MU-ORAM: Dealing with Stealthy Privacy Attacks in Multi-User Data Outsourcing Services

Jinsheng Zhang
Department of Computer Science
Iowa State University
Ames, Iowa, 50010, USA
alexzjs@iastate.edu

Wensheng Zhang
Department of Computer Science
Iowa State University
Ames, Iowa, 50010, USA
wzhang@iastate.edu

Daji Qiao
Department of Electric and Computer Engineering
Iowa State University
Ames, Iowa, 50010, USA
daji@iastate.edu

ABSTRACT

Outsourcing data to remote storage servers has become more and more popular, but the related security and privacy concerns have also been raised. To protect the pattern in which a user accesses the outsourced data, various oblivious RAM (ORAM) constructions have been designed. However, when existing ORAM designs are extended to support multi-user scenarios, they become vulnerable to stealthy privacy attacks targeted at revealing the data access patterns of innocent users, even if only one curious or compromised user colludes with the storage server. To study the feasibility and costs of overcoming the above limitation, this paper proposes a new ORAM construction called *Multi-User ORAM (MU-ORAM)*, which is resilient to stealthy privacy attacks. The key ideas in the design are (i) introduce a chain of proxies to act as a common interface between users and the storage server, (ii) distribute the shares of the system secrets delicately to the proxies and users, and (iii) enable a user and/or the proxies to collaboratively query and shuffle data. Through extensive security analysis, we quantify the strength of MU-ORAM in protecting the data access patterns of innocent users from attacks, under the assumption that the server, users, and some but not all proxies can be curious but honest, compromised and colluding. Cost analysis has been conducted to quantify the extra overhead incurred by the MU-ORAM design.

1. INTRODUCTION

Recent development of cloud computing has witnessed the convenience of remote storage services. Nowadays, many cloud storage providers [1,8,18] are offering large amount of inexpensive storage space to individual and enterprise users. Users can outsource their data and access them remotely from their resource-constrained devices in a pay-per-use manner.

Although a user may encrypt outsourced data to protect confidentiality of the data content, the user's data access pattern remains unprotected and can still reveal the user's private information [20]. To address this issue, various oblivious RAM (ORAM) constructions [13–17, 21, 29, 33, 34, 40, 42, 43] have been proposed to protect a user's data access pattern against a *semi-honest* remote storage server, who honestly hosts the data and serves the user, but is curious to find out the data access pattern of the user.

Most of existing ORAM constructions assume only a single user to interact with the storage server; therefore, the user's device holds all the system secrets about how the outsourced data are encrypted, placed and scrambled in the server's storage. As it is also popular for multiple users to share outsourced data, such constructions [11, 17, 23, 42] have been proposed to extend the single-user ORAM to support parallel accesses from multiple users. In these proposals, however, the users essentially work together as a single user, because either the users need to go through a single proxy which holds the system secrets and interacts with the server on behalf of all users, or each of the users should hold the same system secrets and interacts with the server directly. In either case, the single proxy or any one user becomes a single point of security failure. If it is malicious or compromised, attacks can be launched from the inside, and the security of the whole system can be easily brought down. Observable attacks (e.g., illegitimate deletion or modification of data) launched by the insider attacker can be detected, and the attacker can be identified with some accountability mechanisms (e.g., auditing the logs), but *detecting stealthy attacks targeted at privacy is much more difficult*. A curious or compromised user can collude with the storage server (if the server is also curious or compromised) to reveal the access patterns of all other users; meanwhile, the attackers can keep their attacks stealthy, because they still follow the ORAM protocols without extending any anomaly observable by others.

For instance, a hospital may wish to export the encrypted information of all its patients, to a remote storage organized as an ORAM. To allow each doctor to access the data of any patient who has visited the hospital, all the doctors should share the same secret keys. With such a system, if a doctor is curious or the account of a doctor is compromised by an attacker, the adversary (i.e., the curious doctor or the attacker) may be able to observe the accesses made by all other doctors, through colluding with the storage server which is also curious or compromised, without launching any observable attack to the ORAM.

To study the feasibility and cost of overcoming the above limitation of existing ORAM constructions, we propose, design, and analyze a new ORAM construction called *Multi-User ORAM (MU-ORAM)*. The construction has two design goals. First, it shall support multiple users to share data outsourced to a remote storage. Second, it shall be resilient to the afore-described *stealthy privacy* attacks, in which the curious or compromised insider attackers do not extend observable misbehaviors, but collude stealthily to reveal the data access patterns of innocent users. To the best of our knowledge, this is the first effort aiming to attain these goals.

To tolerate stealthy privacy attacks, the basic principle is to distribute the shares of the system secrets among the users, instead of letting every user to hold all the system secrets. This way, any single user alone will not have sufficient secrets to locate and decrypt a data block of interest to access; collaboration between the users is required. However, when a user needs to access a data block, it is not realistic to require other users to be online and available for collaboration. Hence, the key idea in our design is to introduce a chain of collaborative but mutually independent *proxies* between users and the storage server. These proxies are always online, like the storage server. The shares of the system secrets are distributed delicately to the proxies and the users. When a user needs to query a data block, its request and the storage server's replies shall pass through and be processed by the proxies before they reach the destination.

In practice, the proxies can be implemented as mutually independent hardware components (e.g., computers) or software components (e.g., virtual machines) provided in public or private domains. For instance, in the afore-mentioned "hospital" example, the proxies can be implemented as several physical/virtual machines running in the premise of the hospital or some cloud providers independent of the remote storage server.

Within this architecture, (i) users do not need to hold all the system secrets as they do not interact directly with the storage server; (ii) each user can set up a secure and logically isolated communication channel with the chain of proxies; (iii) multiple proxies, with each holding an independent share of the system secrets, work together to act as a common interface between users and the storage server. They also take non-user-specific workload (e.g., data shuffling). Due to the above features, users are securely isolated from each other, and compromising some but not all proxies cannot capture the system secrets. Thus, the system becomes more resilient to the stealthy privacy attacks.

We propose formal security definitions to quantify the security strength of MU-ORAM in protecting an innocent user's data access patterns against stealthy attacks, and conduct extensive analysis:

- First, we have shown that, like existing single-user ORAMs, MU-ORAM can fully protect the access pattern privacy of each individual user against an semi-honest storage server with a failure probability of $O(N^{-\log\log N})$, where N is the total number of exported data blocks.
- Second, assuming that the server, some users and some but not all proxies are semi-honest and colluding, we study the security strength of MU-ORAM under different scenarios. Particularly, we have shown that, the collusive coalition has an advantage of less than 2ϵ within time period t to reveal an innocent user's access to data that the coalition is not authorized to access, if the Modified Matching Diffie-Hellman (MMDH) problem cannot be solved with an advantage of at least ϵ within the same time period t.

Note that, as our design aims at dealing with stealthy privacy attacks, the threat model of our security analysis assumes that the attackers are semi-honest (i.e., the attackers honestly follow the protocols that they are expected to execute, but may take extra actions to reveal the data access patterns of innocent users).

Cost analysis has been conducted to quantify the costs incurred to provide the protection. The results show that, the communication cost introduced by MU-ORAM is $O(\log^2 N)$ data blocks per query for the user and $O(\log^2 N \log\log N)$ for the proxies. Meanwhile, MU-ORAM does not store any dummy data blocks, which makes the server-side storage to be $O(N)$.

The rest of the paper is organized as follows. System model, design goal, and security definitions are introduced in Section 2. Section 3 elaborates the design details. The results of security and overhead analysis are reported in Sections 4 and 5, respectively. Section 6 provides a brief comparison to related works. Finally, Section 7 concludes the paper.

2. PRELIMINARIES

This section presents the system model, the architecture of our proposed MU-ORAM, and the formal definitions of security.

2.1 System Model

We consider a system where multiple users share N data blocks exported to a storage server. Let F_p be a finite field with p distinct elements, where p is a prime number and $N \ll p$. For example, $\log p$ is usually 128 or larger, while in practice $\log N$ is seldomly greater than 40. Let G_p be a multiplicative, cyclic group with also p distinct elements. Each data block, denoted as D_i, consists of two components: (i) unique data ID denoted as g_i which is an element of G_p; (ii) data content that is a sequence of pieces each being an element of G_p. As the operations on each piece of the data content are the same, we use a single element denoted as d_i to represent the sequence unless stated otherwise. Hereafter, each data block D_i is represented as

$$(g_i,\ d_i) \quad \text{where } g_i \in G_p \text{ and } d_i \in G_p. \tag{1}$$

Each data request from a user, which shall be kept confidential, is one of the following two types: (i) read a data block d_i of unique ID g_i from the storage, denoted as a 3-tuple $(read, g_i, d_i)$; or (ii) write/modify a data block d_i of unique ID g_i to the storage, denoted as a 3-tuple $(write, g_i, d_i)$.

To accomplish a confidential data request, the user may need to access the remote storage multiple times. Each access to the remote storage can be observed by the server and its collusive coalition, and is one of the following two types: (i) retrieve (i.e., read) a data block d_i from a location l at the remote storage, denoted as a 3-tuple $(read, l, d_i)$; or (ii) upload (i.e., write) a data block d_i to a location l at the remote storage, denoted as a 3-tuple $(write, l, d_i)$.

Also, we assume there is a trusted system initialization server. This server is not involved in data access, but only responsible for initializing the system and providing public information for a user when the user joins the system. Note that, once the system initialization finishes, all system secrets are removed from this server. Therefore, we assume the server is immune from attacks.

2.2 Proposed Architecture

As stated in Section 1, MU-ORAM is designed to protect the data access patterns of individual users against stealthy privacy attacks launched by collusive parties in the system. To attain this goal, we

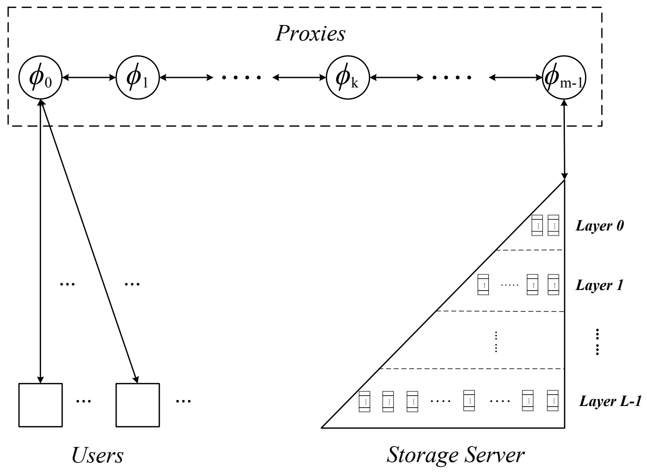


Figure 1: System overview.

propose a new architecture (as shown in Figure 1) composed of a hierarchical storage server, multiple users, and a chain of *proxies* as a bridge between users and the storage server. In practice, proxies can be implemented as mutually independent hardware components (e.g., computers) or software components (e.g., virtual servers). These proxies can be deployed in the premise of the users or some cloud providers independent of the provider of the storage server.

Specifically, the introduced chain of proxies serves as a common interface for all users to access data at the storage server as follows.

- When a user needs to access a certain data block, the request and the data replied from the storage server shall pass through and be processed (i.e., encrypted or decrypted) by all the proxies before they reach either the storage server or the user.
- By introducing proxies to protect users from direct interactions with the storage server, each individual user does not need to maintain the information about storage locations or encryption keys of the data shared with other users. Without exposing such knowledge to individual users, it becomes possible to prevent a user from learning other users' data access patterns through colluding with the storage server or observing their interactions with the storage server.
- Such an architecture also allows each user to establish a secure and logically isolated communication channel with the chain of proxies, which makes it possible to prevent a user from learning other users' data access patterns through observing their interactions with the proxies.
- As all the user/server interactions must go through the entire chain of independent proxies, the user's access pattern privacy is protected, as long as not all of the proxies are compromised and collude with the storage server.

Under this proposed architecture, appropriate algorithms must be designed to guide the interactions between the storage server, proxies, and users. We will present these algorithms in Section 3.

2.3 Security Definitions

As the major goal of our design is to protect individual users' access pattern privacy from stealthy attacks, we assume the storage server, users, and proxies in the system are *honest but curious* or called *semi-honest*. Specifically:

- In response to a data query from a user, the user, the proxies and the storage server follow the query protocol honestly to process the query.
- At the time when data shuffling shall be conducted, we assume that the storage server and the proxies all follow the shuffling protocol honestly to shuffle the data.
- The storage server, each proxy, and each user may be curious to find out the access pattern of other users. To do so, they may collude. However, we assume no collusive coalition will include all proxies.

2.3.1 Security against semi-honest storage server

As a baseline, we first consider the scenario that the storage server does not collude with any user or proxy. Following the security definition of ORAMs [13,36,37], we define the security of an MU-ORAM against an honest but curious storage server as follows.

Definition 1. (*Security against semi-honest storage server*). Let $\vec{x} = \langle\ (op_1, i_1, d_1), (op_2, i_2, d_2), \cdots\ \rangle$ denote a private sequence of a user's intended data requests, where each op is either a $read$ or $write$ operation. Let $A(\vec{x}) = \langle\ (op'_1, l_1, d'_1), (op'_2, l_2, d'_2), \cdots\ \rangle$ denote the sequence of the user's accesses to the remote storage (observed by the server), in order to accomplish the user's private data requests. MU-ORAM is said to be secure if (i) for any two equal-length private sequences $\vec{x}$ and $\vec{y}$ of intended data requests, their corresponding observable access sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable; and (ii) the probability that MU-ORAM fails to operate is $O(N^{-\log\log N})$.

2.3.2 Security against collusive coalition

Next, we consider the more general scenario that the storage server colludes with some users and some (but not all) proxies.

Depending on whether the collusive users have authorized access to the data blocks queried by an innocent user, the security strength of MU-ORAM can be very different. This is because, when the collusive users have access to the data accessed by the innocent user, the collusive users can check if some data blocks have been changed after the innocent user's access to infer the innocent user's access pattern; however, this approach cannot be applied when the collusive users are not authorized to access the data accessed by the innocent user. Hence, we study two cases separately as follows.

Case 1: Users with same access privileges to data. In a system where users have the same access privileges to outsourced data, we study the security strength of MU-ORAM in protecting an innocent user's access pattern to the data that can also be accessed by the collusive users.

To facilitate the study, we define a game between an adversary (i.e., the collusive coalition) and a challenger (i.e., the rest of the system) in the following. Intuitively, the game models the attacks that can be launched by the adversary: it can launch queries and observe how these queries are handled; it can observe the interactions between the innocent user and the server and proxies; it can compromise and thus obtain the secrets of some but not all proxies; it can inspect data stored on the storage server. The adversary is said to have won the game (i.e., defeated the MU-ORAM) if the following happens: the innocent user first selects two data blocks uniformly at random to query; the user is then asked to randomly choose one of these two data blocks to query again; the adversary is able to find out the user's choice.

Definition 2. A game $\mathcal{G}_1(\mathcal{M}, p, N, m, n_{cq})$ between a *challenger* and an *adversary* is defined as follows (Here, $\mathcal{M}$ denotes an MU-ORAM construction):

- *Initialization Phase.* The challenger initializes the storage server and the chain of m proxies, according to the algorithm of $\mathcal{M}$. Here, N data blocks $\{(g_i, d_i)|i = 0, \cdots, N-1; g_i \in G_p; d_i \in G_p\}$ are exported to the storage server. The adversary has access to all the data block IDs.
- *Query Phase I.* The adversary can make any number of queries of the following types.
  - *Proxy Compromising.* The adversary requests to get the information (e.g., secrets) owned by any compromised proxy. We restrict that at most $m - 1$ proxies can be compromised.
  - *Proxy and Server Transcript Inspection.* The adversary requests to get the input/output of any compromised proxy and the storage server.
  - *Data Query.* Two types of queries can be requested:
    * Type I (controlled queries) - The adversary selects an ID and acts as a user to start querying the data block of this ID. In response, if the number of Type I query has exceeds n_{cq}, the request is denied; otherwise, the proxies and the server follow the $\mathcal{M}$ protocol to process the query request.
    * Type II (random queries) - The adversary requests an innocent user to start a query. In response, the challenger secretly selects an ID from the pool of IDs uniformly at random, and then acts as a user to start querying the data block of this ID. The proxies and the server follow the $\mathcal{M}$ protocol to process the query. Note that, this selected ID is unknown to the adversary.
  - *Storage Inspection.* The adversary asks the storage server to return the data blocks in a specified bucket.
- *Selection Phase I.* The challenger secretly selects a data block ID denoted as θ_0 from the pool of IDs uniformly at random, and queries it. Note that, θ_0 is known only by the challenger.
- *Query Phase II.* The phase is the same as Query Phase I, except that the following rule should be added when processing a Type I data query: the challenger aborts the game and declares failure if the queried data ID is θ_0, θ_0 was queried by the adversary before the Selection Phase I, and there is no Type II query for θ_0 between the adversary's last and current query for θ_0. This is because, when the above conditions are satisfied, the adversary will find that the content of data block θ_0 was changed after the Selection Phase I, and thus find θ_0 was queried in the Selection Phase I; therefore, it will know which of θ_0 and θ_1 is selected in the later Challenge Phase by simply querying θ_0 right after the Challenge Phase.
- *Selection Phase II.* The challenger secretly selects another data block ID denoted as θ_1 ($\theta_0 \neq \theta_1$), and queries it.
- *Query Phase III.* The phase is the same as Query Phase I, except that the following rule should be added when processing a Type I data query: the challenger aborts the game and declares failure if either (i) the queried data ID is θ_0, θ_0 was queried by the adversary before the Selection Phase I, and there is no Type II query for θ_0 between the adversary's last and current query for θ_0; or (ii) the queried data ID is θ_1, θ_1 was queried by the adversary before the Selection Phase II, and there is no Type II query for θ_1 between the adversary's last and current query for θ_1. This change is due to the same reason explained in Query Phase II.
- *Challenge Phase.* The challenger decides a binary bit b uniformly at random. Then, it queries the data block of ID θ_b.
- *Query Phase IV.* The phase is the same as Query Phase I, except that the following rule should be added when processing a Type I query: if θ_0 or θ_1 is queried, the challenger aborts the game and declares failure. Note that, the adversary may or may not find out the query target chosen in the Selection Phases if it requests to query θ_0 or θ_1. Hence, by this rule we may under-estimate the security strength of MU-ORAM.
- *Response Phase.* The adversary returns a binary bit b' as a guess of the b.
- *Result.* The adversary wins the game if the challenger declares failure or $b' = b$; otherwise, it loses the game. The advantage for the adversary to win the game is defined as the probability that it wins the game minus $1/2$.

An MU-ORAM construction $\mathcal{M}$ is considered secure against a collusive coalition, if it is *hard* for an adversary with limited computational capability to win the above game. To quantify this notation, we introduce the following definition:

Definition 3. *((ϵ, t, n_{cq})-security against collusive coalition)* An MU-ORAM construction $\mathcal{M}$, in which all users have the same access privileges to the outsourced data, is said to be (ϵ, t, n_{cq})-secure against a collusive coalition of semi-honest storage server, users and some (but not all) proxies if: no adversary can win the game $\mathcal{G}_1(\mathcal{M}, p, N, m, n_{cq})$ with an advantage of at least ϵ under the time complexity of t and the restriction that the adversary cannot make more than n_{cq} Type I data queries (i.e., controlled queries) during the game.

Case 2: Users with different access privileges to data. In a system where users have different access privileges to data, we study the security strength of MU-ORAM in protecting an innocent user's access pattern to the data blocks that cannot be accessed by the collusive users. In the following, we present new game and security definitions.

Definition 4. A game $\mathcal{G}_2(\mathcal{M}, p, N, N', m)$ between a *challenger* and an *adversary* is defined similarly to $\mathcal{G}_1$ (in Definition 2) except for the following differences:

- In the Initialization Phase: the adversary is given only N' IDs from the totally N IDs.
- In the Query Phases I, II and III: there is no limitation on the number of Type I data queries that the adversary can make.
- In the Selection Phase I and II: θ_0 and θ_1 are two distinct IDs selected uniformly at random from the set of IDs that are unknown to the adversary.

To quantify the security strength of MU-ORAM in Case 2, we introduce the following definition:

Definition 5. *((ϵ, t)-security against collusive coalition)* An MU-ORAM construction $\mathcal{M}$, in which users have different access privileges to the outsourced data, is said to be (ϵ, t)-secure in protecting an innocent user's access to the data that a collusive coalition of semi-honest storage server, users and some (but not all) proxies are not authorized to access, if no adversary can win the game $\mathcal{G}_2(\mathcal{M}, p, N, N', m)$, where $N' \leq N - 2$, with an advantage of at least ϵ within time period t.

3. MU-ORAM

This section elaborates our proposed MU-ORAM design, which includes storage structure, system initialization, data query, and data shuffling. Figure 2 illustrates the overall workflow of data query and shuffling.

3.1 Storage Structure

MU-ORAM server organizes its storage as a hierarchy of buckets, and each bucket can store up to $\log N$ data blocks:

- The hierarchy consists of $L = \lceil \log N - \log\log N \rceil$ layers.
- Each layer l ($l = 0, \cdots, L-1$) has $n_l = 2^{l+1} \cdot \log N$ buckets. Hence, the top layer of the hierarchy (i.e., layer 0) has $2\log N$ buckets, while the bottom layer of the hierarchy (i.e., layer $L-1$) has N buckets.
- Each layer l is associated with a public hash function, denoted as $H_l(*)$, which maps each element of group G_p to one bucket at layer l.
- Each layer l has a bitmap to record whether each bucket at this layer is empty or not.

Note that, in MU-ORAM, there is no dummy data in its storage.

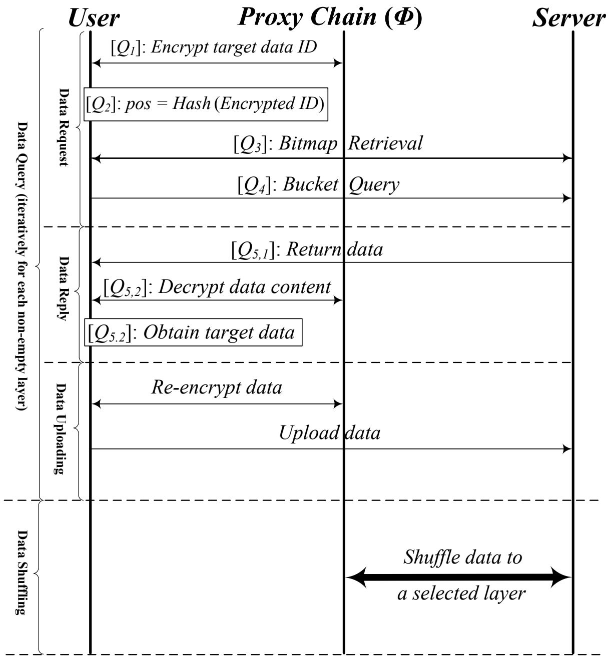
3.2 System Initialization

A trusted authority, which we call system initialization server, is responsible for initializing the system. The system initialization includes *proxy initialization*, *storage initialization*, and *user initialization*.

- The initialization server first picks z from $F_p \setminus \{0\}$ uniformly at random.
- Suppose there are m proxies, denoted as $\phi_0, \cdots, \phi_{m-1}$ in the system. For each ϕ_k ($k = 0, \cdots, m-1$), it is preloaded by the initialization server with the following keys: $x_k(l)$, $y_k(l)$ and $\Delta z_k(l)$ for each layer $l \in \{0, \cdots, L-1\}$, which are randomly picked from $F_p \setminus \{0\}$. These keys are used for encrypting data block IDs and contents. To facilitate presentation, we introduce the following notations:

$$x(l) = \prod_{k=0}^{m-1} x_k(l); \quad y(l) = \prod_{k=0}^{m-1} y_k(l);$$
$$\Delta z(l) = \prod_{k=0}^{m-1} \Delta z_k(l); \quad z(l) = z + \Delta z(l). \tag{2}$$

- The initialization server exports all the N data blocks to the bottom layer (i.e., Layer $L-1$) as follows: for each data block (g_i, d_i), it is encrypted to $(g_i^{x(L-1)}, (g_i^{-z(L-1)} d_i)^{y(L-1)})$, and stored to bucket $H_{L-1}(g_i^{x(L-1)})$.



User
Proxy Chain (Φ)
Server
Data Query (iteratively for each non-empty layer)
Data Request
[Q1]: Encrypt target data ID
[Q2]: pos = Hash (Encrypted ID)
[Q3]: Bitmap Retrieval
[Q4]: Bucket Query
Data Reply
[Q5,1]: Return data
[Q5,2]: Decrypt data content
[Q5,2]: Obtain target data
Data Uploading
Re-encrypt data
Upload data
Data Shuffling
Shuffle data to a selected layer

Figure 2: MU-ORAM Overview. The data query process includes the three phases of data request, data reply and data uploading, which is followed by the data shuffling process.

- For each user, when s/he joins the system, the initialization server preloads to him/her the public hash function $H_l(*)$ for each layer $l \in \{0, \cdots, L-1\}$. For each data block D_i that this user is authorized to access, the user is preloaded with tuple $(g_i, g_i' = g_i^{-z})$, where g_i is the ID of the data block D_i.

3.3 Data Query

When a user wants to query the data block of ID g_i from the storage server, she first needs to randomly select another ID denoted as g_j and also query the data block of ID g_j. Then, for each of the IDs g_i and g_j, the *data request*, *data reply* and *data uploading* phases shall be run sequentially for each of the non-empty layers from the top to the bottom of the storage hierarchy. As the processes for querying g_i and g_j are similar, in the following we only present how these phases are executed for non-empty layer l when g_i is queried.

3.3.1 Phase 1: Data Request

In this phase, the user determines a bucket on layer l and sends a request to retrieve data blocks from the bucket. The phase includes the following steps.

[Q1: Obtain Encrypted ID of the Query Target Data] The goal of this step is to compute the encrypted ID of the query target data block. As MU-ORAM uses the product of all proxies' secret keys as the encryption key, [Q1] requires a collaboration between the user and proxies, as shown in Figure 3. It consists of two sub-steps as follows.

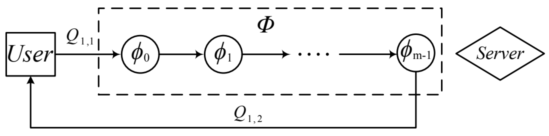


Figure 3: [Q1]: Obtain encrypted target data ID.

[Q1.1] In the first sub-step, the user sends the following message to proxy ϕ_0:

$$\left\langle g_i^{r_0}, g_i^{r_0}, (g_i')^{r_1} \right\rangle, \tag{3}$$

where r_0 and r_1 are two nonces randomly picked from $F_p \setminus \{0\}$.

[Q1.2] Upon receiving the message, each proxy ϕ_k ($k = 0, \cdots, m-2$) updates it and forwards to ϕ_{k+1}:

$$\left\langle (g_i^{r_0})^{\prod_{t=0}^{k} x_t(l)}, (g_i^{r_0})^{\prod_{t=0}^{k} \Delta z_t(l) y_t(l)}, ((g_i')^{r_1})^{\prod_{t=0}^{k} y_t(l)} \right\rangle. \tag{4}$$

Note that $x_k(l), y_k(l)$ and $\Delta z_k(l)$ are secrets preloaded to ϕ_k. After the message has traversed the entire proxy chain, it becomes

$$\left\langle (g_i^{r_0})^{\prod_{t=0}^{m-1} x_t(l)}, (g_i^{r_0})^{\prod_{t=0}^{m-1} \Delta z_t(l) y_t(l)}, ((g_i')^{r_1})^{\prod_{t=0}^{m-1} y_t(l)} \right\rangle$$
$$= \left\langle g_i^{r_0 x(l)}, g_i^{r_0 \Delta z(l) y(l)}, (g_i')^{r_1 y(l)} \right\rangle, \tag{5}$$

according to Equation (2). Then, the message is returned to the user by ϕ_{m-1}. Upon receiving the message, the user can obtain

$$g_i^{x(l)} = \left(g_i^{r_0 x(l)} \right)^{1/r_0} \tag{6}$$

and

$$g_i^{y(l)z(l)} = \left(g_i^{r_0 \Delta z(l) y(l)} \right)^{1/r_0} \cdot \left((g_i')^{r_1 y(l)} \right)^{1/r_1}, \tag{7}$$

respectively, as r_0 and r_1 are its self-generated nonces. Note that, $g_i^{x(l)}$ is the ID of the query target data block encrypted with the product of all proxies' secret keys, which will be used in [Q2]. $g_i^{y(l)z(l)}$ is stored locally at the user and will be used in [Q5: Data Reply].

[Q2: Compute Bucket for Access] Based on $g_i^{x(l)}$, the user computes the position pos of the bucket that may contain the target data block:

$$pos \leftarrow H_l \left(g_i^{x(l)} \right).$$

[Q3: Bitmap Retrieval] This step is to retrieve the bitmap that will be used by the user to decide the buckets to request. This is to avoid the situation where the user may attempt to retrieve an empty bucket at layer l; if this happens, the server would know for sure that D_t is not at this layer, thus leaking the information about D_t.

[Q4: Bucket Request] The user selects the bucket to request based on the retrieved bitmap as follows:

- If D_i has already been found at layer $l' < l$, the user randomly picks a non-empty bucket according to the bitmap.
- Otherwise, the user checks if the bucket at position pos is empty or not. If it is empty, the user randomly picks a non-empty bucket to access; else, the user accesses bucket pos.

Note that in [Q3], the user needs to retrieve a bitmap of $2^l \log N$ bits; when l is large, it is infeasible for the user to do so. To deal with this issue, the bitmap can be stored in a recursive manner. For example, suppose there are α bits in the bitmap. The server can create two bitmaps instead of one. In the first bitmap, it stores α bits and each bit indicates whether the corresponding bucket is empty or not. The second bitmap stores $\sqrt{\alpha}$ bits and each bit i ($0 \le i \le \sqrt{\alpha} - 1$) is set to 0 if all buckets from $\sqrt{\alpha} \cdot i$ to $\sqrt{\alpha} \cdot (i+1) - 1$ are empty. This way, [Q4] becomes:

- If D_i has already been found at layer $l' < l$, the user first requests the second bitmap of $\sqrt{\alpha}$ bits. According to the retrieved bitmap, the user randomly selects a "1" bit, say, at position P. Then, the corresponding segment indicated by P in the first bitmap is retrieved. At last, the user randomly picks a non-empty bucket from the segment.
- Otherwise, the user downloads the second bitmap and checks if the bit of position $P' = \lfloor \frac{pos}{\sqrt{\alpha}} \rfloor$ is 1. If it is 0, the user randomly picks a "1" bit (say, at position P) from the second bitmap; else, let $P = P'$. Next, the user retrieves the segment from the first bitmap that corresponds to P: (1) if bucket at position pos is not empty, it is selected; (2) otherwise, a non-empty bucket is randomly selected.

This way, the communication cost is reduced to $O(\sqrt{\alpha})$ bits. Indeed, the communication cost can be reduced further with more recursive levels introduced in the bitmap.

3.3.2 *Phase 2: Data Reply*

In response to the bucket request from the user, the storage server returns all the data blocks at the requested buckets to the user in two sub-steps: [Q5.1: From Server to User] and [Q5.2: From User to Proxies and back to User], as shown in Figure 4.

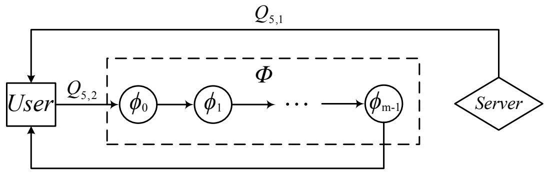


Figure 4: Phase 2: Data reply.

[Q5.1] The storage server returns all encrypted data blocks in the requested buckets to the user. Each data block has the following format:

$$\left(g_{i'}^{x(l)}, (g_{i'}^{-z(l)} d_{i'})^{y(l)} \right). \tag{8}$$

If $g_{i'}^{x(l)} = g_i^{x(l)}$ for a data block, it is the target data block. In this case, the data content part $\hat{d} = d_i^{y(l)}$ is encrypted by multiplying $(g_{i'}^{-z(l)} d_{i'})^{y(l)}$ with $g_i^{y(l)z(l)}$ obtained in step [Q1.2]; then the following step [Q5.2] is executed to decrypt $\hat{d}$ and obtain d_i.

Otherwise (i.e., none of the returned data blocks is the query target), the user randomly selects $\hat{d}$ from G_p and then starts step [Q5.2] to also pretend the decryption process.

[Q5.2] The user randomly picks r_2 from $F_p \setminus \{0\}$, and sends $\hat{d}^{r_2}$ to proxy ϕ_0. Then, each proxy ϕ_k $(k = 0, \cdots, m-2)$ updates it and forwards to ϕ_{k+1}:

$$\hat{d}^{\frac{r_2}{\prod_{t=0}^{k} y_t(l)}}.$$

After the message has traversed the entire proxy train, it becomes

$$\hat{d}^{\frac{r_2}{\prod_{t=0}^{m-1} y_t(l)}} = \hat{d}^{\frac{r_2}{y(l)}},$$

and is then returned to the user.

If $\hat{d} = d_i^{y(l)}$, the returned message is $d_i^{r_2}$ and the user can obtain d_i and access it. Otherwise, the returned message is simply discarded.

3.3.3 *Phase 3: Data Uploading*

In Phase 2, one bucket is downloaded from each non-empty layer of the storage server. After data access, only one data block from each bucket, which must include the query target, need to be uploaded to the shuffling buffer, while other downloaded data blocks are discarded. The storage server updates the corresponding buckets and the bit map to reflect the changes. Note that, the content of the query target data block may have been changed after access. For simplicity, the following description will still use d_i to denote each data block. The *data uploading* phase uploads each of the selected data blocks, denoted as $(g_i^{x(l)}, (g_i^{-z(l)} d_i)^{y(l)})$, to a temporary buffer at the storage server as follows.

Each proxy ϕ_k $(k = 0, \cdots, m-1)$ picks x_k^{temp}, y_k^{temp} and Δz_k^{temp} randomly from $F_p \setminus \{0\}$. We introduce x^{temp}, y^{temp}, Δz^{temp} and z^{temp} as follows:

$$\begin{aligned} x^{\text{temp}} &= \prod_{k=0}^{m-1} x_k^{\text{temp}}; \quad y^{\text{temp}} = \prod_{k=0}^{m-1} y_k^{\text{temp}}; \\ \Delta z^{\text{temp}} &= \prod_{k=0}^{m-1} \Delta z_k^{\text{temp}}; \quad z^{\text{temp}} = z + \Delta z^{\text{temp}}. \end{aligned} \tag{9}$$

The user sends $\left\langle g_i^{r_3 x(l)},\ \hat{d}_i^{r_4} = (g_i^{-z(l)} d_i)^{r_4 y(l)}, \Delta d_i^{r_5} \right\rangle$ to proxy ϕ_0, which updates it to

$$\begin{aligned} &\langle\ g_i^{r_3 x(l) \cdot \frac{x_0^{\text{temp}}}{x_0(l)}},\ g_i^{r_3 x(l) \cdot \frac{\Delta z_0(l) y_0^{\text{temp}}}{x_0(l)}}, \\ &g_i^{r_3 x(l) \cdot \frac{-\Delta z_0^{\text{temp}} y_0^{\text{temp}}}{x_0(l)}},\ \hat{d}_i^{r_4 \frac{y_0^{\text{temp}}}{y_0(l)}}, (\Delta d_i^{r_5})^{y_0^{\text{temp}}}\ \rangle, \end{aligned} \tag{10}$$

and sends it to ϕ_1. Here, r_3, r_4 and r_5 are three random numbers picked by the user from $F_p \setminus \{0\}$ and $\Delta d_i = d_i'/d_i$ if d_i is the target data block (where d_i' denotes the content of the target data after the access), otherwise, Δd_i is randomly selected from G_p.

Upon receiving the message, each proxy ϕ_k $(k = 1, \cdots, m-2)$ updates it and forwards the following to ϕ_{k+1}:

$$\begin{aligned} &\langle\ g_i^{r_3 x(l) \cdot \frac{\prod_{t=0}^{k} x_t^{\text{temp}}}{\prod_{t=0}^{k} x_t(l)}},\ g_i^{r_3 x(l) \cdot \frac{\prod_{t=0}^{k} \Delta z_t(l) y_t^{\text{temp}}}{\prod_{t=0}^{k} x_t(l)}}, \\ &g_i^{r_3 x(l) \cdot \frac{-\prod_{t=0}^{k} \Delta z_t^{\text{temp}} y_t^{\text{temp}}}{\prod_{t=0}^{k} x_t(l)}},\ \hat{d}_i^{r_4 \frac{\prod_{t=0}^{k} y_t^{\text{temp}}}{\prod_{t=0}^{k} y_t(l)}}, (\Delta d_i^{r_5})^{\prod_{t=0}^{k} y_t^{\text{temp}}}\ \rangle. \end{aligned} \tag{11}$$

After proxy ϕ_{m-1} updates, the message becomes:

$$\begin{aligned} &\langle\ g_i^{r_3 x(l) \cdot \frac{\prod_{t=0}^{m-1} x_t^{\text{temp}}}{\prod_{t=0}^{m-1} x_t(l)}},\ g_i^{r_3 x(l) \cdot \frac{\prod_{t=0}^{m-1} \Delta z_t(l) y_t^{\text{temp}}}{\prod_{t=0}^{m-1} x_t(l)}}, \\ &g_i^{r_3 x(l) \cdot \frac{-\prod_{t=0}^{m-1} \Delta z_t^{\text{temp}} y_t^{\text{temp}}}{\prod_{t=0}^{m-1} x_t(l)}},\ \hat{d}_i^{r_4 \frac{\prod_{t=0}^{m-1} y_t^{\text{temp}}}{\prod_{t=0}^{m-1} y_t(l)}}, (\Delta d_i^{r_5})^{\prod_{t=0}^{m-1} y_t^{\text{temp}}}\ \rangle, \end{aligned} \tag{12}$$

which is equal to

$$\begin{aligned} \langle\ g_i^{r_3 x^{\text{temp}}},\ g_i^{r_3 \Delta z(l) y^{\text{temp}}},\ g_i^{-r_3 \Delta z^{\text{temp}} y^{\text{temp}}}, \\ (g_i^{-z(l)} d_i)^{r_4 y^{\text{temp}}}, (\Delta d_i^{r_5})^{y^{\text{temp}}}\ \rangle. \end{aligned} \tag{13}$$

Then, the message is sent to the user and the user removes r_3, r_4 and r_5 and calculates

$$\begin{aligned} & g_i^{\Delta z(l) y^{\text{temp}}} \cdot g_i^{-\Delta z^{\text{temp}} y^{\text{temp}}} \cdot (g_i^{-z(l)} d_i)^{y^{\text{temp}}} \\ =\ & (g_i^{-z^{\text{temp}}} d_i)^{y^{\text{temp}}}. \end{aligned} \tag{14}$$

If the computed entry is the target data block, the user will further multiply Δd_i^{temp} to the data content field to get

$$(g_i^{-z^{\text{temp}}} d_i')^{y^{\text{temp}}}.$$

Without loss of generality, we still use d_i to denote the content of each data block including the target data block.

Then, the user uploads

$$\langle\ g_i^{x^{\text{temp}}},\ (g_i^{-z^{\text{temp}}} d_i)^{y^{\text{temp}}}\ \rangle \tag{15}$$

to the shuffling buffer at the storage server.

3.4 Data Shuffling

After every data query, data shuffling is performed. First, the layer which data blocks should be shuffled to needs to be determined. As a rule, data should be shuffled to layer $l' > 0$ if the total number of data blocks in the temporary buffer and at layers $0, \cdots, l'-1$ is greater than or equal to the total number of buckets at layer $l'-1$, but less than the total number of buckets at layer l'. Otherwise, shuffling should be performed at layer 0 only. For simplicity, we use l' to denote the layer that data blocks are shuffled to.

Data shuffling in MU-ORAM is conducted in the following main steps: (i) Scrambling Round I (oblivious scrambling data blocks that have been uploaded during Phase 2 and thus are already in the temporary buffer before data shuffling); (ii) Data Updating and Appending (updating data blocks at layers $0, \cdots, l'$ that also need to be shuffled and appending them to the temporary buffer); (iii) Scrambling Round II (oblivious scrambling all the data blocks); and (iv) Data Mapping (assigning all the data blocks in the temporary buffer to layer l' according to a hash function). The first three steps are performed through the collaborations between the proxies while the last step is conducted only by the storage server.

To facilitate data shuffling, each proxy maintains a cache that can store $c \cdot \sqrt{N \log N} \cdot \log p$ bits, where $c \geq 1$ is a system parameter and $\log p$ bits is the size of each data block ID or each piece of the data content. Also, each proxy ϕ_k ($k = 0, \cdots, m-1$) selects keys x_k^{new}, y_k^{new} and $\Delta z_k^{\text{new}}(l')$ for layer l', as well as x_k^{shuf} and y_k^{shuf} for the temporary buffer. All these keys are selected from $F_p \setminus \{0\}$ uniformly at random.

3.4.1 Scrambling Round I

The purpose of this round is to re-encrypt and obliviously scramble the data blocks that are in the server's temporary buffer immediately after the data uploading phase ends. Let n_I denote the number of these data blocks. As the total number of layers is L and at most two data blocks are moved from each non-empty layer to the temporary buffer during the query process, at most $2L$ data blocks need to be re-encrypted and scrambled in this round. Hence, $n_I \leq 2L$.

Firstly, each proxy ϕ_k ($k = 0, \cdots, m-1$) determines a permutation function $\pi_k^{n_I}$ that permutes a sequence of n_I elements. The proxy also prepares a local cache with size $3n_I \log p$ bits; note that $\log p$ bits is the size of each data block ID or each piece of data block content.

Secondly, the proxies collaborate in scrambling and re-encrypting the IDs of the data blocks in the temporary buffer of the server. The process is as follows.

Proxy ϕ_0 fetches the encrypted IDs of all the data blocks in the server's temporary buffer, to its own cache and scrambles these IDs using permutation function $\pi_0^{n_I}$. Then, each encrypted ID, denoted as $g_i^{x^{\text{temp}}}$, is updated (i.e., re-encrypted) to tuple

$$\langle\ g_i^{x^{\text{temp}} \cdot \frac{x_0^{\text{shuf}}}{x_0^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{\Delta z_0^{\text{temp}} y_0^{\text{shuf}}}{x_0^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{-\Delta z_0^{\text{new}}(l') y_0^{\text{shuf}}}{x_0^{\text{temp}}}}\ \rangle,$$

and sent to proxy ϕ_1.

Upon receiving the n_I tuples from proxy ϕ_{k-1}, each proxy ϕ_k ($k = 1, \cdots, m-2$) scrambles the tuples using permutation function $\pi_k^{n_I}$, and then updates each tuple to the following and forwards it to ϕ_{k+1}:

$$\begin{aligned}\langle\ g_i^{x^{\text{temp}} \cdot \frac{\prod_{t=0}^{k} x_t^{\text{shuf}}}{\prod_{t=0}^{k} x_t^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{\prod_{t=0}^{k} \Delta z_t^{\text{temp}} y_t^{\text{shuf}}}{\prod_{t=0}^{k} x_t^{\text{temp}}}}, \\ g_i^{x^{\text{temp}} \cdot \frac{-\prod_{t=0}^{k} \Delta z_t^{\text{new}}(l') y_t^{\text{shuf}}}{\prod_{t=0}^{k} x_t^{\text{temp}}}}\ \rangle.\end{aligned} \tag{16}$$

After proxy ϕ_{m-1} scrambles the tuples that it has received and updates them, each tuple becomes:

$$\begin{aligned}\langle\ g_i^{x^{\text{temp}} \cdot \frac{\prod_{t=0}^{m-1} x_t^{\text{shuf}}}{\prod_{t=0}^{m-1} x_t^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{\prod_{t=0}^{m-1} \Delta z_t^{\text{temp}} y_t^{\text{shuf}}}{\prod_{t=0}^{m-1} x_t^{\text{temp}}}}, \\ g_i^{x^{\text{temp}} \cdot \frac{-\prod_{t=0}^{m-1} \Delta z_t^{\text{new}}(l') y_t^{\text{shuf}}}{\prod_{t=0}^{m-1} x_t^{\text{temp}}}}\ \rangle.\end{aligned} \tag{17}$$

which is equal to

$$\langle\ g_i^{x^{\text{shuf}}},\ g_i^{\Delta z^{\text{temp}} y^{\text{shuf}}},\ g_i^{-\Delta z^{\text{new}}(l') y^{\text{shuf}}}\ \rangle,$$

where x^{shuf}, y^{shuf}, Δz^{temp} and $\Delta z^{\text{new}}(l')$ are defined as

$$\begin{aligned}x^{\text{shuf}} = \prod_{k=0}^{m-1} x_k^{\text{shuf}};\quad y^{\text{shuf}} = \prod_{k=0}^{m-1} y_k^{\text{shuf}}; \\ \Delta z^{\text{temp}} = \prod_{k=0}^{m-1} \Delta z_k^{\text{temp}};\quad \Delta z^{\text{new}}(l') = \prod_{k=0}^{m-1} \Delta z_k^{\text{new}}(l').\end{aligned} \tag{18}$$

Proxy ϕ_{m-1} saves the sequence of re-encrypted IDs (i.e., $g_i^{x^{\text{shuf}}}$) back to the server's temporary buffer, but stores the sequence of $\{g_i^{(\Delta z^{\text{temp}} - \Delta z^{\text{new}}(l')) y^{\text{shuf}}}\}$ to its local cache.

Thirdly, the proxies scramble and re-encrypt the contents of the n_I data blocks, piece by piece. As the operations for pieces are similar, we only present the operations on the first piece of the data blocks in the following.

Proxy ϕ_0 fetches the first pieces of all the data blocks from the server's buffer to its own cache, and scramble these pieces using permutation function $\pi_0^{n_I}$. Then, each piece, denoted as $\hat{d}_i = (g_i^{-z^{\text{temp}}} d_i)^{y^{\text{temp}}}$, is updated (i.e., re-encrypted) to $\hat{d}_i^{\frac{y_0^{\text{shuf}}}{y_0^{\text{temp}}}}$, and sent to the next proxy ϕ_1. The following proxies conduct the similar scrambling, re-encryption, and forwarding. After scrambling and re-encryption have been completed in ϕ_{m-1}, each of the n_I pieces in the sequence is in the form of

$$\hat{d}_i^{\frac{\prod_{k=0}^{m-1} y_k^{\text{shuf}}}{\prod_{k=1}^{m-1} y_k^{\text{temp}}}},$$

which is equal to

$$\hat{d}_i^{\frac{y^{\text{shuf}}}{y^{\text{temp}}}} = (g_i^{-z^{\text{temp}}} d_i)^{y^{\text{temp}} \cdot \frac{y^{\text{shuf}}}{y^{\text{temp}}}} = g_i^{-(z + \Delta z^{\text{temp}}) y^{\text{shuf}}}. \tag{19}$$

Finally, proxy ϕ_{m-1} multiplies the piece with its locally-stored $g_i^{(\Delta z^{\text{temp}} - \Delta z^{\text{new}}(l')) y^{\text{shuf}}}$ to obtain $(g_i^{-z^{new}(l')} d_i)^{y^{\text{shuf}}}$, and saves it back to the server's temporary buffer.

3.4.2 Data Updating and Appending

For each data block D_i on layer l ($l = 0, \cdots, l'$), which needs to be shuffled to layer l', it should be updated to

$$\langle\ g_i^{x^{\text{shuf}}},\ (g_i^{-z^{\text{new}}(l')} d_i)^{y^{\text{shuf}}}\ \rangle.$$

The updating is performed collaboratively by the proxies, similar to Phase 2 (Data Uploading). Different from Phase 2, no any user is involved in the process. Hence, the first proxy ϕ_0 directly updates based on $\langle g_i^{x(l)}, \hat{d}_i = (g_i^{-z(l)} d_i)^{y(l)} \rangle$. After the last proxy ϕ_{m-1} has completed its update, it appends the updated data block to the server's temporary buffer. Therefore, at the end of this step, all the data blocks that should be shuffled to layer l' are stored in the server's temporary buffer.

3.4.3 Scrambling Round II

This round is to re-encrypt and scramble all the data blocks in the server's temporary buffer. Let n_{II} denote the total number of these data blocks. As the total number of data blocks stored at the server is N, it holds that $n_{II} \leq N$. Our proposed algorithm for this round is based on the idea of piece-wise shuffling proposed by Zhang et al. [45] and the data scrambling algorithm proposed by Williams et al. [41]. Our algorithm requires the capacity of each proxy's local cache to be $c\sqrt{N \log N} \log p$ bits and incurs the communication cost of $O(N \log \log N)$ data blocks on average.

When $n_{II} \leq \sqrt{N}$, the scrambling round operates as follows.

Initially, each proxy ϕ_k for $k = 0, \cdots, m-1$ determines a secret permutation function $\pi_k^{n_{II}}$ which permutes a sequence of n_{II} elements; therefore, the storage requirement of this function is $n_{II} \log n_{II}$. The proxy also randomly picks new keys $x_k^{\text{new}}(l')$ and $y_k^{\text{new}}(l')$ for layer l'.

Then, proxy ϕ_0 downloads the encrypted IDs of the n_{II} data blocks, and performs the following steps sequentially:

- Re-encryption. Each encrypted ID denoted as $g_i^{x^{\text{shuf}}}$ is re-encrypted to $(g_i^{x^{\text{shuf}}})^{\frac{x_k^{\text{new}}(l')}{y_k^{\text{shuf}}}}$.
- Scrambling. All the n_{II} re-encrypted data IDs are scrambled using permutation function $\pi_0^{n_{II}}$.
- Forwarding. The encrypted IDs are forwarded to the next proxy, which also performs the re-encryption and scrambling using its own key and secret permutation function, and forwards them to its next proxy. The last proxy stores the encrypted IDs back to the server's temporary storage.

In a similar way, the data contents of the n_{II} blocks are also re-encrypted using key $y_k^{\text{new}}(l')$ and scrambled using permutation function $\pi_k^{n_{II}}$ sequentially by each proxy ϕ_k, piece by piece.

When $n_{II} > \sqrt{N}$, the data blocks are also re-encrypted and scrambled sequentially by all the proxies, piece by piece. As different pieces of the same data block are processed in the similar way (the only difference is, the first piece, i.e., the encrypted ID, is re-encrypted with key $x_k^{\text{new}}(l')$ while the content pieces are re-encrypted with key $y_k^{\text{new}}(l')$ by each proxy ϕ_k), we present only the processing of the first pieces (i.e., encrypted IDs) of all the n_{II} data blocks. Furthermore, the processing by different proxies are also similar, except that they use different keys and permutation functions for re-encryption and permutation. Hence, in the following we only elaborate how proxy ϕ_0 processes the encrypted IDs of the data blocks.

As formally presented in Algorithms 1 and 2 (Appendix IV), the data blocks are processed through multiple sub-rounds. In the first sub-round, each of the n_{II} data blocks in the server's temporary buffer forms a single-element group, and every $n_{II}^{1/2}$ groups are randomly merged together, re-encrypted, and uploaded back to the server's temporary buffer. In the second sub-round, these data blocks form $n_{II}^{1/2}$ groups with $n_{II}^{1/2}$ pieces in each group. Then, every $n_{II}^{1/4}$ of such groups are randomly merged together, re-encrypted, and uploaded back to the server. Such merging and re-encryption repeat until all the pieces are merged together.

3.4.4 Data Mapping

In this step, the server assigns each of the n_{II} data blocks, which is in the form of $(g_i^{x^{\text{new}}(l')}, \hat{d}_i)$ into bucket $H_{l'}(g_i^{x^{\text{new}}(l')})$ of layer l'.

4. SECURITY ANALYSIS

This section presents the security analysis of the proposed MU-ORAM. First, we show that MU-ORAM is secure against honest but curious storage server. Then, we show that MU-ORAM is secure against a collusive coalition of honest but curious server, proxies and users.

4.1 Security against Curious Server

MU-ORAM follows the framework of hash-based ORAMs [13] with the following major differences: (i) no dummy data block in the system; (ii) during each query process, two data blocks from each non-empty level are removed from its bucket and uploaded to the top layer; (iii) empty bucket will never be accessed due to the bitmap. In the following, we first find the upper bound of the failure probability (i.e., bucket overflow probability) and then prove that MU-ORAM is secure against an honest but curious server according to Definition 1.

LEMMA 1. *(Probability of bucket overflow).* $\forall 0 \leq l \leq L-1$,

$$\textbf{\textit{Pr}}[\text{A bucket overflows on layer } l] \leq O(N^{-\log\log N}). \quad (20)$$

Please refer to Appendix I for the proof.

THEOREM 1. *MU-ORAM is secure against an honest but curious server.*

PROOF. Given any two equal-length sequence $\vec{x}$ and $\vec{y}$ of data requests, their corresponding observable access sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable, because of the following reasons:

- Firstly, according to the query algorithm, sequences $A(\vec{x})$ and $A(\vec{y})$ should have the same format; that is, they contain the same number of accesses, and each pair of corresponding accesses have the same format.
- Secondly, all data blocks in MU-ORAM are randomly encrypted and each data block is re-encrypted after each access. Hence, the two sequences could not be distinguished based on the appearance of data blocks.
- Thirdly, according to the query algorithm, the j-th accesses ($j = 1, \cdots, |A(\vec{x})|$) of the $A(\vec{x})$ and $A(\vec{y})$ are from the same non-empty layer of the storage. Also, according to the MU-ORAM design, the buckets accessed from each layer are either selected uniformly at random, or determined by a hash function (which is also uniformly random); hence, they are uniformly random in both sequences.

Furthermore, according to Lemma 1, a bucket overflows (i.e., MU-ORAM fails) with probability $O(N^{-\log\log N})$. Therefore, according to Definition 1, MU-ORAM is secure against an honest but curious storage server. $\square$

4.2 Security against Collusive Coalition

To quantify the security strength of MU-ORAM against a collusive coalition, we first introduce the *Modified Matching Diffie-Hellman (MMDH) problem* as follows:

Definition 4. (*Modified Matching Diffie-Hellman (MMDH) Problem*). Let G_p be a multiplicative cyclic group of order p and generator g. The MMDH problem is defined as: given g^{a_0}, g^{a_1}, g^c, and $(g^{a_b c}, g^{a_{1-b} c})$, for some unknown a_0, a_1 and c randomly picked from F_p and an unknown binary bit b randomly picked from $\{0, 1\}$, find out the value of b

Similar to the proofs in [2, 19], it can be shown that MMDH is a computational hard problem as the Decisional Diffie-Hellman and Matching Diffie-Hellman problems.

We study the security strength of MU-ORAM in the following two cases, as described in Section 2.3.2.

Case 1: Users with same access privileges to data. For this case, we study the security strength of MU-ORAM in protecting an innocent user's access pattern to the data that can also be accessed by the collusive users. Specific, we have proved the following theorem based on the game $\mathcal{G}_1$ and the (ϵ, t, n_{cq})-security notion defined in Section 2.3.2:

THEOREM 2. *If the MMDH problem is (ϵ, t)-hard (i.e., there is no algorithm can solve the MMDH problem with an advantage of at least ϵ within time period t), MU-ORAM is $(1.5n_{cq}/N + (1 - 3n_{cq})2\epsilon/N, t, n_{cq})$-secure against a collusive coalition of semi-honest storage server, users and some (but not all) proxies, in the scenario that the collusive users can access all the data accessed by any innocent user.*

PROOF. (sketch) The proof includes two parts: In the first part, we develop an algorithm $\mathcal{B}$ to play as the challenger in game $\mathcal{G}_1$. Note that, there can be two consequences of the game:

- *Consequence I*: $\mathcal{B}$ aborts the game and claims failure because adversary $\mathcal{A}$ succeeds in finding an ID chosen in a Selection Phase through discovering that the data content of this ID has been changed after the Selection Phase.
- *Consequence II*: $\mathcal{B}$ does not "abort the game and declare failure". In this case, $\mathcal{B}$ will attempt to solve the MMDH problem if $\mathcal{A}$ succeeds in the end of the game.

In the second part, we analyze the probabilities of the above two consequences respectively, and the probability for $\mathcal{B}$ to succeed in solving the MMDH problem when Consequence II occurs. We show that, Consequence I occurs with a probability less than $3n_{cq}/N$. When Consequence II occurs, the advantage for $\mathcal{A}$ to win the game is no greater than 2ϵ if the MMDH problem is (ϵ, t)-hard. Based on the above two results, we can finally prove this theorem.

Details of the proof can be found in Appendix II. □

This theorem reveals the following intuition: As the collusive attackers have the access privileges to all data that an innocent user can access, they can attack the access pattern privacy of an innocent user through checking if some randomly selected data blocks have been changed after innocent user accessed a data block. However, to make such attack effective, the attackers need to make a number of queries that is proportional to N; specifically, to gain an advantage A, the attackers need to make $A \cdot N/3$ queries on average. Note that, this will further require the adversary to incur $O(A \cdot N \log^2 N \log\log N \cdot B)$ bits communication cost as the per query communication cost of MU-ORAM is $O(\log^2 N \cdot \log\log N \cdot B)$ bits as shown in Section 5.

Case 2: Users with different access privileges to data. For this case, we study the security strength of MU-ORAM in protecting an innocent user's access pattern to the data that cannot be accessed by the collusive users. To quantify the strength, we have proved the following theorem based on the game $\mathcal{G}_2$ and the notion of (ϵ, t)-security defined in Section 2.3.2:

THEOREM 3. *If the MMDH problem is (ϵ, t)-hard (i.e., there is no algorithm can solve the MMDH problem with an advantage of at least ϵ within time period t), MU-ORAM is $(2\epsilon, t)$-secure in protecting an innocent user's access pattern to the data that cannot be accessed by a collusive coalition of semi-honest storage server, users and some (but not all) proxies.*

As the proof of Theorem 3 is a subset of the proof of Theorem 2, we provide the sketch of the proof in Appendix III, but skip the details.

Theorem 3 reveals the intuition that, MU-ORAM is more effective in protecting an innocent user's access pattern to the data that cannot be accessed by the collusive attackers; specifically, if the advantage is negligible to solve the MMDH problem in a certain time period t, the chance for the collusive attackers to reveal the above data access pattern within time period t is also negligible.

Comparing the security strength of MU-ORAM in the above two cases, we can see that, MU-ORAM is more effective to protect data access pattern in Case 2 than in Case 1.

5. COST ANALYSIS

In this section, we analyze the storage and communication costs of the MU-ORAM with the following assumptions:

- We assume the data block size $B \geq \sqrt[4]{N}$ bits. Note that this is reasonable in practice. For example, if $N \leq 2^{32}$, B is just required to be at least 32 bytes.
- We assume the bitmap recursion depth is 4. Hence, for a layer with n buckets, the total size of the bitmap in bits is:
$$bitmap(n) = n + \sqrt[4]{n^3} + \sqrt[4]{n^2} + \sqrt[4]{n} \leq 2n, \qquad (21)$$
where the inequality holds as long as $n \geq 2$.
- For simplicity, the size of a piece is set to $b = 2048$ bits.

5.1 Storage Costs

We analyze the storage costs for the storage server, each proxy, and each user, respectively.

Storage cost at the server. The storage cost at the server is no more than $N \cdot (2 + B)$ bits, which is $O(N \cdot B)$ because: (i) there is no dummy data stored on the server; (ii) the size of the bitmap in bits is

$$\sum_{l=0}^{L-1} bitmap(n_l) \leq \sum_{l=0}^{L-1} 2 \cdot n_l = \sum_{l=0}^{L-1} 2^{l+1} \log N \qquad (22)$$
$$= 2\log N \cdot (2^L - 1) \leq 2\log N \cdot 2^L \leq 2N.$$

Storage cost at each proxy. Due to the need to perform data shuffling, the storage cost at each proxy is $O(\sqrt{N}\log N \cdot b)$ bits, where b is the size of each data piece. Note that, in practical settings [36] where $N \leq 2^{32}$, the cost is no larger than 2 GB.

Storage cost at each user. For each user, the storage cost is only $O(B + \sqrt[4]{N})$ bits, which is $O(B)$ due to the assumption that $B \geq \sqrt[4]{N}$ bits.

5.2 Communication Costs

The communication costs of each user, each proxy and the server are studied in this subsection.

Communication cost of each user. Each user is involved only in the query process. For step [Q3], the user needs to retrieve the bitmap from each layer. To retrieve the bitmap from layer l, $4\sqrt[4]{n_l}$ bits will be transferred between the server and the user. Thus, the total number of transferred bits for the bitmap is

$$\sum_{l=0}^{L-1} 4\sqrt[4]{n_l} = \sum_{l=0}^{L-1} 4\sqrt[4]{2^{l+1}\log N} = 4\sqrt[4]{2\log N}\sum_{l=0}^{L-1} 2^{l/4}$$
$$= 4\sqrt[4]{2\log N}\frac{2^{L/4}-1}{2^{1/4}-1} \leq 22\sqrt[4]{2\log N}\cdot 2^{L/4} = 22\sqrt[4]{N}. \tag{23}$$

Because there are at most L non-empty layers, the cost of bitmap is at most $22L\sqrt[4]{N}$ bits. For step [Q4], the user needs to retrieve two buckets from each non-empty layer. In step [Q5], since each bucket may contain up to $\log N$ data blocks, the maximum number of data blocks retrieved is $2L\log N$ (i.e., $2L\log N \cdot B$ bits). During data uploading phase, at most $2L$ data blocks are uploaded to the proxy chain and then uploaded to the server, which incurs $2L \cdot B$ bits communication cost. Therefore, the total number of bits transferred to the user during each data query is no more than $O(\log^2 N \cdot B + \log N \cdot \sqrt[4]{N})$ bits. According to the assumption of $B \geq \sqrt[4]{N}$ bits, the cost is $O(\log^2 N \cdot B)$ bits.

Communication cost of each proxy. During data query process, each data block needs to go through each proxy. Thus, each proxy's communication cost for query is the same as a user's communication cost, which is $O(\log^2 N \cdot B)$.

Next, we analyze the communication cost for data shuffling. First of all, as the proxies local storage is large enough to scramble all data blocks from the first layer, the communication cost for data scrambling I on the first layer is $2\log N \cdot B$.

Data shuffling for layer l is triggered when the total number of data blocks on layers 0 to $l-1$ exceeds the total number of buckets on layer $l-1$. Also, each query process moves up to $\log N$ data blocks to the top layer. Hence, the frequency for data shuffling occurring for layer l is at most once per $n_l/\log N$ queries.

The communication cost incurred to each proxy during a query process is

$$S(n_l) = n_l(\lceil\log\log n_l\rceil + 1)\cdot B. \tag{24}$$

Therefore, the amortized communication cost for data shuffling is bounded by:

$$\begin{aligned}
&\sum_{l=0}^{L-1}\frac{S(n_l)}{n_l/\log N} = \sum_{l=0}^{L-1}\log N\cdot(\lceil\log\log n_l\rceil+1)\cdot B\\
&\leq \sum_{l=0}^{L-1}\log N\cdot(\log\log N+1)\cdot B\\
&< \sum_{l=0}^{\log N}\log N\cdot(\log\log N+1)\cdot B\\
&= O(\log^2 N\log\log N\cdot B).
\end{aligned} \tag{25}$$

Overall, data shuffling communication cost is $O(\log^2 N\log\log N\cdot B)$ bits.

Communication cost of the storage server. The communication cost of the storage server is no more than the sum of the costs of each user and each proxy. Hence, the communication cost of the server is $O(\log^2 N\log\log N\cdot B)$ bits.

5.3 Cost Comparison

Table 1 compares MU-ORAM with a couple of representative ORAM constructions, namely, B-ORAM [21] and P-ORAM [37]. B-ORAM is the most communication-efficient hash-based ORAM construction; P-ORAM is the most communication-efficient index-based ORAM that does not require the server to conducte intensive computation. Note that, other ORAM constructions have been briefly introduced and compared to MU-ORAM in Section 6.

Table 1: Cost Comparison. N is the total number of data blocks outsourced to the storage server, B is the size of a data block ($B \geq \sqrt[4]{N}$), and b is the size of a data piece.

Overhead		B-ORAM	P-ORAM	MU-ORAM
Query Comm.	User	$O(B\frac{\log^2 N}{\log\log N})$	$O(B\log N)\omega(1)$	$O(B\log^2 N)$
	Proxy	N/A	N/A	$O(B\log^2 N)$
Shuffle Comm.	User	$O(B\frac{\log^2 N}{\log\log N})$	$O(B\log N)\omega(1)$	N/A
	Proxy	N/A	N/A	$O(B\log^2 N\log\log N)$
Client Storage		$O(B)$	$O(B\log N)\omega(1)$	$O(B)$
Proxy Storage		N/A	N/A	$O(b\sqrt{N}\log N)$
Server Storage		$\geq 4N\cdot B$	$20N\cdot B$	$(B+2)N$

As shown in Table 1, MU-ORAM incurs higher communication overhead compared to both B-ORAM and P-ORAM, which is the cost to support multi-user ORAM model and deal with the stealthy privacy attacks.

6. RELATED WORK

Since Goldreich and Ostrovsky [13] proposed the first ORAM construction, this model has been extensively explored in the past decade. Though it was proposed for single user and single server setting, it has been recently extended to more general settings.

6.1 Single-user and Single-server ORAM

Based on the data lookup techniques used, the single-user and single-server ORAMs can be roughly classified into two categories: hash-based ORAMs and index-based ORAMs.

Index-based ORAMs, such as Tree-ORAM [33] and Partition ORAM [36], use index to locate outsourced data. Tree-ORAM stores outsourced data as a binary tree, while Partition ORAM splits the server storage into multiple smaller ORAM partitions each organized as hierarchies of layers. Two major operations, data query and eviction, are performed obliviously to hide the user's access pattern. Derived from these constructions, numerous variants [3–6, 9, 10, 12, 24, 26–28, 30–32, 35, 38, 39, 44] have been proposed to further improve the performance, especially the communication performance. Among them, P-ORAM [37], which incurs the communication cost of $O(\log N)\omega(1)$ data blocks per query in the general mode (i.e., index is exported to the server) is the most efficient one that does not require the storage server to conduct intensive computation; the most recently-proposed Constant ORAM [7, 27] incurs only constant communication cost and thus is the most efficient, but it requires the server and the user to conduct homomorphic encryption operations on data. As our proposed MU-ORAM scheme shares the same assumption about storage server, we only compare MU-ORAM to P-ORAM in Section 5.

Hash-based ORAMs [13–17, 29, 40, 40, 43] use traditional hash, Bloom Filter, Cuckoo hash, etc., to locate outsourced data in the remote storage. Among them, B-ORAM [21], which is the most efficient hash-based ORAM scheme, incurs the communication cost of $O(\log^2 N/\log\log N)$ data blocks per query, and hence is compared to our proposed MU-ORAM in Section 5.

6.2 Single-user and Multiple-server ORAM

As it is popular for a user to utilize multiple cloud servers simultaneously, some ORAM constructions [22, 34, 36] were proposed based on the idea of having two or more cloud servers to store the data and collaborate in performing data shuffling. These proposals also need to assume that the servers do not collude with each other. Different from them, our proposed MU-ORAM assumes only one cloud storage server.

6.3 Multiple-user and Single-server ORAM

Most of the existing multi-user, single-server ORAMs assume all the users trust each other, and thus they do not collude with the storage server and they only need to protect their data access patterns from the storage server (without the need to protect one user's access pattern from other users). Based on such assumption, all stateless ORAMs (i.e., ORAMs that do not require local storage of data) [11,13–17,23,25,29,33,40,42,45] can be extended to support such scenarios. Particularly, PrivateFS [42] has been proposed to further support parallel accesses from multiple users. As discussed in Sections 1 and 2, MU-ORAM has different security assumptions from these works.

The recently proposed G-ORAM [23] is more relevant to MU-ORAM, which considers the following scenario: A data owner outsources a dataset to a semi-honest cloud storage server, via which the data is shared with a group of untrusted users who may be malicious. The storage server is assumed not to collude with any user. The design goal is to employ the storage server to enforce that a user can only access the data it is authorized to, and meanwhile preserve the obliviousness of the users' access from the server. Among the issues G-ORAM tries to solve, the most relevant issue to MU-ORAM is data access obliviousness. The obliviousness property for G-ORAM is defined as follows: Assuming the server is not allowed to collude with any users in the system, the access pattern of any user is protected against the server. Due to the assumption of non-collusion, it significantly differs from the problem studied in this paper. Also, MU-ORAM considers the scenario where all users are peers obliviously sharing the outsourced data, while G-ORAM assumes a hierarchical model where a data owner (as a central point of control) utilizes the storage server to oversee the data sharing among multiple data users. This architectural difference also makes the designs to be significantly different.

7. CONCLUSION

This paper proposes MU-ORAM, a new ORAM construction to deal with stealthy privacy attack in the application scenarios where multiple users share a data set outsourced to a remote storage server and meanwhile want to protect each individual's data access pattern from being revealed to one another. We propose new security definitions for MU-ORAM, design data storage, query and shuffling algorithms, and conduct extensive security and cost analysis to evaluate the security properties as well as the communication and storage costs of the design.

8. REFERENCES

[1] Amazon. Amazon S3. `https://aws.amazon.com/s3/`.

[2] F. Bao, R. H. Deng, and H. Zhu. Variations of Diffie-Hellman problem. In *LNCS*. SpringerDaMe11, 2003.

[3] B. Chen, H. Lin, and S. Tessaro. Oblivious Parallel RAM: Improved efficiency and generic constructions. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2015.

[4] I. Damgard, S. Meldgaard, and J. B. Nielsen. Perfectly secure Oblivious RAM without random oracles. In *Proc. TCC*, 2011.

[5] J. Dautrich and C. Ravishankar. Combining ORAM with PIR to minimize bandwidth costs. In *Proc. CODASPY*, 2015.

[6] J. Dautrich, E. Stefanov, and E. Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *Proc. USENIX Security*, 2014.

[7] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2015.

[8] DropBox. DropBox Tech Blog. `https://tech.dropbox.com/2012/10/caching-in-theory-and-practice/`.

[9] C. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2015.

[10] C. W. Fletcher, L. Ren, A. Kwon, M. V. Dijk, E. Stefanov, and S. Devadas. Tiny ORAM: A low-latency, low-area hardware ORAM controller. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2014.

[11] M. Franz, P. Williams, B. Carbunar, S. Katzenbeisser, P. Andreas, R. Sion, and M. Sotakova. Oblivious outsourced storage with delegation. In *Proc. FC*, 2012.

[12] C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Proc. PETS*, 2013.

[13] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAM. *Journal of the ACM*, 43(3), May 1996.

[14] M. T. Goodrich and M. Mitzenmacher. Mapreduce parallel cuckoo hashing and oblivious RAM simulations. In *Proc. CoRR*, 2010.

[15] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Proc. ICALP*, 2011.

[16] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proc. CCSW*, 2011.

[17] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proc. SODA*, 2012.

[18] Google. Google Drive. `https://www.google.com/drive/`.

[19] H. Handschuh, Y. Tsiounis, and M. Yung. Decision oracles are equivalent to matching oracles. In *Proc. PKC*, 1999.

[20] M. S. Islam, M. Kuzu, and M. K. Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Proc. NDSS*, 2012.

[21] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proc. SODA*, 2012.

[22] S. Lu and R. Ostrovsky. Multi-server Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2011.

[23] M. Maffei, G. Malavolta, M. Reinert, and D. Schroder. GORAM – Group ORAM for privacy and access control in outsourced personal records. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2015.

[24] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *Proc. NDSS*, 2014.

[25] T. Mayberry, E.-O. Blass, and G. Noubir. Multi-Client Oblivious RAM secure against Malicious Servers. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2015.

[26] T. Moataz, E.-O. Blass, and G. Noubir. Recursive trees for practical ORAM. In *Proc. FC*, 2015.

[27] T. Moataz, T. Mayberry, and E.-O. Blass. Constant communication ORAM with small blocksize. In *Proc. CCS*, 2015.

[28] T. Moataz, T. Mayberry, E.-O. Blass, and A. H. Chan. Resizable tree-based Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2014.

[29] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *Proc. CRYPTO*, 2010.

[30] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Ring ORAM: Closing the gap between small and large client storage Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2014.

[31] L. Ren, C. W. Fletcher, X. Yu, A. Kwon, M. van Dijk, , and S. Devadas. Unified Oblivious-RAM: Improving recursive ORAM with locality and pseudorandomness. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2014.

[32] L. Ren, C. W. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path Oblivious-RAM. In *Proc. HPEC*, 2013.

[33] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT*, 2011.

[34] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *Proc. CCS*, 2013.

[35] E. Stefanov and E. Shi. ObliviStore: high performance oblivious cloud storage. In *Proc. S&P*, 2013.

[36] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Proc. NDSS*, 2011.

[37] 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.

[38] X. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Proc. CCS*, 2015.

[39] X. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. SCORAM: Oblivious RAM for secure computations. In *Proc. CCS*, 2014.

[40] P. Williams and R. Sion. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. CCS*, 2008.

[41] P. Williams and R. Sion. Access privacy and correctness on untrusted storage. In *Proc. TISSEC*, 2013.

[42] P. Williams, R. Sion, and A. Tomescu. PrivateFS: a parallel oblivious file system. In *Proc. CCS*, 2012.

[43] P. Williams, R. Sion, and A. Tomescu. Single round access privacy on outsourced storage. In *Proc. CCS*, 2012.

[44] X. Yu, L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas. Enhancing Oblivious RAM performance using dynamic prefetching. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2014.

[45] J. Zhang, W. Zhang, and D. Qiao. S-ORAM: A segmentation-based Oblivious RAM. In *Proc. AsiaCCS*, 2014.

Appendix I: Proof of Lemma 1

Detailed proof of Lemma 1 is as follows.

In MU-ORAM, there are at most $n_l = 2^{l+1}\log N$ data blocks to be distributed into n_l buckets. Then, according to a standard balls and bins model, we could have the following analysis:

Let us consider a particular bucket buc_j, and for each data block, define $X_1, \cdots, X_{n_l}$ as random variables such that

$$X_i = \begin{cases} 1 & \text{the } i^{\text{th}} \text{data block mapped to } buc_j, \\ 0 & \text{otherwise.} \end{cases} \tag{26}$$

Note that, $X_1, \cdots, X_{n_l}$ are independent of each other, and hence for each X_i, $\mathbf{Pr}[X_i = 1] = \frac{1}{n_l}$. Then, the probability that buc_j has more than t data blocks is:

$$\begin{aligned} &\mathbf{Pr}[\text{\# data blocks in } buc_j \geq t] \\ &\leq \binom{n_l}{t}\left(\frac{1}{n_l}\right)^t \leq \left(\frac{e \cdot n_l}{t}\right)^t \left(\frac{1}{n_l}\right)^t \\ &= \left(\frac{e}{t}\right)^t \end{aligned} \tag{27}$$

Note that the second inequality of Equation 27 is due to $\binom{n}{k} \leq \left(\frac{e \cdot n}{k}\right)^k$ for all $k < n$. Further considering the fact that $n_l \leq N$ and $t = \log N$, we apply the union bound of all buckets on layer l:

$$\begin{aligned} &\mathbf{Pr}[\ \exists \text{ a bucket with more than } t \text{ data blocks}] \\ &\leq n_l \cdot \left(\frac{e}{t}\right)^t \leq N \cdot \left(\frac{e}{\log N}\right)^{\log N} = O(N^{-\log\log N}). \end{aligned} \tag{28}$$

Therefore, we have for any layer l in MU-ORAM, the probability of any buckets to have more than $\log N$ data blocks is negligible in

N, which is $O(N^{-\log\log N})$.

Appendix II: Proof of Theorem 2

The proof includes two parts: In the first part, we develop an algorithm $\mathcal{B}$ to play as the challenger in game $\mathcal{G}_1$. Note that, there can be two consequences of the game: (i) *Consequence I*: $\mathcal{B}$ aborts the game and claims failure because adversary $\mathcal{A}$ succeeds in finding a data ID chosen in a Selection Phase. (ii) *Consequence II*: $\mathcal{B}$ does not "abort the game and declare failure". In this case, $\mathcal{B}$ will attempt to solve the MMDH problem if $\mathcal{A}$ succeeds in the end of the game. In the second part, we analyze the probabilities of the above two consequences respectively, and the probability for $\mathcal{B}$ to succeed in solving the MMDH problem when Consequence II occurs.

Part (1): Algorithm $\mathcal{B}$.

$\mathcal{B}$ acts as the challenger in the game $\mathcal{G}_1(\mathcal{M}, p, N, m, n_{cq})$. $\mathcal{B}$ is given g, g^{a_0}, g^{a_1}, g^c, and $(g^{a_b c}, g^{a_{1-b} c})$, where $b \in \{0,1\}$ and a_0, a_1 and c are randomly picked from F_p.

Initialization Phase - $\mathcal{B}$ simulates to construct and initialize m proxies and the data storage of N encrypted data blocks according to Sections 3.1 and 3.2: A hierarchical storage structure as described in Section 3.1 is constructed and initialized. Three integers are randomly selected from $F_p \setminus \{0\}$, and denoted as z, α and β respectively. For each layer $l \in \{0, \cdots, L-1\}$ in the hierarchical structure, a random oracle (hash function) H_l is introduced to map encrypted data block IDs to buckets. Each proxy ϕ_k ($k = 0, \cdots, m-1$) is preloaded with keys $x_k(l)$, $y_k(l)$ and $\Delta z(l)$ for $l \in \{0, \cdots, L-1\}$, which are all randomly selected from $F_p \setminus \{0\}$. N data blocks are initialized as N distinct (ID, content) pairs as follows: Let u and v be two distinct integers selected from $\{0, \cdots, N-1\}$ uniformly at random. Data blocks D_u and D_v are set to (g^{a_0}, g^{y_u}) and (g^{a_1}, g^{y_v}) respectively, where y_u and y_v are randomly selected from $F_p \setminus \{0\}$. For each of the rest data blocks $D_i = (g_i, d_i)$ where $i \in \{0, \cdots, N-1\} \setminus \{u, v\}$, $g_i = g^{x_i}$ and $d_i = g^{y_i}$ where x_i and y_i are randomly selected from $F_p \setminus \{0\}$. The IDs, i.e., g_i for $i = 0, \cdots, N-1$, are provided to $\mathcal{A}$. Each D_i is then encrypted into $(g_i^{x(L-1)}, (g_i^{-z(L-1)} d_i)^{y(L-1)})$, where $z(L-1) = z + \Delta z(L-1)$, and uploaded to bucket $H_{L-1}(g_i^{x(L-1)})$ of layer $L-1$.

Query Phase I - This phase consists of multiple requests that can be made by $\mathcal{A}$. We describe $\mathcal{B}$'s response to each type of $\mathcal{A}$'s requests as follows:

- *Data Query* - Depending on the type of query requests made by $\mathcal{A}$, the responses are different.
  - For Type I query (i.e., controlled query), if the number of such type of query exceeds n_{cq}, the game aborts. Otherwise, $\mathcal{A}$ simulates the behavior of the user, and $\mathcal{B}$ simulates the behavior of the proxies and the server. They both follow MU-ORAM's query and shuffling algorithms.
  - For Type II query (i.e., random query), $\mathcal{B}$ simulates the querying user, the proxies and the server, following MU-ORAM's algorithms. Note that, $\mathcal{A}$ does not know which ID is selected by $\mathcal{B}$ to query, but it can observe the process through requesting transcripts from the server and compromised proxies.

  Without loss of generality, in this proof we assume that the content of the queried target data block is always changed before it is uploaded.
- *Proxy Compromise* - Upon $\mathcal{A}$ queries to compromise a proxy, the secret keys $x_k(l)$, $y_k(l)$ and $\Delta z_k(l)$, where $l = 0, \cdots, L-1$, are returned to $\mathcal{A}$.
- *Proxy and Server Transcript Checking* - Upon $\mathcal{A}$ queries to check the transcript of a proxy's or the server's certain operations, the input and the output of the operations are returned to $\mathcal{A}$.
- *Storage Inspection* - $\mathcal{A}$ may request to inspect the bitmap of a particular layer or the content of a particular bucket of some layer. As a result, the bitmap and/or the content of a bucket are returned.

Selection Phase I - In this phase, $\mathcal{B}$ launches the process of querying data block D_u, i.e., the data block of ID g^{a_0}.

A new game instance (which we call *Game Instance 1*) is forked from the current game (which we call *Game Instance 0*). In both game instances, the query for data block D_u is executed following the querying and shuffling algorithms described in Sections 3.3 and 3.4. However, the data content of D_u is changed differently in these instances:

- In Game Instance 0, after D_u is queried, the content of D_u, i.e., d_u, is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.
- In Game Instance 1, after D_u is queried, d_u is changed to $g_u^{z+\alpha\cdot c} g^\beta$. Also, from this point, the Δz used in the uploading phase and the shuffling phase should always follow the format of $\Delta z = \alpha \cdot c + \gamma$ where $\gamma \in F_p \setminus \{0\}$ and can vary. This way, the data content of D_u will be encrypted into
  $$g_u^{-(z+\Delta z)} g_u^{z+\alpha\cdot c} g^\beta = g_u^{-\gamma} g^\beta,$$
  which can be computed without knowing the answer to the MMDH problem.

Query Phase II - In this phase, the requests are handled in the same way as in the Query Phase I in both instances of the game, except for the following scenarios. (i) When data block D_u, which was queried in the Selection Phase I, is queried by $\mathcal{A}$ as Type I query request: The rule specified in the definition of $\mathcal{G}_1$ is applied, and $\mathcal{B}$ will abort the game and declare failure if the specified conditions are satisfied. (ii) When data block D_u is queried again in the response to Type II query request: The current Game Instance 1 is aborted, and the current Game Instance 0 forks a new game instance which we call Game Instance 1. In both instances, the query for D_u is processed following the querying and shuffling algorithms described in Sections 3.3 and 3.4, but the content of D_u is changed differently:

- In Game Instance 0, after D_u is queried, d_u is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.
- In Game Instance 1, after D_u is queried, d_u is changed from its current value to $g_u^{z+\alpha\cdot c} g^\beta$. And, from this point, the Δz used in the uploading phase and the shuffling phase should always follow the format of $\Delta z = \alpha\cdot c + \gamma$ where $\gamma \in F_p \setminus \{0\}$ and can vary. Note that, this is the same as in Selection Phase I.

Selection Phase II - In this phase, $\mathcal{B}$ launches the process of querying data block D_v, i.e., the data block of ID g^{a_1}, in both instances of the game. Then, each of the current game instance forks a new game instance. Game Instance 0 forks a new Game Instance 2, and Game Instance 1 forks a new Game Instance 3. All these four game instances follow the same data querying and shuffling algorithms as above, but the content of D_v is changed differently:

- In Game Instances 0 and 1, after D_v is queried, d_v is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.
- In Game Instances 2 and 3, after D_v is queried, d_v is changed to $g_v^{z+\alpha\cdot c}g^{\beta}$.

Furthermore, in Game Instance 3, if D_u and D_v are in the same bucket when the query is launched, the game aborts; otherwise, the following operations should be conducted:

- In the data query phase, both D_u and D_v (i.e., the buckets that contain these two data blocks) should be selected to download. Note that this is attainable because the querying algorithm downloads two buckets from each layer that has two or more buckets, and D_u and D_v are in different buckets. Also this process is oblivious because both blocks are randomly distributed to the buckets.
- In the data uploading phase, both D_u and D_v should be selected to upload to the temporary buffer of the server. This is attainable because the uploading algorithm uploads one data block from each downloaded bucket.
- In the data shuffling phase immediately after the query, D_u and D_v are scrambled during the *Scrambling Round I*, in which $\Delta z^{new}(l')$ (note: l' is the layer that the data blocks should be shuffled to) is set to $r_0 \cdot c$ and x^{shuf} is set to $r_1 \cdot c$ where r_0 and r_1 are randomly selected from $F_p \setminus \{0\}$. Hence, D_u and D_v are encrypted to
$$\left(g^{a_0 c\cdot r_1}, (g^{-a_0 c\cdot(r_0-\alpha)}g^{\beta})^{y^{shuf}}\right)$$
and
$$\left(g^{a_1 c\cdot r_1}, (g^{-a_1 c\cdot(r_0-\alpha)}g^{\beta})^{y^{shuf}}\right),$$
respectively; further after random scrambling, they become
$$\left((g^{a_b c})^{r_1}, (g^{-a_b c})^{(r_0-\alpha)\cdot y^{shuf}}g^{\beta\cdot y^{shuf}}\right)$$
and
$$\left((g^{a_{1-b} c})^{r_1}, (g^{-a_{1-b} c})^{(r_0-\alpha)\cdot y^{shuf}}g^{\beta\cdot y^{shuf}}\right),$$
where b is either 0 or 1 with the same probability.
- In the rest lifetime of this game instance, the key $x(l)$ should always be some $r \cdot c$, where r is selected randomly from $F_p \setminus \{0\}$ and can vary.

Query Phase III - The requests in this phase are handled in the same way as in Query Phase I, except when D_u or D_v is queried. (i) When D_u or D_v is queried by $\mathcal{A}$ as Type I query request, the specified rule is checked to determine if $\mathcal{B}$ should abort the game and declare failure. (ii) When D_u or D_v is queried as $\mathcal{B}$'s response to Type II query request, it is handled as follows.

When D_u is queried, Game Instances 1 and 3 abort; meanwhile, Game Instance 0 forks a new Game Instance 1, and Game Instance 2 forks a new Game Instance 3. These four game instances follow the same data querying and shuffling algorithms as above, but the content of D_u is changed differently:

- In Game Instances 0 and 2, after D_u is queried, d_u is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.
- In Game Instances 1 and 3, after D_u is queried, d_u is changed to $g_u^{z+\alpha\cdot c}g^{\beta}$.

Furthermore, in Game Instance 3, if D_u and D_v are in the same bucket when the query is launched, the game aborts; otherwise, the operations as described in the Selection Phase II should be conducted as well.

When D_v is queried, Game Instances 2 and 3 abort; meanwhile, Game Instance 0 forks a new Game Instance 2, and Game Instance 1 forks a new Game Instance 3. These four game instances follow the same data querying and shuffling algorithms as above, but the content of D_v is changed differently:

- In Game Instances 0 and 1, after D_v is queried, d_v is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.
- In Game Instances 2 and 3, after D_v is queried, d_v is changed to $g_v^{z+\alpha\cdot c}g^{\beta}$.

Furthermore, in Game Instance 3, if D_u and D_v are in the same bucket when the query is launched, the game aborts; otherwise, the operations as described in the Selection Phase II should be conducted as well.

Challenge Phase - If Game Instance 3 does not exist, the game will abort. Otherwise, $\mathcal{B}$ launches the process of querying the data block with ID g^{a_b} in this game instance. Though $\mathcal{B}$ does not know g^{a_b} because b is unknown, the query can be implemented as follows: to execute the first step of data query phase for layer l where $x(l) = rc$ for some $r \in F_p$, $\mathcal{B}$ picks γ_1 and γ_2 from G_p uniformly at random, and then sends out $(\gamma_1, \gamma_1, \gamma_2)$ to the simulated proxies which will execute the algorithm as specified in MU-ORAM design; no matter what are returned from the proxies, $\mathcal{B}$ sets $(g_{a_b})^{x(l)} = (g^{a_b c})$. Then, the rest part of the data query and shuffling algorithms can be implemented trivially.

Query Phase IV - The requests in this phase are handled in the same way as in Query Phase I, except that: (i) If $\mathcal{A}$ requests to query ID g_u or g_v, $\mathcal{B}$ will abort the game and declare failure. (ii) In respond to $\mathcal{A}$'s type II query, $\mathcal{B}$ will randomly select one of the IDs to query. In this case, if the selected ID is g_u, the data block with ID g^{a_b} is queried instead; if the selected ID is g_v, the data block with ID $g^{a_{1-b}}$ is queried instead.

Response Phase - $\mathcal{A}$ responds with b'. Algorithm $\mathcal{B}$ uses b' as the solution to the MMDH problem.

Part (2): Analysis of $\mathcal{B}$.

First, we analyze the probability for Consequence I to occur. Note that, Consequence I refers to the case that $\mathcal{B}$ aborts the game and declares failure according to the rules in processing Type I data queries in Query Phase II and III. As explained in the definition of $\mathcal{G}_1$, such failure of $\mathcal{B}$ is due to that $\mathcal{A}$ finds the ID chosen in a

Selection Phase and thus can discover the access pattern. Specifically, there are three sub-cases for such failure to occur: Sub-case (i): $\mathcal{A}$ finds the ID chosen in Selection Phase I (i.e., g_u - ID of D_u); Sub-case (ii) $\mathcal{A}$ finds the ID chosen in Selection Phase II (i.e., g_v - ID of D_v); Sub-case (iii): $\mathcal{A}$ requests to query D_u or D_v in Query Phase IV. Sub-case (i) occurs if D_u as a Type I data query is requested both before and after Selection Phase I. We can compute the probability for this to occur as follows:

- Supposing x Type I queries are made before Selection Phase I, the probability for the query of D_u to be among the x queries is $\binom{N-1}{x-1} / \binom{N}{x} = x/N$.
- Suppose y Type I queries are made after Selection Phase I. As $\mathcal{B}$ has higher probability to find the target of Selection Phase I if it only chooses the IDs that it has queried before Selection Phase I to query again (Note: this way it can detect the data block whose content is changed), the probability for D_u to be queried among the y IDs is $\binom{x-1}{y-1} / \binom{x}{y} = y/x$.

Therefore, the probability for Sub-case (i) is $(x/N)\cdot(y/x) = y/N$; further due to $x + y \le n_{cq}$ and $y < x$, the probability is at most $n_{cq}/2N$.

Similarly, the probability for Sub-case (ii) is at most $n_{cq}/2N$. Lastly, the probability for Sub-case (iii) is at most $1 - \binom{N-2}{n_{cq}} / \binom{N}{n_{cq}} < 2n_{cq}/N$. Totally, the probability for Consequence I to occur is less than $3n_{cq}/N$.

Second, we consider Consequence II, i.e., $\mathcal{B}$ does not "abort the game and declare failure". Here, there will be two cases: (i) In the first case, the game aborts at the beginning of Challenge Phase because D_u and D_v are in the same bucket in Game Instance 3. This case occurs with a probability of at most $\frac{1}{2\log N}$, due to the facts that each layer has at least $2\log N$ different buckets and data blocks are randomly distributed to the buckets. (ii) In the second case, the game finishes normally, and the adversary returns a binary bit b'. In this case, if $\mathcal{A}$ wins the game (i.e., $b' = b$), $\mathcal{B}$ also obtains the correct answer (i.e., $b = b'$) to the MMDH problem and thus solve the problem.

Considering the above two cases together, if $\mathcal{A}$ can win the game under Consequence II with advantage ϵ' within time period t, $\mathcal{B}$ can solve the MMDH problem with an advantage of at least $\frac{2\log N-1}{2\log N}\epsilon' > 0.5\epsilon'$ with the same time complexity. Hence, with the assumption that the MMDH problem is (ϵ, t)-secure, i.e., no algorithm can solve the MMDH problem with an advantage of at least ϵ within time t, we conclude that $\mathcal{A}$ cannot win the game under Consequence II with an advantage of at least 2ϵ within time t.

To summarize the above analysis for Consequence I and II, $\mathcal{A}$ cannot win the game with a probability greater than $3n_{cq}/N + (1 - 3n_{cq}/N) * (0.5 + 2\epsilon)$ (i.e., an advantage greater than $1.5n_{cq}/N + (1 - 3n_{cq})2\epsilon/N$), if it can issue at most n_{cq} ($n_{cq} < N/3$) Type I data queries and its overall running time is t. That is, MU-ORAM is $(1.5n_{cq}/N + (1 - 3n_{cq})2\epsilon/N, t, n_{cq})$-secure.

Appendix III: Proof of Theorem 3

As the proof of Theorem 3 is actually a subset of the proof of Theorem 2, we only sketch the difference from the proof in Appendix II as follows: This proof also includes two parts: construction of algorithm $\mathcal{B}'$ to play as the challenger in game $\mathcal{G}_2$ with the adversary $\mathcal{A}$, and the analysis of $\mathcal{B}'$.

In the first part, as the definitions of $\mathcal{G}_2$ is different from that of $\mathcal{G}_1$, we describe the major difference of $\mathcal{B}'$ from $\mathcal{B}$ in the proof of Theorem 2: (i) Initially, $\mathcal{A}$ is only provided with a subset of the complete ID sets; that is, there are totally N IDs but $\mathcal{A}$ is only given $N' \le N - 2$ of the IDs. (ii) The IDs of D_u and D_v (i.e., the two data blocks queried in Selection Phase I and II) are unknown to $\mathcal{A}$. (iii) $\mathcal{B}'$ will not "abort the game and declare failure" in the game.

In the second part, as $\mathcal{B}'$ does not "abort the game and declare failure", we only need to analyze the probability for $\mathcal{B}'$ to solve the MMDH problem if $\mathcal{A}$ wins the game. Therefore, based on the assumption that the MMDH problem is (ϵ, t)-hard, we can conclude that $\mathcal{A}$'s advantage to win the game within time t is at most 2ϵ.

Appendix IV: Data Shuffling Algorithms

Algorithm 1 Re-encrypt&Scramble($\{I_0, \cdots, I_{n_{II}-1}\}$): re-encryption and scrambling of encrypted IDs $I_0, \cdots, I_{n_{II}-1}$ by proxy ϕ_0.

```
 1: $n \leftarrow n_{II}$
 2: $k_{old} = x_0^{\text{shuf}}$
 3: while $n > 1$ do
 4:     split $\{I_0, \cdots, I_{n_{II}-1}\}$ evenly to $n$ groups: $\hat{g}_0, \cdots, \hat{g}_{n-1}$
 5:     $n' \leftarrow \lfloor\sqrt{n}\rfloor$
 6:     split local cache evenly to $n'$ segments: $S_0, \cdots, S_{n'}$
 7:     if $n' > 1$ then
 8:         select $k_{new}$ randomly from $F_p \setminus \{0\}$
 9:         $k_{old} = k_{old} * k_{new}$
10:     else
11:         $k_{new} = x_0^{\text{new}}(l')/k_{old}$
12:     end if
        //Merge $n$ groups into $n'$ larger groups
13:     for $i := 0$ to $n' - 1$ do
14:         Re-encrypt&Merge($k_{new}$, $\{\hat{g}_{i*n'}, \cdots, \hat{g}_{(i+1)*n'-1}\}$)
15:     end for
16:     $n \leftarrow n'$
17: end while
```

Algorithm 2 Re-encrypt&Merge(k_{new}, $\{\tilde{g}_0, \cdots, \tilde{g}_{n-1}\}$): merge pieces in groups $\tilde{g}_0, \cdots, \tilde{g}_{n-1}$ and re-encrypt them with key k_{new}

```
 1: $s \leftarrow c\sqrt{N \log N}/n$    //calculate the capacity of each segment
    //fill in half of each segment
 2: for $i := 0$ to $n - 1$ do
 3:     for $j := 0$ to $S/2$ do
 4:         download one piece from $\tilde{g}_i$ (if any) to segment $S_i$
 5:     end for
 6: end for
    //scramble and re-encrypt the pieces
 7: while segments are not empty do
 8:     if groups are not empty then
 9:         for $i := 0$ to $n - 1$ do
10:             download one piece from $\tilde{g}_i$ (if any) to $S_i$
11:         end for
12:     end if
13:     for $i := 0$ to $n - 1$ do
14:         randomly select $r$ from $\{j|S_j$ is not empty $\}$
15:         re-encrypt the first piece in $S_r$ with $k_{new}$ & upload it
            to the server's temporary buffer
16:     end for
17: end while
```