Efficient Secure Storage with Version Control and Key Rotation

Long Chen[1], Hui Guo[2], Ya-Nan Li[3], and Qiang Tang[3]

[1] Institute of Software Chinese Academy of Sciences, Beijing, China
`chenlong@iscas.ac.cn`
[2] The State Key Laboratory of Cryptology, Beijing, China
`guohtech@foxmail.com`
[3] The University of Sydney, Sydney, Australia
`{yanan.li,qiang.tang}@sydney.edu.au`

Abstract. Periodic key rotation is a widely used technique to enhance key compromise resilience. Updatable encryption (UE) schemes provide an efficient approach to key rotation, ensuring post-compromise security for both confidentiality and integrity. However, these UE techniques cannot be directly applied to frequently updated databases due to the risk of a malicious server inducing the client to accept an outdated version of a file instead of the latest one.
To address this issue, we propose a scheme called Updatable Secure Storage (USS), which provides a secure and key updatable solution for dynamic databases. USS ensures both data confidentiality and integrity, even in the presence of key compromises. By using efficient key rotation and file update procedures, the communication costs of these operations are independent of the size of the database. This makes USS particularly well-suited for managing large and frequently updated databases with secure version control. Unlike existing UE schemes, the integrity provided by USS holds even when the server learns the current secret key and intentionally violates the key update protocol.

Keywords: Vector commitment · Updatable encryption · Cloud storage.

1 Introduction

An increasing number of companies, government bodies, and personal users are choosing to store their data on the cloud instead of local devices. However, as a public infrastructure, frequent data breaches from the cloud have been reported. One potential mitigation strategy is to allow users to upload encrypted data and keep the decryption key locally. However, even with encryption mechanisms in place, there is still a risk that users' decryption keys may become compromised over time.

To address this issue, it is widely acknowledged and implemented in the industry to periodically refresh the secret key used to protect the data and to update the corresponding ciphertext in the cloud. For example, the Payment

Card Industry Data Security Standard (PCI DSS)[6,13] requires credit card data to be stored in encrypted form and mandates key rotation, whereby encrypted data is regularly refreshed from an old to a newly generated key. This strategy has also been adopted by many cloud storage providers, such as Google and Amazon [12]. By regularly refreshing encryption keys, the risk of data compromise can be significantly reduced. This approach ensures that even if a decryption key is compromised, it will only affect a limited amount of data that was encrypted with that key. Furthermore, this strategy is relatively easy to implement and can be automated, making it an effective way to improve cloud security.

While standardized encryption tools are available, facilitating key rotation requires careful consideration. A naive solution is to have the client download all encrypted data, decrypt it, choose a new key, encrypt the data, and upload the new ciphertext to the cloud server. However, this approach is inefficient, especially for large amounts of data. To address this issue, Boneh et al. [4] proposed a new primitive called updatable encryption (UE) for efficiently updating ciphertexts with a new key. Everspaugh et al [12] gave a systematic study of Updatable Authenticated Encryption (UAE), especially on the key rotation on *authenticated encryption*, which is the standard practice for encryption. Standard UAE constructions can guarantee the confidentiality and integrity of the plaintext. With UAE, a client only needs to retrieve at most a short piece of information (known as the header) and generate a short update token that enables the server to re-encrypt the data from the existing ciphertext while preserving encryption security.

UAE constructions [17,15,5,9,3] are particularly appealing due to their ability to provide post-compromise security. This ensures that outsourced storage can regain its security, even in the event of a temporary client hack, as long as the system executes the update process by updating both the secret key and ciphertext. Notably, after re-encryption, adversaries cannot determine whether the data has been modified, even if they have seen both the old key and the previous version of the ciphertext.

Integrity vs. frequent data update. In numerous real-world applications, such as E-commerce websites, social media platforms, financial institutions, and logistics companies, users require dynamic databases that can accommodate real-time changes in data. For instance, E-commerce websites need to manage inventory in real-time as products are added, sold, or restocked. Social media platforms must store and update user-generated content, such as posts, comments, and likes, in real-time. Financial institutions require the processing and storage of large volumes of transactional data in real-time, such as stock trades or credit card transactions. Logistics companies need to track and manage shipments in real-time as they move through the supply chain. These databases must be capable of handling continuous updates and modifications. To handle large volumes of data with varying attributes, such databases must be designed to facilitate fast data retrieval and frequent data updates.

However, despite the existence of several proposed constructions for Updatable Encryption (UE) in the literature [12,15,5,9,3], these schemes mainly focus on ensuring the confidentiality and integrity of static databases and cannot be applied directly to dynamic databases. If the client encrypts each file using traditional UE before uploading it to the server, this approach fails to ensure data integrity if the client performs data updates on the database. The main issue with this approach is that the client needs a mechanism to revoke the previous UE ciphertexts associated with outdated data stored by the untrusted server. Otherwise, the server may provide the client with an obsolete version of the file instead of the latest one. Although the client could keep track of every change locally, this contradicts the primary objective of utilizing fewer resources compared to storing the entire database locally.

A promising approach for tracking changes of all files in a database is to use vector commitment (VC), a powerful primitive proposed by Catalano and Fiore [8]. VCs enable the commitment of an ordered sequence of n values $(m_1, \ldots, m_n)$ into a concise commitment while allowing for later opening of the commitment at specific positions with a membership proof to prove that m_i is the i-th committed message. To ensure security, VCs must satisfy the position-binding property, which requires that an adversary cannot open a commitment to two different values at the same position. The size of the commitment and each opening must be independent of the vector degree. To guarantee the integrity of a dynamic database, each file could be treated as an element of the vector.

The vector commitment property, especially its support for element updates, plays a crucial role in ensuring the integrity of a dynamic database. VC has two algorithms to update the commitment and corresponding openings. The first algorithm updates a commitment Com by changing the i-th message from m_i to m'_i, and results in a modified Com' containing the updated message. The second algorithm updates an opening for a message at position j with respect to Com to a new opening with respect to the new Com'. Indeed, Catalano and Fiore [8] have shown that the verifiable database with efficient updates (VDB) [1] can be constructed from the VC scheme.

Key rotation for a verifiable database. Although VC can address the integrity problem of dynamic databases, its compatibility with encryption schemes featuring key rotation is not straightforward. Specifically, applying VC to commit UE ciphertexts as vector elements may result in linear communication costs with the entire storage during each key update. This is because updating the VC content requires linear communication with the updated ciphertext, which constitutes the entire content of the user's encrypted storage.

An alternative approach to reducing communication costs is to apply VC and UE directly to the plaintext, similar to the Enc-and-Mac combination of AE. In this approach, users store the UE ciphertext and VC membership proof of each file on the server while keeping the commitment locally as metadata for integrity checks. However, this construction fails to satisfy the confidentiality requirement if using a general VC without position hiding property, as the vector commitment and membership proofs could potentially leak information about the plaintexts. Although this information leakage can be avoided by committing each file first and running VC on the commitment, it is not sufficient to achieve post-compromise security. Since the membership proofs are not updated during key rotation, an adversary may be able to learn the updated pattern of the files in each vector element. This information leakage can further reveal whether each file has been updated after the epoch has evolved, which is a critical concern for post-compromise security. Therefore, it is important to hide the information of the membership proofs, as well as the plaintext, when considering post-compromise security.

To address the information leakage of VC proofs, one possible suggestion is to encrypt the VC proofs using UE schemes as well. However, this means that updating one file would require updating all other VC proofs in the UE construction. One straightforward solution would be to retrieve all ciphertexts, decrypt them, update their contents, re-encrypt them, and then upload them. However, this approach incurs linear communication costs for each file update in relation to the entire storage. To mitigate the information leakage of the vector commitment, a desirable solution would be to have re-randomizable vector commitment that supports periodic re-randomization.

To tackle these challenges, new encryption with key rotation and vector commitment techniques are needed that can adapt to the evolving needs of dynamic databases. In a nutshell, the following question arises:

Is there an efficient method that enables the highest levels of confidentiality and integrity in a frequently updated database, while minimizing communication overhead?

1.1 Our contributions

This paper introduces a novel primitive called updatable secure storage (short for USS), which provides a secure solution for dynamic databases with version control. The USS scheme ensures both data confidentiality and integrity, even in the event of key compromises. By using efficient key rotation and file update procedures, the communication costs of these operations are independent of the size of the database. This makes the USS scheme particularly well-suited for managing large and frequently updated databases in a secure and efficient manner. The USS scheme is built on the KEM + DEM paradigm, where the DEM part can be any UE ciphertext with IND-CPA security. This allows the USS scheme to benefit from the efficiency of existing UE schemes while also providing strong security guarantees against attacks on data confidentiality and integrity. Overall, the USS scheme provides an effective solution for secure database management in dynamic environments, where frequent data updates are necessary.

Confidentiality in the event of key leakage. The USS scheme is designed to ensure basic content confidentiality even in the event of temporary key leakage or storage breach, as long as the server conducts the key rotation process in an honest manner. This process of key rotation is an essential aspect of the storage system, serving to limit the amount of data that could be compromised in the event of a key breach. More precisely, USS can guarantee the confidentiality of the data unless the attacker can learn the key and the ciphertexts in the same epoch. In more precise terms, the USS scheme can guarantee data confidentiality, unless the attacker can learn both the key and the ciphertexts within the same epoch. Hence USS scheme can effectively mitigate the impact of key breaches, as it reduces the window of opportunity for attackers to gain access to both pieces of information simultaneously.

Integrity for dynamic databases. The USS scheme provides strong integrity guarantees in dynamic databases, where a malicious server may attempt to deceive the client by providing an outdated version of data. More precisely, the strong integrity allows the server to be fully malicious and may not follow the protocol to behave most of the time. In contrast, existing UE schemes, such as those proposed in [12,17,15,5,9,3], assume that the server will honestly proceed with the key rotation procedure. However, this assumption is unrealistic in many scenarios, and it limits the ability of UE schemes to provide comprehensive protection against attacks on data integrity. Furthermore, while UE schemes exclude the case where an adversary forges ciphertexts after learning the current secret key, the USS scheme can guarantee data integrity even if the secret key is leaked. This is a significant advantage of the USS scheme, which offers stronger protection against a wider range of attacks.

Post-compromise security. The USS scheme offers post-compromise security for confidentiality. Specifically, if an adversary compromises both the secret key and storage in some epoch, they cannot gain any advantage in decrypting ciphertexts obtained in epochs after the compromisation. To capture the notion of post-compromise security, we introduce a security game called key update unlinkability. This game requires that attackers cannot distinguish whether an updated ciphertext is key updated from a previously corrupted ciphertext. Additionally, the USS scheme provides file update unlinkability, which guarantees that attackers who corrupt storage before and after a file update operation learn nothing about the update itself, such as whether the file content has changed and what the current content is. These security notions are essential for protecting sensitive data in scenarios where confidentiality is of utmost importance, such as in healthcare, finance, and government applications.

1.2 Technique overview

Intuitively, USS can be regarded as a secure version of Github that provides secure outsourced storage and version control services even when the server is not fully trusted. USS enables users to create remote repositories on the server while keeping a secret key and a public stub on the client side. Its primary goal is to provide the best possible security under the key compromise. To this end, USS employs a periodic key rotation mechanism similar to UE schemes, which prevents an adversary from learning stored data even with a leaked key.

Unlike UE schemes, the integrity of USS relies on the public stub of the repository rather than the secret key. As long as the stub is correctly kept by the user, the server cannot deceive the user. Keeping a public stub is much easier than secretly keeping a key on the client side. Furthermore, the stub changes if any file in the repository gets updated, which makes it convenient for users to track the versions of the entire repository. Even if the server is malicious, it cannot force the client to accept an old version of a file instead of the latest one.

One possible solution is to use a vector commitment to commit to all files in the repository. The commitment value, which is the stub stored on the client side, can be updated efficiently using the vector commitment whenever there are changes to the files [8]. However, this approach raises the concern that the commitment value itself may reveal information about the repository. Another approach is to encrypt the files during updates and use a vector commitment to commit to the resulting ciphertexts. However, this approach faces the challenge that the ciphertexts may change during key updates, causing the commitment value to update accordingly. Since the new ciphertexts are computed on the server side, the client cannot update the stub locally.

An alternative solution is to use a classic commitment to commit to each file and then use a vector commitment to commit to each classic commitment value. The content of each file can then be encrypted using the updated encryption. However, this approach may raise concerns about post-compromise security. Specifically, an attacker who gains access to the server can track the membership proofs and the vector commitment value stored on the server. These values will not change if the files remain unchanged, allowing the attacker to easily determine whether any files have been modified.

Homomorphic vector commitment. To address this dilemma, we introduce the concept of homomorphic vector commitment (HVC), which extends the classical additive homomorphic commitment (e.g., Pedersen commitment [18]). Besides the position-binding property, HVC offers a significant advantage over existing VC constructions [8,16,2,7] by satisfying both the position hiding and homomorphic properties simultaneously. The *position hiding* property states that one cannot distinguish whether a commitment was created to a vector $(m_1, \ldots, m_n)$ or to $(m'_1, \ldots, m'_n)$, even after seeing some openings. Although many existing vector commitment constructions already satisfy the homomorphic property, we observe that augmenting them with position hiding using the hybrid methodology would destroy the homomorphic property. By contrast, HVC provides a more elegant solution that preserves both properties. Specifically, HVC allows a user to commit to a vector of values and later reveal the value at a certain position of the committed vector without revealing any information about other vector elements. Moreover, HVC supports efficient homomorphic operations on both the commitment and the openings. The detailed construction of HVC is in Section 4.

In our proposed USS construction, each file in the repository is represented as an entry in a vector. The commitment of this vector serves as a concise stub that represents the entire repository. The binding property of the HVC ensures that any changes to the files will be detected by the client, even if the stub is public and the key is leaked. The homomorphic property of HVC allows the client to efficiently update the stub by computing a new commitment for the updated vector whenever any file in the repository changes. Finally, the hiding property of HVC ensures that an adversary cannot learn any information about individual files from the public stub and other files' membership proofs. At the end of each epoch, the client can rerandomize the stub by homomorphically adding an HVC of zero vectors. This is because an attacker cannot distinguish between the commitment value of the zero vector or the difference vector of the new and old files, and thus cannot track whether files have changed or not.

Homomorphic updatable encryption. However, the above approach alone is insufficient to guarantee post-compromise security. If an attacker gains access to the membership proof of the vector commitment, they could determine whether a file has changed over two epochs. One possible way is to wrap the vector commitment membership proofs with UE, but this approach may present additional challenges for proof updates, particularly when updating a single file. In VC, changes to one element require updates to membership proofs of all elements[10]. Therefore, all UE ciphertexts of the membership proofs must be updated with the plaintext. To ensure efficient database management, the encryption scheme must enable the server to compute the opening update in a homomorphic manner without requiring the retrieval of the ciphertexts of the membership proofs.

Thus, homomorphic updatable encryption is a critical feature for efficient database management, as it enables updates to be performed on the server side on the ciphertexts of the membership proofs without decryption. This reduces communication costs and enhances the scalability of the system. To the best of our knowledge, only one existing updatable encryption scheme, RISE [17], has homomorphic properties for plaintexts. RISE only supports the homomorphic operation by multiplying a new element. Fortunately, we discovered that the opening update of the bilinear pairing-based VC [8] also involves the multiplication of group elements. As a result, VC membership proofs could be encrypted via RISE [17] and we can leverage RISE's homomorphic property to update the encryption of the VC membership proofs.

2 Preliminary

Here we describe the hardness assumption and several primitives that will be used in our constructions.

2.1 Square-CDH Assumption

Recall the definition of bilinear groups. Let $\mathbb{G}$, $\mathbb{G}_T$ be bilinear groups of prime order p equipped with a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. Let $g \in \mathbb{G}$ be random generators. For an algorithm $\mathcal{B}$, define its advantage as

$$\mathrm{Adv}_{\mathcal{B}}^{\text{Square-CDH}}(\lambda) = |\Pr[\mathcal{B}(g, g^a) = g^{a^2}]$$

where $a \leftarrow\!\!\$ \mathbb{Z}_p$ are randomly chosen. We say that the Square-CDH (Square Computational Diffie-Hellman) assumption holds, if for any probabilistic polynomial time (PPT) algorithm $\mathcal{B}$, its advantage $\mathrm{Adv}_{\mathcal{B}}^{\text{Square-CDH}}(\lambda)$ is negligible in λ, where λ is the security parameter.

2.2 Vector Commitment

A vector commitment is defined with a tuple of algorithms [8]

$$\mathsf{VC} = (\mathsf{VC.Setup}, \mathsf{VC.Com}, \mathsf{VC.Open}, \mathsf{VC.Ver}, \mathsf{VC.Update}, \mathsf{VC.ProofUpdate})$$

that works as follows:

- $\mathsf{VC.Setup}(1^\lambda, \mathcal{M}, n) \to crs_n$: Given the security parameter λ, the description of message space $\mathcal{M}$, and the size of committed vector n, the probabilistic setup algorithm outputs a common reference string crs_n
- $\mathsf{VC.Com}_{crs_n}(m_1, \ldots, m_n) \to (C, aux)$: On input an ordered sequence of n messages $m_1, \ldots, m_n$ and the common reference string crs_n, the commitment algorithm outputs a commitment string C and the auxiliary information aux. We denote the commitment space as $\mathcal{C}$.
- $\mathsf{VC.Open}_{crs_n}(m, i, aux) \to \Lambda_i$: This algorithm is run by the committer to produce a membership proof Λ_i that m is the i-th committed message. We denote the commitment space as $\mathcal{P}$.
- $\mathsf{VC.Ver}_{crs_n}(C, m, i, \Lambda_i) \to 1/0$: The verification algorithm outputs 1 only if Λ_i is a valid proof that m is the i-th committed message to the C.
- $\mathsf{VC.Update}_{crs_n}(C, m, m', i,) \to (C', U)$: This algorithm is run by the committer who produced C and wants to update it by changing the i-th message m to a new message m'. It outputs a new commitment string C' together with an update information U.
- $\mathsf{VC.ProofUpdate}_{crs_n}(C,\ \Lambda_j, m', i, U) \to \Lambda'_j$: This algorithm can be run by any user who holds the membership proof Λ_j for some message on position j w.r.t. C, and it allows the user to compute the updated proof Λ'_j valid w.r.t. C' which contains m' as the new message at position i. Basically, the value U contains the updated information which is needed to compute such values.

A vector commitment (VC) scheme is expected to satisfy correctness, position binding, and conciseness [8]. However, the hiding property of VCs has not been extensively discussed in many existing constructions. Hiding VCs can be obtained by composing a non-hiding VC with a standard commitment scheme.

2.3 Updatable Encryption

Updatable encryption (UE) is a cryptographic technique that allows periodic updates of the secret key of encrypted outsourced data. The syntax of UE is defined as follows:

Definition 1 (Updatable Encryption). *The updatable encryption (UE) consists of the following six algorithms*

$$\mathsf{UE} = (\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Next}, \mathsf{Upd}).$$

- $\mathsf{UE.Setup}(1^\lambda)$ *is a randomized algorithm run by the client. It takes the security parameter λ as input and outputs the public parameter pp which will be shared with the server. Later all algorithms take pp as input implicitly.*
- $\mathsf{UE.Keygen}(e)$ *is a client-run randomized algorithm. It takes the epoch index e as input and outputs a secret key k_e for the epoch e.*
- $\mathsf{UE.Enc}(k_e, m)$ *is a client-run randomized algorithm. It takes the secret key k_e and the message m as inputs, and outputs the ciphertext C_e.*
- $\mathsf{UE.Dec}(k_e, C_e)$ *is a deterministic algorithm run by the client. It takes the secret key k_e and the ciphertext C_e as inputs, and outputs the message m or the symbol $\perp$.*
- $\mathsf{UE.Next}(k_e, k_{e+1})$ *is a randomized algorithm run by the client. It takes the old secret key k_e of the last epoch and the new secret key k_{e+1} of the current epoch as inputs and outputs a re-encrypt token Δ_e or the symbol $\perp$.*
- $\mathsf{UE.Upd}(\Delta_e, C_e)$ *is a deterministic algorithm run by the server. It takes the re-encrypt token Δ_e and the ciphertext C_e as inputs, and outputs a new ciphertext C_{e+1} under the secret key k_{e+1} or the symbol $\perp$.*

The correctness condition of an updatable encryption scheme ensures that an update of a valid ciphertext C_e from epoch e to $e+1$ leads again to a valid ciphertext C_{e+1} that can be decrypted under the new epoch key k_{e+1}. The security definition of UE can be found in Appendix A and [17].

RISE. In this paper, we leverage the homomorphic updatable encryption-RISE [17]. Recall the RISE construction as follows:

- $\mathsf{RISE.Setup}(1^\lambda)$: return pp as public parameter, also an implicit input of the following algorithms.
- $\mathsf{RISE.Keygen}(e)$: $k_e \leftarrow\!\!\$\ \mathbb{Z}_p^*$.
- $\mathsf{RISE.Enc}(k_e, m)$: $y = g^{k_e}$, $r \leftarrow\!\!\$\ \mathbb{Z}_q$, return $C_e \leftarrow\!\!\$\ (y^r, g^r m)$.
- $\mathsf{RISE.Dec}(k_e, C_e)$: parse $C_e = (C_1, C_2)$, return $m \leftarrow C_2 \cdot C_1^{-1/k_{e+1}}$.
- $\mathsf{RISE.Next}(k_e, k_{e+1})$:$\Delta_{e+1} \leftarrow\!\!\$\ (k_{e+1}/k_e, g^{k_{e+1}})$, return Δ_{e+1}.
- $\mathsf{RISE.Upd}(\Delta_{e+1}, C_e)$: parse $\Delta_{e+1} = (\Delta, y')$ and $C_e = (C_1, C_2)$, $r' \leftarrow\!\!\$\ \mathbb{Z}_q$, $C_1' \leftarrow C_1^\Delta \cdot y'^{r'}$, $C_2' \leftarrow C_2 \cdot g^{r'}$, return $C_{e+1} \leftarrow (C_1', C_2')$.

The updatable RISE encryption scheme has been proven to be IND-ENC secure under the decisional Diffie-Hellman (DDH) assumption [17]. Furthermore, it has been observed that RISE is homomorphic under its encryption algorithm Enc and decryption algorithm Dec. Specifically, given two plaintexts m and m', their respective RISE ciphertexts are $\mathsf{RISE.Enc}(k_e, m) = (C_1, C_2)$ and $\mathsf{RISE.Enc}(k_e, m') = (C_1', C_2')$. Then, their product is computed as $\mathsf{RISE.Enc}(k_e, m) \cdot \mathsf{RISE.Enc}(k_e, m') = (C_1 \cdot C_1', C_2 \cdot C_2')$. The decryption algorithm of RISE satisfies $\mathsf{RISE.Dec}(\mathsf{RISE.Enc}(k_e, m) \cdot \mathsf{RISE.Enc}(k_e, m')) = m \cdot m'$.

3 Updatable Secure Storage

As previously introduced, an updatable secure storage (USS) system can be considered an advanced version of GitHub that offers secure outsourced storage and version control services, even in scenarios where the trustworthiness of the server is not fully assured. USS provides users with the capability to create and update remote encrypted repositories on the server while maintaining a secret key and a public stub on the client side. The stored data and its updated version remain confidential and can only be accessed by authorized parties with the secret key. Additionally, USS ensures that a malicious server is unable to manipulate the client into accepting a tampered database or an outdated file, even if it gains access to the client's secret key or violates the protocol during each interactive procedure including file updates, key updates, and data retrieval.

Moreover, the USS system supports *key rotation*, a feature that is similar to updatable encryption schemes [12]. Key rotation is a critical security mechanism that ensures the confidentiality of the database, even if either the key or the storage is compromised, but not both simultaneously. By periodically rotating the secret key, the USS system can prevent an attacker who has gained access to an old key from knowing the current plain version of the database. This is particularly important in scenarios where the key may have been compromised, as it ensures that any data stored on the server remains secure.

Furthermore, we consider the possibility of external attackers gaining access to the repository stored on the server temporarily and occasionally, mirroring frequently reported data breaches. Despite the repository being encrypted, monitoring the alterations in the encrypted repository could unveil its update history, which has the potential to expose users' activities and preferences, thus compromising the privacy of the individual. For instance, if the user updates a file related to their medical records, an attacker who gains access to the update history can infer that the user has medical issues, even if they cannot access the actual contents of the file. To mitigate this issue, we propose a *file update unlinkability* mechanism in USS during key rotation. This mechanism utilizes a rerandomization algorithm to conceal the update history of the database if the attacker has only intermittent access to the stored encrypted repository.

To ensure system efficiency, USS uses efficient data update techniques and secret key refresh/rotation mechanisms. These mechanisms guarantee that communication costs and client workloads remain independent of the number and size of files stored in the system.

3.1 Syntax of USS

To ensure the security of remotely stored data, the USS creates a unique secret key for each encrypted repository. The user will possess the secret key and a unique stub associated with the respective repository on the client side. Each repository can store a predetermined number of encrypted files. When a specific file is required, the user can retrieve it from the remote repository, decrypt its content using the secret key, and verify its integrity. If a file needs to be updated, the client will interactively communicate with the server to modify the file within the repository. Additionally, the client will periodically generate new keys and update the encrypted files in the repository to maintain security.

Accordingly, the syntax of USS should be as follows.

- $\mathsf{USS.ParGen}(1^\lambda, \mathcal{M}, n) \to pp$: Given the security parameter λ, the description of message space $\mathcal{M}$, and the size of vector degree n, the **parameter generation** algorithm generates the public parameter pp.
- $\mathsf{USS.KeyGen}(1^\lambda, pp) \to sk$: Given the security parameter λ and the public parameter pp, the **key generation** algorithm generates the secret key sk.
- $\mathsf{USS.Store}(db, sk, pp) \to (rep, sb)$: Given a database db that contains n independent files $m_1, \ldots, m_n$, and the public parameter pp, the client executes the **data storing** algorithm. The output of this algorithm is a repository $rep = (c_1, \ldots, c_n)$, which will be stored on the server side, along with a stub sb that will be accessible to both the server and the client. Each c_i is the ciphertext corresponding to the file m_i.
- $\mathsf{USS.Rev}_{client}(i, sk, sb, pp) \leftrightarrows \mathsf{USS.Rev}_{server}(rep, sb, pp) \to \langle m_i/\perp; \cdot\rangle$: The **data retrieval** algorithm is an interactive procedure that enables the client to retrieve file i from the server. The client provides the index i, secret key sk, stub sb, and public parameter pp. The server holds the repository rep and public parameter pp. If the data retrieval procedure succeeds, the client will output m_i; otherwise, it will output $\perp$.
- $\mathsf{USS.FileUp}_{client}(i, m_i', sk, sb, pp) \leftrightarrows \mathsf{USS.FileUp}_{server}(rep, sb, pp) \to \langle sb'; sb', rep'\rangle$: The **file update** is an interactive procedure that allows the client to update the ith file to m_i'. Specifically, the client holds the index i, the new ith file m_i', the secret key sk, the stub sb and the public parameter pp, and the server has the storing repository rep together with the public parameter pp. After the interaction, the client will have a new stub sb', and the server will store a new repository rep'.
- $\mathsf{USS.KeyUp}_{client}(sk, sk', sb, pp) \leftrightarrows \mathsf{USS.KeyUp}_{server}(sb, rep, pp) \to \langle sb'; rep'\rangle$: The **key update** is an interactive procedure that makes the server to update the storing file to a new key sk'. After the interaction, the client will have a new stub sb', and the server will store a new repository rep'.

Basically, the USS scheme should satisfy the following properties for correctness and efficiency.

Correctness. The correctness guarantees that when the client invokes the **data retrieval** procedure to fetch the ith file if the server is honest, the client always successfully gets m_i no matter how many times the key has been updated and m_i is the latest updated version of the ith file deposited by the client.

Client storage efficiency. The client storage efficiency requires that the size of the information stored on the client side, including the secret key sk and the stub sb, should be independent of the size of database db and even the number of the files n.[4]

[4] The size of public parameter stored on both client and server should be independent of the size of database db, although it may be related to the number of files n.

Retrieve efficiency. To ensure efficient retrieval, the communication cost for the interactive procedure **data retrieval** should be independent of the size of other plaintext files m_j for $j \neq i$ and the number of files n in the entire repository when retrieving the i-th file. However, it may depend on the size of the retrieved file m_i.

File update efficiency. For efficient file updates, the communication cost for the **file update** procedure should be independent of the size of other plaintext files m_j for $j \neq i$ and the number of files n in the entire repository when updating the i-th file. However, it may depend on the size of the updating file m_i.

Key update efficiency. In order to ensure efficient key updates, it is desirable that the communication cost associated with the key update procedure remains independent of the size of all files m_i for $i = 1, \ldots, n$. If the communication cost is also independent of the number of files n in the repository, we classify these schemes as ciphertext-independent. Conversely, if the communication cost depends on the number of files, we refer to them as ciphertext-dependent schemes.[5]

3.2 Security Models

The security threats associated with USS arise from three main sources. Firstly, the system must safeguard the confidentiality of the stored database against the honest but curious server and external attackers who have temporary and intermittent access to server storage and user secrets with no time overlap. Secondly, the system must retain the integrity of the stored database against the malicious server. Thirdly, given the possibility of the encrypted repository being compromised by external attackers multiple times, and potentially exposing the update history of the repository, the system must offer security guarantees that prevent update history leakage, even if the user's secret is ever revealed once simultaneously.

We present three models aimed at addressing confidentiality security challenges: IND-DD-UP (indistinguishability of dynamic databases under chosen plaintext attacks), IND-REENC-CPA (indistinguishability of re-encryption ciphertext under chosen plaintext attacks), and ciphertext indistinguishability for file update (IND-FileUp-CPA for short). The IND-DD-UP and IND-FileUp-CPA models ensure the fundamental confidentiality of the original database and updated files respectively, while the IND-REENC-CPA model provides post-compromise security and the ability to conceal the update history of the repository. Intuitively, the IND-DD-UP model stipulates that an adversary cannot distinguish between two vectors of messages once they are encrypted. This holds even if the adversary has the ability to corrupt keys, trigger file updates, or initiate key rotations. The IND-FileUp-CPA model ensures that an adversary cannot distinguish between two files used to replace the current file in the repository,

[5] This notion is directly borrowed from the updatable encryption framework, which distinguishes between ciphertext-dependent and ciphertext-independent versions.

even if the attacker knows the previously stored data. On the other hand, the IND-REENC-CPA model ensures that an adversary cannot distinguish whether the ciphertexts have been updated after a key rotation. This is also true even if the adversary has the ability to corrupt keys, trigger file updates, or initiate key rotations. Our proposed models are similar to the IND-ENC and IND-UPD models presented in [17], respectively.

To address the security challenges related to message integrity, we propose a model called ordered full plaintext integrity (OF-PTXT for short). Intuitively, OF-PTXT imposes stricter requirements than the INT-PTXT game for the updatable encryption proposed in [15], as it guarantees message integrity even in the presence of a leaked secret key and non-compliant servers that do not rotate keys as required by the protocol.

Confidentiality. Our confidentiality-related models consider three critical security properties: message confidentiality (IND-DD-UP), file update unlinkability (IND-FileUp-CPA), and re-encryption indistinguishability (IND-REENC-CPA). In these models, the adversary may attempt to compromise the confidentiality of any files in a target repository. The encryption of the target file is referred to as challenge ciphertext. Furthermore, the repository that contains the target files is called the challenge repository. However, since the key is rotated when the epoch evolves, the key-updated version of the challenge ciphertext is referred to as challenge-equal ciphertext. For simplicity, we also use the term "challenge-equal ciphertext" to represent both original and key-updated versions of the challenge ciphertext. Consequently, the repository containing the challenge-equal ciphertexts is called the challenge-equal repository.

More precisely, the challenger maintains the following internal states.

e**:** The current epoch number. It is initialized as 1.

e^***:** The epoch number from which the challenge begins. It is initialized as $\perp$.

sb^***:** The current stub of a repository including challenge-equal ciphertexts, which is initialized as $\perp$.

$\mathcal{K}$**:** The set of epochs in which the adversary has corrupted the epoch key by querying the key corruption oracle.

$\mathcal{C}$**:** The set of epochs in which the adversary corrupts a challenge-equal ciphertext by querying the challenge-equal ciphertext corruption oracle.

$\mathcal{I}$**:** The set of indices of challenge-equal ciphertexts in the repository. Before the adversary submits the challenge, the set is empty $\mathcal{I} = \emptyset$.

$\mathcal{L}$**:** The collection consists of tuples (t, sb, rep, db) which will be used to track all the repositories on the server, where t represents the epoch number, sb represents the corresponding stub, and rep and db represent the corresponding encrypted repository and plaintext database, respectively.

$\mathcal{S}$**:** The collection contains tuples (t, Trans), where t represents the epoch number and Trans represents all the corresponding key update transcripts from epoch $t-1$ to epoch t.

$\mathcal{F}$**:** The collection of challenge-equal ciphertexts' file update transcripts contains tuples (sb, t, i, ft_i), where ft_i represents the file update transcript generated in the file update query $\mathcal{O}.\mathsf{FileUp}(sb, m_i', i)$ at epoch t, and the index i is the index of a challenge-equal ciphertext.

$\mathcal{T}$: The set of epochs in which the adversary queries the key update transcript oracle. If the transcript of the key update procedure from sk_t to sk_{t+1} is corrupted, it is represented as $t \in \mathcal{T}$. The set $\mathcal{T}$ is initially empty, indicating that the adversary has not yet queried the key update transcript oracle.

The adversary is given the following oracles.

– $\mathcal{O}.\mathsf{Store}(db)$: The purpose of this oracle is to enable the adversary to deposit a database on the server. To this end, the challenge invokes the storing algorithm $\mathsf{Store}(sk, db, pp)$ to generate the stored file rep and the stub sb, which are then given to the adversary. Furthermore, the challenger adds the tuple (e, sb, rep, db) to the set $\mathcal{L}$.

– $\mathcal{O}.\mathsf{Next}$: The adversary uses this oracle to initiate the update of all repositories on the server. The adversary will automatically gain access to their updated versions, except for the challenge-equal ciphertexts.
Specifically, the challenger retrieves all entries (e, sb, rep, db) from the database $\mathcal{L}$ having the current epoch number e. Subsequently, the challenger generates a new epoch key sk' and executes the key update procedure to produce new stubs sb' and new repository rep' for each retrieved entry. The transcripts of the generated key updates are denoted as Trans. The current epoch number is then incremented to $e + 1$, and new entries (e, sb', rep', db) are appended to the database $\mathcal{L}$. Additionally, a new entry (e, Trans) is added to the list $\mathcal{S}$. Furthermore, if there exists an entry $(e, sb, *, *) \in \mathcal{L}$ with $sb = sb^*$, then sb^* is also updated accordingly. The challenger provides the adversary with the stub and ciphertext elements having non-challenge indices $\{j\}_{j \notin \mathcal{I}}$ for the current epoch. For all other entries $(e, sb, *, *) \in \mathcal{L}$ such that $sb \neq sb^*$, the adversary is furnished with the updated versions of (sb, rep).

– $\mathcal{O}.\mathsf{KeyCorr}(t)$: The purpose of this oracle is to facilitate the adversary in retrieving the secret key. If the epoch number t is not greater than the current epoch number e, the oracle will provide the adversary with the secret key sk_t corresponding to epoch t. Additionally, epoch t will be included in the set of key corruptions $\mathcal{K}$.

– $\mathcal{O}.\mathsf{KeyUpTrans}(t)$: The purpose of this oracle is to facilitate the adversary in retrieving the key update transcript. If epoch number t is no more than the current epoch number e, retrieve the entry (t, Trans), and return the key update transcript Trans of all ciphertexts from epoch $t - 1$ to epoch t to the adversary. Add epoch t to the set of key corruptions $\mathcal{T}$.

– $\mathcal{O}.\mathsf{FileUp}(sb, m_i', i)$: This oracle enables the adversary to modify the i-th file of the current epoch to be the encryption of m_i'. The input includes the stub sb, the new file m_i', and its index i, where $i \in [1, n]$. The challenger first retrieves the entry (t, sb, rep, db) in $\mathcal{L}$ with the current epoch number $t = e$ and the same stub sb. If the entry is empty, the oracle outputs $\perp$. Otherwise, the challenger executes $\mathsf{FileUp}_{client}(i, m_i', sk_e, sb, pp) \leftrightarrows \mathsf{FileUp}_{server}(rep, sb, pp)$ and updates the entry with (e, sb', rep', db).

If $sb \neq sb^*$, then the challenger returns (sb', rep') and the file update transcript ft_i to the adversary. If $sb = sb^*$, which means that the queried stub sb is the stub of a challenge-equal ciphertext in the current epoch, then the challenger updates $sb^* \leftarrow sb'$ and checks whether index i belongs to a challenge-equal ciphertext. If $i \in \mathcal{I}$, remove i from $\mathcal{I}$, add a tuple (sb, e, i, ft_i) to collection $\mathcal{F}$. The challenger returns the updated stub sb' and each updated ciphertext f'_j with index $j \notin \mathcal{I}$ to the adversary. If $i \notin \mathcal{I}$, the challenger returns the updated stub sb', the file update transcript ft_i, and each updated ciphertext f'_j with index $j \notin \mathcal{I}$ to the adversary.

– $\mathcal{O}.\mathsf{ChaCTCorr}(j)$: This oracle helps the adversary to learn the jth ciphertext of the challenge-equal ciphertext vector in the current epoch. If $j \in \mathcal{I}$, the jth element is a challenge-equal ciphertext for the current epoch. Then the challenger finds the entry (t, sb, rep, db) of $\mathcal{L}$ with the current epoch number $t = e$ and the stub sb is equal to $sb^* \neq \perp$, and add the current epoch e to the challenge-equal ciphertext corruption set $\mathcal{C}$ and give the adversary the jth ciphertext f_j where $rep = (f_1, \ldots, f_n)$. If $j \notin \mathcal{I}$, return $\perp$.
– $\mathcal{O}.\mathsf{ChaFTCorr}(sb, t, i)$: This oracle helps the adversary to learn the file update transcripts of challenge-equal ciphertexts. The challenger finds the entry (sb, t, i, ft_i) of F with the same stub sb, epoch t, and index i, and returns the transcript ft_i to the adversary

Trivial win condition. Adversaries could trivially win the confidentiality game if they corrupt both the epoch key and the challenge ciphertext or the updated version at that epoch. Since adversaries are given access to multiple oracles, where key update transcripts could help to update ciphertexts to the new key due to USS's function and even downgrade ciphertexts to the previous key if USS's update function is bi-directional. To exclude the trivial win conditions, we define an extended ciphertext corruption set $\tilde{\mathcal{C}}$ to record the epochs at which adversaries corrupt the challenge ciphertext via directly querying the challenge ciphertext oracle $\mathcal{O}.\mathsf{ChaCTCorr}$ or indirectly referring to the challenge ciphertext based on queries of $\mathcal{O}.\mathsf{KeyUpTrans}$ and $\mathcal{O}.\mathsf{ChaCTCorr}$. Here we assume the key update transcripts could update/downgrade ciphertexts in bi-direction since the scheme we use to construct USS supports it. Then we have $i \in \tilde{\mathcal{C}}$ if $i \in \mathcal{C}$, or $i-1 \in \mathcal{C}$ & $i \in \mathcal{T}$, or $i+1 \in \mathcal{C}$ and $i+1 \in \mathcal{T}$. The trivial win condition is $\mathcal{K} \cap \tilde{\mathcal{C}} \neq \emptyset$.

Message Confidentiality. Here we defined IND-DD-UP security which aims to capture CPA style message confidentiality in the key updatable and file updatable setting. Concretely, the adversary can query $\mathcal{O}.\mathsf{Store}$ oracle for repository encryption in the storage. The adversary is allowed to engage the key rotation and get the update of non-challenge-equal files via the $\mathcal{O}.\mathsf{Next}$ oracle. The adversary can also corrupt some epoch key and challenge-equal file via the $\mathcal{O}.\mathsf{KeyCorr}, \mathcal{O}.\mathsf{ChaCTCorr}$ oracles. Furthermore, the adversary is allowed to query file update of each repository via $\mathcal{O}.\mathsf{FileUp}$ oracle. To exclude the trivial win of the security game, the adversary is not allowed to see the key and the challenge-equal file encryption simultaneously. Such requirements are similar to the restrictions in the security models of the updatable encryptions [14]. Formally, we have the IND-DD-UP game as Figure 1.

```
$\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, b)$
$pp \leftarrow \mathsf{ParGen}(1^\lambda, \mathcal{M}, n)$, Initialize $e = 1, e^* = \perp, sb^* = \perp$,   Set $\mathcal{K}, \mathcal{C}, \mathcal{I}, \mathcal{L}, \mathcal{S}, \mathcal{T}, \mathcal{F}$ as $\emptyset$
$sk_1 \leftarrow \mathsf{KeyGen}(pp)$
$(db_0, db_1, state) \leftarrow \mathcal{A}_1^{\mathcal{O}.\mathsf{Store}, \mathcal{O}.\mathsf{Next}, \mathcal{O}.\mathsf{FileUp}, \mathcal{O}.\mathsf{KeyCorr}, \mathcal{O}.\mathsf{KeyUpTrans}}(pp)$
Parse $db_0 = (m_{0,1}, \ldots, m_{0,n}), db_1 = (m_{1,1}, \ldots, m_{1,n})$
$(sb^*, rep^*) \leftarrow \mathsf{Store}(db_b, sk_e, pp)$
$e^* = e$,   $\mathcal{L} = \mathcal{L} \cup (e, sb^*, rep^*, db_b)$
for $i = 0$ to $n$
  if $m_{0,i} \neq m_{1,i}$, then $\mathcal{I} = \mathcal{I} \cup \{i\}$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}.\mathsf{Store}, \mathcal{O}.\mathsf{Next}, \mathcal{O}.\mathsf{FileUp}, \mathcal{O}.\mathsf{KeyCorr}, \mathcal{O}.\mathsf{KeyUpTrans}, \mathcal{O}.\mathsf{ChaFTCorr}, \mathcal{O}.\mathsf{ChaCTCorr}}(state, rep^*)$
return $b'$ if $\mathcal{K} \cap \tilde{\mathcal{C}} = \emptyset$
```

Fig. 1. The game of IND-DD-UP.

Definition 2 (IND-DD-UP). *An updatable secure storage scheme USS is called IND-DD-UP secure if for any* PPT *adversary $\mathcal{A}$ the following advantage is negligible in the security parameter λ:*

$$\mathsf{Adv}^{ind\text{-}dd\text{-}up}_{USS,\mathcal{A}}(1^\lambda, \mathcal{M}, n) := \left| \Pr[\mathsf{Exp}^{ind\text{-}dd\text{-}up}_{USS,\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{ind\text{-}dd\text{-}up}_{USS,\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] \right|$$

File update unlinkability. To capture the security that the file update operation does not leak the confidentiality of the updated file, we define the file update unlinkability via the following experiment with the adversary. Intuitively, it ensures that attackers corrupting the storage before and after a file update operation learn nothing about file updates, such as whether the file content has changed, and what the current file content is.

We describe the file update security experiment $\mathrm{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}$ for the key updatable dynamic secure storage scheme USS and adversary $\mathcal{A}$. In $\mathrm{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}$ experiment, $\mathcal{A}$ submits two possible file $(m_{1,0}, m_{1,1})$ for challenge. The challenger updates the first file of the stored storage with one of the two submissions selected randomly and gives the updated ciphertext to the adversary as the challenge ciphertext. $\mathcal{A}$'s goal is to give a correct guess on which file is chosen to update. The trivial win situation is that the adversary corrupts both the epoch key and the challenge-equal ciphertext at the same epoch, i.e., $\mathcal{K} \cap \mathcal{C} \neq \emptyset$.

Definition 3 (IND-FileUp-CPA). *An updatable secure storage scheme USS is called IND-FileUp-CPA secure if for any* PPT *adversary $\mathcal{A}$ the following ad-*

```
$\mathrm{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, b)$
$pp \leftarrow \mathsf{ParGen}(1^\lambda, \mathcal{M}, n)$, Initialize $e = 1, e^* = \perp, sb^* = \perp$, Set $\mathcal{K}, \mathcal{C}, \mathcal{I}, \mathcal{L}, \mathcal{S}, \mathcal{T}, \mathcal{F}$ as $\emptyset$
$sk_1 \leftarrow \mathsf{KeyGen}(pp)$
$(sb, i, m_{0,i}, m_{1,i}, state_1) \leftarrow \mathcal{A}_1^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans}}(pp)$
if $(e, sb, *, *) \notin \mathcal{L}$, or $i \notin [1, n]$, or $|m_{0,i}| \neq |m_{1,i}|$, then return $\perp$
Retrieve $(e, sb, rep, db = (m_1, \dots, m_n))$, Set $e^* = e$, $\mathcal{I} = \mathcal{I} \cup \{i\}$, $\mathcal{C} = \mathcal{C} \cup \{e\}$
Run $\mathsf{FileUp}_{token}(sk_e, sb, m_{b,i}, i, pp) \leftrightarrows \mathsf{FileUp}_{server}(sb, rep, pp) \rightarrow \langle sb^*; rep^* \rangle$
$\mathcal{L} = \mathcal{L} \cup (e, sb^*, rep^*, db_b = (m_1, \dots, m_{b,i}, \dots, m_n))$
Record the file update transcript as $fpt$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans},\mathcal{O}.\mathsf{ChaFTCorr},\mathcal{O}.\mathsf{ChaCTCorr}}(state_1, sb^*, rep^*, fpt)$
return $b'$ if $\mathcal{K} \cap \tilde{\mathcal{C}} = \emptyset$
```

Fig. 2. The game of IND-FileUp-CPA

vantage is negligible in the security parameter λ:

$$\mathsf{Adv}^{\textit{ind-fileup-cpa}}_{\textit{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n) :=$$
$$\left|\Pr[\mathsf{Exp}^{\textit{ind-fileup-cpa}}_{\textit{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\textit{ind-fileup-cpa}}_{\textit{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right|$$

Key update unlinkablity. Intuitively, key update unlinkability is aimed to capture the security for key updates after both corruptions. More concretely, attackers may corrupt both the client and the server at the same epoch. After the key rotation, attackers corrupt the server and obtain the updated ciphertext. Key update unlinkability ensures that attackers cannot detect whether the updated ciphertext contains the same plaintext as the previous corrupted ciphertext. The security is similar to the IND-UPD security of UE [17] since we provide the adversary with all the oracles IND-UPD security provides. In addition, our key update unlinkability allows the adversary to have additional access to the file update oracle.

We define the following security experiment $\mathrm{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}$ for updatable secure cloud storage scheme USS and adversary $\mathcal{A}$, who has access to the oracle tuple $(\mathcal{O}.\mathsf{Store}, \mathcal{O}.\mathsf{Next}, \mathcal{O}.\mathsf{KeyCorr}, \mathcal{O}.\mathsf{FileUp}, \mathcal{O}.\mathsf{KeyUpTrans}, \mathcal{O}.\mathsf{ChaCTCorr})$ like in the above confidentiality games. So, the trivial win condition is triggered in the same case.

Definition 4 (IND-REENC-CPA). *An updatable secure storage scheme USS is called IND-REENC-CPA secure if for any* PPT *adversary $\mathcal{A}$ the following advantage is negligible in the security parameter λ:*

$$\mathsf{Adv}^{\textit{ind-reenc-cpa}}_{\textit{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n) :=$$
$$\left|\Pr[\mathsf{Exp}^{\textit{ind-reenc-cpa}}_{\textit{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\textit{ind-reenc-cpa}}_{\textit{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right|$$

```
$\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, b)$
$pp \leftarrow \mathsf{ParGen}(1^\lambda, \mathcal{M}, n)$, Initialize $e, e^*, sb^*$, Set $\mathcal{K}, \mathcal{C}, \mathcal{I}, \mathcal{L}, \mathcal{S}, \mathcal{T}, \mathcal{F}$ as $\emptyset$
$sk_1 \leftarrow \mathsf{KeyGen}(pp)$
$(sb_0, sb_1, state_1) \leftarrow \mathcal{A}_1^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans}}(pp)$
Retrieve $(e, sb_0, rep_0 = (c_{0,1} \ldots, c_{0,n}), db_0), (e, sb_1, rep_1 = (c_{1,1}, \ldots, c_{1,n}), db_1)$ from $\mathcal{L}$
Set $\mathcal{I} = \{i\}_{i \in \{1,\ldots,n\}, c_{0,i} \neq c_{1,i}}$, $e = e + 1$, $e^* = e$, $\mathcal{C} = \mathcal{C} \cup \{e\}$
$sk_e \leftarrow \mathsf{KeyGen}(pp)$
Run $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb_b, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep_b, pp)$ to output $\langle sb^*; rep^* \rangle$
for each $(e-1, sb, rep, db) \in \mathcal{L}$, where $sb \notin \{sb_0, sb_1\}$
    Run $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb'; rep' \rangle$
    Set $\mathcal{L} = \mathcal{L} \cup (e, sb', rep', db)$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans},\mathcal{O}.\mathsf{ChaCTCorr}}(state_1, sb^*, rep^*,$ all $(sb', rep'))$
return $b'$ if $\mathcal{K} \cap \tilde{\mathcal{C}} = \emptyset$
```

Fig. 3. The game of IND-REENC-CPA

Integrity. We define a kind of strong plaintext integrity notion called *ordered full plaintext integrity* (OF-PTXT for short). For the classic authenticated encryption schemes, plaintext integrity ensures that attackers cannot make any forgery for new plaintext except the queried ones, which work for static storage with appending function. But for dynamic storage, where some storage may be changed or even deleted, the old or deleted messages could be leveraged by dishonest storage providers to cheat users, which is not covered by classic integrity. Here OF-PTXT provides stronger integrity in the dynamic storage setting, where data could be updated dynamically. OF-PTXT ensures attackers cannot forge for a plaintext that does not belong to the current storage. To be formal, we show a security experiment between the adversary who acts as the malicious storage server, and the challenger who acts as the honest user. More precisely, the challenger will maintain the following list to record the latest version of databases.

$\mathcal{R}$: The list recording the latest stub and the message vector pair (sb, db) generated during the integrity game. $\mathcal{R}$ is initialized as empty and will be updated for each file update and key update. $\mathcal{R}$ has only entries for the latest epoch.

We use the stub to track the target stored database. For a certain pair (sb, db) of $\mathcal{R}$ and a certain index i, the adversary aims to make the client accept m_i' as the ith element of the database but $m_i' \neq db[i]$.

Moreover, we allow the adversary to launch active attacks in the integrity game. The server which may be corrupted by the adversary may manipulate the storage and even to not follow the protocol during the file update or key update procedures. To capture adversary's above capability, three special oracles, including the database storing oracle $\mathcal{O}.\mathsf{StoreINT}$, the next oracle $\mathcal{O}.\mathsf{NextINT}$ and the file update token oracle $\mathcal{O}.\mathsf{FileUpINT}$ are provided for the adversary in integrity game. Besides, the adversary in the integrity game can also learn the security key via the key corruption oracle $\mathcal{O}.\mathsf{KeyCorr}$, which means USS can guarantee the integrity even when the key is leaked.

We will elaborate the special oracles for the integrity game in the following. For brevity, please refer to our previous descriptions about the similar oracles in the confidentiality section 3.2.

- $\mathcal{O}.\mathsf{StoreINT}(db)$: This oracle is to let the adversary learn the stored file generated by the storing algorithm. The challenge will invoke the storing algorithm $\mathsf{Store}(sk_e, db, pp)$ to generate the stored file rep and the stub sb, and give rep and sb to the adversary. And add the pair (sb, db) to $\mathcal{R}$.
- $\mathcal{O}.\mathsf{NextINT}$: This oracle is to let the adversary to invoke the client to launch the key update procedure. The challenger updates the epoch number $e = e + 1$, runs the KeyGen algorithm to generate the new epoch key $sk_e = sk'$, and runs the key update client-side algorithm to update all stubs sbs into the corresponding sb's and to generate client-side key update transcripts Trans_c for the server. Then the challenger returns all the new stubs sb's and transcripts Trans_c to the adversary, and updates each entry (sb, db) in $\mathcal{R}$ with the corresponding (sb', db).
- $\mathcal{O}.\mathsf{FileUpINT}(m'_i, i, sb)$: The adversary uses this oracle to invoke the client to launch the file update procedure and replace the ith element of the database to m'_i. More precisely, the input of this oracle contains the new file m'_i, its index i, and the corresponding stub sb. The challenger will first check whether the stub is contained in the list $\mathcal{R}$. During this interaction, the client will communicate with the corrupted server according to the specification of the designed scheme, while the adversary could respond to the client with an arbitrary message and violate the protocol design. If the client finally accepts the update results, the challenger will update the entry (sb, db) of $\mathcal{R}$ with (sb', db').

We describe the integrity experiment $\mathrm{Exp}^{\mathsf{of\text{-}ptxt}}_{\mathsf{USS},\mathcal{A}}$ for key updatable dynamic secure storage scheme USS and adversary $\mathcal{A}$, who has access to the oracle tuple $(\mathcal{O}.\mathsf{StoreINT}, \mathcal{O}.\mathsf{NextINT}, \mathcal{O}.\mathsf{FileUpINT}, \mathcal{O}.\mathsf{KeyCorr})$.

Definition 5 (OF-PTXT). *An updatable secure storage scheme USS is called OF-PTXT secure if for any PPT adversary $\mathcal{A}$ the following advantage is negligible in the security parameter λ:*

$$\mathsf{Adv}^{\textit{of-ptxt}}_{\textit{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n) := \Pr[\mathsf{Exp}^{\textit{of-ptxt}}_{\textit{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n) = 1]$$

4 Homomorphic Vector Commitment

This section presents an introduction to Homomorphic Vector Commitment (HVC) and explores the difficulties involved in constructing an HVC that can simultaneously satisfy both the position hiding and homomorphic properties.

```
$\mathsf{Exp}^{\text{of-ptxt}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n)$
$pp \leftarrow \mathsf{ParGen}(1^\lambda, \mathcal{M}, n)$, Initialize $\mathcal{R}, e$
$sk_e \leftarrow \mathsf{KeyGen}(pp)$
$(sb, i, state_1) \leftarrow \mathcal{A}_1^{\mathcal{O}.\mathsf{StoreINT}, \mathcal{O}.\mathsf{NextINT}, \mathcal{O}.\mathsf{FileUpINT}, \mathcal{O}.\mathsf{KeyCorr}}(pp)$
**if** $(sb, *) \notin \mathcal{R}$ or $i \notin [1, n]$   **return** 0
**else** Run $\mathsf{Rev}_{client}(i, sk_e, sb, pp) \leftrightarrows \mathcal{A}$ to output $\langle m_i^*; \cdot \rangle$
  **for** $\forall \mathbf{m}$ s.t. $(sb, \mathbf{m}) \in \mathcal{R}$
      **if** $m_i^* \neq \mathbf{m}[i]$   **return** 1
  **endfor**
**return** 0
```

Fig. 4. The game of OF-PTXT

4.1 Syntax and Notions

HVC is defined with the following algorithms:

$\mathsf{HVC} = (\mathsf{HVC.Setup}, \mathsf{HVC.Com}, \mathsf{HVC.Open}, \mathsf{HVC.Ver}, \mathsf{HVC.ComHom}, \mathsf{HVC.OpenHom})$

that works as following:

- $\mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n) \to crs_n$: Given the security parameter λ, the description of committed message space $\mathcal{M}$, and the size of committed vector n, the probabilistic setup algorithm outputs a common reference string crs_n.
- $\mathsf{HVC.Com}_{crs_n}(\mathbf{m}) \to (C, aux)$: On input an ordered sequence of n messages $\mathbf{m} = (m_1, \ldots, m_n)$ and the common reference string crs_n, the commitment algorithm outputs a commitment string C and the auxiliary information aux. We denote the commitment space as $\mathcal{C}$. The auxiliary information aux is succinct, say independent of the vector degree n.
- $\mathsf{HVC.Open}_{crs_n}(i, \mathbf{m}, aux) \to \Lambda_i$: This algorithm is run by the committer to produce a proof (also known as opening) Λ_i that the i-th element $\mathbf{m}[i]$ is the committed message. We denote the proof space as $\mathcal{P}$.
- $\mathsf{HVC.Ver}_{crs_n}(C, m, i, \Lambda_i) \to 1/0$: The verification algorithm outputs 1 only if Λ_i is a valid proof that m is the i-th committed message to the C.
- $\mathsf{HVC.ComHom}_{crs_n}(C, C' \in \mathcal{C}) \to C''$: This algorithm can be run by any user who holds two commitment belonging to the commitment space $\mathcal{C}$, and it allows the user to compute another commitment $C'' = C \oplus C' \in \mathcal{C}$, where $\oplus$ denotes the homomorphic operation for the commitment.
- $\mathsf{HVC.OpenHom}_{crs_n}(\Lambda_j, \Lambda_j' \in \mathcal{P}) \to \Lambda_j''$: This algorithm can be run by any user who holds two membership proofs Λ_j and Λ_j' for some message on position j w.r.t. to some C and C'' (which contains m and m' as the message at position j), and it allows the user to compute another proof $\Lambda_j'' = \Lambda_j \otimes \Lambda_j' \in \mathcal{P}$ (w.r.t. some C'' which contains m'' as the new message at position j), where $\otimes$ denotes the homomorphic operation for the proof.

Basically, a HVC scheme should satisfy correctness, conciseness and homomorphic property.

Correctness. A vector commitment is correct if for all honestly generated $crs_n \leftarrow \mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n)$, $\forall i \in [n]$, if C is a commitment on a vector $(m_1, \cdots, m_n) \in \mathcal{M}^n$, Λ_i is a proof for position i generated by $\mathsf{HVC.Open}_{crs_n}$, then $\mathsf{HVC.Ver}_{crs_n}(C, m_i, i, \Lambda_i)$ outputs 1 with overwhelming probability.

Conciseness. A vector commitment is concise if the size of the commitment C and the outputs of $\mathsf{HVC.Open}$ are both independent of the size n of the vector.

Homomorphic property. Formally, $\forall i \in [n]$, for all honestly generated $crs_n \leftarrow \mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n)$, for all honestly generated

$$(C, aux) \leftarrow \mathsf{HVC.ComHom}_{crs_n}(\mathbf{m}), (C', aux') \leftarrow \mathsf{HVC.ComHom}_{crs_n}(\mathbf{m}'),$$

$$\Lambda_i \leftarrow \mathsf{HVC.Open}_{crs_n}(i, \mathbf{m}, aux), \Lambda'_i \leftarrow \mathsf{HVC.Open}_{crs_n}(i, \mathbf{m}', aux'),$$

where $\mathbf{m} = (m_1, \ldots, m_n), \mathbf{m}' = (m'_1, \ldots, m'_n)$, if

$$C'' \leftarrow \mathsf{HVC.ComHom}_{crs_n}(C, C'), \Lambda''_i \leftarrow \mathsf{HVC.OpenHom}_{crs_n}(\Lambda_i, \Lambda'_i)$$

then we have $\mathsf{HVC.Ver}_{crs_n}(C'', m_i + m'_i, i, \Lambda''_i) = 1$.

4.2 Security Models

In this section, we formally define the security models for binding and hiding on the situation that the corresponding membership proofs are leaked.

Position-Binding: It requires that for any well-formed commitment, the *PPT* adversary cannot find two different messages on the same position that the verification algorithm accepts both. Formally, we have the HVC Position-Binding game as Figure 5.

```
$\mathsf{Exp}^{\mathsf{position\text{-}binding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n)$
$crs_n \leftarrow \mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n)$
$(C, i, m_i, m'_i, \Lambda_i, \Lambda'_i) \leftarrow \mathcal{A}_1(crs_n)$
if $m_i \neq m'_i \wedge \mathsf{HVC.Ver}_{crs_n}(C, m, i, \Lambda_i) = 1 \wedge \mathsf{HVC.Ver}_{crs_n}(C, m', i, \Lambda'_i) = 1$
return 1,    else return 0
```

Fig. 5. The game of HVC Position-Binding

The advantage of $\mathcal{A}$ is defined as

$$\mathsf{Adv}^{\mathsf{position\text{-}binding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n) = \Pr[\mathsf{Exp}^{\mathsf{position\text{-}binding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n) = 1].$$

Definition 6. *A HVC scheme satisfies HVC Position-Binding if for every PPT adversary $\mathcal{A}$ the advantage function* $\mathsf{Adv}_{\mathit{HVC},\mathcal{A}}^{\mathit{position\text{-}binding}}(1^\lambda, \mathcal{M}, n)$ *is negligible in* λ.

Position-Hiding: The position hiding property not only requires that the adversary cannot distinguish whether a commitment is for a vector $(m_1, \ldots, m_n)$ or $(m'_1, \ldots, m'_n)$, but also guarantees that the adversary cannot learn any information about m_i from the opening of m_j where $i \neq j$. Formally, we have the HVC Position-Hiding game as Figure 6.

```
$\mathsf{Exp}_{\mathsf{HVC},\mathcal{A}}^{\mathsf{position\text{-}hiding}}(1^\lambda, \mathcal{M}, n, b)$
$crs_n \leftarrow \mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n)$
$(i, m_1, \cdots, m_{i-1}, m_{i,0}, m_{i,1}, m_{i+1}, \cdots m_n, state) \leftarrow \mathcal{A}_1(crs_n)$
$\mathbf{m}_b = (m_1, \ldots, m_{i-1}, m_{i,b}, m_{i+1}, \ldots, m_n)$
$(C, aux) \leftarrow \mathsf{HVC.com}(\mathbf{m}_b)$
$\Lambda_j \leftarrow \mathsf{HVC.open}(j, \mathbf{m}_b, aux), \forall j \in [n]/\{i\}$
$\forall j \in [n]/\{i\}, \Lambda_i \leftarrow VC.open(i, m_{i,b}, aux)$
$b' \leftarrow \mathcal{A}_2(crs_n, C, \{\Lambda_j\}_{j \in [n] \setminus \{i\}}, state),$
**return** $b'$
```

Fig. 6. The game of HVC Position-Hiding

The advantage of $\mathcal{A}$ is defined as

$$\mathsf{Adv}_{\mathsf{HVC},\mathcal{A}}^{\mathsf{position\text{-}hiding}}(1^\lambda, \mathcal{M}, n) =$$
$$\left|\Pr[\mathsf{Exp}_{\mathsf{HVC},\mathcal{A}}^{\mathsf{position\text{-}hiding}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}_{\mathsf{HVC},\mathcal{A}}^{\mathsf{position\text{-}hiding}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right|$$

Definition 7. *A HVC scheme satisfies HVC Position-Hiding if for every PPT adversary $\mathcal{A}$ the advantage function* $\mathsf{Adv}_{\mathit{HVC},\mathcal{A}}^{\mathit{position\text{-}hiding}}(1^\lambda, \mathcal{M}, n)$ *is negligible in* λ.

4.3 Construction

Although there are several vector commitment (VC) constructions [8,16,2,7] that satisfy the homomorphic property, none of them can directly satisfy both the position hiding and homomorphic properties simultaneously. Furthermore, they cannot be made position hiding through the composition approach. For example, if one first commits to each message using a standard commitment scheme and then applies a VC to the resulting sequence of commitments, the resulting hybrid scheme will not satisfy the homomorphic property.

Even if one first commits to each message separately using a homomorphic commitment scheme (such as Peterson commitment) and then applies a VC construction to the obtained sequence of commitments, the compatibility of the algebraic structures of these two underlying primitives is still unclear.[6] Some existing VC schemes are based on bilinear maps [8,16], or RSA groups [8,16,2,7], or lattice assumptions [19]. However, for pairing-based or RSA-based VC constructions, the messages (commitment values themselves in the above composition construction) are encoded into the exponents of group elements, which restricts the operation on messages to addition. This means that we require a homomorphic commitment scheme where the committed values lie in an additive group. Unfortunately, the existence of such a commitment scheme is elusive as most of the well-known computational assumptions do not hold, making it unclear how to construct such a scheme. For example, the homomorphic operation for Peterson commitment is multiplication, making it incompatible with pairing-based or RSA-based VC constructions for obtaining an HVC. Similarly, the message space of lattice-based VC consists of short vectors, while the standard lattice commitment consists of pseudorandom ring elements. It is also challenging to directly combine a lattice-based VC with a lattice-based commitment scheme to obtain an HVC.

Our proposed HVC construction is based on the pairing-based VC scheme introduced by Catalano et al. [8], which already possesses homomorphic properties and position binding. To achieve position hiding without compromising the homomorphic property, we add a dummy position at the end of the vector, which is used to store a random value. The random value is then used to mask the information about the membership proof, thus achieving position hiding. Since the dummy position is not used for any actual file, it does not affect the integrity of the VC scheme. With this approach, we can achieve both homomorphic properties and position hiding in our HVC scheme, which is essential for our proposed construction in an USS system.

Let $\mathbb{G}$, $\mathbb{G}_T$ be bilinear groups of prime order p equipped with a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. Let $g \in \mathbb{G}$ be random generators.

- $\mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n) \to crs_n$: Randomly choose $z_1, \ldots, z_n, z_{n+1} \leftarrow\$ \mathbb{Z}_p$. For all $i = 1, \ldots, n+1$, set $h_i = g^{z_i}$, For all $i, j = 1, \ldots, n+1$, $i \neq j$ set $h_{i,j} = g^{z_i z_j}$. Output $crs_n = \big(g, \{h_i\}_{i \in [n+1]}, \{h_{i,j}\}_{i,j \in [n+1], i \neq j}\big)$.
- $\mathsf{HVC.Com}_{crs_n}(\mathbf{m} = (m_1, \ldots, m_n)) \to (C, aux)$: Randomly select $r \leftarrow\$ \mathbb{Z}_p$, Compute $C = h_1^{m_1} h_2^{m_2} \cdots h_n^{m_n} \cdot h_{n+1}^r$ and output C and the auxiliary information $aux = r$
- $\mathsf{HVC.Open}_{crs_n}(\mathbf{m}, i, aux) \to \Lambda_i$: Compute

$$\Lambda_i = \prod_{j=1, j \neq i}^{n} h_{i,j}^{m_j} \cdot h_{i,n+1}^r = \left(\prod_{j=1, j \neq i}^{n} h_j^{m_j} \cdot h_{n+1}^r \right)^{z_i}$$

- $\mathsf{HVC.Ver}_{crs_n}(C, m, i, \Lambda_i) \to 1/0$: Check $e(C/h_i^m, h_i) = e(\Lambda_i, g)$.

[6] Some recently proposed functional commitment schemes [20,11] may also satisfy similar security requirements of HVC. However, it is unclear how to make these schemes compatible with UE schemes and thus integrate them into our proposed USS construction.

– $\mathsf{HVC.ComHom}_{crs_n}(C, C' \in \mathcal{C}) \rightarrow C''$: Compute $C'' = C \cdot C'$.
– $\mathsf{HVC.OpenHom}_{crs_n}(\Lambda_j, \Lambda'_j \in \mathcal{P}) \rightarrow \Lambda''_j$:Compute $\Lambda''_j = \Lambda_j \cdot \Lambda'_j$.

The correctness and homomorpihc property of the scheme can be easily verified by inspection. We prove its security via the following theorem.

Theorem 1. *If the Square-CDH Assumption holds, then the scheme defined above satisfies the Position-Binding property.*

Proof. We prove the theorem by showing that the scheme satisfies the Position-Binding property. For sake of contradiction assume that there exists an efficient adversary $\mathcal{A}$ who produces two valid openings to two different messages at the same position, then we show how to build an efficient algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to break the Square-CDH Assumption.

To break the Square-CDH Assumption, $\mathcal{B}$ takes as input $g, g^a \in \mathbb{G}$ and its goal is to compute g^{a^2}.

First, $\mathcal{B}$ selects a random $i \leftarrow\$ [n]$ as a guess for the index i on which $\mathcal{A}$ will break the position binding. And set $h_i = g^a$ Next, $\mathcal{B}$ chooses $z_j \leftarrow\$ \mathbb{Z}_p$, $\forall j \in [n+1]\backslash\{i\}$ and it computes:

$$\forall j \in [n+1]\backslash\{i\} : h_j = g^{z_j}, h_{i,j} = h_i^{z_j} = g^{az_j}$$

$$\forall k, j \in [n+1]\backslash\{i\}, k \neq j : h_{k,j} = g^{z_k z_j}$$

and outputs $crs_n = (g, \{h_j\}_{j\in[n+1]}, \{h_{j,k}\}_{j,k\in[n+1],j\neq k})$. Notice that the public parameters are perfectly distributed as the real ones. The adversary is supposed to output a tuple $(C, m, m', \Lambda, \Lambda')$ such that: $m \neq m'$ and both Λ and Λ' correctly verify at position i. If the position is not i, then $\mathcal{B}$ aborts the simulation. Otherwise, it computes $g^{a^2} = (\Lambda/\Lambda')^{(m'-m)^{-1}}$.

To see that the output is correct, observe that since the two openings verify correctly, then it holds: $e(C, h_i) = e(h_i^{m'}, h_i)e(\Lambda', g) = e(h_i^m, h_i)e(\Lambda, g)$. Notice that if $\mathcal{A}$ succeeds with probability ϵ, then $\mathcal{B}$ has probability ϵ/n of breaking the Square-CDH assumption. □

Theorem 2. *The scheme defined above is perfectly position hiding.*

Proof. We prove the theorem by showing that for two given vectors of messages $\mathbf{m}_0 = \{m_1, \ldots, m_{i-1}, m_{i,0}, m_{i+1}, \ldots, m_n\}$ and $\mathbf{m}_1 = \{m_1, \ldots, m_{i-1}, m_{i,1}, m_{i+1}, \ldots, m_n\}$, we can find two random values r_0 and r_1 such that $(\mathbf{m}_0, r_0)$ and $(\mathbf{m}_1, r_1)$ map to the same commitment value C and same proofs $\{\Lambda_j\}_{j\in[n]\backslash\{i\}}$ except Λ_{ib}. Since r_0, r_1 are chosen with equal probabilities according to the commitment algorithm $\mathsf{HVC.Com}$, any adversary $\mathcal{A}$ has a success to win the Position-Hiding game with a probability of exactly 1/2.

Concretely, the challenger $\mathcal{C}$ sets the public parameters as the real environment: randomly choose $z_1, \ldots, z_n, z_{n+1} \leftarrow\$ \mathbb{Z}_p$. For all $i = 1, \ldots, n+1$, set $h_i = g^{z_i}$. For all $i, j = 1, \ldots, n+1$, $i \neq j$ set $h_{i,j} = g^{z_i z_j}$.

Upon receiving $\{m_1, \ldots, m_{i-1}, m_{i,0}, m_{i,1}, m_{i+1}, \ldots, m_n\}$ from $\mathcal{A}$, $\mathcal{C}$ randomly chooses r_0, computes commitment $C = h_1^{m_1} \cdot h_{i-1}^{m_{i-1}} \cdot h_i^{m_{i,0}} \cdot h_{i+1}^{m_{i+1}} \cdots h_n^{m_n} \cdots h_{n+1}^{r_0}$, and proofs $\Lambda_j = (C/{h_j}^{m_j})^{z_j}, j \in [n]\backslash\{i\}$. Obviously, $(C, \{\Lambda_j\}_{j\in[n]\backslash\{i\}})$ is the corresponding commitment and opennings to $(\mathbf{m}_0, r_0)$. $\mathcal{C}$ outputs $(C, \{\Lambda_j\}_{j\in[n]\backslash\{i\}})$. Note that if we set $r_1 = r_0 + (m_{i,0} - m_{i,1})z_i/z_{n+1}$, then the tuple $(C, \{\Lambda_j\}_{j\in[n]\backslash\{i\}})$ mentioned above can also serve as a commitment and openings for $(\mathbf{m}_1, r_1)$.

Since r_0 is randomly chosen, both r_0 and r_1 occur with equal probability. Therefore, the probability for $\mathcal{A}$ to win the Hiding game is exactly 1/2. □

5 Construction of USS

In this section, we present our construction for achieving both confidentiality and integrity in an USS system. Our approach combines UE for confidentiality and VC for integrity. Specifically, we follow the VC-then-UE paradigm, where each file is treated as an element of VC, and its membership proof is appended at the end of the file. The file and its membership proof are then encrypted using UE schemes.

The main challenge in this approach is that updating one file changes all other membership proofs, which are encrypted together with the files using UE. However, general UE does not support file updates, which means that all storage must be retrieved, decrypted, and re-encrypted after each update. To reduce communication and user computation costs, we require a UE with homomorphic properties. To our knowledge, only the scheme RISE [17] satisfies this requirement and is compatible with the update operation of the membership proofs of our HVC scheme in Section 4.

More precisely, let UE be any IND-ENC and IND-UPD secure updatable encryption scheme. RISE [17] is an IND-ENC and IND-UPD secure updatable encryption with homomorphic property described in Subsection 2.3. Let COM be a standard commitment scheme with the hiding and binding property, and HVC be a homomorphic vector commitment with the position hiding and position binding property.

– $\mathsf{USS.ParGen}(1^\lambda, \mathcal{M}, n)$: λ is the security parameter. $\mathcal{M}$ denotes the message space. n specifies the vector degree and the total number of stored files.
  - Run the setup of UE and RISE to generate public parameter $ue.pp, rise.pp$.
  - Let $hvc.crs_n$ be the public parameter of HVC.
  - Let $com.pp$ be the public parameter of COM.

  Then the public parameter $pp = (hvc.crs_n, ue.pp, rise.pp, com.pp)$ will be taken as the implicit input of the following algorithm.
– $\mathsf{USS.KeyGen}(pp)$: take the public parameter pp as input, run the key generation algorithm of UE and RISE to generate the secret key $ue.sk, rise.sk$, and output the secret key $sk = (ue.sk, rise.sk)$.
– $\mathsf{USS.Store}(\mathbf{m}, sk, pp)$: $\mathbf{m} = \{m_1, \dots, m_n\}$, where each m_i denotes one file. The algorithm proceeds as follows:
  1. For each $i \in [n]$, randomly sample $r_i \leftarrow\!\!\$ \{0,1\}^\lambda$, run $\mathsf{COM}(m_i; r_i) \to h_i$.
  2. For each $i \in [n]$, run $\mathsf{UE.Enc}(ue.sk, i\|m_i\|h_i\|r_i) \to \bar{f}_i$, where $\|$ denotes concatenation in this paper. Run $\mathsf{HVC.Com}(\mathbf{h})$ to get the vector commitment C and the auxiliary input aux, where the message vector is $\mathbf{h} = (h_1, \dots, h_n)$.

3. For each $i \in [n]$, compute the proof Λ_i via running $\mathsf{HVC.Open}(i, \mathbf{h}, aux)$.
4. For each $i \in [n]$, run $\mathsf{RISE.Enc}(rise.sk, \Lambda_i) \to \hat{f}_i$.
5. Let $f_i = (\bar{f}_i, \hat{f}_i)$. Upload the total ciphertexts $\mathbf{f} = (f_1, \ldots, f_n)$ to the cloud storage service. Client stores the stub $sb = C$ and the secret key sk for the current epoch in the local storage.

– $\mathsf{USS.Rev}_{client}(i, sk, sb, pp) \leftrightarrows \mathsf{USS.Rev}_{server}(\mathbf{f}, sb, pp)$: The client interacts with the server to retrieve the i-th file through the following procedure.
  - $\mathsf{Rev}_{request}(i, sk, sb, pp) \to (q_{rev}, st_{rev})$: The client sends $q_{rev} = i$ to the server, where $i \in [n]$ and keeps a state $st_{rev} = (\text{“}retrieve\text{”}, i)$.
  - The server holds the public parameter pp, and the storing file $\mathbf{f}$. When given the retrieve request q_{rev}, the server returns the q_{rev}-th file $f_{q_{rev}}$ as the response r_{rev}.
  - $\mathsf{Rev}_{decrypt}(sk, sb, pp, st_{rev}, r_{rev}) \to m_i/\perp$: When given the server's response r_{rev}, the client parses the $sk = (ue.sk, hue.uk)$ and $f_i = (\bar{f}_i, \hat{f}_i)$. Run UEdecryption algorithm $\mathsf{UE.Dec}\left(ue.sk, \bar{f}_i\right) \to i \| m_i \| h_i \| r_i$. Run RISE decryption algorithm $\mathsf{RISE.Dec}\left(rise.sk, \hat{f}_i\right) \to \Lambda_i$.
  - If the commitment verification $\mathsf{Com.Open}(com_i, m_i, r_i) \to 1$ and the homomorphic vector commitment verification $\mathsf{HVC.Ver}(C, h_i, i, \Lambda_i) \to 1$, then the client will output m_i, otherwise output $\perp$.

– $\mathsf{USS.FileUp}_{client}(m'_i, i, sk, sb, pp) \leftrightarrows \mathsf{USS.FileUp}_{server}(\mathbf{f}, pp)$: is an interactive procedure that allows the client to update the i-th file m_i to m'_i with the collaboration of the server. More precisely, the interaction procedure is as follows.
  1. $\mathsf{FileUp}_{request}\,(m'_i, i, sk, sb, pp) \to q_{fup}, st_{fup}$: The client sends the file update request $q_{fup} = (\text{“}FileUpdate\text{”}, i) = st_{fup}$ to the server to request the i-th encrypted file and keep a state st_{fup}.
  2. $\mathsf{FileUp}_{response}\,(\mathbf{f}, pp, q_{fup}) \to (f_i, sr_{fup})$: The server returns the i-th encrypted file f_i to the client as the response r_{fup} and keep the internal state $sr_{fup} = (\text{“}FileUpdate\text{”}, i)$.
  3. $\mathsf{FileUp}_{token}(sk, sb, pp, m'_i, f_i, st_{fup}) \to (sb', tk_{fup})$: The client first parses $sk = (ue.sk, hue.sk)$ and $f_i = (\bar{f}_i, \hat{f}_i)$. Then run UE decryption algorithm $\mathsf{UE.Dec}(ue.sk, \bar{f}_i) \to i \| m_i \| h_i \| r_i/\perp$, and RISE decryption algorithm $\mathsf{RISE.Dec}\left(rise.sk, \hat{f}_i\right) = \Lambda_i/\perp$. If both decryptions are not $\perp$, then run the following verification algorithms. If the commitment opens to the different file $\mathsf{COM.open}(h_i; r_i) \neq m_i$, or the homomorphic vector commitment verification $\mathsf{HVC.Ver}(sb, h_i, i, \Lambda_i) \neq 1$, then output $\perp$, otherwise compute the following procedures.

     The client replaces the plaintext file with m'_i, samples a randomness $r'_i \leftarrow\$ \{0,1\}^\lambda$, and commits it by running $\mathsf{COM}(m'_i; r'_i) \to h'_i$. Encrypt the new file with $\mathsf{UE.Enc}(ue.sk, i \| m'_i \| h'_i \| r'_i) \to \bar{f}'_i$. Set the change of message vector $\mathbf{m}_\delta = (0, \ldots, \delta_i = h'_i - h_i, \ldots, 0)$, and get homomorphic vector commitment $C_{\mathbf{m}_\delta}$ via running $\mathsf{HVC.Com}(\mathbf{m}_\delta) = (C_{\mathbf{m}_\delta}, aux)$. Then update the stub sb by running the vector commitment homomorphic algorithm $\mathsf{HVC.ComHom}(sb, C_{\mathbf{m}_\delta}) = sb'$.

The client keeps the new stub sb' and sends the file update token $tk_{fup} = (\bar{f}'_i, \mathbf{m}_\delta, aux, y = g^{rise.sk})$ to the server. Please note that the change of message vector $\mathbf{m}_\delta$ could be compressed to a constant size independent of the vector degree since it contains redundant 0 with $n-1$ degrees.

4. $\mathsf{FileUp}_{update}\,(\mathbf{f}, pp, sr_{fup}, tk_{fup}) \to \mathbf{f}'$: On receiving $tk_{fup} = (\bar{f}'_i, \mathbf{m}_\delta, aux, y)$ from the client, parse $\mathbf{f} = \Big((\bar{f}_1, \hat{f}_1), \ldots, (\bar{f}_n, \hat{f}_n)\Big)$. For all $j \in [n]$, the server will run $\mathsf{HVC.Open}(j, \mathbf{m}_\delta, aux) = \Lambda_{\delta_j}$, get RISE ciphertext $rise.C_{\delta_j} = (y^r, g^r \cdot \Lambda_{\delta_j})$, and get updated proof encryption

$$\hat{f}'_j = \hat{f}_j \cdot rise.C_{\delta_j} = \mathsf{RISE.Enc}(rise.sk, \Lambda_j) \cdot rise.C_{\delta_j} = \mathsf{RISE.Enc}(rise.sk, \Lambda_j \cdot \Lambda_{\delta_j})$$

(The last equation is because of the homomorphic property of RISE). Then the updated ciphertexts are $\mathbf{f}' = (f'_1, \ldots, f'_n)$, where $f'_j = (\bar{f}_j, \hat{f}'_j)$ for $j \in [n] \setminus i$ and $f'_i = (\bar{f}'_i, \hat{f}'_i)$.

– $\mathsf{USS.KeyUp}_{client}(sk, sk', sb, pp) \leftrightarrows \mathsf{USS.KeyUp}_{server}(\mathbf{f}, pp)$: The interactive procedure between the client and the server updates the stored files to a new key sk'. The details are as follows.

1. $\mathsf{KeyUp}_{token}(sk, sk', sb, pp) \to (tk, sb')$: The client first parses $sk = (ue.sk, rise.sk)$, $sk' = (ue.sk', rise.sk')$. Then get a homomorphic commitment on $\mathbf{0}$ via $\mathsf{HVC.Com}(\mathbf{0}) = (C_\mathbf{0}, aux)$, and re-randomize the stub via running $\mathsf{HVC.ComHom}(sb, C_\mathbf{0}) \to sb'$. Run UE token generation algorithm $\mathsf{UE.Next}(ue.sk, ue.sk') \to \Delta$ to generate the UE key update token Δ. Run RISE token generation algorithm $\mathsf{RISE.Next}(rise.sk, rise.sk') \to rise.\Delta$ to generate the RISE key update token $rise.\Delta = (rise.\Delta_1, y)$. Send $tk = (\Delta, rise.\Delta, \mathbf{0}, aux)$ as the key update token for the repository. The stub is updated as sb'. Please note that $\mathbf{0}$ could be compressed into constant size, so tk is still succinct.
2. $\mathsf{KeyUp}_{update}(\mathbf{f}, pp, tk) \to (\mathbf{f}')$: Server parses $tk = (\Delta, rise.\Delta, \mathbf{0}, aux)$, $rise.\Delta = (rise.\Delta_1, y)$, and $\mathbf{f} = (f_1, \ldots, f_n)$, where $f_i = (\bar{f}_i, \hat{f}_i)$ for $i \in [n]$. For each $i \in [n]$, run UE re-encryption algorithm $\mathsf{UE.Upd}(\Delta, \bar{f}_i) \to \bar{f}'_i$ to update the data part. For each $i \in [n]$, run RISE re-encryption algorithm $\mathsf{RISE.Upd}(rise.\Delta_1, \hat{f}_i) \to \hat{f}'_i$ to re-encrypt the proof part. Then for $i \in [n]$, the server runs $\mathsf{HVC.Open}(i, \mathbf{0}, aux) = \Lambda_{\mathbf{0}_i}$, get RISE its encryption $rise.C_i = (y^r, g^r \cdot \Lambda_{\mathbf{0}_i})$ and to re-randomize the updated proof encryption

$$\hat{f}''_i = \hat{f}'_i \cdot rise.C_i = \mathsf{RISE.Enc}(rise.sk', \Lambda_i \cdot \Lambda_{\mathbf{0}_i}).$$

(The last equation is because of the homomorphic property of RISE) Finally, the updated ciphertexts are $\mathbf{f}' = (f''_1, \ldots, f''_{n-1})$, where $f''_i = \Big(\bar{f}'_i, \hat{f}''_i\Big)$ for $i \in [n]$.

Instantiation. In the USS construction, the HVC is instantiated in section 4, the COM scheme could be instantiated with any secure commitment scheme with hiding and binding property, and the UE could be instantiated with any IND-ENC and IND-UPD secure UE schemes [17]. We know that in USS, the membership proof of HVC is encrypted by the RISE encryption algorithm. We require that the homomorphism of HVC and RISE is compatible.

5.1 Security Proofs

We formally state the security properties of the construction USS in the following theorems and prove our theorems.

Theorem 3. *If UE is an IND-ENC secure updatable encryption scheme, COM is a secure commitment scheme with hiding property, then our USS is IND-DD-UP secure.*

Proof. As IND-DD-UP security is defined in Definition 2, we need to prove that

$$\left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right| = neg(\lambda).$$

We prove our claim via a game sequence.

Game$_0$. This game is same as the experiment $\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$. We define G_0 to be the event that $\mathcal{A}$ outputs 1 in Game$_0$. So,

$$\Pr[G_0] = \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1].$$

Game$_1$. In this game, the challenger $\mathcal{C}$ modifies the behavior of $\mathsf{Store}(db_0, sk_e, pp)$ in response to the challenge, compared to **Game$_0$**. Specifically, for each $i \in \{1, \ldots, n\} \wedge m_{0,i} \neq m_{1,i}$, $\mathcal{C}$ randomly selects $(m_{r,i}, r_{r,i}, h_{r,i})$ from the corresponding message, randomness, and commitment spaces and replaces the invocation of $\mathsf{UE.Enc}(i \| m_{0,i} \| h_{0,i} \| r_{0,i}) \to \bar{f}_i$ with $\mathsf{UE.Enc}(i \| m_{r,i} \| h_{r,i} \| r_{r,i}) \to \bar{f}_{r,i}$. Correspondingly, in rep^*, $\bar{f}_i$ is replaced with $\bar{f}_{r,i}$. Additionally, $\mathcal{C}$ records $h_{0,i}$ for file update queries. All other operations, including answering the challenge and post-processing, remain the same as **Game** $_0$.
Specifically, $\mathcal{C}$ still uses the same $h_{0,i}$ to run the HVC algorithm to generate the stub and openings when answering the challenge. If the file update query $\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$ is related to file $m_{0,i}$, i.e., $sb = sb^* \wedge i \in \mathcal{I}$, $\mathcal{C}$ skips the UE decryption and COM verification steps. Instead, $\mathcal{C}$ retrieves the record $h_{0,i}$ as the decrypted commitment to perform the HVC verification and calculates the change of commitment $\delta_i = h'_i - h_{0,i}$ as a part of the file update transcript ft_i, which will be added to set $\mathcal{I}$. All other operations are identical to those in **Game$_0$**.
Let G_1 be the event that $\mathcal{A}$ outputs 1 in Game$_1$. We claim that

$$|\Pr[G_1] - \Pr[G_0]| = \epsilon_{ind-enc},$$

where $\epsilon_{ind-enc}$ is the IND-ENC advantage of the UE scheme (which is negligible if UE is IND-ENC secure). This claim can be proven by observing that in Game $_0$, $\bar{f_i}$ is a UE encryption of message $i\|m_{0,i}\|h_{0,i}\|r_{0,i}$, while in Game $_1$, it is a UE encryption of message $i\|m_{r,i}\|h_{r,i}\|r_{r,i}$. Since UE is IND-ENC secure, the adversary $\mathcal{A}$ cannot distinguish between the two games.

Game$_2$. In this game, the challenger $\mathcal{C}$ introduces a small modification when running Store to answer the challenge. Specifically, for each $i \in \{1, \ldots, n\} \wedge m_{0,i} \neq m_{1,i}$, instead of using $\mathsf{COM}(m_{0,i}; r_{0,i})$ to generate $h_{0,i}$, $\mathcal{C}$ replaces $h_{0,i}$ with a commitment $h_{1,i} \leftarrow \mathsf{COM}(m_{1,i}; r_{1,i})$ to message $m_{1,i}$. Then, $\mathcal{C}$ uses $h_{1,i}$ to run the HVC algorithm to obtain sb^*, rep^*, and update the set $\mathcal{L}$. Finally, the record for $h_{0,i}$ is replaced with $h_{1,i}$.
Let G_2 be the event that $\mathcal{A}$ outputs 1 in Game $_2$. We claim that $|\Pr[G_2] - \Pr[G_1]| \leq \epsilon_{hiding}$, where ϵ_{hiding} is the hiding-advantage of the COM scheme (which is negligible if COM is a secure commitment scheme with hiding property). This claim can be proven by observing that in Game $_1$, $\mathcal{A}$ obtains the commitment $h_{0,i}$ to message $m_{0,i}$, while in Game $_2$, $\mathcal{A}$ obtains the commitment $h_{1,i}$ to message $m_{1,i}$. However, due to the hiding property of COM, $\mathcal{A}$ cannot distinguish between the two games.

Game$_3$. In this game, challenger $\mathcal{C}$ replaces the UE encryption on the random message to the encryption on a tuple of $i\|m_{1,i}, \|h_{1,i}\|r_{1,i}$ where $h_{1,i}$ is generated from $\mathsf{COM}(m_{1,i}; r_{1,i})$ as in Game$_2$. Then Game$_3$ is the same as experiment $\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$.
We define G_3 to be the event that $\mathcal{A}$ outputs 1 in Game$_3$. Then we claim

$$|\Pr[G_3] - \Pr[G_2]| = \left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] - \Pr[G_2]\right| \leq \epsilon_{ind-enc}$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of UE scheme (which is negligible if UE is IND-ENC secure).
The proof of this claim is identical to the claim $|\Pr[G_1] - \Pr[G_0]| = \epsilon_{ind-enc}$. Thus, noticing the difference between Game$_3$ and Game$_2$ is in negligible probability due to the IND-ENC security of UE.

Then we get

$$\left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] - \mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1]\right| \leq 2\epsilon_{ind-enc} + \epsilon_{hiding}$$

□

Theorem 4. *If UE is an IND-ENC secure updatable encryption scheme, COM is a secure commitment scheme with hiding property, then our USS is IND-FileUp-CPA secure.*

Proof. As IND-FileUp-CPA security is defined in Definition 3, we need to prove that $\left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right| = neg(\lambda)$. We prove the IND-FileUp-CPA security via a sequence of games.

Game$_0$. This game is same as the experiment $\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$. We define G_0 to be the event that $\mathcal{A}$ outputs 1 in Game$_0$. So,

$$\Pr[G_0] = \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1].$$

Game$_1$. In this game, challenger $\mathcal{C}$ changes a bit from **Game$_0$** in running the $\mathsf{FileUp}_{token}(sk_e, sb, m_{0,i}, i, pp) \leftrightarrows \mathsf{FileUp}_{server}(sb, rep, pp) \rightarrow \langle sb^*; rep^* \rangle$ to answer the challenge. Concretely, for challenge index $i \in \mathcal{I}$, $\mathcal{C}$ randomly chooses $(m_{r,i}, r_{r,i}, h_{r,i})$ from the corresponding message, randomness, and commitment spaces and replaces running $\mathsf{UE.Enc}(i \| m_{0,i} \| h_{0,i} \| r_{0,i}) \rightarrow \bar{f}_i^*$ with running $\mathsf{UE.Enc}(i \| m_{r,i} \| h_{r,i} \| r_{r,i}) \rightarrow \bar{f}_{r,i}^*$. Correspondingly, in rep^*, replace the value of $\bar{f}_i^*$ with $\bar{f}_{r,i}^*$. $\mathcal{C}$ still use $h_{0,i}$ to continue the FileUp procedure and the i-th element of $\mathbf{m}_\delta$ is still $h_{0,i} - h_i$ in the file update transcript fpt. Besides, $\mathcal{C}$ record $h_{0,i}$ for file update query. Other operations including answering the challenge and the post-processing remain the same as **Game$_0$**. Specifically, $\mathcal{C}$ still uses the same $h_{0,i}$ to run the HVC algorithm to generate the stub and openings in answering the challenge. If the file update query $\mathcal{O}.\mathsf{FileUp}(sb, m_i', i)$ is related to file $m_{0,i}$, i.e., $sb = sb^* \wedge i \in \mathcal{I}$, $\mathcal{C}$ retrieves e, sb, rep, db from $\mathcal{L}$, changes as follows in running FileUp procedure. $\mathcal{C}$ gets rid of UE decryption and verification check on $\bar{f}_i$, retrieves the record $h_{0,i}$ to calculate the change of commitment $\delta_i = h_i' - h_{0,i}$ as a part of file update transcript ft_i which will be added to set $\mathcal{I}$. Then follow the same operations to continue as in Game$_0$.

We define G_1 to be the event that $\mathcal{A}$ outputs 1 in Game$_1$. We claim that

$$|\Pr[G_1] - \Pr[G_0]| = \epsilon_{ind-enc}$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of UE scheme (which is negligible if UE is IND-ENC secure).

The proof of this claim is essentially the observation that in Game$_0$ $\bar{f}_i^*$ is a UE encryption on message $i \| m_{0,i} \| h_{0,i} \| r_{0,i}$, while in Game$_1$, it is a UE encryption on the message $i \| m_{r,i} \| h_{r,i} \| r_{r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since UE is IND-ENC secure.

Game$_2$. In this game, challenger $\mathcal{C}$ makes one small change to the above game. Specifically, during running FileUp procedure to answer the challenge. For the challenge index $i \in \mathcal{I}$, instead of running $\mathsf{COM}(m_{0,i}; r_{0,i})$ to generate $h_{0,i}$, $\mathcal{C}$ replaces $h_{0,i}$ with a commitment $h_{1,i} \leftarrow \mathsf{COM}(m_{1,i}; r_{1,i})$ to message $m_{1,i}$ to run the HVC algorithm to get sb^*, rep^* and update the set $\mathcal{L}$. Then the record on $h_{0,i}$ is changed to $h_{1,i}$.

We define G_2 to be the event that $\mathcal{A}$ outputs 1 in Game$_2$. We claim that

$$|\Pr[G_2] - \Pr[G_1]| = \epsilon_{hiding}$$

where ϵ_{hiding} is the hiding-advantage of COM scheme (which is negligible if COM is a secure commitment scheme with hiding property).

The proof of this claim is essentially the observation that in Game$_1$, $\mathcal{A}$ could get the commitment $h_{0,i}$ to message $m_{0,i}$, while in Game$_2$, $\mathcal{A}$ could get the commitment $h_{1,i}$ to message $m_{1,i}$. So $\mathcal{A}$ cannot notice the difference due to the hiding property of COM.

Game$_3$. In this game, challenger $\mathcal{C}$ replaces the UE encryption on the random message to the encryption on a tuple of $i\|m_{1,i}, \|h_{1,i}\|r_{1,i}$ where $h_{1,i}$ is generated from $\mathsf{COM}(m_{1,i}; r_{1,i})$ as in Game$_2$. Then Game$_3$ is the same as experiment $\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$.
We define G_3 to be the event that $\mathcal{A}$ outputs 1 in Game$_3$. Then we claim

$$|\Pr[G_3] - \Pr[G_2]| = \left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] - \Pr[G_2]\right| \leq \epsilon_{ind-enc}$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of UE scheme (which is negligible if UE is IND-ENC secure).
The proof of this claim is identical to the claim $|\Pr[G_1] - \Pr[G_0]| = \epsilon_{ind-enc}$. Thus, noticing the difference between Game$_3$ and Game$_2$ is in negligible probability due to the IND-ENC security of UE.

Then we get

$$\left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] - \mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1]\right| \leq 2\epsilon_{ind-enc} + \epsilon_{hiding}$$

□

Theorem 5. *If UE is an IND-UPD secure updatable encryption scheme, RISE is an IND-ENC and IND-UPD secure updatable encryption scheme with homomorphic property, and HVC is a secure vector commitment with homomorphic property and position hiding, then our USS is IND-REENC-CPA secure.*

Proof. We prove the IND-REENC-CPA security via a sequence of games. As IND-REENC-CPA security is defined in Definition 4, we need to prove that $\left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right| = neg(\lambda)$. On the high level, we start from $\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$ as the original game, and make one small change in each of a sequence of games, and end up at $\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$ as the final game. We reduce the advantage of distinguishing between every two adjacent games to the advantage of breaking the security of one of the building blocks including IND-UPD security of UE, IND-UPD security of RISE, IND-ENC security of RISE, and the hiding property of HVC. Since the building blocks satisfy the security requirements, the advantage of distinguishing every two adjacent games is negligible. There are a constant number of games, and the summed advantage is still negligible, so the first game and last game are indistinguishable, which means the probability difference that the adversary outputs the same is negligible.

Game$_0$. This game is same as $\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$. We define G_0 to be the event that $\mathcal{A}$ outputs 1. Then we get

$$\Pr[G_0] = \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1].$$

Game$_1$. In this game, challenger $\mathcal{C}$ changes a bit from **Game$_0$** in running $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb^*; rep^* \rangle$.

Concretely, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ randomly chooses $m_{r,i}, h_{r,i}, r_{r,i}$ from the message space, commitment space and randomness space, respectively, where $|m_{r,i}| = |m_{0,i}|$. Then run $\mathsf{UE.Enc}(sk_{e-1}, i\|m_{r,i}\|h_{r,i}\|r_{r,i}) \rightarrow \bar{f}_{r,i}$ and replaces $c_{0,i} = (\bar{f}_{0,i}, \hat{f}_{0,i})$ in rep_0 with $c_{r,i} = (\bar{f}_{r,i}, \hat{f}_{0,i})$ as input to run the KeyUp procedure. Since the key update version of $\bar{f}_{r,i}$ will be used in answering the $\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$ query when $sb = sb^* \wedge i \in \mathcal{I}$, challenger $\mathcal{C}$ will change the behavior as follows to reply such query. In the FileUp procedure, $\mathcal{C}$ gets rid of the decryption and integrity check of retrieved ciphertext, directly commits and encrypts on the new message m'_i and uses $h_{0,i}$, calculates the change of i-th commitment $\delta_i = h'_i - h_{0,i}$ where $h'_i \rightarrow \mathsf{COM}(m'_i; r'_i)$ to get the file update token and continue. Since the stub sb is generated using $h_{0,i}$ and binds with its vector commitment proof in the second part of i-th ciphertext, the same $h_{0,i}$ is used to update file to ensure the file updated version is a valid ciphertext, consistent with Game$_0$. $\mathcal{A}$ cannot notice the difference via corrupting both the epoch key and ciphertext of the file's updated version.

We define G_1 to be the event that $\mathcal{A}$ outputs 1 in Game$_1$. We claim that

$$|\Pr[G_1] - \Pr[G_0]| \leq \epsilon_{ind-upd}$$

where $\epsilon_{ind-upd}$ is the IND-UPD-advantage of UE scheme (which is negligible if UE is IND-UPD secure).

The claim can be proven by observing that in Game $_0$, $rep^* = \{c^*_{0,1}, \ldots, c^*_{0,n}\}$, where $c^*_{0,i} = (\bar{f}^*_{0,i}, \hat{f}^*_{0,i})$, includes a UE re-encryption of ciphertext $\bar{f}_{0,i}$ that encrypts message $i\|m_{0,i}\|h_{0,i}\|r_{0,i}$, while in Game $_1$, $c^*_{0,i}$ includes a UE re-encryption of ciphertext $\bar{f}_{r,i}$ that encrypts message $i\|m_{r,i}\|h_{r,i}\|r_{r,i}$. Since UE is IND-UPD secure, the adversary $\mathcal{A}$ should not be able to distinguish between the two games.

Game$_2$ This game, makes one small change from Game$_1$ to answer the challenge. Concretely, in $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb^*; rep^* \rangle$, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ randomly chooses $\Lambda_{r,i}$ from the vector commitment proof space to replace $\Lambda_{0,i} \leftarrow \mathsf{RISE.Dec}(sk_{e-1}, \hat{f}_{0,i})$, encrypts it $\mathsf{RISE.Enc}(sk_{e-1}, \lambda_{r,i}) \rightarrow \hat{f}_{r,i}$, replaces $\hat{f}_{0,i}$ with $\hat{f}_{r,i}$ as input to run the KeyUp procedure. Since the key update version of $\hat{f}_{r,i}$ will be used in answering the $\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$ query when $sb = sb^* \wedge i \in \mathcal{I}$, challenger $\mathcal{C}$ will change the behavior as follows to reply such query. In the FileUp procedure, when updating $\hat{f}_{r,i}$, $\mathcal{C}$ addtionally run $\mathsf{RISE.Enc}(sk_e, \Lambda_{0,i}/\Lambda_{r,i}) \leftarrow rise.C_\Delta$, and replace $\hat{f}^*_{r,i}$ with $\hat{f'}^*_{r,i} = \hat{f}^*_{r,i} \cdot rise.C_\Delta$ to answer the query. Since the stub sb is generated using $h_{0,i}$ and binds with its vector commitment proof $\Lambda_{0,i}$, while the second part of i-th ciphertext in set $\mathcal{L}$ is related to $\Lambda_{r,i}$ due to this game's change, $\mathcal{C}$ recover the i-th ciphertext to base on $\Lambda_{0,i}$ by running $\hat{f'}^*_{r,i} = \hat{f}^*_{r,i} \cdot rise.C_\Delta$, which is consistent with Game$_1$. $\mathcal{A}$ cannot notice the difference via corrupting both the epoch key and ciphertext of the file's updated version.

We define G_2 to be the event that $\mathcal{A}$ outputs 1 in Game$_2$. We claim that

$$|\Pr[G_2] - \Pr[G_1]| \leq \epsilon'_{ind-upd}$$

where $\epsilon'_{ind-upd}$ is the IND-UPD-advantage of RISE scheme (which is negligible if RISE is IND-UPD secure).
The proof of this claim is essentially the observation that in Game$_1$ $rep^* = \{c^*_{0,1}, \dots, c^*_{0,n}\}$ where $c^*_{0,i} = (\bar{f}^*_{0,i}, \hat{f}^*_{0,i})$ includes a RISE re-encryption of ciphertext $\hat{f}_{0,i}$ which encrypts proof $\Lambda_{0,i}$, while in Game$_2$, $c^*_{0,1}$ include a RISE re-encryption of ciphertext $\hat{f}_{r,i}$ which encrypts the proof $\Lambda_{r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since RISE is IND-UPD secure.

Game$_3$. This game, makes one small change from Game$_2$ to answer the challenge. Specifically, in $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb^*; rep^* \rangle$, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ randomly chooses $\Lambda_{\delta_r,i}$ from the vector commitment proof space to replace $\Lambda_{\mathbf{0}_i}$ to continue the KeyUp procedure. Since the challenge ciphertext with one change in this game can be updated to a non-challenge ciphertext if $\mathcal{A}$ queries $\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$ where $sb = sb^* \wedge i \in \mathcal{I}$. The non-challenge ciphertext could be decrypted if $\mathcal{A}$ corrupts the epoch key, which is allowed. To make $\mathcal{A}$'s view the same as in Game$_2$, challenger $\mathcal{C}$ will change the behavior as follows to reply to such query. In the FileUp procedure, when updating $\hat{f}_{r,i}$, $\mathcal{C}$ addtionally run $\mathsf{RISE.Enc}(sk_e, \Lambda_{\mathbf{0}_i}/\Lambda_{\delta_r,i}) \leftarrow rise.C'_\Delta$, and replace $\hat{f'}^*_{r,i}$ with $\hat{f''}^*_{r,i} = \hat{f'}^*_{r,i} \cdot rise.C'_\Delta$ to answer the query. Since the stub sb binds with its vector commitment proof $\Lambda_{\mathbf{0}_i}$, while the second part of i-th ciphertext in set $\mathcal{L}$ is related to $\Lambda_{\delta_r,i}$ due to this game's change, $\mathcal{C}$ recover the i-th ciphertext to base on $\Lambda_{\mathbf{0}_i}$ by running $\hat{f''}^*_{r,i} = \hat{f'}^*_{r,i} \cdot rise.C'_\Delta$, which is consistent with Game$_2$. $\mathcal{A}$ cannot notice the difference via corrupting both the epoch key and ciphertext of the file's updated version.
We define G_3 to be the event that $\mathcal{A}$ outputs 1 in Game$_3$. We claim that

$$|\Pr[G_3] - \Pr[G_2]| \leq \epsilon_{ind-enc}$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of RISE scheme (which is negligible if RISE is IND-ENC secure).
The proof of this claim is essentially the observation that in Game$_2$ $rep^* = \{c^*_{0,1}, \dots, c^*_{0,n}\}$ where $c^*_{0,i} = (\bar{f}^*_{0,i}, \hat{f}^*_{0,i})$ includes a RISE encryption of proof $\Lambda_{\mathbf{0}_i}$, while in Game$_3$, $c^*_{0,1}$ include a RISE encryption of the proof $\Lambda_{\delta_r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since RISE is IND-ENC secure.

Game$_4$. This game makes one small change of the stub of challenge ciphertext from Game$_3$ to answer the challenge. Specifically, in running KeyUp procedure to output $\langle sb^*; rep^* \rangle$, for each $i \in \{1, \dots, n\}$, challenger $\mathcal{C}$ runs $\mathsf{UE.Dec}(sk_{e-1}, \bar{f}_{0,i}) \rightarrow (i \| m_{0,i} \| h_{0,i} \| r_{0,i})$ and $\mathsf{UE.Dec}(sk_{e-1}, \bar{f}_{1,i}) \rightarrow (i \| m_{1,i} \| h_{1,i} \| r_{1,i})$ and gets $\mathbf{m}_{\delta_{0-1}} = (h_{1,1} - h_{0,1}, \dots, h_{1,n} - h_{0,n})$. Replace $\mathbf{0}$ with $\mathbf{m}_{\delta_{0-1}}$ to run $\mathsf{HVC.Com}(\mathbf{m}_{\delta 0-1}) \rightarrow (C_{\mathbf{m}_{\delta 0-1}}, aux_\delta)$, updates stub $sb'^* = sb^* \cdot C_{\mathbf{m}_{\delta 0-1}}$, and generates key update token $tk^* = (\Delta^*, rise.\Delta^*, \mathbf{m}_{\delta 0-1}, aux_\delta)$ to continue the KeyUp procedure. In this game each value of $h_{0,i}, \Lambda_{0,i}$ are changed to

$h_{1,i}, \Lambda_{1,i}$. If $i \notin \mathcal{I}$, which means $c_{0,i} = c_{1,i}$, also means $h_{0,i} = h_{1,i}$, the i-th ciphertext is a valid ciphertext that could be decrypted correctly with the corrupted epoch key and does not leak information about $h_{1,j}$ with other index j.
We define G_4 to be the event that $\mathcal{A}$ outputs 1 in Game$_4$. We claim that

$$|\Pr[G_4] - \Pr[G_3]| = \epsilon_{position-hiding}$$

where $\epsilon_{position-hiding}$ is the position-hiding-advantage of HVC scheme (which is negligible if HVC is a secure vector commitment scheme with position hiding and homomorphic property).
The proof of this claim is essentially the observation that in Game$_3$ sb^* is vector commitment corresponding to $rep_0 = \{c_{0,1}, \ldots, c_{0,n}\}$, while in Game$_4$, sb^* is a vector commitment corresponding to $rep_1 = \{c_{1,1}, \ldots, c_{1,n}\}$. For those indices i satisfying $c_{0,i} \neq c_{1,i}$, the vector commitment and other indices' message do not leak information about the i-th element. So the adversary $\mathcal{A}$ should not notice the difference since HVC is position hiding.

Game$_5$ This game makes one small change on Game$_4$, which is a reverse change of Game$_3$ or getting rid of the change in Game$_3$. Specifically, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ changes $\Lambda_{\delta_r,i}$ back to the proof generated from $\mathsf{HVC.Open}$ to continue the KeyUp procedure. We omit the details for simplicity. Please refer to Game$_3$ for details.
We define G_5 to be the event that $\mathcal{A}$ outputs 1 in Game$_5$. We claim that

$$|\Pr[G_5] - \Pr[G_4]| \leq \epsilon_{ind-enc}$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of RISE scheme (which is negligible if RISE is IND-ENC secure).
The proof of this claim is identical to the observation of Game$_3$ that in Game$_5$ $\hat{f}^*_{1,i})$ includes a RISE encryption of proof $\Lambda_{\mathbf{1}_i}$, while in Game$_4$, $\hat{f}^*_{r,i})$ include a RISE encryption of the proof $\Lambda_{\delta_r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since RISE is IND-ENC secure.

Game$_6$ This game makes one small change on Game$_5$, which is a reverse change of Game$_2$ or getting rid of the change in Game$_2$. Specifically, in running $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb^*; rep^* \rangle$, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ replace $\Lambda_{r,i}$ with $\Lambda_{1,i} \leftarrow \mathsf{RISE.Dec}(sk_{e-1}, \hat{f}_{1,i})$, encrypts it $\mathsf{RISE.Enc}(sk_{e-1}, \lambda_{1,i}) \rightarrow \hat{f}_{1,i}$, replaces $\hat{f}_{r,i}$ with $\hat{f}_{1,i}$ as input to run the KeyUp procedure. We omit the details for simplicity. Please refer to Game$_2$ for details.
We define G_6 to be the event that $\mathcal{A}$ outputs 1 in Game$_6$. We claim that

$$|\Pr[G_6] - \Pr[G_5]| \leq \epsilon'_{ind-upd}$$

where $\epsilon'_{ind-upd}$ is the IND-UPD-advantage of RISE scheme (which is negligible if RISE is IND-UPD secure).
The proof of this claim is identical to the observation of Game$_2$. That is in Game$_6$ $rep^* = \{c^*_{0,1}, \ldots, c^*_{0,n}\}$ where $c^*_{0,i} = (\bar{f}^*_{0,i}, \hat{f}^*_{1,i})$ includes a RISE re-encryption of ciphertext $\hat{f}_{1,i}$ which encrypts proof $\Lambda_{1,i}$, while in Game$_5$, $c^*_{0,1}$

include a RISE re-encryption of ciphertext $\hat{f}_{r,i}$ which encrypts the proof $\Lambda_{r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since RISE is IND-UPD secure.

Game$_7$ This game makes one small change on Game$_6$, which is a reverse change of Game$_1$ or getting rid of the change in Game$_1$. Concretely, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ replaces $\bar{f}_{r,i}$ with $\bar{f}_{1,i}$ where $c_{1,i} = (\bar{f}_{1,i}, \hat{f}_{1,i})$ in rep_1, as input to run the Keyup procedure. We omit the details for simplicity. Please refer to Game$_1$ for details.

We define G_7 to be the event that $\mathcal{A}$ outputs 1 in Game$_7$. Game$_7$ is the same as $\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$. So $\Pr[G_7] = \Pr[\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]$. We claim that

$$|\Pr[G_7] - \Pr[G_6]| \leq \epsilon_{ind-upd}$$

where $\epsilon_{ind-upd}$ is the IND-UPD-advantage of UE scheme (which is negligible if UE is IND-UPD secure).

The proof of this claim is identical to the observation of Game$_1$. That is, in Game$_7$ $rep^* = \{c^*_{1,1}, \ldots, c^*_{1,n}\}$ where $c^*_{1,i} = (\bar{f}^*_{1,i}, \hat{f}^*_{1,i})$ includes a UE re-encryption of ciphertext $\bar{f}_{1,i}$ which encrypts message $i \| m_{1,i} \| h_{1,i} \| r_{1,i}$, while in Game$_6$, $c^*_{0,1}$ include a UE re-encryption of ciphertext $\bar{f}_{r,i}$ which encryptps the message $i \| m_{r,i} \| h_{r,i} \| r_{r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since UE is IND-UPD secure.

Then we get

$$\begin{aligned}&\left|\Pr[\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] - \mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1]\right|\\ \leq& 2\epsilon_{ind-upd} + 2\epsilon'_{ind-upd} + 2\epsilon_{ind-enc} + \epsilon_{position-hiding}\end{aligned}$$

□

Theorem 6. *Let USS denote an updatable secure storage scheme. COM is secure commitment with binding property, and HVC is a secure vector commitment with position binding, then USS is OF-PTXT secure.*

Proof. Intuitively, we first assume that USS is not OF-PTXT secure, and then construct contradictions with the existing properties of building blocks to prove the theorem. In the $\mathsf{Exp}^{\text{of-ptxt}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n)$ experiment, given the stub sb_e and the epoch key sk_e, to win the game, $\mathcal{A}$ needs to provide a ciphertext f_i and interact with challenger running $\mathsf{USS.Rev}$ procedure and enable the challenger to retrieve a file m'_i which is different from the i-th element m_i of the latest message vector $\mathbf{m}$ corresponding to the stub sb_e.

Since we assume the USS is not OF-PTXT secure, then there exists $\mathcal{A}$ winning the experiment $\mathsf{Exp}^{\text{of-ptxt}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n)$ by enabling the file $m'_i \neq m_i$ retrieval. That means the commitment of m'_i passes the verification algorithm of HVC. So $\mathcal{A}$ could win in two cases: one case is $\mathcal{A}$ finds a collision for the commitment h_i, i.e., $\mathsf{COM}(m'_i; r'_i) = \mathsf{COM}(m_i; r_i) = h_i$, which is contradictory with the binding property of COM; the other case is that $\mathcal{A}$ finds the collision for the HVC, i.e., two different elements $h_i \neq h'_i$ pass the i-th vector commitment verification for the same stub sb_e, which is contradictory with HVC's position-binding property. So we can reduce USS's OF-PTXT property to COM's binding property and HVC's position binding.

□

6 Future works

Although Updatable Encryption (UE) is a promising approach for secure storage of data, it cannot be directly applied to frequently updated databases due to the risk of a malicious server inducing the client to accept an outdated version of a file instead of the latest one. To address this issue, we propose a scheme called Updatable Secure Storage (USS) that provides a secure and key-rotatable solution for dynamic databases. However, we acknowledge that the USS scheme presented in this paper is still far from practical due to its high computational and communication overheads. Therefore, we suggest several directions for future work to improve the efficiency and usability of the USS scheme.

General construction for general UE. In this paper, we show a construction built on ciphertext-independent UE (CIUE) schemes as the UE syntax we present in the paper describes. This construction can be extended to more general ones, for ciphertext-dependent UE (CDUE). From the construction point of view, the extension could be easy. While there is some subtlety in the security model to be compatible with both CIUE and CDUE, CDUE models are more fine-grained to give the adversary more flexibility to corrupt tokens. It takes more care to formally model and analyze that existing CDUE schemes are secure under some security models that are applicable to CDUE and could be black-box used for USS security proof, which should be true but has never been formalized and proved before.

Towards Full-fledged version control. In this paper, we achieve the first step of version control: ensuring the integrity of the latest version of the database, but do not support a fall-back function. The fall-back function enables users to fall back on the database to any prior version. This requires the database to store all the history versions or the history changes. One simple idea to enable the fall-back function to a limited number of history versions is to pre-config a bunch of versions for one file with empty or meaningless data and to update the corresponding version when the user updates. The stub contains the information of all the versions so that malicious servers cannot manipulate any version to deceive users. During each key update, all versions of ciphertext get key rotation so that the update history is hidden from external attackers.

Acknowlegements. We thank anonymous reviewers from ASIACRYPT'23 for their valuable comments. Long Chen was supported by the National Key R&D Program of China 2022YFB3102500 and the CAS Project for Young Scientists in Basic Research Grant YSBR-035. Hui Guo was supported by the National Natural Science Foundation of China (Grant Nos. 61802021, 62022018, 61932019).

References

1. Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In *Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31*, pages 111–131. Springer, 2011.
2. Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.
3. Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. Improving speed and security in updatable encryption schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 559–589. Springer, 2020.
4. Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic prfs and their applications. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 410–428, 2013.
5. Colin Boyd, Gareth T Davies, Kristian Gjøsteen, and Yao Jiang. Fast and secure updatable encryption. In *Annual International Cryptology Conference*, pages 464–493. Springer, 2020.
6. Martin Bradley and Alexander Dent. Payment card industry data security standard.
7. Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Vector commitment techniques and applications to verifiable decentralized storage. *IACR Cryptol. ePrint Arch.*, 2020:149, 2020.
8. Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.
9. Long Chen, Yanan Li, and Qiang Tang. Cca updatable encryption against malicious re-encryption attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 590–620. Springer, 2020.
10. Miranda Christ and Joseph Bonneau. Limits on revocable proof systems, with applications to stateless blockchains. *IACR Cryptol. ePrint Arch.*, page 1478, 2022.
11. Leo de Castro and Chris Peikert. Functional commitments for all functions, with transparent setup and from sis. In *Advances in Cryptology–EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III*, pages 287–320. Springer, 2023.
12. Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. Key rotation for authenticated encryption. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, pages 98–129, 2017.
13. Payment Card Industry. Data Security Standard. Requirements and Security Assessment Procedures. Version 3.2 PCI Security Standards Council (2016).
14. Michael Klooß, Anja Lehmann, and Andy Rupp. (r) cca secure updatable encryption with integrity protection. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 68–99. Springer, 2019.
15. Michael Klooß, Anja Lehmann, and Andy Rupp. (R)CCA secure updatable encryption with integrity protection. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of*

Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I, pages 68–99, 2019.
16. Russell WF Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *Annual International Cryptology Conference*, pages 530–560. Springer, 2019.
17. Anja Lehmann and Björn Tackmann. Updatable encryption with post-compromise security. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, pages 685–716, 2018.
18. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, 1991.
19. Chris Peikert, Zachary Pepin, and Chad Sharp. Vector and functional commitments from lattices. In *Theory of Cryptography Conference*, pages 480–511. Springer, 2021.
20. Hoeteck Wee and David J Wu. Succinct vector, polynomial, and functional commitments from lattices. In *Advances in Cryptology–EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III*, pages 385–416. Springer, 2023.

A Security Models for UE

Here, we present a revisit of the security models IND-ENC and IND-UPD for ciphertext-independent updatable encryption as proposed in [17]. In these models, the encryption key evolves with the epochs. In addition to the challenge ciphertext and the encryption oracle, the adversary is permitted to obtain keys from certain epochs. This is done to reflect the scenario where the client's keys are leaked. Furthermore, the adversary is capable of obtaining some previous versions of the challenge ciphertexts and update tokens, which captures the situation where previous storage on the server may not have been securely erased in time.

The challenger initializes a UE scheme with global state (k_e, Δ_e, S, e) where $k_0 \leftarrow \mathsf{UE.Setup}(1^\lambda), \delta_0 \leftarrow \perp$, and $e \leftarrow 0$, and S consists of initially empty sets $\mathcal{L}, \tilde{\mathcal{L}}, \mathcal{C}, \mathcal{K}$ and $\mathcal{T}$. Furthermore, let $\tilde{e}$ denote the challenge epoch, and e_{end} denote the final epoch in the game.

- $\mathcal{L}$: List of non-challenge ciphertexts (C_e, e) produced by calls to the $\mathcal{O}_{\mathsf{enc}}$ or $\mathcal{O}_{\mathsf{upd}}$ oracle. $\mathcal{O}_{\mathsf{upd}}$ only updates ciphertexts contained in $\mathcal{L}$.
- $\tilde{\mathcal{L}}$: List of updated versions of the challenge ciphertext. $\tilde{\mathcal{L}}$ gets initialized with the challenge ciphertext $(\tilde{C}, \tilde{e})$. Any call to the oracle $\mathcal{O}_{\mathsf{next}}$ oracle automatically updates the challenge ciphertext into the new epoch, which $\mathcal{A}$ can fetch via a $\mathcal{O}_{\mathsf{updC}}$ call.
- $\mathcal{C}$: List of all epochs e in which $\mathcal{A}$ learned an updated version of the challenge ciphertext.
- $\mathcal{K}$: List of all epochs e in which $\mathcal{A}$ corrupted the secret key k_e.
- $\mathcal{T}$: List of all epochs e in which $\mathcal{A}$ corrupted the update token Δ_e.

$\mathcal{O}_{\mathsf{enc}}(m)$: On input a message $m \in \mathbb{M}$, compute $C \leftarrow \mathsf{UE.Enc}(\mathsf{k_e}, \mathsf{m})$ where k_e is the secret key of the current epoch e. Add C to the list of ciphertexts $\mathcal{L} \leftarrow \mathcal{L} \cup \{(C, e)\}$ and return the ciphertext to the adversary.

$\mathcal{O}_{\mathsf{next}}$: Upon triggering, the oracle $\mathcal{O}_{\mathsf{next}}$ generates new epoch key by running $k_{e+1} \leftarrow \mathsf{UE.Keygen}(e+1)$, and generates a new update token Δ_{e+1} by invoking the function $\mathsf{UE.Next}(k_e, k_{e+1})$. The global state is then updated to $(k_{e+1}, \Delta_{e+1}, S, e+1)$. If a challenge query has been previously made, this operation also updates the challenge ciphertext to the new epoch. Specifically, it executes $C_{e+1} \leftarrow \mathsf{UE.Upd}(\Delta_{e+1}, C_e)$ for each $(C_e, e) \in \tilde{\mathcal{L}}$ and adds the resulting pair $(C_{e+1}, e+1)$ to the set $\tilde{\mathcal{L}} \cup \{(C_{e+1}, e+1)\}$.

$\mathcal{O}_{\mathsf{upd}}(C_{e-1})$: On input ciphertext C_{e-1}, check that $(C_{e-1}, e-1) \in \mathcal{L}$ (i.e., it is a valid ciphertext from the previous epoch $e-1$), compute $C_e \leftarrow \mathsf{UE.Upd}(\Delta_e, C_{e-1})$, add (C_e, e) to the list $\mathcal{L}$, and output C_e to $\mathcal{A}$.

$\mathcal{O}_{\mathsf{corrupt}}(\{\text{token}, \text{key}\}, e^*)$: This oracle models adaptive corruption of the host and owner keys, respectively. The adversary can request a key or update token from the current epoch or any of the previous epochs.

- Upon input token, $e^* \leq e$, the oracle returns Δ_{e^*}, i.e., the update token is leaked. Calling the oracle in this mode sets $\mathcal{T} \leftarrow \mathcal{T} \cup \{e^*\}$.
- Upon input key, $e^* \leq e$, the oracle returns k_{e^*}, that is, the secret key is leaked. Calling the oracle in this mode sets $\mathcal{K} \leftarrow \mathcal{K} \cup \{e^*\}$.

$\mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}$: Returns the current challenge ciphertext C_e from $\tilde{\mathcal{L}}$. Note that the challenge ciphertext gets updated to the new epoch by the O_{next} oracle, whenever a new key is generated. Calling this oracle sets $\mathcal{C} \leftarrow \mathcal{C} \cup \{e\}$.

Extended sets. Since in RISE both the key update and ciphertext update are bi-directional, we define the extended sets $\mathcal{C}^*$ and $\mathcal{K}^*$ as follows.

Recall that $\mathcal{K}^*$ denotes the set of epochs in which the adversary has obtained the secret key:

$$\begin{aligned}\mathcal{K}^* \leftarrow \{e \in \{0, \ldots, e_{\text{end}}\} \mid \text{ corrupt-key } (e) = \text{ true }\}\\ \text{and true } \leftarrow \text{ corrupt-key } (e) \text{ iff:}\\ (e \in \mathcal{K}) \vee (\text{ corrupt-key } (e-1) \wedge e \in \mathcal{T})\\ \vee(\text{ corrupt-key } (e+1) \wedge e+1 \in \mathcal{T})\end{aligned}$$

Define the set $\mathcal{C}^*$ containing all challenge-equal epochs:

$$\begin{aligned}\mathcal{C}^* \leftarrow \{e \in \{0, \ldots, e_{\text{end}}\} \mid \text{ challenge-equal } (e) = \text{ true }\}\\ \text{and true } \leftarrow \text{ challenge-equal } (e) \text{ iff:}\\ (e = \tilde{e}) \vee (e \in \mathcal{C})\\ \vee (\text{ challenge-equal } (e-1) \wedge e \in \mathcal{T}^*)\\ \vee (\text{challenge-equal } (e+1) \wedge e+1 \in \mathcal{T}^*)\end{aligned}$$

Now, let's review the IND-ENC security notion. It ensures that ciphertexts obtained from the $\mathsf{UE.Enc}$ algorithm do not reveal any information about the underlying plaintexts even when $\mathcal{A}$ adaptively compromises a number of keys and tokens before and after the challenge epoch:

Definition 8 (IND-ENC). *An updatable encryption scheme* UE *is said to be IND-ENC secure if for any probabilistic polynomial-time adversary* $\mathcal{A}$ *it holds that* $\left|\Pr[\mathrm{Exp}_{\mathsf{UE},\mathcal{A}}^{\mathsf{ind\text{-}enc\text{-}cpa}}(1^\lambda, 0) = 1] - \Pr[\mathrm{Exp}_{\mathsf{UE},\mathcal{A}}^{\mathsf{ind\text{-}enc\text{-}cpa}}(1^\lambda, 1) = 1]\right|$ *is negligible in* λ.

```
$\mathsf{Exp}_{\mathsf{UE},\mathcal{A}}^{\mathsf{ind\text{-}enc\text{-}cpa}}(1^\lambda, b)$
$k_0 \leftarrow \mathsf{UE.Setup}(1^\lambda)$
$e \leftarrow 0;\ \tilde{e} \leftarrow \perp;\ \mathcal{L} \leftarrow \emptyset$
$(m_0, m_1, \mathrm{state}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{enc}}, \mathcal{O}_{\mathsf{next}}, \mathcal{O}_{\mathsf{upd}}, \mathcal{O}_{\mathsf{corrupt}}}(1^\lambda)$
proceed only if $|m_0| = |m_1|$
$\tilde{e} \leftarrow e$
$\tilde{C} \leftarrow \mathsf{UE.Enc}(k_{\tilde{e}}, m_b)\,,\ \tilde{\mathcal{L}} \leftarrow \{(\tilde{C}, \tilde{e})\}$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{enc}}, \mathcal{O}_{\mathsf{next}}, \mathcal{O}_{\mathsf{upd}}, \mathcal{O}_{\mathsf{corrupt}}, \mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}}(\mathrm{state})$
return $b'$   if $\mathcal{C}^* \cap \mathcal{K}^* = \emptyset$
```

Fig. 7. The game of IND-ENC for UE

Then, recall the IND-UPD security notion. It ensures that an updated ciphertext obtained from the $\mathsf{UE.Upd}$ algorithm does not reveal any information about the previous ciphertext, even when $\mathcal{A}$ adaptively compromises a number of keys and tokens before and after the challenge epoch adaptive manner:

Definition 9 (IND-UPD). *An updatable encryption scheme* UE *is said to be IND-UPD secure if for any probabilistic polynomial-time adversary* $\mathcal{A}$ *it holds* $\left|\Pr[\mathrm{Exp}_{\mathsf{UE},\mathcal{A}}^{\mathsf{ind-upd-cpa}}(1^\lambda, 0) = 1] - \Pr[\mathrm{Exp}_{\mathsf{UE},\mathcal{A}}^{\mathsf{ind-upd-cpa}}(1^\lambda, 1) = 1]\right|$ *is negligible in* λ.

```
$\mathrm{Exp}^{\mathsf{ind\text{-}upd\text{-}cpa}}_{\mathsf{UE},\mathcal{A}}(1^\lambda, b)$
$k_0 \leftarrow \mathsf{UE.Setup}(1^\lambda)$
$e \leftarrow 0;\ \tilde{e} \leftarrow \perp;\ \mathcal{L} \leftarrow \emptyset$
$(C_0, C_1, \mathrm{state}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{enc}}, \mathcal{O}_{\mathsf{next}}, \mathcal{O}_{\mathsf{upd}}, \mathcal{O}_{\mathsf{corrupt}}(1^\lambda)}$
 proceed only if $(C_0, \tilde{e}-1) \in \mathcal{L}$ and $(C_1, \tilde{e}-1) \in \mathcal{L}$ and $|C_0| = |C_1|$
$\tilde{e} \leftarrow e$
$\tilde{C} \leftarrow \mathsf{UE.Upd}\left(\Delta_{\tilde{e}}, C_b\right),\ \tilde{\mathcal{L}} \leftarrow \{(\tilde{C}, \tilde{e})\}$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{enc}}, \mathcal{O}_{\mathsf{next}}, \mathcal{O}_{\mathsf{upd}}, \mathcal{O}_{\mathsf{corrupt}}, \mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}}(\mathrm{state})$
**return** $b'$ **if** $\tilde{e} \notin \mathcal{T}^* \ \wedge\ \mathcal{C}^* \cap \mathcal{K}^* = \emptyset \wedge$ if UE.Upd is deterministic, then
 $\mathcal{A}$ has neither queried $\mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}(C_0)$ nor $\mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}(C_1)$ in epoch $\tilde{e}$
```

Fig. 8. The game of IND-UPD for UE