Publicly Verifiable Batch Data Deletion Supporting Version Control Without TTP

Abstract

Secure data deletion (SDD) is an important problem in the file system, which guarantees the deleted data is permanently inaccessible to any entity, including the data owner. With the rapid development of cloud storage, the need arises to carry out data deletion over outsourced data in the cloud service provider (CSP). However, due to the possibility of CSP's misbehavior, it is essential for the data owner not only to ensure that the data has been deleted honestly by the CSP but also to protect the privacy of the outsourced data. To resolve these problems, plenty of SDD schemes have been proposed. However, the existing works cannot be able to deal with the following cases simultaneously: (i) no trusted third party (TTP), (ii) version control, (iii) public verification, (iv) batch operations, (v) constant communication cost. To resolve these problems, we first introduce a novel primitive called batch vector commitment (BVC) as the building block, which additionally allows the data owner to open, verify and update the commitment in a batch way. Then, we construct a publicly verifiable batch data deletion scheme supporting version control without any TTP, which satisfies all the above requirements. Besides, the CSP cannot refuse to admit the existence of the outsourced file if the file has not been deleted by the data owner. Finally, the theoretical analysis and simulation show that our proposed scheme is more practical in the real world.

Keywords

Data Assured Deletion, Batch Vector Commitment, Batch Operations, Version Control, Cloud Storage.

1 Introduction

Secure data deletion (SDD) is an important problem in the file system. It requires permanently deleting data from a physical disk such that the deleted data is irrecoverable and inaccessible by any party. Due to its importance, several SDD protocols[1–10] have been proposed. These schemes can be regarded as a "one-bit-return" protocol[11]. Namely, the data owner sends a command to delete the data from the storage system and receives a one-bit response (Success/Failure) indicating the status of the deletion operation.

With the rapid development of cloud computing, storage outsourcing is becoming a trend. Despite of tremendous benefits, the past real-world incidents [12] and recent researches [13–16] have shown that the cloud service provider (CSP) cannot be fully trusted and it may misbehave by exposing or tampering with data owner's sensitive data, or forging the computation results for some economic incentives. So, it is essential for the data owner not only to force the CSP to delete the data honestly but also to protect the confidentiality of their outsourced data. Bonehand and Lipton [11] proposed the first solution based on cryptography in 1996. Their idea is to encrypt each data using a different key before uploading them and then delete the plaintext, discard the key to make the ciphertext irrecoverable when deleting the data. Despite the CSP may not erase the outsourced ciphertext or furtively back up it before deletion, no one including the data owner can be able to decode the ciphertext. Deleting the key will permanently make them invalid, so the data has been completely deleted from the view of the data owner. Following this pioneering work, plenty of similar works [7, 11, 17–20] have been proposed in the past decades. This kind of method is very efficient since it erases a large amount of data by deleting the corresponding short keys. Although removing the data by discarding the key is indeed valid, the main challenge is key management since the data owner is a resource-constrained entity. Most of the existing SDD schemes shift their heavy burden to the trusted third party (TTP). However, these schemes will be no longer secure if the TTP is compromised by any malicious adversary. Therefore, it is essential to design an SDD scheme without any TTP. On the other hand, public verifiability [7, 13, 16, 21] also is a desired property in cloud computing. To the best of our knowledge, Yang et al.[17] introduced the first and only one publicly verifiable SDD scheme without any TTP based on the blockchain technique. This scheme takes advantage of the Merkle hash tree to record the deletion operations. To prove the validity of the deletion operation, the CSP needs to return all sibling nodes in the path from the leaf node to the root node as proof. The communication cost is proportional to the logarithm of the number of deletion records stored in the leaf nodes. When the number of deletion records is large, this scheme is not suitable for a network with narrow bandwidth. Sometimes, the communication overhead might be even more important than the computation overhead since it is often easier to add computation power than to improve the outgoing communication. Meanwhile, there is no mechanism dealing with the conflict that the data owner has already outsourced his file to the CSP but the CSP does not admit this fact. Besides, it does not support version control and batch operations.

1.1 Main Design Goals

Our main design goals can be summarized as follows:

(1) **No TTP.** All interaction between the data owner and the CSP does not need any help of TTP.
(2) **Privacy.** The malicious CSP cannot be able to expose the sensitive data of the data owner.
(3) **Batch Operations.** The data owner can deal with multiple files at the same time. Besides, the running time of performing such a batch operation should be less than the sum of that of carrying out the operation on each data one by one.
(4) **Version Control.** There may exist several versions of the data since the data can be overwritten many times. If the CSP returns a previous version of data as the download response, the misbehavior can be detected with an overwhelming probability.
(5) **Public Verifiability.** Any verifier including the data owner can check not only whether the data has been deleted irreversibly and the download data is the newest one but also the integrity of the data.

(6) **Existence Traceability.** If the file has already been uploaded and not been deleted by the data owner, the CSP cannot refuse to admit the existence of the outsourced file.
(7) **Constant Communication Complexity.** The communication complexity between the verifier and the CSP is independent of the number of deletion records.

1.2 Our Results and Contributions

Our major results and contributions can be summarized as follows:

(1) To enhance the expressiveness of vector commitment, we first extend it to a new notion called batch vector commitment (BVC) as the building block. It additionally allows the data owner to open, verify and update the commitment in a batch manner. For security, it should additionally satisfy the property of batch position binding. Namely, the commitment cannot be opened to two different message sets under the same opened position set.
(2) To strengthen the practicability of secure data deletion, we construct the first one publicly verifiable batch data deletion supporting version control without any TTP (PVBDC). It can check whether CSP has deleted the data honestly without any help of TTP. Besides, our proposed scheme additionally supports version control, batch operations, constant communication complexity, and existence traceability. Finally, the theoretical analysis and simulation show that our proposed scheme is more practical.

1.3 Technical Overview

The high-level idea behind our construction is similar to the general cryptography-based solution. We apply the IND-CPA secure encryption to protect the confidentiality of outsourced files. Each file F_i is encrypted using a different key before being uploaded to CSP. Later, the data owner deletes the decryption key to make the ciphertext invalid. In such a way, no one can recover the plaintext without the key, so it achieves secure data deletion from the view of the data owner. To resist the rollback attack, a unique counter count_{F_i} is incorporated into the encryption key K_{F_i} during each upload phase. As a result, the ciphertext of the same file is different in different upload phases. At the setup phase, the counter count_{F_i} is equal to 0. After each upload, the counter count_{F_i} will be increased by 1. When deleting the file F_i, the counter count_{F_i} will be set to 0. It achieves file assured deletion as the decryption key of the deleted file cannot be recovered. Since the data owner is a resource-limited entity, it cannot be able to maintain the counter information of each file. Otherwise, it will deviate from the original intention of the cloud storage. Hence, the main challenge is how to force CSP to maintain the counter information honestly. This can be achieved by using IBF and vector commitment (see Remark 4). To support batch operations, we replace vector commitment with our proposed batch vector commitment in Section 3. If there is a false positive (i.e., all cells associated with this new file are occupied) when a new file is uploaded, CSP has to batch open the commitment at these positions and check whether $\mathsf{hashSum}$ equals to 0 in these cells (see Step 2(f)iv). If the file has already been uploaded or there is a vacant cell when uploading a new file, CSP just needs to find its corresponding position in IBF (see Step 2(c)i) and open the commitment at this position (see Step 2(c)ii).

1.4 Related Work

Secure data deletion (SDD) guarantees the deleted data is permanently inaccessible to any entities including the data owner. Over the past decades, researchers have paid much attention to this problem and proposed lots of state-of-the-art schemes [1, 7, 17–20, 22–24]. These schemes can be regarded as a "one-bit-return" protocol [11]. Namely, the data owner sends a command to delete the data from the storage system and receives a one-bit response (Success/Failure) indicating the status of the deletion operation. These "one-bit-return" protocols can be roughly categorized into three folds: deletion by unlinking [1, 22], deletion by overwriting [23, 24] and deletion by cryptography [7, 11, 17–20]. The first one is to remove the link of the file from the underlying file system. Although the link of the file has been deleted, the content of the file remains on the disk. An attacker with a forensic tool can easily recover the deleted file by scanning the disk. The second one is to overlay the deleted data with some random data. Overwriting the data with some random data cannot guarantee the deleted data can be covered completely, so the malicious adversary still can partially get the content of the data. Besides, these two methods cannot deal with the case where the malicious adversary stealthily makes multiple data backup copies. The third one is to encrypt the data before saving it to the disk and later deleting the data by discarding the decryption key. Neither deletion by unlinking nor by overwriting can be applied to the real world due to the problems of security and confidentiality. In the following, we will focus on the problem of cryptography-based SDD.

With the growing impact of cloud computing, data outsourcing is becoming a trend. Despite tremendous benefits, cloud storage inevitably suffers from some new security challenges [13, 14, 25, 26] such as data disclosure, data tampering, and data deletion. *Privacy* and *verifiability* have been the key roadblock to popularize this technology because enterprises or individual will lose control over their data once the data is outsourced to the remote CSP. All operations over their data will be executed by the malicious CSP on behalf of the data owner. For the problem of SDD, the main challenge is how to assure the data has been completely deleted by CSP. Deleting the data by discarding the key can meet the above requirements, but the biggest problem is key management since the data owner is a resource-limited entity. To relieve this heavy burden of the data owner, most of the prior works introduce the trusted third party (TTP). However, these schemes will be no longer secure if the TTP is compromised by the malicious CSP. Hence, it is really impractical in the real world. To the best of our knowledge, Yang et al. [17] proposed the first and only one publicly verifiable SDD scheme without any TTP based on blockchain, which takes the deletion record as the leaf node of the Merkle tree. The communication cost is proportional to the logarithm of the number of the deletion records stored in the leaf nodes. When the number of the deletion records is large, this scheme is not suitable for the network with narrow bandwidth. Meanwhile, this scheme does not support version control, batch operations. Besides, it cannot deal with the conflict that CSP refuses to admit the existence of the file while it

has been outsourced and not been deleted by CSP. Therefore, it is essential to construct *a publicly verifiable SDD scheme without any TTP satisfying all the above requirements simultaneously.*

1.5 Paper Organization

The rest of this paper is organized as follows. Section 2 reviews some knowledge needed beforehand. Section 3 introduces the main building block called batch vector commitment and proves its security. Section 4 proposes a detailed construction of our scheme and proves its security. Section 5 provides a detailed theoretical analysis and simulation. Finally, Section 6 concludes this paper.

2 Preliminaries

In this section, we first give some necessary definitions that are going to be used in the rest of the paper. Let $\kappa \in \mathbb{N}$ denote the security parameter. If a function $negl(\kappa)$ is a negligible function of κ, $negl(\kappa)$ must be less than $1/poly(\kappa)$ for arbitrary polynomial $poly(\kappa)$. PPT is the 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 $\mathcal{D}$. We use $|\mathcal{D}|$ to denote the size of $\mathcal{D}$. If q is an integer, we denote the set $\{1, \cdots, q\}$ with $[q]$. Table 3 give a quick reference for the notations in Appendix A.

2.1 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 [27] which satisfies the following properties:

(1) Bilinearity. $\forall u, v \in \mathcal{G}_1$ and $a, b \in \mathbb{Z}_p$, $e\left(u^a, v^b\right) = e(u, v)^{ab}$.
(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 $e(u, v)$.

2.2 Bilinear Function Collision Resistance Assumption [28]

Given a set of elements $\mathcal{D}$, a function $f\left(d_1, \cdots, d_q\right) = \sum_{i=1}^{q} g^{d_i \cdot \alpha^i}$, t $g, g^{\alpha}, g^{\alpha^2}, \cdots, g^{\alpha^q}$ for some t, α chosen at random from $\mathcal{Z}_p^*$, there exists no PPT algorithm $\mathcal{A}$ that can find a set $\mathcal{D}' \neq \mathcal{D}$ such that $f\left(\mathcal{D}'\right) = f(\mathcal{D})$ except with a negligible probability $negl(\kappa)$.

2.3 Invertible Bloom Filter

Invertible bloom filter (IBF) [29] is an extension of counting bloom filter [30], which additionally provides two rows idSum, hashSum for generating the hash table. The rows idSum and hashSum store the sum of all elements and the sum of all the hash values, respectively. It can be used to recover all elements of the latest set in the dynamic setting. The basic idea behind it is that there exists at least one pure cell for each element whose value can only be affected by himself. Once a pure cell is found (see **Algorithm** 1), the corresponding element can be recovered by dividing its idSum by its count. Unlike standard bloom filters and counting bloom filters, IBF additionally needs three hash functions: f_1, f_2, g for generating an additional hash table B, where $f_1, f_2 : [n] \to [m]$ and $g : [n] \to \left[n^2\right]$. The table B serves as a fallback hash table for "catching" elements that are difficult to recover using the hash table A alone. The construction of B is similar to that of A, but only uses the hash functions f_1 and f_2 to map elements to its cells. In the dynamic setting, IBF consists of two update operations: Deletion (see **Algorithm** 2) and Insertion (see **Algorithm** 3). To delete/insert an element x from/to the IBF, we first need to find each cell that x maps to and decrease/increase its count, subtract/add x from/to its idSum, and subtract/add $g(x)$ from/to its hashSum. In the setup phase, all elements of each cell are set to 0 (vacant cell). Figure 1 illustrates these operations.

Algorithm 1: FindPureCell(x)

Input: an element x
Output: a pure cell P
/*$A[i]$ is the i-th cell of hash table A (i.e., the i-th column of A)*/;
for $i \in [k]$ **do**
  if $A[h_i(x)]$.count $= 0 \wedge A[h_i(x)]$.idSum $= 0$ **then**
    break
  if $g(A[h_i(x)]$.idSum$/A[h_i(x)]$.count$) =$ $A[h_i(x)]$.hashSum$/A[h_i(x)]$.count **then**
    return $A[h_i(x)]$
/*$B[i]$ is the i-th cell of hash table B (i.e., the i-th column of B)*/;
for $i \in \{1, 2\}$ **do**
  if $B[f_i(x)]$.count $= 0 \wedge B[f_i(x)]$.idSum $= 0$ **then**
    break
  if $g(B[f_i(x)]$.idSum$/B[f_i(x)]$.count$) =$ $B[f_i(x)]$.hashSum$/B[f_i(x)]$.count **then**
    return $B[f_i(x)]$
return *null*;

Algorithm 2: Delete(x)

Input: an element x
Output: an invertible bloom filter IBF
/*$A[i]$ is the i-th cell of hash table A (i.e., the i-th column of A)*/;
for $i \in [k]$ **do**
  Decrement $A[h_i(x)$.count$]$ by one;
  Subtract the value x from $A[h_i(x)]$.idSum;
  Subtract the hash value $g(x)$ from $A[h_i(x)]$.hashSum;
/*$B[i]$ is the i-th cell of hash table B (i.e., the i-th column of B)*/;
for $i \in \{1, 2\}$ **do**
  Decrement $B[f_i(x)$.count$]$ by one;
  Subtract the value x from $B[f_i(x)]$.idSum;
  Subtract the hash value $g(x)$ from $B[f_i(x)]$.hashSum;
return IBF;

3 Batch Vector Commitment

In this section, we first introduce a novel primitive called batch vector commitment (BVC) as the building block. Then, a detailed

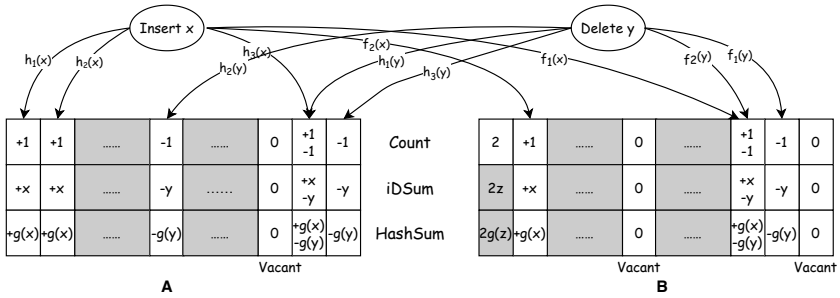


Figure 1: The Updates Performed by Deletion and Insertion Operations in IBF

Algorithm 3: Insert(x)

Input: an element x
Output: an invertible bloom filter IBF
/*$A[i]$ is the i-th cell of hash table A (i.e., the i-th column of A)*/;
for $i \in [k]$ **do**
  Increment $A[h_i(x)]$.count] by one;
  Add the value x from $A[h_i(x)]$.idSum;
  Add the hash value $g(x)$ from $A[h_i(x)]$.hashSum;
/*$B[i]$ is the i-th cell of hash table B (i.e., the i-th column of B)*/;
for $i \in \{1, 2\}$ **do**
  Increment $B[f_i(x)$.count] by one;
  Add the value x from $B[f_i(x)]$.idSum;
  Add the hash value $g(x)$ from $B[f_i(x)]$.hashSum;
return IBF;

construction of BVC is presented. Finally, we give a rigorous proof of the security required in BVC.

3.1 Model Definition

Vector commitment (VC) [31] allows to commit to an ordered sequence of values $(m_1, \cdots, m_q)$ in such a way that it is later possible to open the commitment at a single specific position. It requires that the commitment can not be opened to two different values at the same position (*position binding*). Besides, the size of the commitment and the proof should be independent of the size of the sequence q (*concise*). To the best of our knowledge, there is no prior work supporting the batch opening operation. Namely, the commitment can be simultaneously opened at several positions. Meanwhile, the communication and computation costs should be less than the sum of that of performing the opening algorithm one by one. To facilitate the flexibility of the existing VC schemes [31, 32], we introduce a new notion called batch vector commitment (BVC), in which the commitment can be additionally opened, verified and updated in a batch manner. In terms of security, our new paradigm should additionally satisfy *batch position binding*, which guarantees that no one can open the commitment to two different value sets under the same opened position set. Later we will discuss how to use it to construct our SDD scheme (see Section 4).

Definition 1 (Batch Vector Commitment). *The batch vector commitment scheme is a tuple of nine algorithms as follows:*

$\mathsf{pp} \leftarrow \mathsf{BVC.KeyGen}(1^\kappa, q)$. *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 $\mathcal{M}$).*

$(C, \mathsf{aux}) \leftarrow \mathsf{BVC.Com}_{\mathsf{pp}}(m_1, \cdots, m_q)$. *The committing algorithm is run by the committer. It takes as input the public parameters* pp *and a sequence of q messages $m_1, \cdots, m_q \in \mathcal{M}$ and outputs a commitment value C and an auxiliary information* aux.

$\Lambda_i \leftarrow \mathsf{BVC.Open}_{\mathsf{pp}}(i, m_i, \mathsf{aux})$. *The opening algorithm is run by the committer. It takes as input the public parameters* pp*, the opened position i, the opened message m_i and the auxiliary information* aux *and outputs a proof Λ_i.*

$\{0, 1\} \leftarrow \mathsf{BVC.Ver}_{\mathsf{pp}}(C, m, i, \Lambda_i)$. *The verification algorithm is run by any verifier. It takes as input the public parameters* pp*, the commitment value C, the opened message m, the opened position i and the proof Λ_i. It outputs 1 if Λ_i is a valid proof that C is a commitment to the sequence $(m_1, \cdots, m_q)$ such that $m = m_i$ and 0 otherwise.*

$(C', U) \leftarrow \mathsf{BVC.Upd}_{\mathsf{pp}}(C, m, m', i)$. *The update algorithm is run by the committer who wants to update C by changing the i-th message m to m'. It takes as input the public parameters* pp*, the commitment C, the old message m, the new message m' and the position i and outputs a new commitment C' together with an update information U.*

$\Lambda'_j \leftarrow \mathsf{BVC.ProofUpd}_{\mathsf{pp}}(\Lambda_j, U)$. *The proof update algorithm is run by any verifier who holds a proof Λ_j for some message at the 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 $(m_1, \cdots, m_{i-1}, m', m_{i+1}, \cdots, m_q)$.*

$\Lambda_I \leftarrow \mathsf{BVC.BatchOpen}_{\mathsf{pp}}(I, M, \mathsf{aux})$. *The batch opening algorithm is run by the committer. It takes as input the public parameters* pp*, the opened position set $I = \{i_1, \cdots, i_k\}$ with size k, the the corresponding opened message set M and the auxiliary information* aux *and outputs a proof Λ_I.*

$\{0, 1\} \leftarrow \mathsf{BVC.BatchVer}_{\mathsf{pp}}(C, M, I, \Lambda_I)$. *The batch verification algorithm is run by any verifier. It takes as input the public parameters* pp*, the commitment value C, the opened message set M, the opened*

position set I and the proof Λ_I. It outputs 1 *if Λ_I is a valid proof and* 0 *otherwise.*

$C' \leftarrow \mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C, SU, I)$. *The batch update algorithm is run by the original committer who wants to update C by changing message set M to the message set M' both associated with the same position set I with size k. It takes as input the public parameters* pp, *the commitment C, the update information set SU and the position set I and outputs a new commitment C'.*

3.2 Security Definitions

Intuitively, a batch vector commitment is correct if whenever its algorithms are executed honestly, a valid proof will never be rejected. More formally:

Definition 2 (Correctness). *Let M_a denote the sequence constructed after several invocations of update algorithms* $\mathsf{BVC.Upd}_{\mathsf{pp}}$ *and* $\mathsf{BVC.BatchUpd}_{\mathsf{pp}}$ *(starting from the original sequence M_0 with size q) and likewise for $C_a, U_a, SU_a, \mathsf{aux}_a, I_a, M_{I_a}$. A batch vector commitment scheme is correct if the following holds:*

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{BVC.KeyGen}\,(1^\kappa, q)\,; \\ (C_0, \mathsf{aux}_0) \leftarrow \mathsf{BVC.Com}_{\mathsf{pp}}(M_0); \\ \text{From } a = 0 \text{ to } \ell = poly(\kappa)\,: \\ \quad j \xleftarrow{\$} \{1, \cdots, q\}; \\ \quad (C_a, U_a) \leftarrow \mathsf{BVC.Upd}_{\mathsf{pp}}\left(C_a, m_j, m'_j, j\right); \\ \quad I_a \xleftarrow{\$} \{1, \cdots, q\}; \\ \quad C_{a+1} \leftarrow \mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C_a, SU_a, I_a); \\ i \xleftarrow{\$} \{1, \cdots, q\}, I \xleftarrow{\$} \{1, \cdots, q\}; \\ \Lambda_i \leftarrow \mathsf{BVC.Open}_{\mathsf{pp}}\,(i, m_i, \mathsf{aux}_\ell)\,; \\ \Lambda_I \leftarrow \mathsf{BVC.BatchOpen}_{\mathsf{pp}}(I, M, \mathsf{aux}_\ell)\,: \\ \mathsf{BVC.Ver}_{\mathsf{pp}}\,(C_\ell, m_i, i, \Lambda_i) = 1 \wedge \\ \mathsf{BVC.BatchVer}_{\mathsf{pp}}\,(C_\ell, M, I, \Lambda_I) = 1 \end{array}\right] \geq 1 - negl(\kappa)$$

Remark 1. *The newest auxiliary information* aux_{a+1} *in above experiment can be generated by the updated informations U_a, SU_a and old auxiliary information* aux_a*, where $a \in \{0, \cdots, \ell\}$.*

Intuitively, a batch 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 3 (Position Binding). *Let BVC be a batch vector commitment scheme, and let $\mathcal{A}(\cdot) = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5)$ be a six-tuple of PPT machine. We define security via the following experiment.*

$\mathsf{Exp}^{\mathsf{PB}}_{\mathcal{A}}[BVC, \mathcal{M}, \kappa]$:
$\mathsf{pp} \leftarrow \mathsf{BVC.KeyGen}\,(1^\kappa, q)$;
$M_0 \leftarrow \mathcal{A}_0(\mathsf{pp}, \kappa, q)$;
$(C_0, \mathsf{aux}_0) \leftarrow \mathsf{BVC.Com}_{\mathsf{pp}}(M_0)$;
From $a = 0$ to $\ell = poly(\kappa)$:
  $j \leftarrow \mathcal{A}_1\,(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a)$;
  $\Lambda_a \leftarrow \mathsf{BVC.Open}_{\mathsf{pp}}\,(j, m_j, \mathsf{aux}_a)$;
  $I'_a \leftarrow \mathcal{A}_2\,(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a, \Lambda_0, \cdots, \Lambda_a)$;
  $\Lambda_{I'_a} \leftarrow \mathsf{BVC.BatchOpen}_{\mathsf{pp}}(M_a, I'_a, \mathsf{aux}_a)$;
  $\left(i, m'_i\right) \leftarrow \mathcal{A}_3\,(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a, \Lambda_0, \cdots, \Lambda_a)$;
  $C_a \leftarrow \mathsf{BVC.Upd}_{\mathsf{pp}}\,\left(C_a, m_i, m'_i, i\right)$;
  $(I_a, SU_a) \leftarrow \mathcal{A}_4\,(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a, \Lambda_0, \cdots, \Lambda_a)$;
  $C_{a+1} \leftarrow \mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C_a, SU_a, I_a)$;
$\left(i^*, m'_{i^*}, \Lambda'_{i^*}\right) \leftarrow \mathcal{A}_5(\mathsf{pp}, \left\{C_a, I_a, I'_a, U_a, SU_a, \Lambda_a\right\}_{a \in [\ell]})$;
$b \leftarrow \mathsf{BVC.Ver}_{\mathsf{pp}}\left(C_\ell, m'_{i^*}, i^*, \Lambda'_{i^*}\right)$;
If $m'_{i^*} \neq m_{i^*}$ *and* $b = 1$:
  output 1;
else
  output 0;

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

$$Adv^{\mathsf{PB}}_{\mathcal{A}}(BVC, \mathcal{M}, \kappa) = \Pr\left[\mathsf{Exp}^{\mathsf{PB}}_{\mathcal{A}}[BVC, \mathcal{M}, \kappa] = 1\right]$$

We say that BVC achieves position binding if $Adv^{\mathsf{PB}}_{\mathcal{A}}(BVC, \mathcal{M}, \kappa) \leq negl(\kappa)$.

Intuitively, a batch vector commitment scheme satisfies batch position binding if it is infeasible to open a commitment to two different sets under the same opened position set I. More formally:

Definition 4 (Batch Position Binding). *Let BVC be a batch vector commitment scheme, and let $\mathcal{A}(\cdot) = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_5)$ be a six-tuple of PPT machine. We define security via the following experiment.*

$\mathsf{Exp}^{\mathsf{BPB}}_{\mathcal{A}}[BVC, \mathcal{M}, \kappa]$:
$\mathsf{pp} \leftarrow \mathsf{BVC.KeyGen}\,(1^\kappa, q)$;
$M_0 \leftarrow \mathcal{A}_0(\mathsf{pp}, \kappa, q)$;
$(C_0, \mathsf{aux}_0) \leftarrow \mathsf{BVC.Com}_{\mathsf{pp}}(M_0)$;
From $a = 0$ to $\ell = poly(\kappa)$:
  $j \leftarrow \mathcal{A}_1\,(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a)$;
  $\Lambda_a \leftarrow \mathsf{BVC.Open}_{\mathsf{pp}}\,\left(j, m_j, \mathsf{aux}_a\right)$;
  $I'_a \leftarrow \mathcal{A}_2\,(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a, \Lambda_0, \cdots, \Lambda_a)$;
  $\Lambda_{I'_a} \leftarrow \mathsf{BVC.BatchOpen}_{\mathsf{pp}}(C_a, I'_a, \mathsf{aux}_a)$;
  $\left(i, m'_i\right) \leftarrow \mathcal{A}_3\,(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a, \Lambda_0, \cdots, \Lambda_a)$;
  $C_a \leftarrow \mathsf{BVC.Upd}_{\mathsf{pp}}\,\left(C_a, m_i, m'_i, i\right)$;
  $(I_a, SU_a) \leftarrow \mathcal{A}_4\,(\mathsf{pp}, \kappa, q, C_0, \cdots, C_a, \Lambda_0, \cdots, \Lambda_a)$;
  $C_{a+1} \leftarrow \mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C_a, SU_a, I_a)$;
$\left(I^*, M'_{I^*}, \Lambda'_{I^*}\right) \leftarrow \mathcal{A}_5(\mathsf{pp}, \{C_a, I_a, U_a, \Lambda_a\}_{a \in [\ell]})$;
$b \leftarrow \mathsf{BVC.BatchVer}_{\mathsf{pp}}\left(C_\ell, M'_{I^*}, I^*, \Lambda'_{I^*}\right)$;
If $M'_{I^*} \neq M_{I^*}$ *and* $b = 1$:
  output 1;
else
  output 0;

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

$$Adv_{\mathcal{A}}^{\mathsf{BPB}}(BVC, \mathcal{M}, \kappa) = \Pr\left[\mathrm{Exp}_{\mathcal{A}}^{\mathsf{BPB}}[BVC, \mathcal{M}, \kappa] = 1\right]$$

We say that BVC achieves batch position binding if

$$Adv_{\mathcal{A}}^{\mathsf{BPB}}(BVC, \mathcal{M}, \kappa) \leq negl(\kappa).$$

It is worth noting that position binding (in vector commitment) is a special case of batch position binding, where the opened position set size $|I| = 1$. In other words, it is only opened at a particular position.

3.3 Detailed Construction

The construction of our scheme is detailed as follows:

$\mathsf{pp} \leftarrow \mathsf{BVC.KeyGen}\,(1^\kappa, q)$. 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 $e : \mathcal{G}_1 \times \mathcal{G}_1 \rightarrow \mathcal{G}_2$. Let g be a generator of $\mathcal{G}_1$. After that, it also chooses two randoms α and β from $\mathbb{Z}_p$. Finally, it sets the public parameters:

$$\mathsf{pp} = \left\{\left\{g^{\alpha^i}, g^{\beta^i}\right\}_{i\in[q]}, \left\{g^{\alpha^i\beta^j}\right\}_{i,j\in[2q-1]\setminus\{q\}}\right\}$$

and publishes it. The message space is $\mathcal{M} = \mathbb{Z}_p$.[1]

$(C, \mathsf{aux}) \leftarrow \mathsf{BVC.Com}_{\mathsf{pp}}\left(m_1, \cdots, m_q\right)$. The committer first parses public parameters pp as $\{\{g^{\alpha^i}, g^{\beta^i}\}_{i\in[q]}, \{g^{\alpha^i\beta^j}\}_{i,j\in[2q-1]\setminus\{q\}}\}$. After that, it computes the commitment $C = \prod_{i=1}^{q} g^{m_i\alpha^i}$. Finally, it outputs C and the auxiliary information $\mathsf{aux} = \left(m_1, \cdots, m_q\right)$. Note that the term $g^{m_i\alpha^i} = \left(g^{\alpha^i}\right)^{m_i}$ can be easily computed using the set $\{g^{\alpha^j}\}_{j\in[q]}$ in pp.

$\Lambda_i \leftarrow \mathsf{BVC.Open}_{\mathsf{pp}}\,(i, m_i, \mathsf{aux})$. The committer first parses public parameters pp as $\{\{g^{\alpha^i}, g^{\beta^i}\}_{i\in[q]}, \{g^{\alpha^i\beta^j}\}_{i,j\in[2q-1]\setminus\{q\}}\}$. After that, it constructs two polynomials $\mathcal{R}(x)$ and $\mathcal{Q}(x, y)$ satisfying:

$$\mathcal{R}(x) = \sum_{j=1}^{q} m_j \cdot x^j \tag{1}$$

$$\mathcal{R}(x) \cdot x^{q-i}y^i = m_i \cdot x^q \cdot y^i + \mathcal{Q}(x, y) \tag{2}$$

Finally, it computes the proof $\Lambda_i = g^{\mathcal{Q}(\alpha,\beta)}$ using the public parameters pp especially the set $\{g^{\alpha^i\beta^j}\}_{i,j\in[2q-1]\setminus\{q\}}$. Note that $\mathcal{Q}(x, y) = \sum_{j=1,j\neq i}^{q} m_j \cdot x^j \cdot x^{q-i}y^i = \sum_{j=1,j\neq i}^{q} m_j \cdot x^{q-i+j}y^i$ based on Equation 2.

$\{0, 1\} \leftarrow \mathsf{BVC.Ver}_{\mathsf{pp}}\,(C, m, i, \Lambda_i)$. After the verifier receives the opened message m_i and the proof Λ_i, it parses public parameters pp as $\{\{g^{\alpha^i}, g^{\beta^i}\}_{i\in[q]}, \{g^{\alpha^i\beta^j}\}_{i,j\in[2q-1]\setminus\{q\}}\}$. After that, it outputs 1 (valid) or 0 (invalid) by checking whether the following equation holds:

$$e\left(C, g^{\alpha^{q-i}\beta^i}\right) = e\left(g^{m_i\cdot\beta^i}, g^{\alpha^q}\right) \cdot e\left(\Lambda_i, g\right) \tag{3}$$

$(C', U) \leftarrow \mathsf{BVC.Upd}_{\mathsf{pp}}\,(C, m, m', i)$. The committer computes $\delta = m' - m$ and generates the new commitment $C' = C \cdot g^{\delta\cdot\alpha^i}$. Finally, it outputs the updated commitment C' and the updated information $U = (\delta, i)$.

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

$\Lambda'_j \leftarrow \mathsf{BVC.ProofUpd}_{\mathsf{pp}}\left(\Lambda_j, U\right)$. A verifier 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 $U = (\delta, i)$ to produce a new proof Λ'_j for m_j with respect to the update sequence $(m_1, \cdots, m_{i-1}, m', m_{i+1}, \cdots, m_q)$. We distinguish two cases:

(1) $j \neq i$. Compute the updated proof $\Lambda'_j = \Lambda_j \cdot g^{\delta\cdot\alpha^{q-j+i}\beta^j}$.
(2) $j = i$. The updated proof remains same as Λ_i.

$\Lambda_I \leftarrow \mathsf{BVC.BatchOpen}_{\mathsf{pp}}(I, M, \mathsf{aux})$. The construction of this algorithm is similar to that of opening algorithm BVC.Open. It constructs three polynomials $\mathcal{R}(x)$, $\mathcal{S}(x, y)$ and $\mathcal{Q}(x, y)$ as follows:

$$\mathcal{R}(x) = \sum_{j=1}^{q} m_j \cdot x^j \tag{4}$$

$$\mathcal{S}(x, y) = \sum_{j=1}^{k} x^{q-i_j} \cdot y^{i_j} \tag{5}$$

$$\mathcal{R}(x) \cdot \mathcal{S}(x, y) = \sum_{j=1}^{k} m_{i_j} \cdot x^q \cdot y^{i_j} + \mathcal{Q}(x, y) \tag{6}$$

Finally, it computes the proof $\Lambda_I = g^{\mathcal{Q}(\alpha,\beta)}$ using the public parameters pp especially the set $\{g^{\alpha^i\beta^j}\}_{i,j\in[2q-1]\setminus\{q\}}$.

$\{0, 1\} \leftarrow \mathsf{BVC.BatchVer}_{\mathsf{pp}}\,(C, M, I, \Lambda_I)$. Upon receiving the opened message set $M = \{m_{i_1}, \cdots, m_{i_k}\}$ and the proof Λ_I, the verifier computes $\mathcal{I}$ as follows:

$$\mathcal{I} = \prod_{j=1}^{k} g^{m_{i_j}\cdot\beta^{i_j}} \tag{7}$$

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

$$\prod_{j=1}^{k} e\left(C, g^{\alpha^{q-i_j}\beta^{i_j}}\right) = e\left(\mathcal{I}, g^{\alpha^q}\right) \cdot e\left(\Lambda_I, g\right) \tag{8}$$

$C' \leftarrow \mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C, SU, I)$. The committer generates the new commitment $C' = C \cdot \prod_{j=1}^{k} g^{\delta_{i_j}\cdot\alpha^{i_j}}$, where $\delta_{i_j} = m'_{i_j} - m_{i_j}$ is the j-th item in SU. Finally, it outputs the updated commitment C'.

Remark 2. *It is not hard to find that the opening algorithm* BVC.Open *can be regarded as the special case of the batch opening algorithm* BVC.BatchOpen *when* $|I| = 1$. *Similarly, the verification algorithm* BVC.Ver *can be regarded as the special case of the batch verification algorithm* BVC.BatchVer. *In our construction, the verifier can check the validity of the opened message set in a batch manner based on Equation 8 rather than check the validity of each opened message based on Equation 3. Therefore, our BVC scheme saves a lot of communication and computation costs. We will use this desired property to reduce the cost of batch operations in our proposed SDD scheme.*

3.3.1 Correctness and Security Analysis. The correctness of our proposed scheme can be found in Appendix B. In what follows, we will focus on proving its security.

Theorem 1. *If there exists a PPT adversary $\mathcal{A}$ winning the position binding experiment as defined in Definition 3 with non-negligible probability $negl(\kappa)$, then the adversary $\mathcal{A}$ can construct an efficient algorithm $\mathcal{B}$ to break bilinear Diffie-Hellman inverse assumption.*

Proof. If there exists a PPT adversary $\mathcal{A}$ with non-negligible advantage against the above security game for position binding, $\mathcal{A}$ can leverage this forgery to construct a method $\mathcal{B}$ to solve bilinear Diffie-Hellman inverse problem.

Supposed $\mathcal{A}$ outputs a forgery $(m'_{i^*}, \Lambda'_{i^*})$ passing the verification, the following equation holds:

$$e\left(C_\ell, g^{\alpha^{q-i^*}\beta^{i^*}}\right) = e\left(g^{m'_{i^*}\cdot\beta^{i^*}}, g^{\alpha^q}\right)\cdot e\left(\Lambda'_{i^*}, g\right) \tag{9}$$

Similarly, for the correct opened message m_{i^*} and proof Λ_{i^*}, we have:

$$e\left(C_\ell, g^{\alpha^{q-i^*}\beta^{i^*}}\right) = e\left(g^{m_{i^*}\cdot\beta^{i^*}}, g^{\alpha^q}\right)\cdot e\left(\Lambda_{i^*}, g\right) \tag{10}$$

Due to Equations 9 and 10, it is not hard to see that:

$$\begin{aligned} e\left(g^{m'_{i^*}\cdot\beta^{i^*}}, g^{\alpha^q}\right)\cdot e\left(\Lambda'_{i^*}, g\right) &= e\left(g^{m_{i^*}\cdot\beta^{i^*}}, g^{\alpha^q}\right)\cdot e\left(\Lambda_{i^*}, g\right) \\ e\left(g^{\left(m'_{i^*}-m_{i^*}\right)\cdot\beta^{i^*}}, g^{\alpha^q}\right) &= e\left(\Lambda_{i^*}/\Lambda'_{i^*}, g\right) \end{aligned} \tag{11}$$

The left hand of Equation 11 can be evaluated by the adversary $\mathcal{A}$, so $\mathcal{A}$ can get the value of $e\left(\Lambda_{i^*}/\Lambda'_{i^*}, g\right)$. By knowing g and $e\left(\Lambda_{i^*}/\Lambda'_{i^*}, g\right)$, if $\mathcal{A}$ succeeds to forge the witness Λ'_{i^*} passing the verification check, $\mathcal{A}$ will provide an efficient method $\mathcal{B}$ to break bilinear Diffie-Hellman inverse assumption. □

Theorem 2. *If there exists a PPT adversary $\mathcal{A}$ winning the batch position binding experiment as defined in Definition 4 with non-negligible probability $negl(\kappa)$, then the adversary $\mathcal{A}$ can construct an efficient algorithm $\mathcal{B}$ to break bilinear Diffie-Hellman inverse assumption and bilinear function collision resistance assumption.*

Proof. If there exists a PPT adversary $\mathcal{A}$ with non-negligible advantage against the above security game for batch position binding, $\mathcal{A}$ can leverage this forgery to construct a method $\mathcal{B}$ to solve either bilinear Diffie-Hellman inverse problem or bilinear function collision resistance problem.

Supposed $\mathcal{A}$ forges a forgery $(M'_{I^*}, \Lambda'_{I^*})$ passing the verification, the following equation holds:

$$\prod_{j=1}^{k} e\left(C, g^{\alpha^{q-i_j}\beta^{i_j}}\right) = e\left(\mathcal{I}', g^{\alpha^q}\right)\cdot e\left(\Lambda'_{I^*}, g\right), \tag{12}$$

where $\mathcal{I}' = \prod_{j=1}^{k} g^{m'_{i_j}\cdot\beta^{i_j}}$.

Similarly, for a correct pair (M, Λ_{I^*}), we have as below:

$$\prod_{j=1}^{k} e\left(C, g^{\alpha^{q-i_j}\beta^{i_j}}\right) = e\left(\mathcal{I}, g^{\alpha^q}\right)\cdot e\left(\Lambda_{I^*}, g\right). \tag{13}$$

Due to Equations 12 and 13, it is not hard to find that:

$$e\left(\mathcal{I}', g^{\alpha^q}\right)\cdot e\left(\Lambda'_{I^*}, g\right) = e\left(\mathcal{I}, g^{\alpha^q}\right)\cdot e\left(\Lambda_{I^*}, g\right). \tag{14}$$

According to Equation 14, it is not hard to see that there are two cases for break batch position binding property:

- Case 1: $\mathcal{I}' = \mathcal{I} \wedge \Lambda'_{I^*} = \Lambda_{I^*}$. $\mathcal{I}' = \mathcal{I}$ means that $\prod_{j=1}^{p} g^{m'_{i_j}\cdot\beta^{i_j}} = \prod_{j=1}^{p} g^{m_{i_j}\cdot\beta^{i_j}}$. If $\mathcal{A}$ succeeds to forge $\mathcal{I}'$ such that $\mathcal{I}' = \mathcal{I}$, it can leverage this forgery to construct a method $\mathcal{B}$ to solve the bilinear function collision resistance assumption.
- Case 2: $\mathcal{I}' \neq \mathcal{I} \wedge \Lambda'_{I^*} \neq \Lambda_{I^*}$. The process of proof is similar to that of proof of Theorem 1. According to Equation 14, we can have:

$$e\left(\mathcal{I}'/\mathcal{I}, g^{\alpha^q}\right) = e\left(\Lambda_{I^*}/\Lambda'_{I^*}, g\right) \tag{15}$$

By knowing g and $e(\mathcal{I}'/\mathcal{I}, g^{\alpha^q})$, if $\mathcal{A}$ succeeds to forge the witness Λ'_{I^*} passing the verification check, $\mathcal{A}$ will provide an efficient method $\mathcal{B}$ to break bilinear Diffie-Hellman inverse assumption.

□

4 PVBDC

4.1 System Model

Our PVBDC protocol comprises three different entities which are illustrated in Figure 2. We describe them in the following:

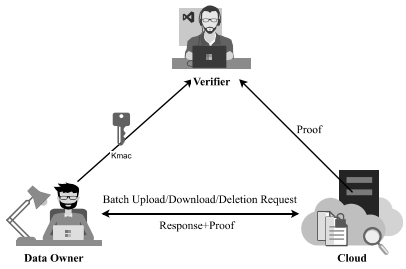


Figure 2: The Architecture of PVBDC Scheme

(1) Data owner O, an honest entity, who can upload his personal data to CSP and does not store any copy locally. When O does not need the uploaded data anymore, he requires CSP to delete the data permanently. At the same time, he can also update the uploaded file. When he needs some file, he would try to download the latest file.
(2) CSP C, an untrusted entity with huge storage and powerful computation resource, who can keep and delete the data on behalf of the data owner. The CSP may not erase the data and return an error result to mislead the data owner. Meanwhile, he may also return a previous version file as the download response.
(3) Verifier V, an honest entity, who knows the public key and public parameters. He can not only check whether the data has been deleted by CSP but also verify whether the file returned by CSP is the latest one.

4.2 Security Requirements

The security model of secure data deletion in this paper is similar to that in [7, 17, 18]. They assume that CSP is malicious and may return a wrong result as the deletion operation response to mislead the data owner. Besides, it may also return an invalid file as the download response since the data owner does not keep any copy of the file. In general, a secure data deletion scheme should satisfy the following security requirements:

(1) Correctness: If all entities follow the proposed protocol honestly, a deletion proof generated by the CSP can always pass the deletion verification. Furthermore, the data owner can upload and download the file correctly.
(2) File Integrity: Although the data owner does not keep any copies of the outsourced files locally, it is still infeasible for the malicious CSP to tamper the files.
(3) Deletion Traceability: Whenever the malicious CSP does not delete the file honestly (i.e., no one can decode the deleted file despite the file being still stored in the physics disk), the misbehavior of the CSP can always be detected with an overwhelming probability.
(4) Existence Traceability: If the file has already been uploaded and not been deleted by the data owner, the CSP cannot refuse to admit the existence of the outsourced file.
(5) Version Control: Whenever the malicious CSP returns an older version of the file F_i, the misbehavior of the CSP can always be detected with an overwhelming probability. Namely, the proposed protocol can resist the rollback attack.
(6) False Positive Traceability: Whenever the malicious CSP deliberately claims that there exists a false positive or not in IBF when uploading a new file F_i, the misbehavior of the CSP can always be detected with an overwhelming probability. Note that this is not a general requirement since we utilize IBF in our proposed scheme.

4.3 Construction

For simplicity, we assume that all messages transmitted between these entities are valid in the following. This can be achieved by using a digital signature scheme. We use the hash-based message authenticated code (HMAC) to guarantee the integrity of the outsourced file F_i. When downloading a file, the data owner needs to check whether the HMAC is valid before decrypting the file. Our proposed scheme consists of seven phases: setup, upload, batch upload, batch deletion, batch download, deletion, download. In the following, we just show the processes of the first five phases. After that, we will show how to achieve the process of download/deletion phase from that of the batch download/deletion phase.

(1) **Setup:** To bootstrap the system parameters, O works as follows:
   (a) O chooses $(k+3)$ hash functions $h_1, \cdots, h_k, f_1, f_2, g$, which are used in IBF (see Section 2.3).
   (b) O chooses hash functions $\mathcal{H}_1, \mathcal{H}_2$, where $\mathcal{H}_1 : \{0,1\}^* \to [n], \mathcal{H}_2 : \{0,1\}^* \to \{0,1\}^\kappa$. Note that $\mathcal{H}_1$ and $\mathcal{H}_2$ are used to generate the tag and the encryption key, respectively.
   (c) O chooses three keys $\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_{mac} \in \{0,1\}^\kappa$, where K_i is the key of hash function $\mathcal{H}_i$ and K_{mac} is the HMAC key.
   (d) O conducts $\mathsf{BVC.KeyGen}(1^\kappa, m)$ to generate the public parameters pp.
   (e) O sets public key $\mathsf{PK}_o = \{h_1, \cdots, h_k, f_1, f_2, g, \mathsf{pp}\}$ and private key $\mathsf{SK}_o = \{\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_{mac}\}$. Then, it publishes the public key PK_o and keeps the private key SK_o secret.
(2) **Upload:** To protect the privacy, O encrypts the outsourced file F_i, and uploads the corresponding ciphertext C_{F_i} to C. If the uploaded file is a new file, O needs to find a vacant cell P (i.e., $P.\mathsf{count} = 0$) to store the corresponding information. If there is no vacant cell in IBF, the data owner has to rename the file. If the file has already been uploaded, O has to find the pure cell P associated with the file to get the counter. O and C work as follows:
   (a) O computes $\mathsf{tag}_{F_i} = \mathcal{H}_1(\mathsf{K}_1 \| \mathsf{name}_{F_i})$ and sends the upload request $R_u = \{\text{“upload”}, \mathsf{tag}_{F_i}\}$ to C.
   (b) Upon receiving the upload request R_u, C sets a set $CS = \emptyset$ and performs **Algorithm** 1 on input tag_{F_i} to get the corresponding pure cell P.
   (c) If $\{P \neq null\} \vee \{P = null \wedge \{\{\exists j \in [k], A[h_j(\mathsf{tag}_{F_i})].\mathsf{hashSum} = 0\} \vee \{\exists j \in [2], B[f_j(\mathsf{tag}_{F_i})].\mathsf{hashSum} = 0\}\}\}$:
      (i) C finds the index ℓ of the cell P in the table $T \in \{A, B\}$ (i.e., $P = T[\ell]$). Note that the cell P is pure or vacant.
      (ii) C conducts $\Lambda_\ell^{(T)} \leftarrow \mathsf{BVC.Open}(\ell, P.\mathsf{hashSum}, \mathsf{aux}^{(T)})$ to prove that there is no false positive in IBF.
      (iii) C sends the response $\mathsf{Resp}_u = \{\text{“the description of T”}, \ell, P, \Lambda_\ell^{(T)}, CS\}$ to O.
   (d) If $P = null \wedge \{A[h_i(\mathsf{tag}_{F_i})].\mathsf{hashSum} \neq 0\}_{i\in[k]} \wedge \{B[f_i(\mathsf{tag}_{F_i})].\mathsf{hashSum} \neq 0\}_{i\in[2]}$:
      (i) C computes $I_{F_i}^{(A)} = \{h_1(\mathsf{tag}_{F_i}), \cdots, h_k(\mathsf{tag}_{F_i})\}$ and $I_{F_i}^{(B)} = \{f_1(\mathsf{tag}_{F_i}), f_2(\mathsf{tag}_{F_i})\}$. C adds $\{A[h_1(\mathsf{tag}_{F_i})], \cdots, A[h_k(\mathsf{tag}_{F_i})], B[f_1(\mathsf{tag}_{F_i})], B[f_2(\mathsf{tag}_{F_i})]\}$ to CS.
      (ii) C conducts $\Lambda_{I_{F_i}^{(A)}} \leftarrow \mathsf{BVC.BatchOpen}_{\mathsf{pp}}(I_{F_i}^{(A)}, M^{(A)}, \mathsf{aux}^{(A)})$, where $M^{(A)} = \{A[h_1(\mathsf{tag}_{F_i})].\mathsf{hashSum}, \cdots, A[h_k(\mathsf{tag}_{F_i})].\mathsf{hashSum}\}$. Similarly, C conducts $\Lambda_{I_{F_i}^{(B)}} \leftarrow \mathsf{BVC.BatchOpen}_{\mathsf{pp}}(I_{F_i}^{(B)}, M^{(B)}, \mathsf{aux}^{(B)})$, where $M^{(B)} = \{B[f_1(\mathsf{tag}_{F_i})].\mathsf{hashSum}, B[f_2(\mathsf{tag}_{F_i})].\mathsf{hashSum}\}$. Note that this step is used to prove that the file F_i has not been stored and there is a false positive in IBF.
      (iii) C sends the response $\mathsf{Resp}_u = \{\Lambda_{I^{(A)}}, M^{(A)}, \Lambda_{I^{(B)}}, M^{(B)}, CS\}$ to O.
   (e) Upon receiving the response Resp_u, O sets $I_{F_i}^{(B)} = \{f_1(\mathsf{tag}_{F_i}), f_2(\mathsf{tag}_{F_i})\}$ and $I_{F_i}^{(A)} = \{h_1(\mathsf{tag}_{F_i}), \cdots, h_k(\mathsf{tag}_{F_i})\}$.
   (f) If $CS = \emptyset$:
      (i) O conducts $\mathsf{BVC.Ver}_{\mathsf{pp}}(C^{(T)}, P.\mathsf{hashSum}, \ell, \Lambda_\ell^{(T)})$ to check whether the opened hash sum $P.\mathsf{hashSum}$ is valid. If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.
      (ii) O checks whether the following equation holds:

$$\ell \in I_{F_i}^{(T)} \tag{16}$$

      If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.
      (iii) If $P.\mathsf{hashSum} \neq 0$, O checks whether the following equation holds:

$$\frac{P.\mathsf{idSum}}{\mathsf{tag}_{F_i}} = \frac{P.\mathsf{hashSum}}{g(\mathsf{tag}_{F_i})} = P.\mathsf{count} \tag{17}$$

      If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(iv) If $P.\mathsf{hashSum} = 0$, O checks whether the following equation holds:

$$\{T[\ell].\mathsf{count} = 0 \wedge T[\ell].\mathsf{idSum} = 0\} = 1. \quad (18)$$

If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(g) If $CS \neq \emptyset$:

(i) O checks whether 0 belongs to $M^{(A)}$ and $M^{(B)}$. If so, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(ii) O conducts $\mathsf{BVC.BatchVer}_{\mathsf{pp}}(C^{(A)}, M^{(A)}, I^{(A)}_{F_i}, \Lambda_{I^{(A)}_{F_i}})$ and $\mathsf{BVC.BatchVer}_{\mathsf{pp}}(C^{(B)}, M^{(B)}, I^{(B)}_{F_i}, \Lambda_{I^{(B)}_{F_i}})$ to check whether the opened message sets $M^{(A)}$ and $M^{(B)}$ are valid. If either of them is invalid, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(iii) For each element $P \in CS$, O checks whether Equation 17 holds. If so, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(iv) O renames the file F_i and repeats the above steps until $CS = \emptyset$.

(h) O sets $\mathsf{count}_{F_i} = P.\mathsf{count} + 1$ and computes $\mathsf{K}_{F_i} = \mathcal{H}_2(\mathsf{K}_2 \| \mathsf{tag}_{F_i} \| \mathsf{count}_{F_i})$. Next, O encrypts the file F_i as $C_{F_i} = Enc(\mathsf{K}_{F_i}, F_i)$ and generates the message authentication code $\mathsf{MAC}_{F_i} = \mathsf{HMAC}(\mathsf{K}_{mac}, C_{F_i} \| \mathsf{count}_{F_i} \| \mathsf{tag}_{F_i})$. Note that Enc is an IND-CPA secure encryption scheme.

(i) O updates commitments $C^{(A)}$ and $C^{(B)}$ by conducting $\mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C^{(A)}, \{g(\mathsf{tag}_{F_i})\}^k, I^{(A)}_{F_i})$ and $\mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C^{(B)}, \{g(\mathsf{tag}_{F_i})\}^2, I^{(B)}_{F_i})$, respectively. Then, O publishes the updated commitments $C^{(A)}$ and $C^{(B)}$.

(j) O deletes the local file F_i and sends $\mathsf{tag}_{F_i}, C_{F_i}, \mathsf{MAC}_{F_i}$ to C.

(k) Upon receiving $\mathsf{tag}_{F_i}, C_{F_i}, \mathsf{MAC}_{F_i}$, C updates IBF by conducting **Algorithm** 3 on input tag_{F_i} and stores C_{F_i} and MAC_{F_i}.

(3) **Batch Upload:** The process of batch upload is similar to that of upload, where conducting the opening algorithm and the verification algorithm multiple times will be replaced with conducting the batch opening and the batch verification algorithm one single time. When the uploaded file set is U, O and C work as follows:

(a) O sets the set $\mathsf{Tag} = \{\mathsf{tag}_{F_i} | F_i \in U\}$, where $\mathsf{tag}_{F_i} = \mathcal{H}_1(\mathsf{K}_1 \| \mathsf{name}_{F_i})$. Next, O sends the upload request $R_u =$ { "upload", Tag} to C.

(b) Upon receiving the request R_u, C sets eight empty sets $I^{(A)}$, $I^{(B)}$, $M^{(A)}$, $M^{(B)}$, $CS^{(A)}$, $CS^{(B)}$, $TG^{(A)}$, $TG^{(B)}$. Note that $TG^{(A)}/TG^{(B)}$ is used to identify the tag, where a pure cell associated with the tag can be found in the hash table A or B.

(c) For any $\mathsf{tag}_{F_i} \in \mathsf{Tag}$:

(i) C finds the corresponding pure cell P by performing **Algorithm** 1 on input tag_{F_i} .

(ii) If $\{P \neq null\} \vee \{P = null \wedge \{\{\exists j \in [k], A[h_j(\mathsf{tag}_{F_i})].\mathsf{hashSum} = 0\} \vee \{\exists j \in [2], B[f_j(\mathsf{tag}_{F_i})].\mathsf{hashSum} = 0\}\}\}$:

(A) C finds the index ℓ of the cell P in the table $T \in \{A, B\}$.

(B) C appends the index ℓ, $P.\mathsf{hashSum}$, P and tag_{F_i} to the opening position set $I^{(T)}$, the opened message set $M^{(T)}$, the cell set $CS^{(T)}$ and the tag set $TG^{(T)}$, respectively.

(iii) If $P = null \wedge \{A[h_i(\mathsf{tag}_{F_i})].\mathsf{hashSum} \neq 0\}_{i\in[k]} \wedge \{B[f_i(\mathsf{tag}_{F_i})].\mathsf{hashSum} \neq 0\}_{i\in[2]}$:

(A) C appends $I^{(A)}_{F_i} = \{h_1(\mathsf{tag}_{F_i}), \cdots, h_k(\mathsf{tag}_{F_i})\}$, all corresponding hash sum, all corresponding cells and tag_{F_i} to $I^{(A)}$, $M^{(A)}$, $CS^{(A)}$ and $TG^{(A)}$, respectively.

(B) C appends $I^{(B)}_{F_i} = \{f_1(\mathsf{tag}_{F_i}), f_2(\mathsf{tag}_{F_i})\}$, all corresponding hash sum, all corresponding cells and tag_{F_i} to $I^{(B)}$, $M^{(B)}$, $CS^{(B)}$ and $TG^{(B)}$, respectively.

(d) C conducts $\Lambda_{I^{(A)}} \leftarrow \mathsf{BVC.BatchOpen}(I^{(A)}, M^{(A)}, \mathsf{aux}^{(A)})$ and $\Lambda_{I^{(B)}} \leftarrow \mathsf{BVC.BatchOpen}(I^{(B)}, M^{(B)}, \mathsf{aux}^{(B)})$.

(e) C sends the response

$$\mathsf{Resp}_u = \left\{ \begin{array}{l} I^{(A)}, M^{(A)}, I^{(B)}, M^{(B)}, CS^{(A)}, \\ CS^{(B)}, TG^{(A)}, TG^{(B)}, \Lambda_{I^{(A)}}, \Lambda_{I^{(B)}} \end{array} \right\}$$

to O.

(f) Upon receiving the response Resp_u, O sets a re-upload set $RU = \emptyset$.

(g) O conducts $\mathsf{BVC.BatchVer}_{\mathsf{pp}}(C^{(A)}, M^{(A)}, I^{(A)}, \Lambda_{I^{(A)}})$ and $\mathsf{BVC.BatchVer}_{\mathsf{pp}}(C^{(B)}, M^{(B)}, I^{(B)}, \Lambda_{I^{(B)}})$ to check whether the opened message sets $M^{(A)}$ and $M^{(B)}$ are valid. If either of them is invalid, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(h) O checks whether the following equation holds:

$$\{TG^{(A)} \subseteq \mathsf{Tag} \wedge TG^{(B)} \subseteq \mathsf{Tag} \wedge TG^{(A)} \cup TG^{(B)} = \mathsf{Tag}\} = 1 \quad (19)$$

If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(i) For each $\mathsf{tag}_{F_i} \in TG^{(A)} \cap TG^{(B)}$:

(i) O computes index sets $I^{(A)}_{F_i} = \{h_1(\mathsf{tag}_{F_i}), \cdots, h_k(\mathsf{tag}_{F_i})\}$ and $I^{(B)}_{F_i} = \{f_1(\mathsf{tag}_{F_i}), f_2(\mathsf{tag}_{F_i})\}$. Next, it checks whether the following equation holds:

$$\{I^{(A)}_{F_i} \subseteq I^{(A)} \wedge I^{(B)}_{F_i} \subseteq I^{(B)}\} = 1 \quad (20)$$

If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(ii) O checks whether Equation 17 holds, where $P \in CS^{(T)}$. If so, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(iii) O renames the file F_i and adds the file F_i to RU.

(j) For each $T \in \{A, B\}$ and $\mathsf{tag}_{F_i} \in TG^{(T)} - TG^{(A)} \cap TG^{(B)}$:

(i) O computes index sets $I^{(A)}_{F_i} = \{h_1(\mathsf{tag}_{F_i}), \cdots, h_k(\mathsf{tag}_{F_i})\}$ and $I^{(B)}_{F_i} = \{f_1(\mathsf{tag}_{F_i}), f_2(\mathsf{tag}_{F_i})\}$.

(ii) is similar to Step 2(f)ii.

(iii) is similar to Step 2(f)iii.

(iv) is similar to Step 2(f)iv.

(v) is similar to Step 2h.

(vi) O deletes the local file F_i, keeps the file name name_{F_i} locally, and sends $\mathsf{tag}_{F_i}, C_{F_i}, \mathsf{MAC}_{F_i}$ to C.
(vii) Upon receiving $\mathsf{tag}_{F_i}, C_{F_i}, \mathsf{MAC}_{F_i}, C$ updates IBF by conducting **Algorithm** 3 on input tag_{F_i} and stores C_{F_i} and MAC_{F_i}.

(k) O repeats the above steps until $RU = \emptyset$.

(l) O updates commitments $C^{(A)}$ and $C^{(B)}$ by conducting $\mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C^{(A)}, \{\{g(\mathsf{tag}_{F_i})\}^k\}_{F_i \in U}, \{I^{(A)}_{F_i}\}_{F_i \in U})$ and $\mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C^{(B)}, \{\{g(\mathsf{tag}_{F_i})\}^2\}_{F_i \in U}, \{I^{(B)}_{F_i}\}_{F_i \in U})$, respectively. Then, O publishes the updated commitments $C^{(A)}$ and $C^{(B)}$.

(4) **Batch Deletion:** When the deleted file set is E, O and C work as follows:

(a) O computes the tag set $\mathsf{Tag} = \{\mathsf{tag}_{F_i} | F_i \in E\}$, where $\mathsf{tag}_{F_i} = \mathcal{H}_1(\mathsf{K}_1 \| \mathsf{name}_{F_i})$. Next, O sends the batch delete request $R_e = \{\text{"delete"}, \mathsf{Tag}\}$ to C.

(b) Upon receiving the request R_e, C sets eight empty sets $I^{(A)}, I^{(B)}, M^{(A)}, M^{(B)}, CS^{(A)}, CS^{(B)}, TG^{(A)}, TG^{(B)}$.

(c) For any $\mathsf{tag}_{F_i} \in \mathsf{Tag}$:

(i) C finds the corresponding pure cell P by performing **Algorithm** 1 on input tag_{F_i} and finds the index ℓ of the pure cell P in the table $T \in \{A, B\}$.
(ii) C appends the index ℓ, $P.\mathsf{hashSum}$, P and tag_{F_i} to the opening position set $I^{(T)}$, the opened message set $M^{(T)}$, the cell set $CS^{(T)}$ and the tag set $TG^{(T)}$, respectively.
(iii) C update IBF by conducting **Algorithm** 2 on input tag_{F_i} $P.\mathsf{count}$ times.

(d) C conducts $\Lambda_{I^{(A)}} \leftarrow \mathsf{BVC.BatchOpen}(I^{(A)}, M^{(A)}, \mathsf{aux}^{(A)})$ and $\Lambda_{I^{(B)}} \leftarrow \mathsf{BVC.BatchOpen}(I^{(B)}, M^{(B)}, \mathsf{aux}^{(B)})$.

(e) C sends the response

$$\mathsf{Resp}_e = \left\{ \begin{array}{l} I^{(A)}, M^{(A)}, I^{(B)}, M^{(B)}, CS^{(A)}, \\ CS^{(B)}, TG^{(A)}, TG^{(B)}, \Lambda_{I^{(A)}}, \Lambda_{I^{(B)}} \end{array} \right\}$$

to O.

(f) Upon receiving the response Resp_e, O conducts $\mathsf{BVC.BatchVer}_{\mathsf{pp}}(C^{(A)}, M^{(A)}, I^{(A)}, \Lambda_{I^{(A)}})$ and $\mathsf{BVC.BatchVer}_{\mathsf{pp}}(C^{(B)}, M^{(B)}, I^{(B)}, \Lambda_{I^{(B)}})$ to check whether the opening messages $M^{(A)}$ and $M^{(B)}$ are valid. If either of them is invalid, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(g) For any $F_i \in E$:

(i) O sets $I^{(A)}_{F_i} = \{h_1(\mathsf{tag}_{F_i}), \cdots, h_k(\mathsf{tag}_{F_i})\}$ and $I^{(B)}_{F_i} = \{f_1(\mathsf{tag}_{F_i}), f_2(\mathsf{tag}_{F_i})\}$.
(ii) O checks whether Equation 16 holds. If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.
(iii) O checks whether Equation 17 holds, where $P \in CS^{(T)}$. If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.

(h) O updates commitments $C^{(A)}$ and $C^{(B)}$ by respectively conducting

$$\mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C^{(A)}, \{I^{(A)}_{F_i}\}_{F_i \in E}, \{\{-\mathsf{count}_{F_i} \cdot g(\mathsf{tag}_{F_i})\}^k\}_{F_i \in E})$$

$$\mathsf{BVC.BatchUpd}_{\mathsf{pp}}(C^{(B)}, \{I^{(B)}_{F_i}\}_{F_i \in E}, \{\{-\mathsf{count}_{F_i} \cdot g(\mathsf{tag}_{F_i})\}^2\}_{F_i \in E}),$$

where count_{F_i} is the counter of F_i. Then, O publishes the updated commitments $C^{(A)}$ and $C^{(B)}$.

(5) **Batch Download:** To download and decrypt the file F_i, O needs to get the counter count_{F_i} of each file F_i to generate the decryption key K_{F_i}. When the deleted file set is D, O and C work as follows:

(a) O computes the tag set $\mathsf{Tag} = \{\mathsf{tag}_{F_i} | F_i \in D\}$, where $\mathsf{tag}_{F_i} = \mathcal{H}_1(\mathsf{K}_1 \| \mathsf{name}_{F_i})$. Next, O sends the batch download request $R_d = \{\text{"download"}, \mathsf{Tag}\}$ to C.

(b) is similar to Step 4b.

(c) is similar to Step 4c.

(d) is similar to Step 4d.

(e) C sends the response

$$\mathsf{Resp}_d = \left\{ \begin{array}{l} I^{(A)}, M^{(A)}, I^{(B)}, M^{(B)}, CS^{(A)}, CS^{(B)}, \\ TG^{(A)}, TG^{(B)}, \Lambda_{I^{(A)}}, \Lambda_{I^{(B)}}, C_{F_i}, \mathsf{MAC}_{F_i} \end{array} \right\}$$

to O.

(f) is similar to Step 4f.

(g) For each $F_i \in D$:

(i) O uses MAC_{F_i} to check the integrity of $C_{F_i} \| P.\mathsf{count}$. Note that $P \in CS^{(T)}$, where $T \in \{A, B\}$. If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.
(ii) O checks whether Equation 16 holds. If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.
(iii) O checks whether Equation 17 holds. If not, O outputs **Deny** and aborts; otherwise, O proceeds to the next step.
(iv) O generates the key $\mathsf{K}_{F_i} = \mathcal{H}_2(\mathsf{K}_2 \| \mathsf{tag}_{F_i} \| \mathsf{count}_{F_i})$ and use K_{F_i} to decrypt F_i where $\mathsf{count}_{F_i} = P.\mathsf{count}$.

Due to space constraints, security analysis associated with security requirements can be found in our full version of the paper, which is available at https://www.openssl.org/.

Remark 3. *In our construction, the counter* count_{F_i} *is used to generate the encryption key* K_{F_i} *of the file* F_i*. The key* K_{F_i} *is totally different during each upload phase despite they have the same name* name_{F_i} *(see Step 2h). Trivially, the counter* count_{F_i} *will be set to 0 when the file* F_i *is deleted. Therefore, it is required to conduct* ***Algorithm*** *2 on input* tag_{F_i} count_{F_i} *times (see Step 4(c)iii). Since O does not maintain any information about* F_i *except the name, O has to first get the counter* count_{F_i} *no matter in which phase except setup. Furthermore, O just keeps the secret key* SK_o *and two latest commitments* $C^{(A)}, C^{(B)}$ *locally. It is not hard to find that the commitment is an element of the group* $\mathcal{G}_1$*. Therefore, the storage cost of the data owner is independent of the number of outsourced files. At the setup phase, the two commitments* $C^{(A)} = C^{(B)} = 1$ *since all items of each cell (including hash sum) are set to* 0.

Remark 4. *Since hash functions* $h_1, \cdots, h_k, f_1, f_2$ *are public, the verifier can generate the index sets* $I^{(A)}_{F_i}$ *and* $I^{(B)}_{F_i}$ *both associated with the file* F_i*. Trivially, Equation 16 guarantees that CSP can not use other positions to cheat the verifier. The property of position binding of vector commitment guarantees that the hash sum* $P.\mathsf{hashSum}$ *can not be forged. The verifier knows* tag_{F_i} *and hash function g, so he can get the correct counter* count_{F_i} *by dividing* $P.\mathsf{hashSum}$ *by* $g(\mathsf{tag}_{F_i})$*. As a result, the verifier can compute* $P.\mathsf{idSum} = \mathsf{count}_{F_i} \cdot \mathsf{tag}_{F_i}$*. Therefore,*

Equation 17 guarantees the integrity of IBF. Namely, the misbehavior of CSP will always be found if he tampers the IBF.

Remark 5. *Due to the special data structure, there must exist at least one corresponding pure cell P in IBF if F_i has already been uploaded. When uploading a new file F_i, there may exist a false positive in IBF. If the false positive occurs, C needs to batch open the commitments $C^{(A)}$ and $C^{(B)}$ at all positions associated with the file F_i to prove this fact (see Step 2(d)ii). The communication complexity of this case is $O(k)$. In other case, C only needs to open the commitment C^T at one single position ℓ (see Step 2(c)ii), where $T \in \{A, B\}$. The communication complexity is $O(1)$. Although the communication overhead $O(k)$ is far greater than the communication overhead $O(1)$, the false positive occurs with a very small probability [29].*

Remark 6. *A straightforward solution to make our proposed scheme suitable for one file's case (i.e., $|U| = |D| = |E| = 1$) is to replace algorithms* BVC.BatchOpen *and* BVC.BatchVer *with algorithms* BVC.Open *and* BVC.Ver *in the batch deletion and download phases. As discussed in Remark 2,* BVC.Open/BVC.Ver *is the special case of* BVC.BatchOpen/BVC.BatchVer. *Therefore, without any change, the above construction can also suit for one file's case.*

4.4 Security Analysis

Theorem 3. *The proposed batch data deletion scheme satisfies the property of correctness.*

Proof. The proof of the correctness of our proposed scheme is trivial. The correctness of our proposed scheme relies on that of BVC and IBF. The correctness of BVC has been proved in Appendix B. The proof of the correctness of IBF scheme can be found in [29]. Therefore, our proposed scheme is correct. □

Theorem 4. *The proposed data deletion scheme satisfies the property of false positive traceability.*

Proof. Trivially, there are two cases breaking the security of false positive traceability. The first one is CSP still claims that there is no false positive despite there exists a false positive for the new upload file F_i. The second one is that CSP still claims that there exists a false positive despite there is no false positive for the file F_i. Note that the returned cell P is null in both of these two cases. For the first case, CSP must forge two opened message sets not including 0. If CSP forges these two sets successfully, it will break the property of batch position binding in batch vector commitment. The property of batch position binding has been proved in Theorem 2. Therefore, it is infeasible to successfully forge these two opened message sets. For the second case, CSP must forge an opened message $P.\mathsf{hashSum} = 0$ at the position $\ell \in \{h_1(\mathsf{tag}_{F_i}), \cdots, h_k(\mathsf{tag}_{F_i}), f_1(\mathsf{tag}_{F_i}), f_2(\mathsf{tag}_{F_i})\}$. If CSP forges such an opened message successfully, it will break the property of position binding in batch vector commitment. The property of position binding has been proved in Theorem 1. Therefore, it is infeasible to successfully forge such an opened message.

From all the above, our proposed batch data deletion achieves false positive traceability. □

Theorem 5. *The proposed batch data deletion scheme satisfies the property of deletion traceability.*

Proof. The proof of Theorem 5 is trivial. As discussed in Remark 4, the commitments $C^{(A)}$ and $C^{(B)}$ sufficiently represent the IBF. Whether a file F_i is uploaded or deleted, the data owner always keeps the latest commitments $C^{(A)}$ and $C^{(B)}$. If CSP does not delete the file and update IBF, it will break the batch position binding of batch vector commitment. Namely, only the witness corresponding to the newest IBF can pass the verification. The batch position binding of our proposed batch vector commitment has been proved in Theorem 2. Therefore, our scheme achieves deletion traceability. □

Theorem 6. *The proposed batch data deletion scheme satisfies the property of existence traceability.*

Proof. If CSP does not admit that the file F_i already exists, CSP has to find a vacant cell P in the IBF, where $P.\mathsf{hashSum} = 0$ and the index of P belongs to $\{h_1(\mathsf{tag}_{F_i}, \cdots, h_k(\mathsf{tag}_{F_i}, f_1(\mathsf{tag}_{F_i}), f_2(\mathsf{tag}_{F_i}\}$. If the file F_i exists, there must be a pure cell P in the IBF, where $P.\mathsf{hashSum} \neq 0$. As discussed in Theorem 5, the data owner always keep the latest commitments $C^{(A)}$ and $C^{(B)}$, which sufficiently represent the latest IBF. Due to the property of position binding in batch vector commitment, it cannot break the property of existence traceability. □

Theorem 7. *The proposed batch data deletion scheme achieves version control.*

Proof. If CSP returns a previous version of file F_i when downloading the file F_i, CSP must return its corresponding pure cell P under the previous version. Otherwise, the misbehavior of CSP will be detected by using message authenticated code $\mathsf{MAC}_{F_i} = \mathrm{HMAC}(\mathsf{K}_{mac}, C_{F_i} \| \mathsf{count}_{F_i})$, where C_{F_i} is the ciphertext of F_i and $\mathsf{count}_{F_i} = P.\mathsf{count}$. As discussed in Theorem 5, the data owner always keep the latest commitments $C^{(A)}$ and $C^{(B)}$, which sufficiently represent the latest IBF. Due to the property of position binding in batch vector commitment, the data owner can always get the latest hash sum $P.\mathsf{hashSum}$. The tag tag_{F_i} can be generated by the data owner, so the he can get the newest count count_{F_i} by dividing $P.\mathsf{hashSum}$ with tag_{F_i}. Therefore, our scheme achieves version control. □

5 Evaluation

In this section, we first theoretically compare our proposed scheme (We call it as PVBDC) with *the first and only one* publicly verifiable data deletion scheme without any TTP [17] (we call it as BPVDD) in two folds: the desired property and the computation complexity. Then, we provide a prototypal implementation of our proposed scheme.

5.1 Desired Properties

Table 1 summarizes the comparison of the desired properties. Both of these two schemes are under the amortized model [33]. Namely, they both allow a one-time expensive computational overhead in the setup phase. BPVDD is not a two-party protocol since it additionally requires the honest timestamp server to check the validity of the Merkle tree generated by CSP. It is very impractical in the real world since the timestamp server has to check the validity of each deletion record and regenerate the root of the Merkle tree.

Usually, the timestamp server does not have any computational ability. BPVDD requires the data owner to assign a unique name for each file. When the uploaded file is modified, the data owner needs to assign a new name. Therefore, it does not support overwrite and version control. However, the file may be overwritten many times and the file name cannot be modified in the real-life application. Our proposed scheme generates the HMAC taking as input the version number. Due to the position binding of batch vector commitment, the data owner can always get the newest version number. As a result, the data owner can detect the misbehavior if CSP returns a previous version file. Meanwhile, BPVDD cannot support batch operations while our proposed scheme can. Whether in BPVDD or PVBDC, the storage overhead of the verifier is all independent of the number of files. There is no mechanism dealing with the conflict that the data owner has already outsourced their file to CSP but CSP does not admit this fact. Our proposed scheme can resolve this dispute since the commitment is updated by the honest data owner in the upload phase (see Steps 2i and 3l). BPVDD takes advantage of the Merkle tree to periodicity record all deletion operations. BPVDD is more efficient than our proposed scheme since it just needs to conduct some hash operations. However, it is not suitable for the network with narrow bandwidth since the communication cost proportional to the logarithm of the number of deletion operations in the time interval. However, the communication complexity of the verifier is constant in our proposed scheme since the proof is one single element of the group $\mathcal{G}_1$. Sometimes, the communication overhead might be even more important than the computation overhead since it is often easier to add computation power than to improve the outgoing communication.

Table 1: Comparison of Desired Properties

Property	BPVDD [17]	PVBDC
Amortized Model	√	√
Two-party Protocol	×	√
Public Verifiability	√	√
Overwrite	×	√
Version Control	×	√
Batch Operations	×	√
Existence Traceability	×	√
Constant Storage Cost on Verifier	√	√
Constant Communication Cost on Verifier	×	√

5.2 Computation Complexity

Since the computation cost is mainly dominated by bilinear pairing, exponentiation, and multiplication over the group, we evaluate it by counting the number of such operations. Although the time cost of hashing is very small, there are so many hashing operations in BPVDD scheme. So, we take it as an evaluation index as well. Table 2 summarizes the comparison of the computation cost between BPVDD and PVBDC. In Table 2, we denote by Pairing the time cost of bilinear pairing, by Exp the time cost of exponentiation over group $\mathcal{G}_1$ or $\mathcal{G}_2$, by Mul the time cost of multiplication over group $\mathcal{G}_1$ or $\mathcal{G}_2$, by H the time cost of hashing, by N the number of all deletion records in the time interval, and by m_T the number of vacant cells in the hash table $T \in \{A, B\}$. For simplicity, we set $|U| = |D| = |E| = p = p_A + p_B$, where p_A and p_B are respectively the number of opened positions in the hash table A and B. The

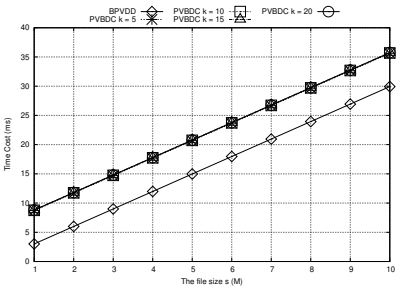


Figure 3: Comparison of Data Owner's Computation Cost in Upload Phase

BPVDD protocol [17] is more efficient than our proposed scheme, but it does not satisfy the requirements mentioned above. As discussed in Section 4.3, the data owner in our scheme needs to first get the true counter no matter in which phase. Take as an example the upload phase, to achieve this, the data owner has to carry out $(k+2)$ hashing operations in Step 2e, $(k+2)$ multiplications and 2 exponentiations in Step 2i, and 1 multiplication, 1 exponentiation and 3 pairing operations in Step 2(f)i. In the batch upload phase, the batch verification algorithm will be carried out twice in Step 3g if the opened position sets $I^{(B)}$ is not empty. Therefore, the data owner will perform at most 6 pairings. In general, the data owner only needs to conduct 3 pairing operations since there always exists a pure cell associated with each file in the hash table A. It is easy to find that the batch verification algorithm costs less effort than performing the verification algorithm one by one. Therefore, it saves a lot of time. The process of the data owner in the deletion/batch deletion phase is similar to that of the data owner in the upload/batch upload phase, since the data owner both needs to get the counter and update the commitments $C^{(A)}$ and $C^{(B)}$ in these two phases. Updating the commitments $C^{(A)}$ and $C^{(B)}$ requires the data owner to conduct $p(k+2)$ hashing operations, $p(k+2)-2$ multiplications and $2p$ exponentiations. The process of the data owner in the download phase is different from that of the data owner in the upload or deletion phase, since the commitments $C^{(A)}$ and $C^{(B)}$ stay the same. No matter in which phase, CSP all needs to find the pure cell and open the commitment before the data owner gets the counter. It is worth noting that the computation complexity of the data owner in each phase is not influenced by the hash table size n while the computation complexity of CSP does.

5.3 Simulation

In this section, we provide a thorough experimental evaluation of our proposed scheme. To precisely evaluate the computation cost at the data owner, CSP sides, all simulations were conducted on Linux with Inter(R) Core(TM) i7-8550U CPU @ 1.80GHZ processor and 8GB memory. The cryptographic algorithms are implemented using PBC library[3] on type A with a 160-bit group order. The cryptographic hash function and encryption algorithm are respectively implemented with SHA-256 and the 128-bit AES algorithm with the cipher block chaining (CBC) mode in OpenSSL[4].

[3]Available: https://crypto.stanford.edu/pbc/.
[4]Available: https://www.openssl.org/.

Table 2: Comparison of Computation Cost

Phase		BPVDD [17]	PVBDC
Upload	Data Owner	1·Enc+3·H	1·Enc+$(k+5)$·H+$(k+1)$·Mul+3·Exp+3·Pairing
	CSP	N/A	$(k+2)$·H+$(m-m_T-2)$·Mul+$(m-m_T-1)$·Exp
Batch Upload	Data Owner	N/A	p·Enc+$p(k+5)$·H+$(pk+4p-4)$·Mul+3p·Exp+3\6·Pairing
	CSP	N/A	$p(k+2)$·H+$p_A(m-m_A-1)$·Mul+$p_A(m-m_A-1)$·Exp+$p_B(m-m_B-1)$·Mul+$p_B(m-m_B-1)$·Exp
Deletion	Data Owner	$(\log N+2)$·H	$(k+3)$·H+$(k+1)$·Mul+3·Exp+3·Pairing
	CSP	$(2N-1)$·H	$(k+2)$·H+$(m-m_T-2)$·Mul+$(m-m_T-1)$·Exp
Batch Deletion	Data Owner	N/A	$p(k+3)$·H+$(pk+4p-4)$·Mul+3p·Exp+3\6·Pairing
	CSP	N/A	$p(k+2)$·H+$p_A(m-m_A-2)$·Mul+$p_A(m-m_A-1)$·Exp+$p_B(m-m_B-2)$·Mul+$p_B(m-m_B-1)$·Exp
Download	Data Owner	1·Dec+3·H	1·Dec+$(k+5)$·H+1·Mul+1·Exp+3·Pairing
	CSP	N/A	$(k+2)$·H+$(m-m_T-2)$·Mul+$(m-m_T-1)$·Exp
Batch Download	Data Owner	N/A	p·Dec+$p(k+5)$·H+$(2p-2)$·Mul+p·Exp+3\6·Pairing
	CSP	N/A	$p(k+2)$·H+$p_A(m-m_A-2)$·Mul+$p_A(m-m_A-1)$·Exp+$p_B(m-m_B-2)$·Mul+$p_B(m-m_B-1)$·Exp

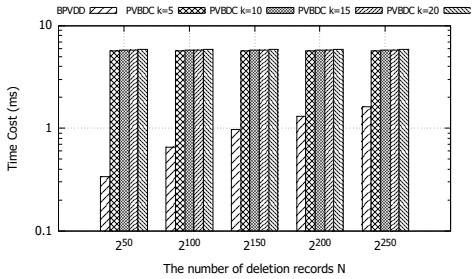


Figure 4: Comparison of Data Owner's Computation Cost in Deletion Phase

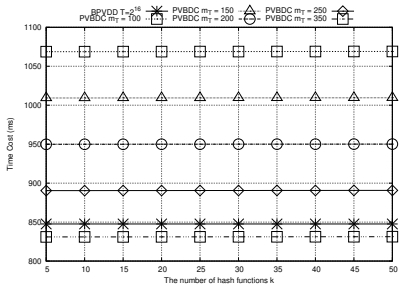
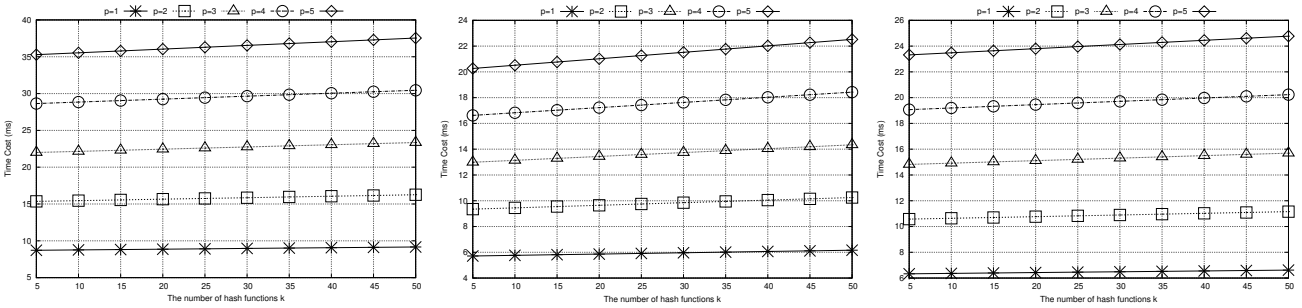


Figure 5: Comparison of CSP's Computation Cost in Deletion Phase

The hash table parameters affect the performance of our proposed scheme. In our experiment, we fix the number of cells $m = 1000$ and range the number of hash functions k between 5 and 50. Figure 3 depicts the comparison of the data owner's computation cost in the upload phase. The file size s varies from 1M to 10M. As can be seen from Figure 3, the computation cost is almost independent of the number of hash functions and linear to the file size. The running time of the data owner in BPVDD is the most efficient, but it is almost less than 5ms than others. Figure 4 shows the comparison of the data owner's running time in the deletion phase. The computation cost of the data owner in our scheme is not affected by the number of the deletion records N, while proportional to the logarithm of the number of deletion records to the number of the deletion records in BPVDD. Although BPVDD is relatively more efficient than our scheme, our scheme is still efficient since the time cost (just 7ms) is acceptable. Figure 5 depicts the comparison of the CSP's computation cost in the deletion phase. We fix the file size s = 1M and range the number of vacant cells m_T from 100 to 350. The running time is almost independent of the number of hash functions, as the hash function costs little overhead. When the number of deletion records reaches to 2^{16} in the time interval, the time cost of BPVDD is greater than that of PVBDC where $m_T = 350$. Therefore, BPVDD will cost the biggest amount of time when the number of deletion records is so large. Besides, the CSP's computation complexity in our proposed scheme decreases linearly to the number of vacant cells m_T in the hash table $T \in \{A, B\}$. In our scheme, the time cost of CSP in the deletion and download phase is the same as that in the upload phase. The time cost of CSP in the BPVDD scheme is negligible since the CSP just needs to add or delete the link. Hence, we omit the computation comparisons of CSP in the deletion and download phases. Figure 6 gives a comparison of time cost on the data owner's computation in different phases. We set the file size s = 1M. The execution time in the deletion phase is less than that in the upload phase when p is the same since the data owner does not conduct the encryption algorithm. Similarly, the time cost in the download phase is less than that in the deletion phase, since the data owner does not need to update the commitments $C^(A)$ and $C^(B)$. Figure 6 presents the time cost of the data owner's computation in different batch operation phases. It is noted that the execution time when $p = 1$ is almost half of that when $p = 2$ in each phase while relatively more than 1 over p of that when $p > 2$. Figure 7 presents a comparison of the CSP's time cost in the batch upload/deletion/download phase. We fix $m_B = 500$ and range m_A between 50 and 250. The running time decreases linearly to the number of vacant cells in the table A of IBF.

6 Conclusion

To enhance the flexibility of secure data deletion in cloud storage, we first proposed a novel primitive called batch vector commitment as the building block, which can open the commitment in a batch manner. Based on batch vector commitment, we constructed the first one two-party secure data deletion without any trusted third party. Compared with it, our scheme additionally supports version control, batch operations, constant communication complexity, and existence traceability. Finally, we provided detailed performance analyses, simulations and security proofs for our proposed scheme.



(a) Batch Upload (b) Batch Deletion (c) Batch Download

Figure 6: Comparison of Data Owner's Computation

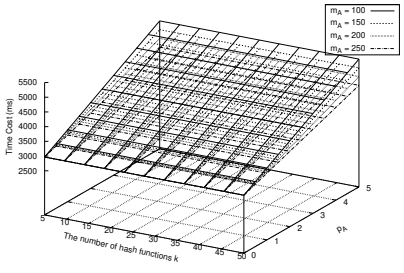


Figure 7: Time Cost of CSP's Computation in Batch Upload/Deletion/Download Phase

References

[1] Steven Bauer and Nissanka Bodhi Priyantha. 2001. Secure Data Deletion for Linux File Systems.. In *Usenix Security Symposium*, Vol. 174.

[2] Joel Reardon, David Basin, and Srdjan Capkun. 2013. Sok: Secure data deletion. In *2013 IEEE symposium on security and privacy*. IEEE, 301–315.

[3] Jaeheung Lee, Sangho Yi, Junyoung Heo, Hyungbae Park, Sung Y Shin, and Yookun Cho. 2010. An Efficient Secure Deletion Scheme for Flash File Systems. *J. Inf. Sci. Eng.* 26, 1 (2010), 27–38.

[4] Daniele Perito and Gene Tsudik. 2010. Secure code update for embedded devices via proofs of secure erasure. In *European Symposium on Research in Computer Security*. Springer, 643–662.

[5] Joel Reardon, Srdjan Capkun, and David Basin. 2012. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 17–17.

[6] Joel Reardon, Hubert Ritzdorf, David Basin, and Srdjan Capkun. 2013. Secure data deletion from persistent media. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 271–284.

[7] Feng Hao, Dylan Clarke, and Avelino Francisco Zorzo. 2015. Deleting secret data with public verifiability. *IEEE Transactions on Dependable and Secure Computing* 13, 6 (2015), 617–629.

[8] Sarah M Diesburg and An-I Andy Wang. 2010. A survey of confidential data storage and deletion methods. *ACM Computing Surveys (CSUR)* 43, 1 (2010), 2.

[9] Yong Yu, Liang Xue, Yannan Li, Xiaojiang Du, Mohsen Guizani, and Bo Yang. 2018. Assured data deletion with fine-grained access control for fog-based industrial applications. *IEEE Transactions on Industrial Informatics* 14, 10 (2018), 4538–4547.

[10] Liang Xue, Yong Yu, Yannan Li, Man Ho Au, Xiaojiang Du, and Bo Yang. 2019. Efficient attribute-based encryption with attribute revocation for assured data deletion. *Information Sciences* 479 (2019), 640–650.

[11] Dan Boneh and Richard J Lipton. 1996. A Revocable Backup System.. In *USENIX Security Symposium*. 91–96.

[12] BBC-NEW. [n. d.]. The interview: a guide to the cyber attack on hollywood. https://www.bbc.com/news/entertainment-arts-30512032. accessed September 20, 2019.

[13] Qiang Wang, Fucai Zhou, Su Peng, and Zifeng Xu. 2018. Verifiable Outsourced Computation with Full Delegation. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 270–287.

[14] Claudio A Ardagna, Rasool Asal, Ernesto Damiani, and Quang Hieu Vu. 2015. From security to assurance in the cloud: A survey. *ACM Computing Surveys (CSUR)* 48, 1 (2015), 2.

[15] Xixun Yu, Zheng Yan, and Athanasios V Vasilakos. 2017. A survey of verifiable computation. *Mobile Networks and Applications* 22, 3 (2017), 438–453.

[16] Qiang Wang, Fucai Zhou, Chunyu Chen, Pengkai Xuan, and Qiyu Wu. 2017. Secure collaborative publicly verifiable computation. *IEEE Access* 5 (2017), 2479–2488.

[17] Changsong Yang, Xiaofeng Chen, and Yang Xiang. 2018. Blockchain-based publicly verifiable data deletion scheme for cloud storage. *Journal of Network and Computer Applications* 103 (2018), 185–193.

[18] Changsong Yang, Xiaoling Tao, Feng Zhao, and Yong Wang. 2019. A New Outsourced Data Deletion Scheme with Public Verifiability. In *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 631–638.

[19] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. 2009. Vanish: Increasing Data Privacy with Self-Destructing Data.. In *USENIX Security Symposium*, Vol. 316.

[20] Yang Tang, Patrick PC Lee, John CS Lui, and Radia Perlman. 2012. Secure overlay cloud storage with access control and assured deletion. *IEEE Transactions on dependable and secure computing* 9, 6 (2012), 903–916.

[21] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. 2009. Enabling public verifiability and data dynamics for storage security in cloud computing. In *European symposium on research in computer security*. Springer, 355–370.

[22] Simson L Garfinkel and Abhi Shelat. 2003. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security & Privacy* 1, 1 (2003), 17–27.

[23] Yuchuan Luo, Ming Xu, Shaojing Fu, and Dongsheng Wang. 2016. Enabling assured deletion in the cloud storage by overwriting. In *Proceedings of the 4th ACM International Workshop on Security in Cloud Computing*. ACM, 17–23.

[24] Craig Wright, Dave Kleiman, and Shyaam Sundhar RS. 2008. Overwriting hard drive data: The great wiping controversy. In *International Conference on Information Systems Security*. Springer, 243–257.

[25] Wei Huang, Afshar Ganjali, Beom Heyn Kim, Sukwon Oh, and David Lie. 2015. The state of public infrastructure-as-a-service cloud security. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 68.

[26] Eun-Jun Yoon, Kee-Young Yoo, Jeong-Woo Hong, Sang-Yoon Yoon, Dong-In Park, and Myung-Jin Choi. 2011. An efficient and secure anonymous authentication scheme for mobile satellite communication systems. *EURASIP Journal on Wireless Communications and Networking* 2011, 1 (2011), 86.

[27] Dan Boneh and Matt Franklin. 2001. Identity-based encryption from the Weil pairing. In *Annual international cryptology conference*. Springer, 213–229.

[28] Lan Nguyen. 2005. Accumulators from bilinear pairings and applications. In *Cryptographers' Track at the RSA Conference*. Springer, 275–292.

[29] David Eppstein and Michael T Goodrich. 2010. Straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters. *IEEE Transactions on Knowledge and Data Engineering* 23, 2 (2010), 297–306.

[30] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting bloom filters. In *European Symposium on Algorithms*. Springer, 684–695.

[31] Dario Catalano and Dario Fiore. 2013. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*. Springer, 55–72.

[32] Johannes Krupp, Dominique Schröder, Mark Simkin, Dario Fiore, Giuseppe Ateniese, and Stefan Nuernberger. 2016. Nearly optimal verifiable data streaming. In *Public-Key Cryptography–PKC 2016*. Springer, 417–445.

[33] Rosario Gennaro, Craig Gentry, and Bryan Parno. 2010. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Annual Cryptology Conference*. Springer, 465–482.

A Notations

Table 3: Notations

Notations	Descriptions
A/B	The first/second hash table in IBF
m	Number of cells in hash tables A and B
q	Size of the committed vector
pp	Public parameters
C	Commitment value
aux	Auxiliary information
m_i	The i-th message of the sequence $(m_1, \cdots, m_q)$
Λ_i	Proof that m_i is the i-th committed message
C'	Updated commitment
U	Updated information
Λ'_j	Updated proof for m_j
I	Opened position set
M	Opened message set associated with I
Λ_I	Proof that M is the opened message set
SU	Update information set
$\mathsf{K}_1/\mathsf{K}_2$	Key of hash function $\mathcal{H}_1/\mathcal{H}_2$
K_{mac}	Key of message authentication code
F_i	The i-th file
C_{F_i}	Ciphertext of the file F_i
count_{F_i}	Counter of the file F_i
name_{F_i}	Name of the file F_i
tag_{F_i}	Tag of the file F_i
K_{F_i}	Encryption key of the file F_i
$D/E/U$	Batch download/deletion/upload file set
$R_d/R_e/R_u$	Download/Deletion/Upload request
$C^{(A)}/C^{(B)}$	Commitment on the hash sum row of table A/B
$M^{(A)}/M^{(B)}$	Opened message set associated with table A/B
$\mathsf{aux}^{(A)}$ / $\mathsf{aux}^{(B)}$	Auxiliary information associated with table A/B
$\Lambda_{I^{(A)}}/\Lambda_{I^{(A)}}$	Proof that $M^{(A)}/M^{(B)}$ is the opened message set

B Proof of Correctness

In the following, we will show the correctness of our construction mainly based on Equations 8 and 3. According to Equation 7, the right hand side of Equation 8 can be expressed as:

$$\begin{aligned} e\left(\mathcal{I}, g^{\alpha^q}\right) \cdot e\left(\Lambda_I, g\right) &= \prod_{j=1}^{k} e\left(g^{m_{i_j} \cdot \beta^{i_j}}, g^{\alpha^q}\right) \cdot e\left(\Lambda_I, g\right) \\ &= \prod_{j=1}^{k} e\left(g^{m_{i_j} \cdot \beta^{i_j}}, g^{\alpha^q}\right) \cdot e\left(g^{Q(\alpha,\beta)}, g\right) \\ &= e(g,g)^{\sum_{j=1}^{k} m_{i_j} \cdot \beta^{i_j} \cdot \alpha^q} \cdot e\left(g^{Q(\alpha,\beta)}, g\right) \\ &= e(g,g)^{\sum_{j=1}^{k} m_{i_j} \cdot \beta^{i_j} \cdot \alpha^q + Q(\alpha,\beta)} \\ &= e\left(g^{\mathcal{R}(\alpha)}, g^{\mathcal{S}(\alpha,\beta)}\right) \\ &= e\left(g^{\mathcal{R}(\alpha)}, g^{\sum_{j=1}^{k} \alpha^{q-i_j} \cdot \beta^{i_j}}\right) \\ &= \prod_{j=1}^{k} e\left(C, g^{\alpha^{q-i_j} \beta^{i_j}}\right) \end{aligned} \tag{21}$$

As discussed in Remark 2, the verification algorithm BVC.Ver is the special case of the batch verification algorithm BVC.BatchVer when $|I| = 1$. Hence, Equation 3 will hold as well. Therefore, if the data owner and CSP follow all procedures described above honestly, the opened messages and the proofs will always pass the verifier's check.