Privacy-Preserving Accountable Cloud Storage

Ka Yang
Iowa State University, yangka@iastate.edu

Jinsheng Zhang
Iowa State University, alexzjs@iastate.edu

Wensheng Zhang
Iowa State University, wzhang@iastate.edu

Daji Qiao
Iowa State University, daji@iastate.edu

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Data Storage Systems Commons

Recommended Citation
Yang, Ka; Zhang, Jinsheng; Zhang, Wensheng; and Qiao, Daji, "Privacy-Preserving Accountable Cloud Storage" (2015). Computer Science Technical Reports. 370.
http://lib.dr.iastate.edu/cs_techreports/370

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Privacy-Preserving Accountable Cloud Storage

Ka Yang*, Jinsheng Zhang†, Wensheng Zhang† and Daji Qiao*
* Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa 50010
† Department of Computer Science, Iowa State University, Ames, Iowa 50010

Abstract—In cloud storage services, a wide range of sensitive information may be leaked to the host server via the exposure of access pattern albeit data is encrypted. Many security-provable schemes have been proposed to preserve the access pattern privacy; however, they may be vulnerable to attacks towards data integrity or availability from malicious users. This is due to the fact that, preserving access pattern privacy requires data to be frequently re-encrypted and re-positioned at the storage server, which can easily conceal the traces that are needed for accountability support to detect misbehaviors and identify attackers. To address this issue, this paper proposes a scheme that integrates accountability support into hash-based ORAMs. Security analysis shows that the proposed scheme can detect misconduct committed by malicious users and identify the attackers, while preserving the access pattern privacy. Overhead analysis shows that the proposed accountability support incurs only slightly increased storage, communication, and computational overheads.

I. INTRODUCTION

Along with the increasing popularity of cloud-based data sharing, security and privacy concerns also arise. Although encrypting data content has been commonly used for data protection, it alone cannot eliminate the concerns completely. This is because users’ data access pattern is not preserved and researchers have found that a wide range of private information could be conveniently revealed by observing the access pattern [1]. To address this issue, more and more efficient designs [2]–[17] have been developed to implement oblivious RAM (ORAM) [18], which was originally proposed for software protection but also is a provable solution to data access pattern preservation.

Though initially designed for single-user scenarios, ORAM has also been applied to multi-user scenarios [14], [19]. However, these systems all assume that the outsourced data is accessible to the data owner and trusted users only; hence, they do not provide any protection against attacks from malicious users on data integrity or availability. Unfortunately, users may not always behave faithfully in practice. Thus, it is highly desirable to introduce accountability support into ORAM to provide privacy-preserving accountable cloud storage services. For example, a company may export its financial records to a cloud storage; while it may want to share the financial data with its stake holders, it is critical to protect the integrity of the data and hold a malicious user accountable if data is altered without authorization.

Many schemes [20], [21] have been proposed to provide accountability support in multi-user storage systems. However, these schemes cannot be readily applied to ORAM because of the following conflicting design goals: preserving access pattern privacy in ORAM requires data blocks to be frequently re-encrypted and re-positioned at the storage server, which can easily conceal the traces that are needed for accountability support to detect misbehaviors and identify attackers. In this paper, we propose a unique accountability solution for hash-based ORAMs such as the one proposed in [18]. It is capable of detecting misconduct by malicious users and identifying the attackers, while not interfering with the access pattern preservation mechanism inherent from the underlying ORAM. This goal is achieved via a creative application and integration of two well-known security techniques: Merkle hash tree [22] and group signature [23], as well as a delicate design of the data block signature. With our scheme, selected traces of accesses are properly recorded for the purpose of attack detection but without revealing the private information of innocent users that shall be kept confidential to protect their access pattern privacy. We have conducted both security analysis and overhead evaluation for the proposed scheme. Results show that our scheme has achieved the design goals of providing accountability support to ORAM and preservation of data access pattern privacy, at the cost of slightly increased storage, communication, and computational overheads.

The rest of the paper is organized as follows. Section II presents the related work. The system model is described in Section III and the proposed design is elaborated in Section IV. Sections V and VI report the security and overhead analysis. Finally, Section VII concludes the paper.

II. RELATED WORK

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [18], is a well-known technique to hide a user’s access pattern from an outsourced data storage server. In recent years, various schemes [2]–[17] have been proposed to improve the performance of ORAM from both theoretical and practical aspects in terms of communication overhead, storage overhead, and query latency.

While ORAM has become more practical in terms of performance, there are security issues that need to be addressed before ORAM can be deployed in real systems. There are only a few works that address these practical security concerns under the framework of ORAM. For example, in [24], an integrity verification scheme for path ORAM [13] was proposed to ensure data integrity against a malicious server with a Merkle hash tree solution. In another example, the problem of ORAM delegation was studied in [25]. It proposes a scheme with which the data owner can delegate controlled access to third parties for the outsourced dataset, while preserving the access pattern privacy. However, this scheme can only be applied to the square-root ORAM [18], which is inefficient in terms of communication overhead. As a third example, a write-once-read-many ORAM (WORM-ORAM) was proposed in [26], which enables the data owner to offer a read-only data service to third parties. It proposes a zero-knowledge proof based scheme to verify the integrity of the encrypted data...
in a write-once-read-many setting. But, it imposes expensive communication and computational overheads over the regular ORAM, which may limit its practical applications.

Different from these existing works, we study user accountability in this paper, which is an important security feature required by most data sharing services. We propose a low-overhead scheme to provide user accountability support to hash-based ORAM schemes [4]–[6], [9], [10], [18] without compromising the access pattern privacy provided by the underlying ORAM.

Private information retrieval (PIR) [27], [28] is another popular technique to preserve user’s access pattern. However, PIR protocols work on read-only data. In this paper, we consider a system where users can both read and write data.

### III. System Model and Design Goals

In this paper, we study a system where the owner of a dataset outsources data to a remote storage server and shares data with multiple users. Based on the access control requirement of the application, the owner may authorize different users with different access privileges. For each data item, we consider three types of access privileges: no access, read-only access, and write access. Moreover, in order to prevent leakage of access pattern privacy, the owner is assumed to follow a hash-based Oblivious RAM (ORAM) design to deploy the data on the storage server.

![System Architecture and Threat Model](image)

**A. Scheme Overview: The Intuitions**

Built on top of a hash-based ORAM such as [18], our proposed scheme has the ultimate goal of user accountability and data access pattern preservation, which is attained through accomplishing each of the following more specific subgoals:

- Data owner is trusted.
- Cloud server is honest but curious, and tries to reveal user’s private information, such as:
  - Data access frequency
  - Data access order
  - Data linkability

Users can be misbehaving and may attack data integrity or availability via unauthorized operations:
- Modification
- Replacement
- Misplacement

**G1** If the content of a data block is modified by an unauthorized user, such a modification attack shall be detected and the attacker shall be identified.

**G2** During a query or shuffling process, the set of data blocks that a user uploads to the server shall be the same as the set downloaded earlier. If a user fails to do so by replacing some of the data blocks with others, such a replacement attack shall be detected and the user shall be identified.

**G3** When being uploaded, data blocks shall be placed properly to the buckets decided by the underlying ORAM; that is, the position of each data block shall be determined by a designated hash function. If a user fails to do so, such a misplacement attack shall be detected and the user shall be identified.

**Accomplishment of Subgoal (G1)** In our scheme, subgoal (G1) is accomplished via group signatures. Initially, when exporting a data block to the server, the data owner generates two group signatures: $s$ – a group signature for the entire data block (on behalf of all the users in the system), and $g$ – a group signature for the content and ID of the data block only (on behalf of the users who have the write privilege to the
data block). When a user accesses a data block, it verifies both signatures. If $s$ is valid but $g$ is invalid, it is detected that the user who last accessed the data block has modified the data block without proper authorization, and the identity of the attacking user can be traced out by the data owner based on the group signature $s$. If both signatures are valid, the user proceeds to access the data and generates a new group signature $s$ before uploading the data block back to the server. Note that, if the user has the write privilege and has modified the data content, it also needs to generate a new group signature $g$ for the updated data content. At the server side, after it receives a data block from the user, it simply checks the group signature $s$. If it is invalid, the uploading user is detected to have committed a misconduct.

**Accomplishment of Subgoal (G2)** During a query or shuffling process, data blocks are re-encrypted and re-arranged. Therefore, it is challenging to ensure that the user always uploads the same set of data blocks that it downloaded from the server. In our scheme, this is accomplished via group signatures and Merkle hash tree. Specifically, after a user requests and downloads a set of data blocks, the server constructs a Merkle hash tree using the hash values of the IDs of the requested blocks as leaf nodes, and calculates the root hash of the tree. Then, the server informs the user of the root hash as well as the co-path information in the Merkle hash tree for each data block. After access to the data, the user updates the following information in each data block before uploading it back to the server: $c = \text{hash of the block ID with a new random nonce}$, $e = \text{root hash of the Merkle hash tree (informed by the server)}$, and $e' = \text{an encryption of the co-path information (also informed by the server)}$ with a new random nonce. Recall that the data content and block ID are protected by the group signature $g$.

This way, if a user fails to upload the same set of data blocks back to the server (by replacing some of the data blocks with others), such a replacement attack can be detected by the server or the user who next accesses the data block. For example, if a user replaces the content of a data block with that of another data block, together with block ID, or co-path information, or root hash, or a combination (but not all) of them, this can be detected by the user who next accesses the data block, since the root hash carried in the data block would be different from the one calculated using the block ID and co-path information. However, if the user replaces the entire data block, this can be detected by the server, since either the root hash of the data block is different from all other uploaded blocks, or the encrypted co-path information of the data block is identical to another uploaded block. Also note that the server cannot identify a data block using the hash of the block ID or the co-path information, because both hash and encryption operations use a new random nonce.

**Accomplishment of Subgoal (G3)** With the above solutions to subgoals (G1) and (G2) in place, the solution to subgoal (G3) becomes straightforward as follows. To ensure that a data block is placed properly according to the underlying ORAM, our scheme requires the user to include the position information (i.e., layer and bucket) as part of the block before generating the group signature $s$. Next time when another user accesses the block, it checks (i) whether the block was placed at the position specified in the data block, and (ii) whether the position is consistent with the output of the hash function designated by the underlying ORAM.

Next, we elaborate the details of our scheme which integrates the above solutions to subgoals (G1)-(G3).

**B. System Initialization**

System initialization is conducted by the owner of the dataset. It consists of four operations: selection of system parameters, user authorization, preparation of data blocks, and uploading of data blocks to the storage server. In the following, we describe these operations in detail. Particularly, when describing how to prepare and upload data blocks, the data format and storage structure will also be introduced.

1) **Selection of System Parameters:** As the first step of system initialization, the data owner classifies all data blocks into read groups and write groups according to the access policy. Specifically,

- a read group (denoted by $R_i$) is defined as the $i$th set of data blocks that have the same read access policy, i.e., they can be read by the same set of users;
- a write group (denoted by $W_j$) is defined as the $j$th set of data blocks that have the same write access policy, i.e., they can be written by the same set of users.

Each data block in the system belongs to exactly one read group and one write group. Let $n$ be the total number of data blocks exported to the storage server. Let $m_R$ and $m_W$ denote the number of read groups and write groups, respectively. We have $m_R \leq n$ and $m_W \leq n$, where the equality holds when every data block has a different read/write access policy. In practice, we usually have $m_R \ll n$ and $m_W \ll n$. Table I lists the system parameters.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_i$ ($W_j$)</td>
<td>the $i^{th}$ read (write) group</td>
</tr>
<tr>
<td>$m_R$ ($m_W$)</td>
<td>number of read (write) groups</td>
</tr>
<tr>
<td>$\mathcal{U}$</td>
<td>the set of all data users</td>
</tr>
<tr>
<td>$H_{R_i}$</td>
<td>the set of users that can read blocks in $R_i$</td>
</tr>
<tr>
<td>$H_{W_j}$</td>
<td>the set of users that can write blocks in $W_j$</td>
</tr>
<tr>
<td>$k$</td>
<td>a system symmetric key for all users</td>
</tr>
<tr>
<td>$k_{R_i} (j \geq 1)$</td>
<td>a symmetric key that is used to encrypt the plain-text content of all blocks in $R_i$</td>
</tr>
<tr>
<td>$k_{W_j} (j \geq 1)$</td>
<td>the group public key for write group $W_j$</td>
</tr>
<tr>
<td>$K_{u,W_j} (j \geq 1)$</td>
<td>a group private key assigned to user $u$ who can write all data blocks in $W_j$</td>
</tr>
<tr>
<td>$G_{u}\text{-}\text{Sig}_u(\cdot)$</td>
<td>group signature signed using $K_{u,W_j}$</td>
</tr>
<tr>
<td>$K_u$</td>
<td>the group public key for all data blocks</td>
</tr>
<tr>
<td>$K_{u,0}$</td>
<td>a group private key assigned to user $u \in \mathcal{U}$</td>
</tr>
<tr>
<td>$G_{u}\text{-}\text{Sig}_u(\cdot)$</td>
<td>group signature signed using $K_{u,0}$</td>
</tr>
<tr>
<td>$h_l(x,y)$</td>
<td>hash function that maps $x$ and $y \in {1, \ldots, n}$ to an integer in ${1, \ldots, 2^l}$ ($l = 1, \ldots, \log n$)</td>
</tr>
<tr>
<td>$H(\cdot)$</td>
<td>a general one-way hash function</td>
</tr>
<tr>
<td>$E_{\theta}(\cdot)$</td>
<td>a block cipher algorithm with key $\theta$</td>
</tr>
</tbody>
</table>

Let $\mathcal{U}$ denote the set of all users in the system. For each $R_i$ ($j \geq 1$), the data owner selects a symmetric key $k_{R_i}$ to encrypt the plain-text content of all data blocks in $R_i$. For each $W_j$ ($j \geq 1$), the owner also selects one group public key $(K_{W_j}^+)$ and a set of group private keys $(K_{u,W_j}^-)$, where $K_{u,W_j}^-$ denotes the private key assigned to user $u$. We
use $G_i\cdot\text{Sig}_u()$ to denote the group signature generated with $K_{u,W_i}^{-}$. In addition, the data owner also selects the following system parameters:

- $k$: a system symmetric key for general encryption purposes. It is known to all users in $U$.
- $K_0^+, \{K_{u,0}^-\}_{u \in U}$: one group public key ($K_0^+$) and a set of group private keys ($\{K_{u,0}^-\}_{u \in U}$), where $K_{u,0}^-$ denotes the private key assigned to user $u$. We use $G_0\cdot\text{Sig}_u()$ to denote the group signature generated with $K_{u,0}^-$.
- $h_l(x, y)$ for $l = 1, \cdots, \log n$: a set of hash functions, where each $h_l(x, y)$ hashes a pair of positive integers $x$ and $y \in \{1, \cdots, n\}$ to an integer in $\{1, \cdots, 2^l\}$. They are known to all users in $U$.
- $H(x)$: a hash function that randomly maps one or a sequence of integers to an $L$-bit integer, where $L$ is a security parameter.
- $E_{\theta}(\alpha, \beta)$: a symmetric block cipher algorithm working in CBC mode, where $\theta$ is the encryption key, $\alpha$ is the initialization vector, and $\beta$ is the plain-text to be encrypted. We also use $E_{\theta}(\alpha, \beta, \gamma)$ to denote the encryption of the concatenation of $\beta$ and $\gamma$ with the initialization vector $\alpha$ (i.e., the first parameter is always the initialization vector).

2) User Authorization: Among the parameters, $K_0^+$ and hash function $H(x)$ are available to the public. For each user $u \in U$, the owner provides the symmetric key $k$, a distinct group private key $K_{u,0}^-$, and the set of group public keys $K_{W_j}^+$. Let $U_{R_j}$ and $U_{W_j}$ denote the set of users who can read blocks in $R_j$ and the set of users who can write blocks in $W_j$, respectively. If a user $u$ is authorized to read data blocks in $R_j$, i.e., $u \in U_{R_j}$, the owner provides the symmetric key $k_{R_j}$ to the user; if a user $u$ is authorized to write data blocks in $W_j$, i.e., $u \in U_{W_j}$, the owner provides a group private key $K_{u,W_j}^-$ to the user.

Table II summarizes how the keys are distributed to the server and users. Basically, a user needs to store (i) $k$, $K_0^+$, and one group private key $K_{u,0}^-$ (O(1) overhead), (ii) a number of $k_{R_j}$ for the read groups that it is allowed to read (O($m_{R_j}$) overhead), (iii) a number of group private keys $K_{u,W_j}^-$ for the write groups that it is allowed to write (O($m_{W_j}$) overhead), (iv) $m_{W_j}$ group public keys $K_{W_j}^+$ (O($m_{W_j}$) overhead). If a misbehaving user is identified, the user could be revoked from the system with existing schemes such as [30], [31].

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>DISTRIBUTION OF KEYS TO THE SERVER AND USERS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \in U$</td>
<td>$k$</td>
</tr>
<tr>
<td>server</td>
<td>$\sqrt{}$</td>
</tr>
<tr>
<td>$u \in U_{R_j}$</td>
<td>$\sqrt{}$</td>
</tr>
<tr>
<td>$u \in U_{W_j}$</td>
<td>$\sqrt{}$</td>
</tr>
</tbody>
</table>

3) Preparation of Data Blocks: We use $i$ to denote the ID of a data block and use $D_{i,t_i}$ to denote the data block associated with ID $i$ and time stamp $t_i$. To simplify the presentation, when describing a data block, we use $k_i$ to denote the encryption key and use $K_{i,R_j}^+$ and $K_{u,i}$ to denote the group public key and private keys that are used by data block $D_{i,t_i}$. For example, if $D_{i,t_i} \in R_j$ and $D_{i,t_i} \in W_j$, we denote $k_i = k_{R_j}^+$, $K_i^+ = K_{W_j}^+$, and $K_{u,i}^- = K_{u,W_j}^-$.

The plain-text content carried in $D_{i,t_i}$ is denoted as $w_{i,v_i}$, where $v_i$ denotes the version number of the data content and is incremented when the content is updated. To enforce the access control policy, $w_{i,v_i}$ is encrypted together with its ID $i$ by its corresponding encryption key $k_i$. We use $d_{i,v_i}$ to denote the encrypted data content in $D_{i,t_i}$, i.e.,

$$d_{i,v_i} = E_{k_i}(r_{i,v_i}, v_i, w_{i,v_i})$$

where $r_{i,v_i}$ is a random initialization vector chosen upon each encryption of $w_{i,v_i}$. Overall, a data block $D_{i,t_i}$ has the following format:

$$D_{i,t_i} = (d_{i,t_i}^{(1)}, e_{i,t_i}, e_{i,t_i-1}, l_{i,t_i}, b_{i,t_i}, s_{i,t_i})$$

where the fields are explained below. A high-level explanation of the purpose of each data field is shown in Fig. 2, while detailed explanations about how these data fields are used will be presented in Section IV-C and Section V.

![Fig. 2](image-url)
include another random nonce \( \mu_i,t \), because it will be used as the initialization vector for another data field \( e'_i,t,1 \), which will be explained in the following. The purpose of the group signature \( g_{i,v},i \) is to protect the integrity of \( d'_{i,t,1} \), so that only a user who is authorized to write the data block (and hence owns the group private key \( K^*_u,i \)) can update \( r_{i,v},i;v,t_i \) and \( d_{i,v} \) legitimately. Meanwhile, it also enables the data owner to track the identity of the user who made the last modification.

\[
c_i,t_i = H(\mu_i,t_i, i, r_{i,v}').
\]

During each query or resuffling process, \( c_i,t_i \) is used by the server to build a Merkle hash tree based on the data blocks requested.

\[
e_i,t_i,1 = H(d_i,t_i,1, e_i,t_i,1, r_{i,v}').
\]

\( e_i,t_i,1 \) is the root hash of the Merkle hash tree that was constructed by the server when \( D_i,t_i \) was last accessed (for simplicity, we use \( t_i - 1 \) to denote the time stamp when \( D_i,t_i \) was last accessed). In other words, the leaf nodes of the Merkle hash tree are the ID hashes of all the blocks downloaded together with \( D_i,t_i,1 \). The purpose of \( e_i,t_i,1 \) (together with \( e'_i,t_i,1 \) which is explained below) is to let the user and the server collaboratively verify that, when a data block was previously accessed, the set of data blocks uploaded back to the server after the access was the same as the set downloaded.

\[
e'_i,t_i,1 = E_k(\mu_i,t_i,1, S_i,t_i,1).
\]

where \( S_i,t_i,1 \) denotes the co-path hash values for leaf hash \( c_i,t_i,1 \) on the Merkle hash tree when the data block was retrieved at time \( t_i - 1 \). Note that the same initialization vector \( \mu_i,t_i,1 \) is used in the encryption of \( e'_i,t_i,1 \) and \( d_i,t_i,1 \).

\( l_i,t_i \) and \( b_i,t_i \) are the layer and bucket of block \( D_i,t_i \) in the storage hierarchy respectively. \( S_i,t_i \) is a group signature of the entire block generated by the owner or any user in the system:

\[
s_i,t_i = G_0^*\text{Sig}_u(d_i,t_i,1, c_i,t_i,1, c_i,t_i,1, e'_i,t_i,1, l_i,t_i,1, b_i,t_i).
\]

Similar to \( g_{i,v},i \), the purpose of \( s_i,t_i \) is to allow the server and users to verify the integrity of the entire data block, as well as to enable the owner to track the identity of the user who last accessed the data block.

Initially, the time stamp \( t_i \) and the version number \( v_i \) are set to 0, both \( e_i,t_i,1 \) and \( e'_i,t_i,1 \) are set to empty, \( l_i,t_i \) is set to \( \log n \), and \( b_i,t_i \) is set to \( h_{\log n}(0, i) \).

4) Uploading of Data Blocks to the Storage Server: All the data blocks are stored to a hierarchy of layers according to the underlying ORAM. Specifically, the hierarchy consists of \( \log n \) layers. Each layer \( l \) (1 \( \leq l \leq \log n \)) includes 2^\( l \) buckets and each bucket contains \( \log n \) blocks. Hence, the store servers \( (2^n - 2) \log n \) data blocks in total, which includes \( n \) real data blocks that contain meaningful data exported by the owner. The other \( (2^n - 2) \log n - n \) data blocks in the hierarchy are called dummy data blocks. They simply contain random stuffing data and no user (except the owner) is authorized to write a dummy block. To create a dummy data block, the owner randomly generates a unique block ID and some data content, and then creates the data block in the same format as the real data block. Each dummy data block \( D_i,t_i \) is initialized with \( t_j \) set to 0, \( e_i,t_i,1 \) and \( e'_i,t_i,1 \) set to empty, and \( l_i,t_i \) and \( b_i,t_i \) set to some layer and bucket so that dummy data blocks fill up all the storage locations not occupied by real data blocks. In addition, the server maintains a counter \( C_q \) (with an initial value of 0) which keeps track of the number of queries that have been processed.

C. Query Process

The query process is executed when a user needs to retrieve a target data block (denote its ID as \( T \)). We show the flowchart of the proposed query process in Fig. 3(b), together with the flowchart of the query process in the original ORAM in Fig. 3(a). Specifically, to retrieve the target block, a user performs the following operations.

(Q1) Both buckets at the top layer of the storage hierarchy are retrieved from the server. If the target data block is found in the buckets, the flag \( \text{found} \) is set to \( \text{true} \); else, it is set to \( \text{false} \).

(Q2) Counter \( C_q \) is retrieved from the server and is incremented by 1.

(Q3) For each layer \( i \) from 2 to \( \log n \), the following is performed:

- If \( \text{found} = \text{true} \), a bucket is selected uniformly at random from the layer and all the data blocks in the bucket are retrieved.
- If \( \text{found} = \text{false} \), all data blocks in bucket \( h_1([C_q/2^{i}], T) \) are retrieved. If target block is found among the retrieved blocks, \( \text{found} \) is set to \( \text{true} \).

(Q4) After all the requested data blocks have been retrieved, the server constructs a Merkle tree hash based on the data ID hashes (i.e., \( c_i,t_i \) of the retrieved blocks, calculates the root hash (i.e., \( c_i,t_i \) ) of the tree, and saves it for future verification purposes. Then, the server notifies the user of the root hash and the co-path hash values (i.e., \( S_i,t_i \) ) for each retrieved data block \( D_i,t_i \).

(Q5) The user verifies all the retrieved data blocks, accesses the content of the target data block, re-formats all the retrieved data blocks, and then uploads them back to the server. More specifically, this step involves the operations of verification, updating, and uploading, as explained below.

1) Verification: Each retrieved data block \( D_i,t_i \) is verified as follows.

(V1) Verification of group signature \( s_i,t_i \): \( s_i,t_i \) is verified with the public key \( K^*_g \).

(V2) Verification of group signature \( g_{i,v},i \): \( g_{i,v},i \) is verified with the group public key \( K^*_g \). If it is invalid, a user misconduct has been detected, as the user who last accessed the block must have altered the data content without proper authorization.

(V3) Verification of ID hash \( c_i,t_i \): The correctness of \( c_i,t_i \) is verified by checking whether it is equal to \( H(\mu_i,t_i, i, r_{i,v}'). \) Note that, \( \mu_i,t_i, i \) and \( r_{i,v} \) are parts of \( d_i,t_i \) and hence protected by the group signature \( g_{i,v},i \). If \( c_i,t_i \) is incorrect, a user misconduct has been detected.
(V4) Verification of root hash $e_{i,t_1-1}$: Recall that $e_{i,t_1-1}$ is the root hash of the Merkle hash tree constructed by the server when the data block was last accessed. To verify it, the user needs the following information: the ID hash $e_{i,t_1-1}$ and the co-path hashes $S_{i,t_1-1}^l$ when the data block was last accessed. As we will explain in the “Updating” operation below, if the last access to the data block was a write access, a new random nonce $r_{v_1}$ would be generated. Therefore, as the user is unsure about the last access, it verifies $e_{i,t_1-1}$ against both $r_{v_1}$ and $r_{v_1-1}$. In other words, it checks whether $e_{i,t_1-1}$ is the same as the root hash calculated based on using either $H(\mu_{i,t_1-1}, i, r_{v_1})$ or $H(\mu_{i,t_1-1}, i, r_{v_1-1})$ as the ID hash $e_{i,t_1-1}$, and the co-path hashes $S_{i,t_1-1}^l$. If both fail, a user misconduct has been detected. Note that $r_{v_1}$ and $r_{v_1-1}$ can be obtained from $d'_{i,t_1}$, while $\mu_{i,t_1-1}$ and $S_{i,t_1-1}^l$ can be obtained from $e_{i,t_1-1}$.

(V5) Verification of data block placement: Suppose $D_{i,t_1}$ is retrieved from bucket $b$ of layer $l$ ($l > 1$). It is checked whether $l_{i,t_1} = l$ and $b_{i,t_1} = b$. Also, if the block is not a dummy, it is verified whether $b = h_l(\lceil C_q/2^l \rceil, i)$.

If verification fails, a misplacement attack has been detected.

2) Updating: If all the verifications succeed, the user accesses the target data and then updates all the retrieved data blocks as follows.

For each data block $i$, a new random initialization vector $\lambda_{i,\text{new}}$ and a new random nonce $\mu_{i,\text{new}}$ are selected. Then, the user sets $\lambda_{i,t_1+1} = \lambda_{i,\text{new}}$ and $\mu_{i,t_1+1} = \mu_{i,\text{new}}$. In addition, if a data block is the target block and the current access is a write access, a new random nonce $r_{i,\text{new}}$ is selected and $r_{i,v_1+1} = r_{i,\text{new}}$. Then, for each block, the user calculates $\mu'$ and $c$ with the updated data content and random nonces, $\nu'$ and $e$ are updated based on the co-path hashes ($S$) and the root hash ($e$) of the Merkle hash tree, which were notified by the server during query step (Q4).

Next, according to the underlying ORAM, a dummy block is randomly picked from the top layer of the storage hierarchy, and the target block swaps its position with the dummy by swapping the values of their $b$ and $l$ fields. For all other blocks, $b$ and $l$ fields remain unchanged. Finally, a new group signature $s$ is generated for each block.

3) Uploading: After all the retrieved data blocks have been updated, they are uploaded back to the server at the positions determined by their $b$ and $l$ values. The order in which the blocks are uploaded is arbitrary. In addition, if the counter $C_q$ is a multiple of $2^l$ for certain $l \in \{1, \cdots, \log n\}$ but not a multiple of any $2^l$ where $l' > l$, a shuffling process for layer $l$ should be conducted. The shuffling process is elaborated in Section IV-D.

4) Server Operations: Upon receiving an uploaded data block, the server performs the following verifications. If any of them fails, a user misconduct has been detected and the server refuses to accept the block.

(O1) Verify the validity of the group signature $s$;
(O2) Verify that the root hash $e$ carried in the block is the same as the root hash of the Merkle hash tree constructed by the server during query step (Q4);
(O3) Verify that the $e'$ value carried in the block is different from all other uploaded blocks. This is to prevent replacement attacks launched by the user.
(O4) Increase the counter $C_q$ by one.

D. Shuffling Process

The shuffling process for layer $l$ is to obliviously re-position and re-format all the blocks residing at layers $1, \cdots, l$ so that: (i) after the shuffling, each bucket contains the same number (i.e., $\log n$) of blocks; (ii) each real data block $D_{i,t_1}$ is placed to bucket $h_l(\lceil C_q/2^l \rceil, i)$ of layer $l$, where $C_q$ is the afore-mentioned counter keeping track of the number of queries that have been processed. As in the original ORAM,
our proposed shuffling process also contains several rounds of scanning, tagging, and oblivious sorting. However, the original ORAM shuffling scheme must be modified in order to provide accountability support. Due to space limitation, please refer to Appendix 2 for details.

E. Insertion and Deletion

Occasionally, the file system may be updated by adding new data blocks and/or removing existing data blocks. Due to space limitation, please refer to Appendix 3 for how the proposed scheme handles data insertion and deletion.

V. Security Analysis

In this section, we present the security analysis of the proposed scheme.

A. Access Pattern Privacy

We first show that the proposed scheme ensures the same access pattern privacy offered by the underlying ORAM, which is formally presented in Theorem 1. In order to prove Theorem 1, we first present the following two lemmas (Lemma 1 and Lemma 2).

Lemma 1: Compared with the original ORAM, the proposed data format does not provide extra information that helps the server identify a data block.

Proof: To hide access pattern, ORAM requires that the server cannot identify a data block from its appearance when multiple data blocks are put back to the server during a query or shuffling process. To achieve this, each data block in the original ORAM, in its simplest format, is just ciphertext generated over the real data content by a probabilistic symmetric encryption algorithm. In our proposed scheme, as shown in Eq. (2), each data block contains seven data fields that are visible to the server, including $d_{i;t}^t, c_{i;t}, e_{i;t}−1, e_{i;t}−1, l_{i;t}, b_{i;t}$, and $s_{i;t}$. We now show that, compared with the original ORAM, none of them leaks extra information that may help the server identify a data block.

- $d_{i;t}^t$, which contains the real data content, is equivalent to the encrypted data block as in the original ORAM. It is a ciphertext output by a probabilistic encryption function $E(\cdot)$ which is semantically secure. Given that (i) the size of the real content in each data block (i.e., $w_{i;1,v_1}$) is the same, and (ii) $\mu_{i,t}, \mu_{r,v}$, $r_{i,v}$, $l_{i,v}$, $b_{i,v}$, and $s_{i,v}$, each block have the same size, $d_{i;t}^t$ of each data block has the same length. As a result, compared with the original ORAM, the server does not gain extra information from $d_{i;t}^t$.

- $c_{i;t}$ is the hash of the concatenation of $\mu_{i,t}, i$ and $r_{i,v}$. Because $\mu_{i,t}$ is picked randomly for each data block downloaded during a query or shuffling process, $c_{i;t}$ appears random for each uploaded data block. Therefore, the server does not gain extra information from $c_{i;t}$.

- $e_{i;t}−1$ is the root hash of a Merkle hash tree that is constructed during a query or shuffling process. In the proposed scheme, data blocks that are put back to the server in the same query or shuffling process will have the same $e_{i,t}−1$. As a result, $e_{i,t}−1$ cannot help the server identify a data block.

- $e_{i,t}−1$ is the ciphertext of $\mu_{i,t}−1$ and $s_{i,t}−1$ with a semantically secure function $E(\cdot)$. Because $\mu_{i,t}−1$ is randomly picked for each data block, the server does not gain extra information from $e_{i,t}−1$.

- $l_{i,t}$ and $b_{i,t}$ are the location information of each data block, which are actually observable by the server. Therefore, inclusion of them in a data block does not leak extra information to the server.

- $s_{i,t}$ is a group signature re-generated for each uploaded data block and is random because the content to be signed has changed. Thus, it does not help the server identify a data block.

Lemma 2: Compared with the original ORAM, neither the query process nor the shuffling process in the proposed scheme leaks additional information that helps the server identify a data block.

Proof: As for query, in our proposed scheme, the bucket to be retrieved on each layer during downloading is the same as the original ORAM. As explained in the proof of Lemma 1, the Merkle tree construction (i.e., the computation of $e_{i,t}−1$ during downloading) and the server verification of the Merkle tree root (during uploading) do not provide the server any additional information to identify a data block. The user verification and update processes all happen at the client side. In the uploading process, the target data block swaps its location with a random dummy block at the top layer. But, because every uploaded data block is updated, the server cannot do better in identifying the target data block than random guessing. Lastly, due to the randomness of group signature generation (i.e., $s_{i,t}$), the server cannot identify a data block by the verification of the group signature. As a result, the query process does not leak additional information.

As for shuffling, similar to query, the Merkle tree construction, the server-side verification, and the client-side verification do not leak additional information. On the other hand, from the algorithm, it is straightforward to see that sorting is oblivious. Therefore, the shuffling process also does not leak additional information.

Theorem 1: The proposed scheme provides the same level of access pattern privacy as the underlying ORAM.

Proof: Because neither the modified data format nor the data query/shuffling process downgrades the security of the scheme in term of access pattern privacy, Theorem 1 is a straightforward conclusion from Lemma 1 and Lemma 2.

B. User Accountability

We now show that any un-authorized modification, replacement, or misplacement conducted by a malicious user can be detected either by the server or by the user who next accesses the data block.
1) Modification Attacks: Modification attacks are referred to as the attacks where a malicious user modifies the actual content of a data block without proper authorization. This can be detected thanks to the presence of group signature $g_i$ which can be signed only by the users who are authorized to write the data block. The identity of the user who last accessed the data block can be identified by opening the group signature $s_i$.

2) Replacement Attacks: Instead of modifying the data content, a malicious user may launch replacement attacks by replacing some or all fields of a data block $D_{i,t_i}$, with those of another block $D_{j,t_j}$ or an older version of the same block $D_{i,t_i'}$ ($t_i' < t_i$), and then generating a valid group signature $s_i$. Therefore, similar to the discussions above, such an attack would fail the verification process. Now, let’s consider the following replacement attacks: replacement of $d'$ and $c$; or replacement of $d'$ and $c$ and $e$'.

a) Replacement of $d'$ and $c$: This can be detected by the user who next accesses the block during the root hash verification step (V4). As described in (V4), to verify the root hash $e_{i,t_i-1}$ carried in $D_{i,t_i}$, which was calculated by the server when the block was last accessed, the following information are needed: $c_{i,t_i-1}$ – the ID hash, and $S_{i,t_i-1}$ – the co-path hashes. $c_{i,t_i-1}$ can be derived as $c_{i,t_i-1} = H (s_{i,t_i-1}, i, r_{i,v_i})$, where $v_i$ is the version number of the data content when the block was last accessed, and both $i$ and $r_{i,v_i}$ are embedded in $d_{i,t_i}$. Clearly, replacement of $d_{i,t_i}$ with $d_{i,t_i}'$ of another block would result in an incorrect ID being used to derive $c_{i,t_i-1}$ and hence would fail the verification process. Now, let’s consider the attacks by replacing $d_{i,t_i}'$ with an older version of the same block, i.e., $d_{i,t_i}'$, whose data content has an earlier version number of $v_i' < v_i$. As the nonces embedded in $d_{i,t_i}'$ are $r_{i,v_i'}$ and $r_{i,v_i'-1}$, we have $v_i > v_i' > v_i'-1$. This means that the random nonce $r_{i,v_i}$ needed to derive $c_{i,t_i-1}$ cannot be found in the replaced $d'$, which also would result in a verification failure. Here, we assume that the space for the random nonce is large enough so that the probability of two randomly picked nonces being the same is negligibly small.

b) Replacement of $d'$, $c$, and $e'$: As described in the server operation step (O3), the server verifies that the $e'$ values are distinct in all the uploaded data blocks during the same query/shuffling process. Therefore, it may only be possible that a malicious user attempts to replace $d_{i,t_i}'$, $c_{i,t_i}$, and $e_{i,t_i-1}'$ with those of another block $D_{j,t_j}$ from a different query/shuffling process, or an older version of the same block $D_{i,t_i'}$ ($t_i' < t_i$). In either case, a different Merkle hash tree was constructed. Therefore, similar to the discussions above, such an attack can be detected by the user who next accesses the block, as it would result in a root hash verification failure due to the replaced co-path hashes. Here, we assume that the hash function $H$ is well designed so that for two different Merkle hash trees, the probability that they have the same root hash is negligibly small.

3) Misplacement Attacks: A malicious user may place a data block in the wrong bucket and/or layer of the storage hierarchy. Such attacks can be detected in one of the following ways: (i) the position of the block is inconsistent with the hash of its ID using the designated hash function; or (ii) the data block cannot be found at the specified position. In the latter case, the server may work with the dataset owner to scan the storage hierarchy to locate the misplaced block and identify the attacker through group signature $s_i$.

VI. OVERHEAD ANALYSIS

A. Storage and Communication Overhead

1) Server Storage Overhead: As our scheme requires additional fields in each data block $D_{i,t_i}$, including $s_{i,t_i}$, $r_{i,v_i}$, $r_{i,v_i-1}$, $i$, $g_{i,v_i}$, $c_{i,t_i}$, $e_{i,t_i-1}$, $e_{i,t_i-1}'$, $l_{i,t_i}$, $b_{i,t_i}$, and $s_{i,t_i}$, extra storage is needed on the server. We assume that each hash value ($s_{i,t_i}$, $r_{i,v_i}$, $r_{i,v_i-1}$, $c_{i,t_i}$, and $e_{i,t_i-1}$) is 32-byte long and one group signature ($g_{i,v_i}$ and $s_{i,t_i}$) takes less than 200 bytes [31]. $i$ is the block ID, which is of $\log n$ bits, $l_{i,t_i}$ and $b_{i,t_i}$ store the layer and bucket indices. Since there are $\log n$ layers in the storage hierarchy and at most $n$ buckets at each layer, $l_{i,t_i}$ is of $\log \log n$ bits and $b_{i,t_i}$ is of $\log n$ bits. $e_{i,t_i-1}'$ introduces a larger overhead, because it stores the information that corresponds to a path from root to leaf in the Merkle hash tree. In the worst case, the Merkle hash tree may contain $n \log n$ data blocks (i.e., when the entire database is shuffled) and thus $e_{i,t_i-1}'$ may store up to $\log (n \log n)$ hash values. Fortunately, this is still considered a small amount of storage overhead in practice. For example, assuming $n \leq 2^{32}$ and each hash value is of 32 bytes, the size of $e_{i,t_i-1}'$ is $32 \log (n \log n)$ bytes, which is less than 1.1 KB, when $n \leq 2^{32}$. To summarize, the overall extra overhead per data block is $O(\log n)$ bytes.

TABLE III shows the storage overhead with different numbers of data blocks. Considering that a data block is typically 64 KB or 256 KB in practice [12], the extra overhead is less than 3% of data block size.

<table>
<thead>
<tr>
<th># blocks ($n$)</th>
<th>$2^{20}$</th>
<th>$2^{24}$</th>
<th>$2^{28}$</th>
<th>$2^{32}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extra Overhead</strong></td>
<td>1.32 KB</td>
<td>1.45 KB</td>
<td>1.58 KB</td>
<td>1.71 KB</td>
</tr>
</tbody>
</table>

2) User Storage Overhead: A user also needs an extra storage space to store (i) the encryption keys for the blocks that the user is authorized to read, (ii) the group private keys for the blocks that the user is authorized to write, and (iii) all the group public keys for verifying each block. As a result, a user needs $O(m_R + m_W)$ storage overhead, where $m_R$ and $m_W$ are the number of read groups and number of write groups respectively. Note that in the worst case, the owner may classify each data block as a read/write group, which results in $m_R = m_W = n$, where $n$ is the number of blocks in the system. However, in practice, a shared data set typically may have a limited number of access policies (e.g., no more than a few hundred). Thus, the overhead for storing the keys is acceptable (e.g., no more than a few megabytes). Moreover, as explained in Section IV-C, the Merkle hash tree is constructed by the server and the user only needs to verify the Merkle hash tree information it retrieves; hence, no storage overhead is introduced for this process.
3) Communication Overhead: As explained in the storage overhead, the proposed scheme will increase the data block size slightly. On the other hand, the number of data blocks transmitted during query and shuffling are asymptotically the same between the proposed scheme and the original ORAM. As a result, the proposed scheme adds only small communication overhead to the original ORAM.

B. Computational Overhead

1) Server Overhead: For each data query, the additional (amortized) computational overhead at the server side includes $O(\log^2 n)$ hash computations (to construct a Merkle hash tree) and $O(\log^2 n)$ group signature verifications.

2) User Overhead: For each data block $D_{i,t}$ retrieved in a query or shuffling process, a user needs to perform the following extra computations: (i) verification of the group signatures $g_{i,v_i}$ in the retrieved data block (and generation of $g_{i,v_i+1}$ for the updated data block for a write operation); (ii) verification of $e_{i,t,i}$ in the retrieved data block and generation of $c_{i,t,i+1}$ for the updated data block, both of which are hash computations; (iii) decryption of $e_{i,t,i-1}$ and re-encryption of $e'_{i,t,i}$; (iv) verification of the root hash $e_{i,t,i-1}$ of the Merkle hash tree, which is composed of a sequence of hash computations; (v) verification of placement of $D_{i,t}$, which is one hash computation; (vi) verification of group signature $s_{i,t,i}$ and generation of $s_{i,t,i+1}$; and (vii) verification of $e_{i,t,i}$ and $S_{i,t,i}$ received from the server, which is composed of a sequence of hash computations. Using a modern pairing-based cryptography library such as PBC library [32], each of the above computations can be done efficiently from several milliseconds to several hundreds of milliseconds.

VII. CONCLUSIONS AND FUTURE WORK

This paper proposes an accountability solution for hash-based ORAM schemes such as [18]. It detects misconduct committed by malicious users and identify the attacker, without interfering with the access pattern preservation mechanisms inherent from the underlying ORAM. This is achieved through an integration of Merkle hash tree and group signature techniques, as well as a delicate design of the data block format. As a tradeoff, the storage, communication, and computational overheads are increased slightly. Our proposed accountability support can also be extended to work with other hash-based hierarchical ORAM schemes [4]–[6], [9], [10], as long as the extended data format does not give the server non-negligible advantages in inferring users’ access pattern in the integrated schemes. However, its integration with Bloom filter-based or index-based ORAM schemes [2], [3], [7], [8], [11]–[17] is nontrivial, and will be studied in our future work.

REFERENCES


APPENDIX 1
In this section, we explain the three building blocks of our proposed scheme: Oblivious RAM, group signature, and Merkle hash tree.

Oblivious RAM

Existing ORAM schemes can be roughly classified into two categories depending on how the user decides the storage location of a data block: (i) hash-based ORAMs where a data block’s storage location is decided according to hash functions, and (ii) index-based ORAMs where a block’s storage location is retrieved from an index map. Our scheme is designed to integrate with most hash-based ORAM schemes. In this paper, we present how our scheme works with the seminal hash-based ORAM scheme proposed in [18], which provides a framework for all other hash-based ORAMs. We now briefly explain how this ORAM scheme works:

Storage: In ORAM, user’s data are stored as a set of encrypted and equal-sized data blocks, as shown in Fig. 4. All data blocks are organized in a hierarchical structure which consists of \( \log n \) layers (where \( n \) is the number of data blocks). Layer \( i \) contains \( 2^i \) buckets, each of which can store \( \log n \) data blocks. Layer \( i \) may store up to \( 2^i \) real data blocks, while the rest are dummy blocks. Real data blocks are stored into buckets based on the hash of their IDs.

Query: The data query is executed whenever the user wants to retrieve a target block from the server. To hide the actual target block from the server, the ORAM query works in a manner so that all layers will be accessed, and the buckets accessed at each layer appear random to the server. Specifically, to retrieve a target block with ID \( T \) at the \( C_q \)’th query, the user performs the following operations.

- Both buckets at the top layer are retrieved from the server. If the target data block is found in either bucket, the flag found is set to true; else, it is set to false.
- For each layer \( i \) from 2 to \( \log n \),
  - If found = true, a bucket is selected uniformly at random from the layer and all data blocks in this bucket are retrieved.
  - If found = false, all data blocks in bucket \( b_i([C_q/2^i], T) \) are retrieved, where \( b_i(\cdot) \) is a hash function for layer \( i \). If the target block is found among the retrieved blocks, found is set to true.

- At the end of the query, the user randomly selects a dummy data block from the top layer and replaces it with the target data block. Then, all data blocks of the top layer are re-encrypted and uploaded back to the top layer.

3) Shuffling: As target blocks are gradually moved from bottom layers to upper layers, the shuffling process is triggered periodically to avoid layer overflow. For layer \( i \), the shuffling process takes place every \( 2^i \) accesses. The shuffling process for layer \( i \) essentially re-positions all real data blocks on and above layer \( i \) into layer \( i + 1 \) in an oblivious manner. Due to space limitation, please refer to [18] for details.

Group Signature and Merkle Hash Tree

Group signatures [23] are often used to allow each member of a group to anonymously sign a message on behalf of the group. Typically, there is a group manager who is in charge of the membership management and has the ability to reveal the identity of the signer in the event of disputes. For simplicity, we assume that a group signature scheme provides a primitive \( G_i \cdot \text{Sig}_u(m) \) which generates the signature for message \( m \) with the secret key of group member \( u \), where we use subscript \( i \) to differentiate different groups.

A Merkle hash tree [22] is typically used to allow efficient verification of the integrity of a large set of data. In a Merkle hash tree, the leaves are hashes of data blocks and each interior node (including the root) is the hash of its child nodes. The root of a Merkle hash tree is called the root hash, and the set of hash values that are needed to calculate the root hash from a leaf node is called the co-path (denoted as \( S \)) from the leaf node to the root. For example, in Fig. 5, the co-path for data block \( D_1 \) from the leaf node \( h_{0-1} \) to the root \( h_{2-1} \) is \( S_1 = \{ h_{0-2}, h_{1-2} \} \).

\[
\begin{align*}
h_{2-1} &= H(h_{0-2} || h_{1-2}) \\
h_{2-2} &= H(h_{2-3} || h_{2-3}) \\
h_{2-3} &= H(h_{2-4} || h_{2-4}) \\
h_{2-4} &= H(h_{2-5} || h_{2-5}) \\
h_{2-5} &= H(h_{2-6} || h_{2-6}) \\
h_{2-6} &= H(h_{2-7} || h_{2-7}) \\
h_{2-7} &= H(h_{2-8} || h_{2-8}) \end{align*}
\]

Fig. 5. A Merkle hash tree for verifying the integrity of data blocks \( D_1-D_4 \) (\( H \) is a hash function).

APPENDIX 2
The shuffling process in the proposed scheme works in the same fashion as in the original ORAM, which contains several rounds of tagging and oblivious sorting. However, because we do not allow a user to add new data blocks to the server, the original ORAM shuffling process must be modified to suit this requirement. In addition, the proposed scheme also requires the server to construct the Merkle hash tree for all the data blocks to be shuffled at the beginning and verify the Merkle hash tree information in each data block when they are finally put back to the server. The user that performs the shuffling also needs to perform various verifications as described in the query process.
when data blocks are downloaded. Incorporating accountability mechanisms, the detailed operations of the shuffling process are as follows. An example where \( l = 2 \) is shown in Fig. 6.

\[
\begin{array}{c}
\text{Layer 1} \\
\begin{array}{ccc}
\sim & D1 & \sim \\
\sim & \sim & \sim \\
\end{array} \\
\downarrow \\
\text{Buffer} \\
\begin{array}{ccc}
\sim & D1 & \sim \\
\sim & D2 & \sim \\
\sim & D4 & \sim \\
\end{array} \\
\end{array}
\]

(S1) Data blocks from two layers are merged. Each data block’s Merkle hash tree information is attached.

\[
\begin{array}{c}
\text{Layer 2} \\
\begin{array}{ccc}
\sim & \sim & \sim \\
\end{array} \\
\downarrow \\
\text{Buffer} \\
\begin{array}{ccc}
\sim & D4 & \sim \\
\sim & \sim & \sim \\
\sim & D3 & \sim \\
\sim & D1 & \sim \\
\end{array} \\
\end{array}
\]

(S2) The user verifies and updates each data block and augments each data block with a tag from \([0, 4]\).

\[
\begin{array}{c}
\text{Buffer} \\
\begin{array}{cccc}
1, & \sim & 1, & \sim \\
\sim & D1 & \sim & \sim \\
\sim & D2 & \sim & \sim \\
\sim & D3 & \sim & \sim \\
\sim & D4 & \sim & \sim \\
\end{array} \\
\end{array}
\]

(S3) Data blocks are obliviously sorted.

\[
\begin{array}{c}
\text{Buffer} \\
\begin{array}{cccc}
\sim & \sim & \sim & \sim \\
\sim & \sim & \sim & \sim \\
\sim & \sim & \sim & \sim \\
\sim & \sim & \sim & \sim \\
\sim & \sim & \sim & \sim \\
\end{array} \\
\end{array}
\]

(S4) Scan the blocks and leave exactly \( \log n \) blocks with tag \( j \), for each \( j \) from 1 to \( 2^l \).

\[
\begin{array}{c}
\text{Buffer} \\
\begin{array}{cccc}
\sim & \sim & \sim & \sim \\
\sim & \sim & \sim & \sim \\
\sim & \sim & \sim & \sim \\
\sim & \sim & \sim & \sim \\
\sim & \sim & \sim & \sim \\
\end{array} \\
\end{array}
\]

(S5)-(S6) Obliviously sort the buffer, put each data block to its bucket.

Fig. 6. Example of the shuffling process. Each bucket has \( \log n \) data blocks. We use symbol ‘\(\sim\)’ to represent one or more adjacent dummy data blocks.

(S1) The server merges all the data blocks at layers \( l, \cdots, l \) to a shuffling buffer, builds a Merkle hash tree with the data ID hash fields of these blocks as leaf nodes, and calculates the root hash. Then, for each block \( D_{i,t} \) in the buffer, the root hash value \( e_{i,t} \) and its corresponding co-path values \( S_{i,t} \) are attached to the block. After this, the following five steps are performed by the user who conducts the shuffling process.

(S2) In this step, the user scans (i.e., downloads, processes, and re-uploads back) the blocks in the shuffling buffer, one by one. More specifically, after a block \( D_{i,t} \) has been downloaded, the validity of the data block, including the attached \( e_{i,t} \) and \( S_{i,t} \), is first checked in a similar manner as in the verification process during a regular query (please refer to Section IV-C). If the check fails, the user stops the shuffling process and informs the server. If all data blocks have been downloaded, the validity of the data block, including the user's version of its \( S_{i,t} \) (without a random nonce) to allow the server to check whether the block carries a unique \( e' \) value.

Note that, in the last time when all data blocks are uploaded back to buckets, the server needs to check whether the uploaded set is the same as the downloaded set.

APPENDIX 3

Occasionally, the file system may be updated by adding new data blocks and/or removing existing data blocks. In the proposed scheme, data addition and deletion is handled by the data owner and is processed in the same fashion as in the underlying ORAM scheme.
To delete a data block, the owner first creates a new dummy data block (as described in Section IV-B) whose ID is set to be the same as the block to be removed. This new dummy data block will replace the data block to be removed. To achieve this, the data owner makes a query (as described in Section IV-C) with the target set as the data block to be removed. Then when uploading the data blocks back to the server, the owner replaces the target data block with the new dummy data block. Note that due to the Merkle hash tree verification, the new dummy data block’s ID must be the same as the ID of the deleted data block, because the root hash is calculated using the removed data block’s ID information. Also note that only the data owner can write a dummy data block.

To insert a new data block, the owner first determines if the new block falls into an existing read (write) group. If it doesn’t belong to any existing read groups, a new read group $R_x$ is created and the owner selects an encryption key $k_{R_x}$ for this read group. Similarly, if it doesn’t belong to any existing write groups, a new write group $W_y$ is created and the owner selects a group public key $K_{W_y}^+$ and a set of group private keys $\{K_{W_y}^-\}$ for this write group. Then, the owner (i) publishes $K_{W_y}^+$, (ii) disseminates $k_{R_x}$ to users who can read the new data block, and (iii) disseminates a unique $K_{W_y}^-$ to each user who can write the new data block. After that, the owner creates the new data block as described in Section IV-B. In order to put the new data block onto the server, the data owner swaps it with a dummy data block on the server whose ID will be assigned to the new data block. This swapping process is similar to the one described in the last paragraph.

Note that in ORAM, under certain conditions, addition/deletion of a real data block may change the number of layers in the ORAM storage. In such cases, the owner may first change the number of layers per the rules defined in the underlying ORAM scheme and then add/delete the data block as afore-explained.