An Accountability Scheme for Oblivious RAMs

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

Abstract. In outsourced data services, revealing users' data access pattern may lead to the exposure of a wide range of sensitive information even if data is encrypted. Oblivious RAM has been a well-studied provable solution to access pattern preservation. However, it is not resilient to attacks towards data integrity from the users or the server. In this paper, we study the problem of protecting access pattern privacy and data integrity together in outsourced data services, and propose a scheme that introduces accountability support into a hash-based ORAM design. The proposed scheme can detect misconduct committed by malicious users or server, and identify the attacker, while not interfering with the access pattern preservation mechanisms inherent from the underlying ORAM. This is accomplished at the cost of slightly increased computational, storage, and communication overheads compared with the original ORAM.

1 Introduction

Along with the increasing popularity of outsourcing data services to remote cloud servers, arise also security and privacy concerns. Although encrypting data content has been a common practice for data protection, it does not eliminate the concerns, 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 [2–14] have been developed to implement oblivious RAM (ORAM) [15], which was originally proposed for software protection but also is a provable solution to data access pattern preservation.

Existing ORAM designs, however, share a common problem which may limit their practical application. Specifically, these designs all assume that the outsourced data is accessible to trusted users only and the storage server is curious but honest; hence, they are not equipped with any protection against attacks towards data integrity from the users or the server. Unfortunately, such assumptions are unrealistic in practice. 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 user or the storage server accountable if data is altered.

To deal with the above problem, it is highly desirable to introduce accountability support into ORAM. However, existing accountability solutions cannot be readily applied because of the conflicting goals of these two types of mechanisms: preserving access pattern privacy in ORAM requires data blocks to be frequently re-encrypted and re-positioned, which however can easily conceal the traces that are needed for detecting misbehavior and identifying attackers.

In this paper, we propose an accountable ORAM scheme, which is developed on top of a hash-based ORAM design such as [15]. The scheme can detect misconduct by malicious users or server and identify the attackers, while not interfering with the access pattern preservation mechanisms inherent from the underlying ORAM. The goal is achieved through a combined leverages of Merkle hash tree [16], digital signature and group signature [17] techniques, as well as a delicate design of the data block format. 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.

Security analysis and overhead evaluation have been conducted. The results show that the proposed scheme has achieved the goals of accountability of users and the server's behavior and preservation of data access pattern privacy, at the cost of slightly increased computational, storage, and communication overheads.

The rest of the paper is organized as the follows. Section 2 presents the related work. Section 3 describes the system model and Section 4 explains the building blocks of the proposed scheme. The proposed scheme is elaborated in Section 5. Section 6 and Section 7 present the security analysis and overhead analysis respectively. Finally, Section 8 concludes the paper.

2 Related Work

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [15], is a well-known technique to hide the pattern of a user's accesses to an outsourced data storage. Following it, various schemes [2–14] have been proposed to improve the performance of ORAM in terms of data access latency, storage overhead and communication/computational complexity.

While ORAM is becoming more and more practical in terms of performance, there are security issues that need to be addressed before ORAM can be widely adopted in practice. There are only a few existing works that address these practical security concerns under the framework of ORAM. For example, in [18], the problem of access right delegation was studied. It proposes a scheme with which the data owner can delegate controlled access to third parties for the outsourced dataset, while preserving full access pattern privacy. However, this scheme can only be applied to the square-root ORAM [15], which is inefficient in terms of communication overhead. In another example, a write-once-read-many ORAM (WORM-ORAM) was proposed in [19], 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. However, it imposes expensive communication and computation overheads upon the regular ORAM, which also makes the scheme impractical.

Different from existing works, we study user and storage server accountability in this paper, which is an important security feature required by most data sharing services. We propose a scheme that adds user and storage server accountability to existing ORAM schemes while ensuring the access pattern privacy provided by the ORAM. To our knowledge, this is the first work that attempts to introduce accountability to ORAM.

The proposed scheme introduces low overhead and can be integrated with most of the existing hash-based ORAM schemes.

3 System Model and Design Goals

In this paper, we study a system where an owner of a dataset outsources the data to a remote server and allows a group of users to read (but not modify) the data. In order to hide the users' access pattern from the storage server, the owner is assumed to follow a hash-based Oblivious RAM (ORAM) design to deploy the data on the server, which is further explained in Section 4.1.

We assume the dataset owner is trusted while the server and the authorized users could be misbehaving. Specifically, a user may attempt to attack the integrity of data blocks through modification, replacement, addition, or deletion. However, we assume the user will not attack the access pattern preservation mechanism, for its own benefit. One the other hand, the server may attempt to attack both data integrity and the access pattern preservation mechanism. Also, we do not consider the collusion between the server and any user.

The design goals of the proposed scheme include: (i) preserving the users' access pattern of the dataset; (ii) detecting attacks launched by a user towards data integrity, and upon detection, enabling the dataset owner to identify the attacking user; and (iii) detecting attacks launched by the server towards data integrity or data access pattern preservation mechanisms (i.e., the underlying ORAM scheme).

4 Preliminaries

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

4.1 Oblivious RAM

The proposed scheme is designed to be atop and integrated seamlessly with most of the hash-based ORAM schemes. In general, a hash-based ORAM scheme works in the following way. At the server, the outsourced data are stored as a set of equal-sized data blocks and all data blocks are organized in a hierarchical storage structure. At each layer of the hierarchy, real data blocks and dummy data blocks are encrypted and obliviously stored in hash tables (e.g., buckets in [5, 6, 9, 10, 15] or cuckoo hash table in [4–6, 9, 10]) for future lookup. Typically, there are two major operations in ORAM: *data query* and *data shuffling*. During a data query, the user retrieves one or more data blocks from each layer based on the hash functions. Among the retrieved data blocks, only one of them is the actual target. At the end of each query, the target data block is re-encrypted and uploaded to the top layer of the hierarchy. As data queries proceed, the shuffling process is triggered periodically to protect access pattern privacy and avoid layer overflow. The shuffling process essentially re-positions all data blocks into a lower layer in an oblivious manner. Please refer to [2–10] for details of the ORAM operations.

In the rest of this paper, we choose the seminal hash-based ORAM scheme proposed by O. Goldreich and R. Ostrovsky [15] as the underlying ORAM scheme. This is because this ORAM scheme proposes a framework for other hash-based ORAMs; hence, integration of the proposed scheme with this ORAM scheme can be easily extended to other hash-based ORAMs (as to be explained in Appendix 2).

4.2 Group Signature

Group signatures [17] 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 membership management and has the ability to reveal the identity of the signer in the event of disputes. In general, a group signature scheme has the following properties:

- *Traceability:* Given any valid signature, the group manager should be able to trace the user who generated the signature.
- *Anonymity:* Given a message and its signature, the identity of the individual signer cannot be determined without the group manager's secret key.
- *Unlinkability:* Given two messages and their signatures, it cannot be determined if the signatures were generated by the same signer.
- *Non-framing:* Even if all other group members (including the manager) collude, they cannot forge a signature of an innocent group member.

To leverage the group signature mechanism without delving into its implementation details, we assume that a group signature scheme provides the following primitives: (We use prefix "G" to differentiate group signature from regular digital signature, which is also used in the proposed scheme.)

- $G\text{-}Sig_u(m)$: To generate the signature for message m with the secret key of group member u.
- $G\text{-}Verify(s, m, p)$: To verify whether signature s was generated by a valid group member for message m using the public key p.
- $G\text{-}Open(s, m)$: To reveal (by the group manager) the identify of the group member who generated signature s for message m.

In the proposed scheme, the owner of the dataset acts as the group manager and each user is a unique member of the group. Moreover, the storage server is given the public key so that it can verify the group signature carried in each data block; details will be explained in Section 5.

4.3 Merkle Hash Tree

A Merkle hash tree [16] 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 children 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 $\mathbb{S}$) from the leaf node to the root. For example, in Fig. 1, the co-path for data block D_1 from the leaf node h_{0-1} to the root h_{2-1} is $\mathbb{S}_1 = \{h_{0-2}, h_{1-2}\}$.

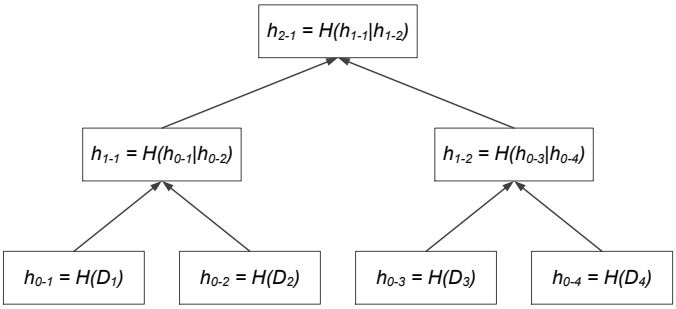

Fig. 1. A Merkle hash tree for verifying the integrity of data blocks D_1-D_4 (H is a hash function).

5 Proposed Design

This section presents the intuitions and detailed design of the proposed scheme.

5.1 Scheme Overview: The Intuitions

Built on top of a hash-based ORAM such as [15], our proposed scheme has the ultimate goal of user and server accountability and data access pattern preservation, which is attained through accomplishing each of the following more specific subgoals:

(G1) The content of each data block shall not be modified by any user or server. If the modification occurs, it shall be detected and the user or server committing the modification shall be identified.
(G2) After 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, the misconduct 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, the misconduct shall be detected and the user shall be identified.
(G4) The server shall execute the underlying ORAM algorithm honestly. Particularly, when a user uploads a data block to the server, the block shall be placed to the layer and bucket specified by the user; the server shall inform the user of the number of queries to ensure that data shuffling can be conducted according to the timing required by the underlying ORAM. If the server fails to do so, the misconduct shall be detected.

Accomplishment of Subgoal (G1) To accomplish subgoal (G1), the proposed scheme employs both digital signature and group signature mechanisms. More specifically, to export a data block to the server, the dataset owner (i) generates a digital signature for the plain-text content of the data block using a private key known only to itself, (ii) encrypts the data content together with the digital signature and a random nonce, (iii)

generates a group signature for the entire block, and (iv) include them as part of the data block and upload to the server.

After a user downloads a data block from the server, it performs the following steps. Firstly, it verifies whether the group signature is valid; if not, the server is detected to have committed a misconduct and the process is aborted. Secondly, the user decrypts the block to obtain the plain-text data content and its digital signature, and checks whether the digital signature is valid by using the public key published by the dataset owner; if the digital signature is invalid, it is detected that the user who last accessed the data block has modified the data content, and the identity of the malicious user can be traced out by the dataset owner based on the group signature associated with the block.

If both signatures are valid, the user proceeds to (i) access the data content, (ii) re-encrypt the data content together with the same digital signature but with a different random nonce, (iii) generate a new group signature for the entire block, and (iv) upload the updated data block back to the server. Upon receiving a data block from a user, the server checks whether the group signature is valid; if not, the user is detected to have committed a misconduct.

Accomplishment of Subgoal (G2) During a query or shuffling process, it is challenging to ensure that a user always uploads the same set of data blocks that it downloaded from the server, while the user is allowed to re-encrypt the data and change the order between blocks. To address this issue, we propose a solution that leverages the collaboration between users and the server, and utilizes the Merkle hash tree mechanism.

Specifically, the proposed solution requires each data block (denoted as D_i) to include three extra fields: (i) block hash field (denoted as c_i), (ii) root hash field (denoted as e_i), and (iii) auxiliary information field (denoted as e'_i). Initially, both e_i and e'_i are set to empty, and c_i is set to the hash value of the plain-text data content together with a random nonce.

During a query or shuffling process, a user needs to download a set of data blocks. The server constructs a Merkle hash tree using the block hash fields of these blocks as leaf nodes, and computes the root hash value of the tree. When the set of blocks are uploaded to the server later, it is required that the root hash field of each block shall be set to the root hash value computed by the server. Moreover, the auxiliary field of each block D_i shall contain (i) an encrypted version of the random nonce which was used when D_i was last accessed, and (ii) the co-path $\mathbb{S}_i$ of hash values needed to calculate the root hash from block D_i.

These requirements are critical to accomplish subgoal (G2) because they enable the server and the user who will next access these blocks to collaboratively check whether the uploaded set is the same as the downloaded set. More specifically:

- For any of the uploaded blocks, say D_i, the user who next access it will check whether it belongs to the set of the downloaded blocks, by checking whether the root hash value stored in e_i is the same as the one computed using the data content of D_i and the hash values in $\mathbb{S}_i$.
- The server checks all the uploaded blocks to ensure that all blocks carry the same root hash value for other users to check, and the auxiliary values are different for different blocks so that all the uploaded blocks are distinct.

If no violation is found in the above collaborative check, it indicates that each uploaded block belongs to the downloaded set and all uploaded blocks are distinct; furthermore, considering that the number of downloaded blocks is the same as the number of uploaded blocks, the downloaded and uploaded sets must be identical. If there is a violation, the user who uploads the blocks must have committed attacks, and its identity can be traced out by the dataset owner based on the group signatures that the user left with the blocks.

Accomplishment of Subgoals (G3) and (G4) With the above solutions to subgoals (G1) and (G2) in place, the solutions to subgoals (G3) and (G4) become straightforward.

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 a group signature. Next time when another user accesses the block, it performs the following checks. (i) It first checks whether the group signature is valid. (ii) Then, it checks whether the block is indeed placed at the position specified in the block. (iii) Finally, the user decrypts the block, extracts its ID, and applies the designated hash function to check whether the position of the block is consistent with the output of the hash function.

To ensure that data shuffling is performed according to the timing required by the underlying ORAM, users and the server collaborate to maintain a counter to keep track of the number of queries that has been processed. Initially, the counter is set to 0 and a group signature is generated by the dataset owner for the concatenation of the counter value and the content of the first block at the top layer. Later on, during each query process, the querying user retrieves the counter and the associated group signature from the server. Note that, the user should also retrieve every block at the top layer during the query process. Hence, it can verify the validity of the retrieved counter value. Based on the authenticated counter value, it can decide whether shuffling should be performed according to the underlying ORAM. After a query completes, the user increments the counter by one and re-generates a group signature for the concatenation of the new counter value and the new content of the first block at the top layer, and stores the above information back to the server. This way, the server cannot cheat on the counter value because (i) it cannot generate valid group signatures by itself, and (ii) it cannot replay an old counter value and an old group signature as the first block at the top layer is updated after every query. Also note that, if a user does not update the counter honestly, the misbehavior can be detected and reported by the server.

Integrating the above solutions that accomplish subgoals (G1)-(G4), we obtain the whole scheme, as elaborated in the following subsections.

5.2 System Initialization

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

Selection of System Parameters Let n be the total number of data blocks exported to the storage server; for simplicity, n is assumed to be a power of 2. The dataset owner selects the following system parameters:

- k: the system key used to encrypt plain-text data content of blocks;
- K^-, K^+: the pair of private key (K^-) and public key (K^+) for generating and verifying digital signatures, respectively;
- $H(x)$: a hash function that randomly maps one or a sequence of integers to an L-bit integer, where L is a security parameter;
- $h_l(x, y)$ 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 $\{1, \cdots, 2^l\}$.
- One group public key and a set of group private keys prepared for users: the group private keys can be used for generating group signatures, while the group public key can be used for verifying group signatures.

Among the parameters, only K^+ and the group public key are disseminated to the public, while the rest are kept secret either solely known to the dataset owner or shared between the owner and the users authorized to access the dataset.

Preparation of Data Blocks Each exported data block has a unique ID which is an integer belonging to set $\{1, \cdots, n\}$. We use $D_{i,t}$ to denote a data block of ID i and time stamp t. The actual content (i.e., plain-text data) that has been encrypted and embedded in $D_{i,t}$ is denoted as d_i. More specifically, $D_{i,t}$ has the following format:

$$D_{i,t} = \langle d'_{i,t}, c_{i,t}, e_{i,t-1}, e'_{i,t-1}, l_{i,t}, b_{i,t}, s_{i,t} \rangle, \tag{1}$$

where the fields are explained below.

- $d'_{i,t}$ is a one-time encryption of the concatenation of d_i and its digital signature generated by the owner, i.e.,

$$d'_{i,t} = E_k(r_{i,t}, d_i, Sig_{K^-}(d_i)), \tag{2}$$

  where $r_{i,t}$ is a random nonce, $E_k(X)$ is the symmetric encryption of X with key k, and $Sig_{K^-}(d_i)$ is a digital signature generated for d_i.
- $c_{i,t}$ (i.e., block hash field) is a one-time hash of d_i, i.e.,

$$c_{i,t} = H(r_{i,t}, d_i). \tag{3}$$

- $e_{i,t-1}$ (i.e., root hash field) is the root hash of a Merkle hash tree that was constructed when $D_{i,t}$ was last accessed (for simplicity, we use $t-1$ to denote the time stamp when $D_{i,t}$ was last accessed). The leaf node of the Merkle hash tree were the block hash fields of all the blocks downloaded together with $D_{i,t}$ during the last access.
- $e'_{i,t-1}$ (i.e., auxiliary information field) is a one-time encryption of the hash tree information that is needed to calculate root hash $e_{i,t-1}$. Specifically,

$$e'_{i,t-1} = E_k(r_{i,t-1}, \mathbb{S}_{i,t-1}), \tag{4}$$

where $\mathbb{S}_{t-1,i}$ denotes the co-path of hash values needed to calculate $e_{i,t-1}$, as aforedefined in Section 4.3.

- $l_{i,t}$ and $b_{i,t}$: the layer and bucket of block $D_{i,t}$ in the storage hierarchy.
- $s_{i,t}$ is a group signature of the entire block, i.e.,

$$s_{i,t} = G\text{-}Sig_u(d'_{i,t}, c_{i,t}, e_{i,t-1}, e'_{i,t-1}, l_{i,t}, b_{i,t}), \tag{5}$$

where u is the ID of either the dataset owner or a user authorized to access the data block, and $G\text{-}Sig_u(X)$ stands for a group signature generated by u for X.

Initially, the time stamp t is set to 0, both $e_{i,-1}$ and $e'_{i,-1}$ are set to empty, $l_{i,t}$ is initialized to $\log n$, and $b_{i,t}$ is initialized to $h_{\log n}(0, i)$.

Uploading of Data Blocks to the Storage Server All the data blocks are stored to a hierarchy of layers according the underlying ORAM. Specifically, the hierarchy consists of $\log n$ layers. Each layer l ($1 \leqslant l \leqslant \log n$) includes 2^l buckets and each bucket can contain $\log n$ blocks. Hence, the server stores $(2n-2)\log n$ data blocks in total, which includes n *real* data blocks that contain meaningful data exported by the owner.

The other $(2n-2)\log n - n$ data blocks in the hierarchy are called *dummy data blocks*. They simply contain random stuffing data. To generate a dummy data block, the owner randomly generates some data content, and then creates the data block in the same format as the real data block. To help quick identification of a dummy data block, the owner may put a special dummy tag (for example, integer 0 if all real data blocks have IDs greater than 0) at the beginning of the dummy block's content. Each dummy data block $D_{j,t}$ is initialized with t set to 0, $e_{j,-1}$ and $e'_{j,-1}$ set to empty, and $l_{j,t}$ and $b_{j,t}$ set to some layer and bucket such that all dummy data blocks fill up the storage locations not occupied by the real data blocks.

After the data blocks have been initialized, they are uploaded to the server. In addition, the owner also uploads to the server the initial value 0 of the counter C_q that keeps track of the number of queries having been processed, and its associated group signature which is generated over the concatenation of C_q and the content of the first block at the top layer.

User Authorization For each user that is authorized to access the data blocks, the dataset owner provides the symmetric key k, the public key K^+, the group public key, a distinct group private key, and the hash functions $H(x)$ and $h_l(x, y)$ for $l = 1, \cdots, \log n$.

5.3 Query Process

To retrieve a target data block with ID T, a user performs the following operations.

(Q1) Both buckets at the top layer are retrieved. If the target data block is found in the buckets, the flag `found` is set to `true`. Otherwise, `found` is set to `false`.

(Q2) Counter C_q is retrieved together with its associated group signature, from the server. If the group signature is found to be consistent with the value of C_q, let $m = C_q + 1$; otherwise, server misconduct is detected and the process aborts.

(Q3) For each layer i from 2 to $\log n$, the following is performed:
- If `found` = `true`, a bucket is selected uniformly at random from the layer and all the data blocks in the bucket are retrieved.
- If `found` = `false`, all data blocks in bucket $h_i(\lfloor m/2^i \rfloor, T)$ are retrieved. If target block is found in the retrieved blocks, `found` is set to `true`.

(Q4) The user needs to validate all retrieved data blocks, access the content of the target data block, re-format all the retrieved data blocks, and finally upload them to the server. More specifically, this step involves the operations of *validating*, *updating*, and *uploading*, as explained in the following.

Validating Without loss of generality, suppose the user retrieves q data blocks in a query, denoted as $D_{1,t_1}, D_{2,t_2}, \cdots, D_{q,t_q}$. These blocks are validated as follows.

(V1) Verification of digital signature: For each D_{i,t_i}, d'_{i,t_i} is decrypted to obtain d_i and its digital signature, and the signature is verified with public key K^+. This step checks if the data content of the block is unaltered.

(V2) Verification of c_{i,t_i}: For each D_{i,t_i}, the correctness of c_{i,t_i} is verified by checking if it is equal to $H(r_{i,t_i}, d_i)$.

(V3) Verification of e_{i,t_i-1}: For each D_{i,t_i}, e'_{i,t_i-1} is decrypted to obtain r_{i,t_i-1} and $\mathbb{S}_{i,t_i-1}$. The correctness of e_{i,t_i-1} is verified by checking if it is equal to the root hash computed based on $c_{i,t_i-1} = H(r_{i,t_i-1}, d_i)$ and the hashes included in $\mathbb{S}_{i,t_i-1}$.

(V4) Verification of placement of D_{i,t_i}: Suppose the block retrieved from bucket b of layer l ($l > 1$). It is checked whether $l_{i,t_i} = l$ and $b_{i,t_i} = b$. Also, if the block is not a dummy, the block ID i is extracted and then it is verified whether $b = h_l(\lfloor m/2^l \rfloor, i)$.

If any of the above verification fails, the user stops the query process and informs the owner of the potential tampering of the data blocks.

Updating After the validation succeeds and the target data has been accessed, the user updates each of the retrieved blocks D_{i,t_i} to D_{i,t_i+1} as follows.

(U1) A Merkle hash tree is built with $c_{1,t_1}, c_{2,t_2}, \cdots, c_{q,t_q}$ as leaf hashes. The root hash of the tree is stored in e_{i,t_i} of each D_{i,t_i+1} for $i = 1, \cdots, q$.

(U2) For each $i = 1, 2, \cdots, q$, the following operations are also conducted:
- A new nonce r_{i,t_i+1} is picked randomly;
- $d'_{i,t_i+1} = E_k(r_{i,t_i+1}, d_i, Sig_{K^-}(d_i))$;
- $c_{i,t_i+1} = H(r_{i,t_i+1}, d_i)$;
- $e'_{i,t_i} = E_k(r_{i,t_i}, \mathbb{S}_{i,t_i})$, where $\mathbb{S}_{i,t_i}$ is obtained from the Merkle hash tree built in (U1);
- According to the underlying ORAM, target block D_{T,t_T} and a dummy block D_{j,t_j} randomly picked from the top layer swap their positions, that is, $l_{T,t_T+1} = l_{j,t_j}$, $b_{T,t_T+1} = b_{j,t_j}$, $l_{j,t_j+1} = l_{T,t_T}$ and $b_{j,t_j+1} = b_{T,t_T}$; for any other block D_{i,T_i}, its placement remains unchanged, that is, $l_{i,t_i+1} = l_{i,t_i}$ and $b_{i,t_i+1} = b_{i,t_i}$;
- The user u generates a group signature $s_{i,t_i+1} = G\text{-}Sig_u(\ d'_{i,t_i+1}, c_{i,t_i+1}, e_{i,t_i}, e'_{i,t_i}, l_{i,t_i+1}, b_{i,t_i+1})$ and attaches the signature to D_{i,t_i+1}.

Uploading After each retrieved data block D_{i,t_i} has been updated to D_{i,t_i+1}, the block should be uploaded back to the server at layer l_{i,t_i+1} and bucket b_{i,t_i+1}. The order in which the blocks are uploaded is arbitrary. Also, the value of C_q is incremented by one and a new group signature is generated for the concatenation of the new C_q and the new content of the first block at the top layer; then, these two are also uploaded to the server. If the new value of C_q is a multiple of 2^l for certain $l \in \{1, \cdots, \log n\}$ but not a multiple for any $2^{l'}$ where $l' > l$, a shuffling process for layer l should be conducted by this user. The shuffling process is elaborated in Appendix 1.

Server Operations Upon receiving an uploaded block D_{i,t_i+1}, the server needs to check whether (i) the block has a valid group signature; (ii) the root hash value carried by the block, i.e., e_{i,t_i}, is the same as the root hash of the Merkle hash tree constructed when the blocks were downloaded during the previous access; and (iii) the value in auxiliary field e'_{i,t_i} is different from that of other blocks.

6 Security Analysis

In this section, we present the security analysis of the proposed scheme. We firstly explain that the proposed scheme ensures the same access pattern privacy offered by the underlying ORAM. Then we show that any misconduct committed by malicious users or the server can be detected and the identity of the intruder can be traced.

6.1 Access Pattern Privacy

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

Proof Sketch: Due to space limitation, we only provide an informal sketch of the proof. Firstly, the proposed format of the data block, which consists of $d'_{i,t}$, $c_{i,t}$, $e_{i,t-1}$, $e'_{i,t-1}$, $l_{i,t}$, $b_{i,t}$, and $s_{i,t}$, does not leak additional information to the server. In other words, the data re-encryption semantics remain intact. Secondly, the modified query and shuffling processes do not break the randomness and obliviousness of the underlying ORAM. For example, in the modified query process, the target data block is swapped with one of the dummy data blocks at the top layer; this is indeed equivalent to the query process of the original ORAM, which uploads the target data block to the top layer while the data block at its original location essentially becomes a dummy.

□

6.2 User Accountability

We now show that any misconduct by a malicious user will be detected either by the server or by the honest user who next accesses the modified data block. Note that, in the following analysis, we assume that the server always performs the validating operations honestly as described in Section 5.3. The detection of server misbehavior is analyzed in Section 6.3.

Modification Attacks As the user does not have access to the private key K^-, any arbitrary modification to the data content will be detected due to the presence of digital signature $Sig_{K^-}(d_i)$ in each data block D_{i,t_i}, and the user who last accessed the data block is identified as the attacker through $G\text{-}Open(s_{i,t_i}, d_i)$.

Replacement Attacks Even though a malicious user cannot modify the content of a data block arbitrarily, it may replace data content of a block D_{i,t_i} with data content of another block D_{j,t_j}. In this case, data content d_i is essentially lost and the digital signature $Sig_{K^-}(d_i)$ alone cannot detect such type of misconduct. Our proposed scheme deals with replacement attacks as follows.

If a malicious user simply replaces d'_{i,t_i} in D_{i,t_i} with d'_{j,t_j} from D_{j,t_j} (or $d'_{i/j,t'_{i/j}}$ where $t'_i < t_i$ and $t'_j < t_j$), without replacing other parts in D_{i,t_i}, this can be detected via (i) a mismatch between d'_{i,t_i} and its hash value c_{i,t_i}, or (ii) a failed Merkle hash tree verification for d'_{i,t_i} based on e_{i,t_i-1} and e'_{i,t_i-1}.

Instead, a malicious user may replace d'_{i,t_i}, c_{i,t_i}, e_{i,t_i-1}, and e'_{i,t_i-1} in D_{i,t_i} with their counter parts from D_{j,t_j}. This can be detected too as the server would observe two data blocks with the same auxiliary information field e'_{i,t_i-1} and hence reject the uploading operation and identify the user as the attacker through $G\text{-}Open(s_{j,t_j}, d_j)$.

Misplacement Attacks A malicious user may place a data block in the wrong bucket and/or layer of the storage hierarchy. This attack may 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 bucket. 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 of that block.

Addition/Deletion Attacks In the proposed scheme, during a query or shuffling, as the server always checks to ensure that the set of downloaded data blocks and the set of uploaded data blocks are identical, any addition/delection attacks can be detected and rejected by the server.

6.3 Server Accountability

In this section, we first explain how the proposed scheme detects the server misbehavior when it does not perform the required validating operations when a user uploads data blocks to the server.

- If the server does not verify the group signature of a data block, the misconduct will be detected as soon as an honest user finds that a data block carries an invalid signature.
- If the server does not verify the root hash e_{i,t_i} and/or the auxiliary information field e'_{i,t_i} of a data block, the misconduct will be detected when an honest user finds that a data block is not what it wants (i.e., a mismatch of the data block IDs); in other words, a replacement attack has passed through the server without being detected.

The server may also launch attacks in an active manner, including modification of data blocks and disruption of the underlying ORAM. Modification of data blocks can be detected as the server cannot generate a valid group signature, while disruption of the underlying ORAM can be detected by the user via validating l_{i,t_i}, b_{i,t_i}, and s_{i,t_i}.

7 Overhead Analysis

We now present the overhead analysis of the proposed scheme. In this paper, we only show the extra storage, communication and computational overhead that are added to the underlying ORAM scheme.

7.1 Storage Overhead

Server Overhead Extra storage overhead on the server is introduced as a result of expanding data block. For each data block D_{i,t_i}, the extra storage overhead includes r_{i,t_i}, $Sig_{K^-}(d_i)$, 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}. Table 1 shows a practical example, where $n = 2^{30}$ real data blocks are in the system. We assume each hash value is of 32 bytes and the group signature s_{i,t_i} takes 230 bytes. Thus, each of r_{i,t_i}, $Sig_{K^-}(d_i)$, c_{i,t_i}, and e_{i,t_i-1} is of 32 bytes. l_{i,t_i} and b_{i,t_i} store layer and bucket number. Since there are $\log n$ layers and at most n buckets on one 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) thus e'_{i,t_i-1} stores at most $\log(n \log n)$ hash values. But in practice, this is still a small amount of extra overhead. For example, assuming $n = 2^{30}$ 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. Considering that a data block is typically 64 KB or 256 KB, the extra overhead is less than 2 KB, which is acceptable in practice.

Table 1. Storage overhead ($n = 2^{30}$) per block

r_{i,t_i}	$Sig_{K^-}(d_i)$	c_{i,t_i}	e_{i,t_i-1}	e'_{i,t_i-1}	l_{i,t_i}	b_{i,t_i}	s_{i,t_i}
32 bytes	32 bytes	32 bytes	32 bytes	$\leq$ 1.1 KB	$\leq$ 1 byte	4 bytes	230 bytes

User Overhead A user also needs an extra storage space to store the hash values to construct the Merkle hash tree. In a query process, the number of leaf nodes in the Merkle hash tree is $O(\log^2 n)$, which in turn requires the user to store $O(\log^2 n)$ hash values. This is a small amount of overhead compared to the memory space available on a typical user device. For example, assuming $n = 2^{30}$ and each hash value is 32 bytes, the storage overhead for storing the Merkle hash tree is around 60 KB, which is comparable to the size of a data block. Note that during a shuffling process, there could be as many as $O(n \log n)$ leaf nodes in a Merkle hash tree. However, as explained in

Appendix 1, the Merkle hash tree is pre-computed by the server and the user only needs to verify the Merkle hash tree information it retrieves; hence, no storage overhead is introduce for this process.

7.2 Communication Overhead

Downloading Ignoring the extra overhead caused by the increase of data block size, the downloading communication overhead is exactly the same as that of the original ORAM scheme.

Uploading For a query process, our proposed scheme has a bigger uploading communication overhead than the original ORAM scheme, as the user needs to upload all retrieved $O(\log^2 n)$ data blocks back to the server. For a shuffling process, however, our proposed scheme does not introduce extra uploading overhead compared to the original ORAM scheme. Consequently, the amortized communication overhead (which is $O(\log^3 n)$ or $O(\log^4 n)$, depending on the choice of the sorting algorithm) is not different for the two schemes.

7.3 Computational Overhead

Server Overhead During a query or shuffling process, the server needs to perform the following extra computations: (i) calculation of a Merkle hash tree, which needs $O(n \log n)$ hash computations in the worst case; (ii) verification of group signature for each data block, which can be $O(n \log n)$ in the worst case. Note that, both of the above calculations can be parallelized on the server. Hence, the computation overhead introduced by the proposed scheme should not be an obstacle for the massively parallelized cloud computing platforms.

User Overhead For each data block D_{i,t_i} retrieved in a query or shuffling process, a user needs to perform the following extra computations: (i) verification of digital signature $Sig_{K^-}(d_i)$; (ii) verification of c_{i,t_i} in the old data block and generation of c_{i,t_i+1} for the new 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 Merkle hash tree root e'_{i,t_i-1}, which is composed of a sequence of hash computations; (v) verification of placement of D_{i,t_i}, which is one hash computation; and (vi) verification of group signature s_{i,t_i}. Using a modern pairing-based cryptography library such as PBC library [20], each of the above computations can be done efficiently from several milliseconds to several hundreds of milliseconds.

8 Conclusions

In this paper, we propose an accountable ORAM scheme, which is developed on top of a hash-based ORAM design such as [15]. It can detect misconduct committed by

malicious users or server and identify the attacker, while not interfering with the access pattern preservation mechanisms inherent from the underlying ORAM. The goal is achieved through a combined leverages of Merkle hash tree, digital signature and group signature techniques, as well as a delicate design of the data block format. Results of security analysis and overhead evaluations show that the proposed scheme has achieved the goals of accountability of users and the server's behavior and preservation of data access pattern privacy, at the cost of slightly increased computational, storage, and communication overheads.

References

1. M. Islam, M. Kuzu, and M. Kantarcioglu., "Access pattern disclosure on searchable encryption: ramification, attack and mitigation," in *NDSS*, 2012.
2. P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: practical access pattern privacy and correctness on untrusted storage," in *Proc. CCS*, 2008.
3. P. Williams and R. Sion, "Usable private information retrieval," in *Proc. NDSS*, 2008.
4. B. Pinkas and T. Reinman, "Oblivious ram revisited," in *Proc. Crypto*, 2010.
5. M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious ram simulation," in *Proc. ICALP*, 2011.
6. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Oblivious ram simulation with efficient worst-case access overhead," in *Proc. CCSW*, 2011.
7. P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *Proc. CCS*, 2012.
8. ——, "Privatefs: A parallel oblivious file system," in *Proc. CCS*, 2012.
9. E. Kushilevitza, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious ram and a new balancing scheme," in *Proc. SODA*, 2012.
10. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious ram simulation," in *Proc. SODA*, 2012.
11. E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o((logn)3) worst-case cost," in *Proc. ASIACRYPT*, 2011.
12. E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," in *Proc. ASIACRYPT*, 2011.
13. E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *Proc. CCS*, 2013.
14. E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," in *Proc. S&P*, 2013.
15. O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious ram," in *JACM'96*, 1996.
16. R. Merkle, "A digital signature based on a conventional encryption function," in *Proc. Crypto*, 1987.
17. D. Chaum and E. van Heyst, "Group signatures," in *Proc. EUROCRYPT*, 1991.
18. M. Franz, P. Williams, B. Carbunar, S. Katzenbeisser, A. Peter, R. Sion, and M. Sotakova, "Oblivious outsourced storage with delegation," in *Proc. FC*, 2011.
19. B. Carbunar and R. Sion, "Write-once read-many oblivious ram," *Trans. Info. For. Sec.*, vol. 6, no. 4, Dec. 2011.
20. B. Lynn, "On the implementation of pairing-based cryptosystems," Ph.D. dissertation, Stanford University, 2008.

Appendix 1 Shuffling Process

Essentially, the shuffling process for layer l is to obliviously reposition and reformat all the blocks residing at layers $1,\cdots, l$ such 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_i} is placed to bucket $h_l(\frac{C_q}{2^i}, i)$ of layer l, where C_q is the afore-mentioned counter keeping track of the number of queries that have been processed. Incorporating accountability mechanisms, the detailed operations are as follows. An example where l=2 is shown in Fig. 2.

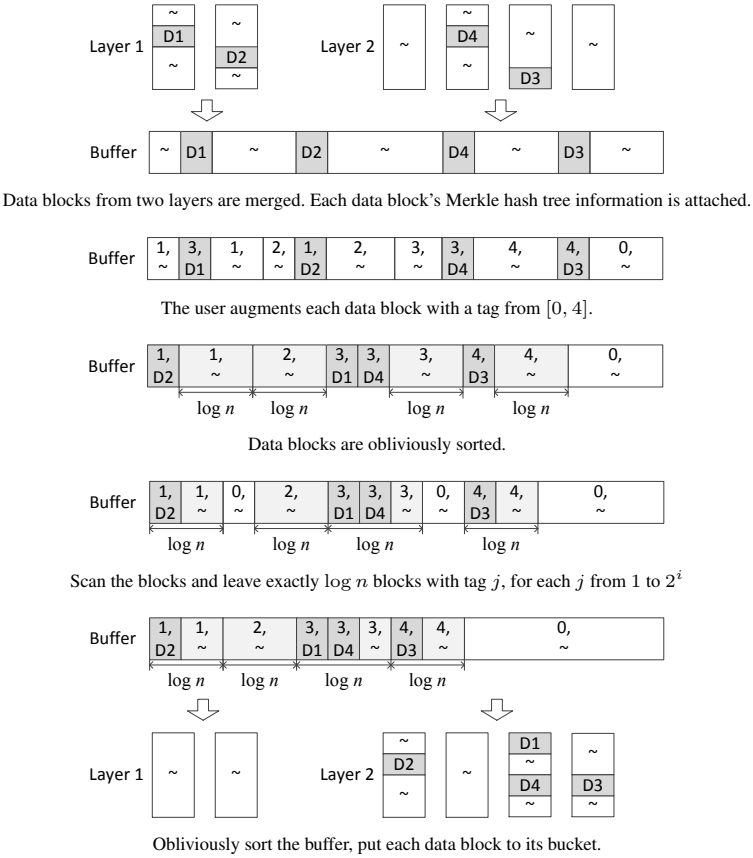

Fig. 2. Example of the shuffling process. Each bucket has $\log n$ data blocks. We use the symbol $\sim$ to represent one or more adjacent dummy data blocks.

(S1) The server merges all the data blocks at layers $1, \cdots, l$ to a *shuffling buffer*, builds a Merkle hash tree with the block hash fields of these blocks as leaf nodes, and calculates the root hash. Then, for each block D_{i,t_i} in the buffer, the corresponding $\mathbb{S}_{i,t_i}$ is attached to the block, and the root hash value is saved to field e_{i,t_i-1} of 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_i} has been downloaded, the validity of the attached $\mathbb{S}_{i,t_i}$ and e_{i,t_i-1} is first checked as follows: (i) If this is not the first downloaded block, it is checked if e_{i,t_i-1} is the same as the root hash saved in the previously downloaded blocks; (ii) The root hash is re-computed based on c_{i,t_1} and $\mathbb{S}_{i,t_i}$ to check if it is the same as e_{i,t_i-1}. If either of the checks fails, the server is detected to be misbehaving. Then, the block is further processed as follows to obtain D_{i,t_i+1} of different appearance: (i) A new nonce r_{i,t_i+1} is picked uniformly at random from $\{0, \cdots, 2^L - 1\}$. (ii) e'_{i,t_i} is updated to $E_k(r_{i,t_i+1}, \mathbb{S}_{i,t_i})$ to save an encrypted version of $\mathbb{S}_{i,t_i}$. (iii) A tag T_i is assigned to indicate which bucket the block should reside after the shuffling. Particularly, if the block is a real data block of ID i, $T_i = h_l(C_q/2^l, i)$; otherwise (i.e., the block is a dummy block), T_i is picked from $\{0, \cdots, 2^i\}$ such that, after the assignment has been performed for all the dummy blocks in the buffer, $\log n$ dummy blocks are assigned with tag j for each $j = 1, \cdots, 2^i$ while the rest dummy blocks are assigned with tag 0. (iv) The tag T_i is encrypted and saved in field b_{i,t_i+1}; that is, $b_{i,t_i+1} = E_k(r_{i,t_i+1}, T_i)$. And l_{i,t_i+1} are all set to l. (v) d'_{i,t_i+1} is re-computed as $E_k(r_{i,t_i+1}, d_i, sig_{K^-}(d_i))$, and $e_{i,t_i} = e_{i,t_i-1}$ to save the root hash value. (vi) the group signature is re-computed for the whole block.

(S3) In this step, the user conducts an oblivious sorting for all the blocks in the shuffling buffer, based on the tags carried by the blocks. As a result, the blocks are placed in the shuffling buffer according to the ascending order of their tags, and for blocks of the same tag, the real data blocks are placed before dummy blocks.

(S4) Again, the user scans the blocks in the shuffling buffer, one by one. This time, the tags of some dummy blocks are adjusted to ensure that, for each $j = 1, \cdots, 2^l$, exactly $\log n$ (real or dummy) data blocks are assigned with tag j, while other dummy blocks are assigned with tag 0. Based on the sorting result of Step 2, the tag adjustment can be conducted as follows by using a temporary counter: For each $j = 1, \cdots, 2^l$, when the first block with tag j is scanned, the counter is initialized to 1. Later on, when a block of the same tag is scanned, the tag of the block remains unchanged and the counter is incremented by 1, if the counter is smaller than $\log n$; otherwise (i.e., the counter is equal to $\log n$), the block should be re-tagged with 0.

(S5) In this step, the user conducts another oblivious sorting for all the blocks in the shuffling buffer, based on the tags carried by the blocks. As a result, the blocks are placed in the shuffling buffer according to the ascending order of their tags; for blocks of the same tag, however, the data blocks are placed randomly.

(S6) In this step, a third scan is performed in this step to specify the new location of each block in the shuffling buffer. Specifically, according to the order produced by Step 4, the blocks are placed into the buckets from layer 1 to layer l and from bucket 1 to bucket 2^j for each layer j. Also, for each block D_{i,t_i+1}, the l_{i,t_i+1} and b_{i,t_i+1} fields save its assigned layer number and bucket number; the e'_{i,t_i+1} field

just contains an encrypted version of its $\mathbb{S}_{i,t_i}$ (without random nonce) to allow the server to check if the block carries a unique e'_{i,t_i+1} value.

Note that, similar to the download and upload operations in the query process, the user needs to check the integrity of the content and placement of each block every time when it is downloaded to detect if the server has misbehaved, and the server also needs to check the integrity of a block every time when it is uploaded; specially, in the last time when all data blocks are uploaded back to buckets, the server needs to check if the uploaded set is the same as the downloaded set.

Appendix 2 Integration with General Hash-Based ORAMs

For the sake of simplicity, we use a specific ORAM [15] to explain the proposed scheme. However, the proposed scheme can also be extended to work with other hash-based hierarchical ORAM scheme. The high level insights of the proposed scheme is to add validation information to each data block and verify these information collaboratively between the server and the users, without interfering with the original ORAM operations. The proposed scheme can be integrated with any hash-based ORAM as long as:

- The extended data format proposed in the scheme does not give the server non-negligible advantages in inferring users' access pattern in the ORAM scheme.
- During a regular query, the ORAM scheme still operates successfully if the user puts all retrieved data blocks back to the server, in the way as explained in Section 5.3.
- During the shuffling process, the re-hashing algorithm and the underlying oblivious sorting algorithm still works if no new dummy data blocks are added as explained in Appendix 1.

Many existing ORAM schemes [2–10] satisfy these requirements. Note that there also exist other ORAM schemes [11–14] that are not hash-based. How to achieve user accountability in these ORAM schemes will be studied in our future work.