If the algorithm Verify returns 1, it is required that both of algorithms $\mathsf{MIHT.Verify}(k, \Omega_x, \mathsf{MPK})$ and $\mathsf{VC.Ver}_{\mathsf{pp}}(C, \triangle_k, k, \Lambda_k)$ output 1. The correctness of MIHT and VC has been proved in Sections 5.1.1 and 4.3.1 , respectively. Therefore, our proposed scheme is correct.

Theorem 5. *The proposed VDB scheme is secure.*

Proof. If the cloud forges a wrong result v'_x and the witness $\pi'_{v_x} = \left\{k', \Omega'_x, \triangle'_k, \Lambda'_k\right\}$ successfully, the verification algorithm $\mathsf{Verify}(\mathsf{PK}, x, v'_x, \pi'_{v_x})$ will output 1. If the algorithm Verify returns 1, it is required that both of algorithms $\mathsf{MIHT.Verify}(k', \Omega'_x, \mathsf{MPK})$ and $\mathsf{VC.Ver}_{\mathsf{pp}}(C, \triangle'_k, k', \Lambda'_k)$ output 1. If the forged pair (k', Ω'_x) passes the verification algorithm MIHT.Verify (i.e., this algorithm outputs 1), it will break the security of MIHT proved in Theorem 3. Therefore, k' must be equal to k. As a result, the commitment must be opened at the position k. If the forged pair $(\triangle'_k, \Lambda'_k)$ passes the verification algorithm $\mathsf{VC.Ver}_{\mathsf{pp}}(C, \triangle'_k, k, \Lambda'_k)$, it will break the security of our proposed VC scheme proved in Theorem 1. Hence, $\triangle'_k$ must be equal to $\triangle_k$. Therefore, our proposed VDB scheme is secure.

Theorem 6. *The proposed VDB scheme is publicly verifiable.*

Proof. In verification phase, the inputs of algorithm Verify are the public key PK, the query x, the result v_x and the witness π_{v_x}. These inputs are all public to the client. Therefore, our proposed VDB scheme is publicly verifiable.

Theorem 7. *The proposed VDB scheme is secure against the forward automatic update attack [9].*

Proof. As pointed out by Chen et al. [9], Catalano–Fiore's VDB framework from vector commitment [7] is vulnerable to forward automatic update attack since algorithm $\mathsf{VC.Update}_{\mathsf{pp}}$ does not take as input any secret knowledge. Without the secret knowledge, any adversary can update database arbitrarily and maliciously generate a valid public key PK*. As a result, the forward update public key PK* and the real one PK are computationally indistinguishable for any third party. As can be seen in our algorithm $\mathsf{VC.Update}_{\mathsf{pp}}$, the private key VSK is contained in it. Therefore, our proposed VDB scheme is secure under the forward automatic update attack.

6. Evaluation

In this section, we first theoretically compare our proposed scheme (we call it as WVDB) with Benabbas et al.'s scheme (we call it as BVDB), Chen et al.'s scheme (we call it as CVDB), Miao et al.'s scheme (we call it as MVDB), Catalano and Fiore's scheme (we call it as FVDB), and Shen et al.'s scheme (we call it as SVDB) in two folds: the desired properties and the computation complexity. Then, we provide a prototypal implement to compare their computation cost.

6.1. Properties

Table 2 summarizes the comparison of desired properties. These schemes are all in the amortized model [46], which requires a one-time expensive computational overhead in the setup phase. All of these schemes are based on the bilinear map, while the order of the group in BVDB [6] and MVDB [8] schemes are composite. The operation in composite order group costs much more computational overload than that in prime order group. Furthermore, BVDB and MVDB schemes do not support public verification. The security of SVDB [10] relies on q-strong bilinear Diffie–Hellman assumption [47], while the others are based on the constant-size assumption. The FVDB scheme [7] does not involve any secret knowledge of data owner when the update occurs, so it cannot resist FAU attack. Only WVDB, MVDB, and SVDB schemes can support all update operations, simultaneously. However, SVDB requires a trusted third party(TTP) to interact with the cloud and data owner in the update phase. Their scheme is efficient, but the TTP knows the private key of the data owner. If the TTP has colluded with a malicious adversary, the scheme will be no longer secure. It is really impractical in the real world, so we omit this scheme in the following.

6.2. Computation cost

Since the computation cost is mainly dominated by bilinear pairing, exponentiation, multiplication over group and hashing, we evaluate it by counting the number of such operations. Note that the bilinear pairing and exponentiation operations cost more effort than others. Table 3 summarizes the comparison of the

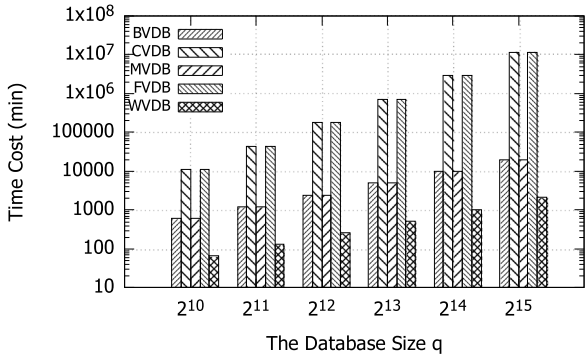

Fig. 5. The comparison of computation cost in setup phase.

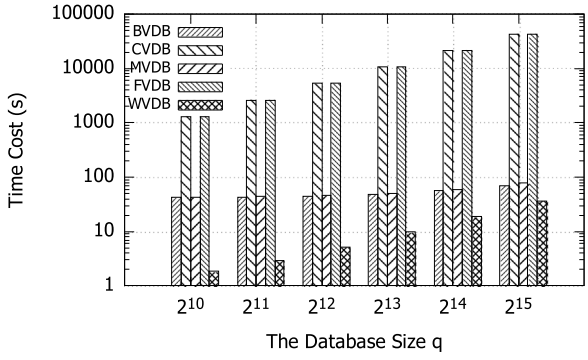

Fig. 6. The comparison of computation cost in query phase.

computation cost of the schemes mentioned above. In Table 3, we denote by P the time cost of bilinear pairing over composite order group or prime order group, by E the time cost of exponentiation, by M the time cost of multiplication over group $\mathcal{G}_1$ or $\mathcal{G}_2$, and by H the hash operation to $\mathbb{Z}_p$ or $\mathcal{G}_1$. In the setup phase, the computation costs of CVDB and FVDB schemes grow in the quadratic function of the size q of the committed vector. The time cost of WDVB, BVDB, and MVDB schemes is all linear to the size q, while BVDB and MVDB schemes are both based on composite order group. Hence, our proposed scheme is most efficient in the setup phase as discussed in Section 6.1. For the update algorithm, the computation cost of these three operations is identical in MVDB scheme since the MHT will be updated no matter which update operation occurs. Nevertheless, as discussed in Remark 6, our proposed WVDB scheme does not need to conduct algorithm MIHT.Update to update the MIHT when a modification operation occurs. It just conducts the algorithm VC.Update to generate the latest commitment. As shown in Remark 3, the algorithm VC.Update needs to conduct 1 exponentiation and at most q multiplications. Hence, the time cost of the modification operation is $1 \cdot E + q \cdot M$ in the worst case. For the other two update operations, it requires to additionally conduct $(\ell + 1)$ hash operations to generate the latest MIHT.

6.3. Simulation

In this section, we provide a thorough experimental evaluation of our proposed scheme. To precisely evaluate the computation cost at all data owner, the cloud and client sides, all simulations were conducted on Linux with Inter(R) Core(TM) i7-8550U CPU

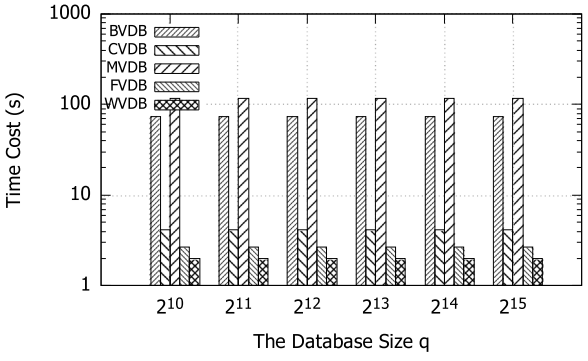

Fig. 7. The comparison of computation cost in verify phase.

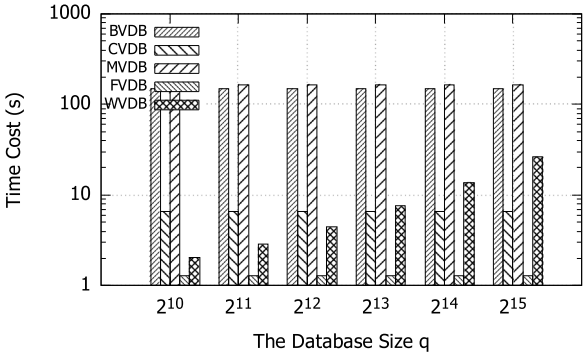

Fig. 8. The comparison of computation cost in modification phase.

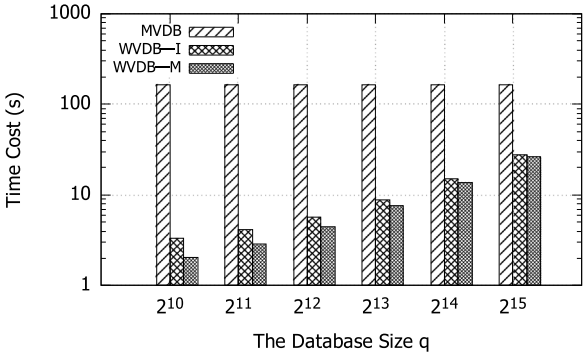

Fig. 9. The comparison of computation cost in insertion phase.

@ 1.80GHZ processor and 8GB memory. The cryptographic algorithms are implemented using PBC library[7] on a type A with a 160-bit group order. The cryptographic hash function is implemented with SHA-256 in OpenSSL.[8] We also use the BLS signature scheme [43] to generate the signature in implements of MVDB and WVDB schemes.

Figs. 5, 6, 7, 8, 9 and 10 respectively depict the comparison of time cost in setup, query, verification, modification, insertion and deletion phases. MVDB and BVDB schemes cost almost the same effort no matter in which phase. Similarly, FVDB scheme costs almost the same effort as the CVDB scheme in each algorithm. In both two cases, the latter one is relatively higher than the former one since it needs to conduct additional operations to support

[7] Available: https://crypto.stanford.edu/pbc/.
[8] Available: https://www.openssl.org/.

Table A.4
Notations.

Notations	Description
κ	The security parameter
ℓ	The parameter that determines the size of APRF
K	The key of APRF
params	The public parameters in APRF
SSK	The private key of signature scheme
SPK	The public key of signature scheme
σ	The signature
q	The size of the committed vector
pp	The public parameter in vector commitment
VSK	The private key in vector commitment scheme
m_i	The ith message in committed vector
$a_i/a_{i_1,\ldots,i_\ell}$	The ith coefficient of polynomial $f(x_1,\ldots,x_\ell)$
C	The commitment value
C'	The updated commitment value
aux	The auxiliary information in vector commitment
Λ_i	The proof that m_i is the ith committed message
Λ_i'	The updated proof for m_i
DB	The outsourced database
PK	The public key in verifiable database scheme
SK	The private key in verifiable database scheme
$\|DB\|$	The outsourced database size
v_i	The ith value returned by the cloud in DB
π_{v_i}	The witness that v_i is equal to $DB[i]$.
PK′	The updated public key
MPK	The public key of MIHT
MSK	The private key of MIHT
auth(D)	The authenticated data structure of data set D
I	The index such that the queried value x is exactly located in the interval of leaf node n_I
Ω_x	The proof that the queried value x is exactly located in the interval of leaf node n_I
auth(D')	The updated authenticated data structure
σ'	The updated signature
n_j	The jth node in MIHT
n_R	The root node of MIHT
v_{n_j}	The value of node n_j
Υ_{n_j}	The hash value of node n_j
Υ'_{n_j}	The updated hash value of node n_j

some special property. For example, CVDB needs to conduct hash and signature operations to resist the FAU attack. As can be seen in Fig. 5, the time cost of BVDB and MVDB schemes is far less than that of CVDB and FVDB scheme, while they are all higher than ours. All of these schemes require an expensive computational effort, which can be amortized over future executions. Furthermore, this algorithm can be conducted offline or in a multi-threading way. The result of Fig. 6 is similar to that of Fig. 5. Note that our scheme remains most efficient in the query phase. As shown in Fig. 7, the computation cost of algorithm Verify in each scheme is almost independent of the database size q. The time cost of BVDB and MVDB is far greater than that of CVDB and FVDB, while they are all greater than WVDB's. As discussed above, the time cost of update operation of our proposed scheme is different in different cases, while the others are same in different cases. In Fig. 8, we compare our proposed scheme in the worst case with others. As can be seen in Fig. 8, the execution time of modification operation of our proposed scheme is linear to the database size q, while others are almost independent of data size q. Our scheme is not the most effective one, but it is acceptable for a resource-limited client since it just costs several seconds. Fig. 9 illustrates the comparison of time cost in the insertion phase. WVDB-I/WVDB-M represents our proposed scheme WVDB when an insertion/modification operation occurs. It is not hard to find that the time cost of WVDB-M and WVDB-I is far less than that of MDVB even in the worst case. What is more, WVDB-M is more efficient than WVDB-I since WVDB-M does not need to update the MIHT. Fig. 10 depicts the comparison of time cost in the deletion phase. As can been seen in Fig. 10, our proposed VDB scheme is more efficient than MDVD. By comparing Figs. 9 and 10, we can find that the time cost of deletion operation is identical to that of insertion operation. This experiment result is consistent with the theoretical analysis in Section 6.2.

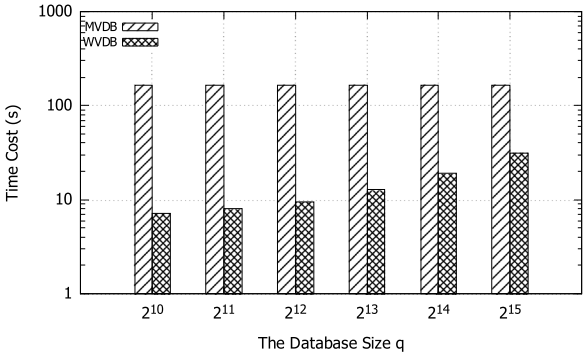

Fig. 10. The comparison of computation cost in deletion phase.

7. Conclusion

The primitive of the verifiable database is useful to solve the problem of verifiable outsourced storage. However, the previous schemes do not either support public verifiability and all update operations or resist forward automatic update attack. To resolve this problem, we first introduce two new primitives called vector commitment and Merkle interval hash tree. We use these two primitives as the building-blocks to construct efficient VDB schemes that achieving the properties above simultaneously. Security analysis shows that our proposed VDB scheme can achieve real-world security requirements. The detailed performance analyses and simulations show that our proposed schemes are more practical in the real world.

CRediT authorship contribution statement

Qiang Wang: Conceived the presented idea, Developed the theory, Discussed the scheme and gave the construction of the proposed scheme, Carried out the simulations, Proved the security of the proposed scheme, Writing - original draft, Provided critical feedback and helped shape the research, analysis, and manuscript, Proofread the whole manuscript, Writing - review & editing. **Fucai Zhou:** Conceived the presented idea, Developed the theory, Discussed the scheme and gave the construction of the proposed scheme, Provided critical feedback and helped shape the research, analysis, and manuscript, Proofread the whole manuscript, Writing - review & editing. **Jian Xu:** Conceived the presented idea, Important inspiration for the construction of the vector commitment, Discussed the scheme and gave the construction of the proposed scheme, Proved the security of the proposed scheme, Provided critical feedback and helped shape the research, analysis, and manuscript, Proofread the whole manuscript, Writing - review & editing. **Zifeng Xu:** Conceived the presented idea, Important inspiration for the construction of the vector commitment, Discussed the scheme and gave the construction of the proposed scheme, Carried out the simulations, Provided critical feedback and helped shape the research, analysis, and manuscript, Proofread the whole manuscript, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We thank the anonymous reviewers and Vladimir Kolesnikov for their fruitful suggestions. This work was supported by the Natural Science Foundation of China under Grant Nos. 62072090, 61872069, 61772127 and 61703088, the National Science and Technology Major Project of China under Grant No. 2013ZX03002006, the Fundamental Research Funds for the Central Universities of China under Grant No. N171704005.

Appendix. Notations

See Table A.4.

References

[1] P. Mell, T. Grance, et al., The NIST Definition of Cloud Computing, Computer Security Division, Information Technology Laboratory, National, 2011.
[2] BBC, Data on 540 million Facebook users exposed, 2019, https://www.bbc.com/news/technology-47812470, (Accessed 5 October 2019).
[3] ISC, 2019 cloud security report, 2019, https://www.isc2.org/Resource-Center/Reports/Cloud-Security-Report, (Accessed 7 March 2020).
[4] Q. Wang, F. Zhou, S. Peng, Z. Xu, Verifiable outsourced computation with full delegation, in: International Conference on Algorithms and Architectures for Parallel Processing, Springer, 2018, pp. 270–287.
[5] J. Xu, L. Wei, Y. Zhang, A. Wang, F. Zhou, C.-z. Gao, Dynamic fully homomorphic encryption-based merkle tree for lightweight streaming authenticated data structures, J. Netw. Comput. Appl. 107 (2018) 113–124.
[6] S. Benabbas, R. Gennaro, Y. Vahlis, Verifiable delegation of computation over large datasets, in: Annual Cryptology Conference, Springer, 2011, pp. 111–131.
[7] D. Catalano, D. Fiore, Vector commitments and their applications, in: International Workshop on Public Key Cryptography, Springer, 2013, pp. 55–72.
[8] M. Miao, J. Ma, X. Huang, Q. Wang, Efficient verifiable databases with insertion/deletion operations from delegating polynomial functions, IEEE Trans. Inf. Forensics Secur. 13 (2) (2017) 511–520.
[9] X. Chen, J. Li, X. Huang, J. Ma, W. Lou, New publicly verifiable databases with efficient updates, IEEE Trans. Dependable Secure Comput. 12 (5) (2014) 546–556.
[10] J. Shen, D. Liu, M.Z.A. Bhuiyan, J. Shen, X. Sun, A. Castiglione, Secure verifiable database supporting efficient dynamic operations in cloud computing, IEEE Trans. Emerg. Top. Comput. (2017).
[11] X. Chen, J. Li, J. Weng, J. Ma, W. Lou, Verifiable computation over large database with incremental updates, IEEE Trans. Comput. 65 (10) (2015) 3184–3195.
[12] R. Gennaro, C. Gentry, B. Parno, Non-interactive verifiable computing: Outsourcing computation to untrusted workers, in: Annual Cryptology Conference, Springer, 2010, pp. 465–482.
[13] B. Parno, M. Raykova, V. Vaikuntanathan, How to delegate and verify in public: Verifiable computation from attribute-based encryption, in: Theory of Cryptography Conference, Springer, 2012, pp. 422–439.
[14] C. Cachin, E. Ghosh, D. Papadopoulos, B. Tackmann, Stateful multi-client verifiable computation, in: International Conference on Applied Cryptography and Network Security, Springer, 2018, pp. 637–656.
[15] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, S. Zahur, Geppetto: Versatile verifiable computation, in: 2015 IEEE Symposium on Security and Privacy, IEEE, 2015, pp. 253–270.
[16] S. Peng, F. Zhou, J. Li, Q. Wang, Z. Xu, Efficient, dynamic and identity-based remote data integrity checking for multiple replicas, J. Netw. Comput. Appl. 134 (2019) 72–88.
[17] C.B. Tan, M.H.A. Hijazi, Y. Lim, A. Gani, A survey on proof of retrievability for cloud data integrity and availability: Cloud storage state-of-the-art, issues, solutions and future trends, J. Netw. Comput. Appl. 110 (2018) 75–86.
[18] K. He, J. Chen, Q. Yuan, S. Ji, D. He, R. Du, Dynamic group-oriented provable data possession in the cloud, IEEE Trans. Dependable Secure Comput. (2019).
[19] N. Bitansky, R. Canetti, A. Chiesa, E. Tromer, From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again, in: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ACM, 2012, pp. 326–349.
[20] S. Goldwasser, Y.T. Kalai, G.N. Rothblum, Delegating computation: interactive proofs for muggles, J. ACM 62 (4) (2015) 27.
[21] Y.T. Kalai, R. Raz, Probabilistically checkable arguments, in: Annual International Cryptology Conference, Springer, 2009, pp. 143–159.
[22] B. Parno, J. Howell, C. Gentry, M. Raykova, Pinocchio: Nearly practical verifiable computation, in: 2013 IEEE Symposium on Security and Privacy, IEEE, 2013, pp. 238–252.
[23] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, M. Virza, SNARKs for C: Verifying program executions succinctly and in zero knowledge, in: Annual Cryptology Conference, Springer, 2013, pp. 90–108.
[24] E. Ben-Sasson, A. Chiesa, E. Tromer, M. Virza, Scalable zero knowledge via cycles of elliptic curves, Algorithmica 79 (4) (2017) 1102–1160.
[25] E. Ben-Sasson, A. Chiesa, E. Tromer, M. Virza, Succinct non-interactive zero knowledge for a von Neumann architecture, in: 23rd {*USENIX*} Security Symposium ({*USENIX*} Security 14), 2014, pp. 781–796.
[26] B. Braun, A.J. Feldman, Z. Ren, S. Setty, A.J. Blumberg, M. Walfish, Verifying computations with state, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 341–357.
[27] M. Etemad, A. Küpçü, Verifiable database outsourcing supporting join, J. Netw. Comput. Appl. 115 (2018) 1–19.
[28] A. Miller, M. Hicks, J. Katz, E. Shi, Authenticated data structures, generically, ACM SIGPLAN Not. 49 (1) (2014) 411–423.
[29] C. Xu, Q. Chen, H. Hu, J. Xu, X. Hei, Authenticating aggregate queries over set-valued data with confidentiality, IEEE Trans. Knowl. Data Eng. 30 (4) (2017) 630–644.
[30] Y. Zhang, J. Katz, C. Papamanthou, IntegriDB: Verifiable SQL for outsourced databases, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 1480–1491.
[31] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, C. Papamanthou, VSQL: Verifying arbitrary SQL queries over dynamic outsourced databases, in: 2017 IEEE Symposium on Security and Privacy, SP, IEEE, 2017, pp. 863–880.
[32] H. Pang, K.-L. Tan, Authenticating query results in edge computing, in: Proceedings. 20th International Conference on Data Engineering, IEEE, 2004, pp. 560–571.
[33] Y. Yang, D. Papadias, S. Papadopoulos, P. Kalnis, Authenticated join processing in outsourced databases, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, ACM, 2009, pp. 5–18.
[34] Z. Zhang, X. Chen, J. Li, X. Tao, J. Ma, HVDB: a hierarchical verifiable database scheme with scalable updates, J. Ambient Intell. Hum. Comput. 10 (8) (2019) 3045–3057.
[35] J. Camenisch, A. Lysyanskaya, Dynamic accumulators and application to efficient revocation of anonymous credentials, in: Annual International Cryptology Conference, Springer, 2002, pp. 61–76.
[36] R. Canetti, O. Paneth, D. Papadopoulos, N. Triandopoulos, Verifiable set operations over outsourced databases, in: International Workshop on Public Key Cryptography, Springer, 2014, pp. 113–130.
[37] L. Nguyen, Accumulators from bilinear pairings and applications, in: Cryptographers' Track At the RSA Conference, Springer, 2005, pp. 275–292.
[38] H. Pang, J. Zhang, K. Mouratidis, Scalable verification for outsourced dynamic databases, Proc. VLDB Endow. 2 (1) (2009) 802–813.
[39] B. Palazzi, M. Pizzonia, S. Pucacco, Query racing: fast completeness certification of query results, in: IFIP Annual Conference on Data and Applications Security and Privacy, Springer, 2010, pp. 177–192.
[40] J. Wang, X. Chen, J. Li, J. Zhao, J. Shen, Towards achieving flexible and verifiable search for outsourced database in cloud computing, Future Gener. Comput. Syst. 67 (2017) 266–275.
[41] J. Baek, Y. Zheng, Identity-based threshold signature scheme from the bilinear pairings, in: International Conference on Information Technology: Coding and Computing, 2004. Proceedings, Vol. 1, ITCC 2004, IEEE, 2004, pp. 124–128.
[42] J.H. Cheon, Security analysis of the strong Diffie–Hellman problem, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2006, pp. 1–11.
[43] D. Boneh, B. Lynn, H. Shacham, Short signatures from the Weil pairing, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2001, pp. 514–532.
[44] R.C. Merkle, A certified digital signature, in: Conference on the Theory and Application of Cryptology, Springer, 1989, pp. 218–238.
[45] T. Sauer, Y. Xu, On multivariate Lagrange interpolation, Math. Comp. 64 (211) (1995) 1147–1170.
[46] M. Walfish, A.J. Blumberg, Verifying computations without reexecuting them, Commun. ACM 58 (2) (2015) 74–84.
[47] D. Boneh, M. Franklin, Identity-based encryption from the Weil pairing, SIAM J. Comput. 32 (3) (2003) 586–615.

Wang Qiang received his B.S. degree of Information Security and the M.S. degree of Software Engineering from Northeastern University, China, in 2014 and 2016, respectively. He is currently working toward the Ph.D. degree in Software College, Northeastern University, China. His research interests include verifiable computation, private set intersection and outsourced database.

Zhou Fucai received the Ph.D degree of Computer Software and Theory from Northeastern University, China, in 2001. He is currently a Professor and Doctoral Supervisor of Software College in Northeastern University, China. His research interests include cryptography, network security, trusted computing, secure cloud storage, software security, basic theory and critical technology in electronic commerce.

Jian Xu received his Ph.D. degree in computer application technology from Northeastern University in 2013. He is currently an associated Professor of Software College in Northeastern University. His research interests include cryptography and network security.

Zifeng Xu received his B.S. degree of Computer Science and the M.S. degree of Computer Science from Bristol University, United Kingdom, in 2011and 2012, respectively. He is currently working towards the Ph.D degree in Software College, Northeastern University, China. His research interests include verifiable computation, multi-party computation and searchable encryption.