2014

A Multi-user Oblivious RAM for Outsourced Data

Zhang Jinsheng
Iowa State University, alexzjs@alumni.iastate.edu

Zhang Wensheng
Iowa State University, wzhang@iastate.edu

Daji Qiao
Iowa State University, daji@iastate.edu

A Multi-user Oblivious RAM for Outsourced Data

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) systems have been proposed. However, existing ORAM designs assume a single user or a group of mutually-trusted users to access a remote storage, which makes them inapplicable to many practical scenarios where multiple users share data but may not trust each other. Even if the data-sharing users do trust each other, such systems are vulnerable to the compromise of even a single user. To study the feasibility and costs for overcoming the limitation of existing ORAMs in multi-user scenarios, this paper proposes a new type of ORAM system called Multi-user ORAM (M-ORAM). The key idea is to introduce a new component, i.e., a chain of anonymizers, to act as a common proxy between users and the storage server. M-ORAM can protect the data access pattern of each individual user from others as long as not all anonymizers are compromised. Extensive security and overhead analysis has been conducted to quantify the strength of the scheme in protecting an individual user's access pattern and the costs incurred to provide the protection.

Disciplines
Information Security | OS and Networks

A Multi-user Oblivious RAM for Outsourced Data

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

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) systems have been proposed. However, existing ORAM designs assume a single user or a group of mutually-trusted users to access a remote storage, which makes them inapplicable to many practical scenarios where multiple users share data but may not trust each other. Even if the data-sharing users do trust each other, such systems are vulnerable to the compromise of even a single user. To study the feasibility and costs for overcoming the limitation of existing ORAMs in multi-user scenarios, this paper proposes a new type of ORAM system called *Multi-user ORAM (M-ORAM)*. The key idea is to introduce a new component, i.e., a chain of anonymizers, to act as a common proxy between users and the storage server. M-ORAM can protect the data access pattern of each individual user from others as long as not all anonymizers are compromised. Extensive security and overhead analysis has been conducted to quantify the strength of the scheme in protecting an individual user's access pattern and the costs incurred to provide the protection.

1. INTRODUCTION

Recent development of cloud computing has witnessed the convenience of remote storage services. Many online storage providers 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 restricted devices in a pay-per-use manner.

Although a user may encrypt outsourced data to protect confidentiality of the data content, her pattern in accessing the data remains unprotected and may reveal the user's private information. The emerging oblivious RAM (ORAM) techniques [1, 3–7, 9–14] have been proposed to protect a user's access pattern privacy, however, with restrictions. One of the fundamental restrictions is that all existing ORAM schemes essentially assume only a single user to the remote storage. Even though multi-user ORAM schemes have been proposed in [6, 13], the users essentially work together as a single user, because they share secret keys with each other and thus they must trust each other in order to share the remote storage. In practice, users who share a data storage (e.g., file system or data base) may not trust each other. Even if they do, compromising just one of them can easily jeopardize the all.

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 secret keys and appear as a single user to the remote storage. Such a system, however, may violate patient-doctor privacy. As an example, the pattern in which a patient visits a doctor, which shall be kept confidential between the patient and the doctor, may now be observed by other doctors through observing this doctor's pattern in accessing the data from the remote storage. Moreover, if the account of a doctor is compromised by an outsider, the outsider can also observe the accesses made by all other doctors.

To study the feasibility and cost for overcoming the above limitation of existing ORAM systems, we propose, design, and analyze a new type of ORAM system called *Multi-user ORAM (M-ORAM)*, with which (i) multiple users can share a remote storage, and meanwhile (ii) the access pattern of each individual user can be well protected from other users or the storage server. To the best of our knowledge, this is first design to accomplish the above goals.

The key idea in our design is to introduce a chain of collaborative but mutually-independent *anonymizers* between users and the storage server. When a user needs to query a data item, its request and the storage server's replies shall pass through and be processed by the anonymizers before they reach the storage server or the user. In practice, the anonymizers 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 anonymizers can be implemented as several physical/virtual machines running in the premise of the hospital or some cloud providers independent of the provider of the remote storage server.

Within this architecture, (i) users do not need to share secrets as they do not interact independently with the shared

storage server; (ii) each user sets up a secure and logically isolated communication channel with the chain of anonymizers; (iii) multiple anonymizers, each holding an independent share of the system-wide secret, work together to act as a proxy between users and the storage server, and also take non-user-specific workload (e.g., data shuffling). Due to the above features, users are securely isolated from each other, compromising some but not all anonymizers cannot capture the system-wide secret, and therefore the system can be more privacy-preserving and secure. Our design is based on a customized ORAM system. The system follows the framework of existing hierarchical, bucket-based, and hash-based ORAMs [1,3,7], with a few modifications to facilitate the implementation of the afored-described ideas.

Extensive security analysis has been conducted to quantify the strength of the system in protecting users' data access pattern. Overhead analysis has also been provided to quantify the costs incurred to provide the protection. Compared to the single-user ORAM with the best-known performance [7], our design yields

- a lower communication overhead per query by the user: $O(\log N \log\log N)$ vs. $O\left(\frac{\log^2 N}{\log\log N}\right)$;
- a higher communication overhead (amortized per query) for data shuffling per anonymizer: $O(\log^3 N \log\log N)$ vs. $O\left(\frac{\log^2 N}{\log\log N}\right)$;
- a moderately increased local cache at the user: $O(\log N \log\log N)$ vs. $O(1)$.

The rest of the paper is organized as follows: System model and design goal are introduced in Section 2. Sections 3 and 4 elaborate the design details. The results of security and overhead analysis are reported in Sections 5 and 6, respectively. Section 7 provides a brief comparison to related works. Finally, Section 8 concludes the paper.

2. SYSTEM MODELS AND DESIGN GOAL

2.1 System Models

Multiple users, who may not trust each other, share N data items, which are exported to an un-trusted storage server. Let F_p be a finite field of p distinct elements, where p is a large prime number. Let G_p be a multiplicative, cyclic group with also p distinct elements. Hence, for any element $g \in G_p$, elements $g^0, g^1, g^2, \cdots, g^{p-1}$ should all belong to G. Each data item, denoted as D_i, consists of two components: a unique data ID and the data content that is a sequence of elements of G_p. As the operations on each element of the sequence are the same, we focus our study on the operations on a single element in this work. In practice, operations on a realistic data content are simply a sequence of operations on each element of the data content. Hereafter, each data item D_i is represented as $(g_i,\ d_i)$, where $g_i \in G_p$ is the data ID and $d_i \in G_p$ is the data content.

The server provides the raw storage services of storing and fetching data items with the following two primitives:

- $store(data, pos)$ to store $data$ at physical address pos;
- $fetch(pos)$ to retrieve data from physical address pos.

Moreover, there is a system initialization server that is trusted by all the users. It is not involved in data access, but only responsible for initializing the system and initializing a user when the user joins the system. Therefore, it does not need to stay online after the system starts, and thus is assumed to be immune from attacks.

2.2 Design Goal and Our Approach

A number of oblivious RAM (ORAM) schemes [1, 3–5, 7, 9–12, 14] have been proposed to protect the data access pattern of a user from the storage server. But these schemes only consider the situation when there is only a single user in the system, and thus cannot be applied when there exist multiple users who may not trust each other. The situation becomes even more challenging if users can collude with the storage server.

Our *design goal* is to protect the data access pattern of each user of a multi-user storage system from being revealed to the storage server or other users. To attain this goal, we propose a new architecture (as shown in Figure 1) composed of a hierarchical storage server, multiple users, and a chain of *anonymizers* as a bridge between users and the storage server. In practice, anonymizers can be implemented as mutually-independent hardware components (e.g., computers) or software components (e.g., virtual servers).

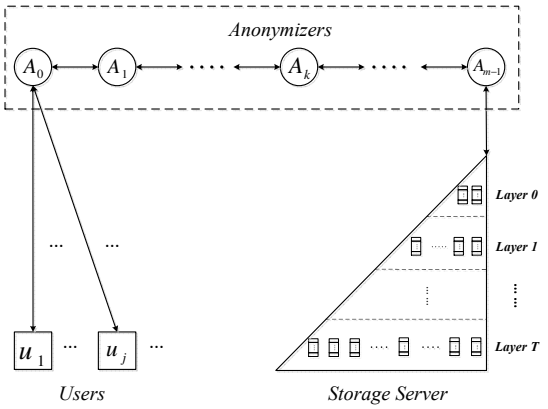

Figure 1: System overview.

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

- When a user needs to access certain data items, its request and the data replied from the storage server should pass through and be processed by the anonymizers before they reach the storage server or the user.

- With such a proxy 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 observing their interactions with the storage server.

- Such an architecture also allows each user to establish secure and logically isolated communication channel with the chain of anonymizers, which makes it possible to prevent a user from learning other users' data access patterns through observing their interactions with the anonymizers.

- As the proxy consists of multiple collaborative but independent anonymizers, the access pattern privacy of users can still be under protection as long as not all of the anonymizers have been compromised.

Under the proposed architecture, appropriate algorithms must be designed to guide the interactions between the storage server, anonymizers, and users. For readability, we describe the algorithms in two steps in the following sections. We first introduce a customized ORAM (C-ORAM), in which the anonymizers and users are treated as a *virtual* single user of the storage server, and the interactions between this virtual single user and the storage server are specified. Then, we present the complete scheme, called multi-user oblivious RAM (M-ORAM), in which the detailed interactions between the users and anonymizers are elaborated under the umbrella of C-ORAM.

3. A CUSTOMIZED ORAM

This section presents a customized ORAM (C-ORAM) that specifies the interactions between the storage server and the rest of the system (i.e., the anonymizers and users, which are treated as a virtual single user).

3.1 System Initialization

Similar to several existing ORAMs [1, 10], C-ORAM organizes the storage as a hierarchy of buckets:

- The hierarchy consists of $T+1$ layers, where $T = \lceil \log N - \log(\log N \log\log N) \rceil - 1$.

- Each layer l ($l = 0, \cdots, T$) has ϕ_l buckets where $\phi_l = 2^{l+1} \log N \log\log N$. Hence, the top layer of the hierarchy (i.e., layer 0) has $2 \log N \log\log N$ buckets, while the bottom layer of the hierarchy (i.e., layer T) has at least 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 two integers uniformly at random between 0 and $\phi_l - 1$, i.e., two of the ϕ_l buckets at layer l.

- Each bucket can contain up to $B = 4 \log\log N$ data items.

- A counter is maintained for each bucket to track the number of data items stored at the bucket.

The single user of C-ORAM is preloaded with two keys $x(l)$ and $y(l)$ for each layer l. Here, a key is an element of $F_p \setminus \{0\}$ and the encryption of an element $g \in G_p$ with a key k is $g^k \in G_p$. Initially, the user encrypts and exports all N data items to the bottom layer of storage hierarchy (i.e., layer T) at the storage server. For example, for each data item $D_i = (g_i,\ d_i)$, where $g_i \in G_p$ is the data ID and $d_i \in G_p$ is the data content, it is encrypted with $x(T)$ and $y(T)$ to obtain $\left(g_i^{x(T)},\ d_i^{y(T)}\right)$, and then exported to one of the two buckets produced by $H_T\left(g_i^{x(T)}\right)$.

Figure 2 gives an overview of the interactions between the single user and the storage server in C-ORAM. As shown in the figure, to obtain a desired data item from the storage server, C-ORAM operates in the following three phases: *data query*, *data uploading*, and *data shuffling*.

3.2 Phase 1: Data Query

Let $D_i = (g_i,\ d_i)$ denote the desired data item. To obtain it from the storage, C-ORAM executes the *data query* phase in an iterative manner at each nonempty layer l of the storage hierarchy from the top layer $l = 0$ to the bottom layer $l = T$. Each iteration consists of six steps from [Q1] to [Q6].

[Q1] The user computes the encrypted ID $g_i^{x(l)}$.

[Q2] The user computes the positions pos_0 and pos_1 of the buckets that may contain the desired data item:

$$(pos_0,\ pos_1) \ \leftarrow\ H_l\left(g_i^{x(l)}\right).$$

[Q3: Bitmap Retrieval] This step is to retrieve the bitmap that will be used by the user to decide the buckets to request. To facilitate the presentation, we introduce the following notations:

$$a \ =\ \min\left\{2^l,\ 8\log\log N(4\log\log N+1)\right\}\log N\log\log N,$$

and

$$b \ =\ \left\lceil \frac{\phi_l}{a} \right\rceil,$$

where ϕ_l is the number of buckets at layer l.

If D_i has already been found at a layer higher than l, the user generates two integers c_0 and c_1 uniformly at random from $\{0, \cdots, b-1\}$; otherwise, $c_0 = pos_0 \mod b$ and $c_1 = pos_1 \mod b$. In case the above procedure produces the same value for c_0 and c_1, the user re-generates c_1 uniformly at random from $\{0, \cdots, b-1\} \setminus \{c_0\}$. Then, the user requests the storage server to return a bitmap of $2a$ bits to indicate the emptiness of the buckets at positions $c_0 + b*i$ and $c_1 + b*i$ for $i = 0, \cdots, a-1$. Note that the $2a$ queried

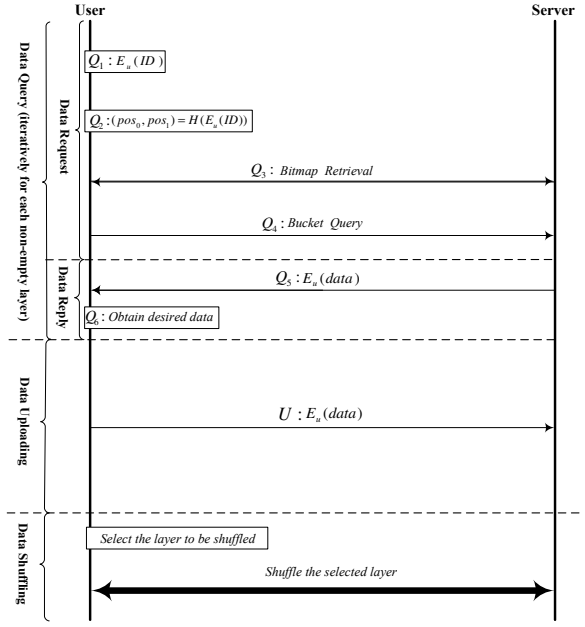

Figure 2: C-ORAM overview.

positions must include the positions of the buckets where D_i may be stored, if D_i has not been found yet. The emptiness of these buckets will be used in the next step to decide which buckets should be retrieved. Furthermore, as to be shown in Section 5.1.2, the above a value ensures that the probability for all queried positions to be empty is negligible, which guarantees negligibly low probability of failure in running the query scheme.

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

- If D_i has already been found at a layer higher than l, then for each $x \in \{0,1\}$, the user randomly picks a nonempty position from $c_x + bt$, $t \in \{0, \cdots, a-1\}$ to request.
- Otherwise, for each $x \in \{0,1\}$, if the bucket at position pos_x is nonempty, the bucket is requested; else, the user randomly picks a nonempty position from $c_x + bt$, $t \in \{0, \cdots, a-1\}$ to request.

[Q5: Data Reply] In response to the request, the storage server returns all the encrypted data items at the requested buckets directly to the user.

[Q6: Obtain Desired Data] For each returned data item, the user uses keys $x(l)$ and $y(l)$ to decrypt its ID and content. If a data item has an ID of g_i, the desired access is performed to the data content; otherwise, the data item is stored temporarily in a local cache at the user.

3.3 Phase 2: Data Uploading

After all nonempty layers have been queried and the desired data item has been accessed, all the returned data items are re-encrypted and then uploaded to the storage server. Specifically, the user picks a new pair of keys $x(l)$ and $y(l)$ uniformly at random from F_p, and re-encrypts each returned data item with these keys. Then, the re-encrypted data items are uploaded in an arbitrary order to a temporary *shuffling buffer* at the storage server.

3.4 Phase 3: Data Shuffling

Suppose S_l denote the dataset from layer l and S denote the data in the shuffling buffer. Data shuffling is conducted as follows.

Algorithm 1 Data Shuffling in C-ORAM

S1: Determine Shuffling Layer
$l := 0$
$S := S \cup S_0$
while $\phi_l < |S|$ **do**
  $S := S \cup S_{l+1}$
  $l := l + 1$
end while
$l_s := l$
S2: Data Encryption
for $D_i \in S$ **do**
  user.Download(S, D_i)
  D'_i := user.Transform($(D_i, \frac{x(l_s)}{x(l)}, \frac{y(l_s)}{y(l)})$
  $E_u(D'_i)$:= user.Encrypt(D'_i, u)
  user.Upload(S, $E_u(D'_i)$)
end for
S3: Data Sorting
user.Obliviously-Sort(S)
S4: Data Decryption
for $D'_i \in S$ **do**
  user.Download(S, $E_u(D'_i)$)
  D'_i := user.Decrypt(D'_i, u)
  user.Upload(S, D'_i)
end for
S5: Server Locates Data into Buckets
Server.Map(l_s, S)

[S1] Determine the layer for which data shuffling should be performed. As a rule, shuffling should be performed for layer $l_s > 0$ only if (i) the number of data items in the shuffling buffer and at layers $0, \cdots, l_s - 1$ is greater than or equal to the total number of buckets at layer $l_s - 1$, and (ii) the number of data items in the shuffling buffer and at layers $0, \cdots, l_s$ is less than the total number of buckets at layer l_s.
[S2] All data items at layers $0, \cdots, l_s$ should be updated such that the ID of each data item becomes encrypted by $x(l_s)$ and the content of each data item becomes encrypted by $y(l_s)$. For this purpose, the user should download each $D_i = \left(g_i^{x(l)}, d_i^{y(l)}\right)$ at layer l $(0 \le l \le l_s)$, raise the ID part of the data item to power $x(l_s)/x(l)$ and the content part to power $y(l_s)/y(l)$ to get $D'_i = \left(g_i^{x(l_s)}, d_i^{y(l_s)}\right)$. Then, the user further encrypts D'_i using his/her own private key and uploads $E_u(D'_i) = \left(E_u(g_i^{x(l_s)}), E_u(d_i^{y(l_s)})\right)$, to the shuffling buffer at the storage server. Similar to existing ORAM scheme, it is required that the private-key encryption on each data be a probabilistic encryption which gives adversary no advantage in distinguishing data items from their appear-

ance.

[S3] For the $|S|$ data items stored on the shuffling buffer, the user performs data-oblivious sorting, based on the "new" ID of D'_i, $g_i^{x(l_s)}$. In order to perform data-oblivious sorting, any sorting with obliviousness property can be used. For the sake of lower cost, we can deploy the one proposed by Goodrich in [2], which takes $O(n \log n)$ steps to finish sorting on n data items.

[S4] After the end of data shuffling, the user further scans all data in the data buffer to remove the private-key encryption such that each D'_i is finally transformed back to $\left(g_i^{x(l_s)}, d_i^{y(l_s)}\right)$.

[S5] The storage server moves the data items at the shuffling buffer to the buckets at layer l_s as follows: $\log N$ random hash functions $H_{l_s}^{\theta}\left(g_i^{x(l_s)}\right)$ ($\theta \in \{1, \cdots, \log N\}$) are selected by storage server. For each hash function, storage server executes the following procedure: each data item $D'_i = \left(g_i^{x(l_s)},\ d_i^{y(l_s)}\right)$ is moved to one of the two buckets generated by $H_{l_s}^{\theta}\left(g_i^{x(l_s)}\right)$ which has no less empty spaces than the other one. Once bucket overflow happens during the simulation, stop the execution for this hash function.

As proved in theorem 1, the probability that all $\log N$ hash functions lead to bucket overflow will be negligible in N. Therefore, the first hash function with none bucket overflowing will be selected as the one used for l_s during current data shuffling phase, denoted as $H_{l_s}\left(g_i^{x(l_s)}\right)$.

4. THE COMPLETE SCHEME

This section presents the complete scheme, called Multi-user Oblivious RAM (M-ORAM), which is built upon C-ORAM. Comparing Figure 3 with Figure 2, we can see that, M-ORAM operates in the same three phases as C-ORAM: *data query*, *data uploading*, and *data shuffling*. However, the protocol becomes more complicated due to the introduction of anonymizers, which: (i) serve as a common proxy for all users to access data at the storage server; (ii) maintain shares of the system-wide secret that is used to re/encrypt the data items before uploading them to the storage server; and (iii) handle data shuffling if needed. In the following, we start the description of M-ORAM with system initialization, followed by the detailed descriptions of each phase.

4.1 System Initialization

The system initialization includes *anonymizer initialization*, *storage initialization*, and *user initialization*.

4.1.1 Anonymizer Initialization

Suppose there are m anonymizers, denoted as A_0, $\cdots$, A_{m-1}, in the system. For each A_k, $k = 0, \cdots, m-1$, it is preloaded by the initialization server with the following information:

- Two keys $x_k(l)$ and $y_k(l)$ for each $l \in \{0, \cdots, T\}$, which are randomly picked from $F_p \setminus \{0\}$;

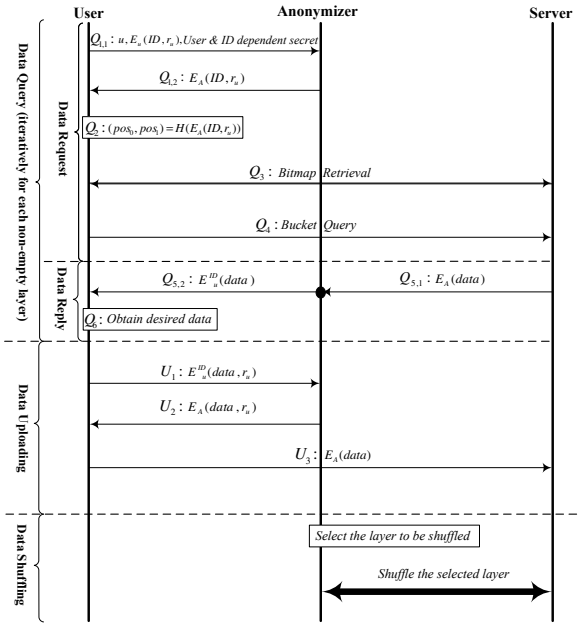

Figure 3: M-ORAM overview.

- Two numbers s_k, z_k randomly picked from $F_p \setminus \{0\}$;

4.1.2 Storage Initialization

Same as C-ORAM, M-ORAM organizes the storage as a hierarchy of buckets. Initially, each data item $D_i = (g_i, d_i)$ is encrypted before being exported to the bottom layer of the storage hierarchy (i.e., layer T). The difference between M-ORAM and C-ORAM lies in how the data item is encrypted. In M-ORAM:

- g_i is encrypted to $g_i^{x(T)}$, where

$$x(l) = \prod_{k=0}^{m-1} x_k(l) \text{ for each } l = 0, \cdots, T, \qquad (1)$$

  and $x_k(l)$ is a secret key known to A_k only.

- d_i is encrypted to $\left(g_i^{-z} d_i\right)^{y(T)}$, where

$$z = \prod_{k=0}^{m-1} z_k, \qquad (2)$$

  and

$$y(l) = \prod_{k=0}^{m-1} y_k(l) \text{ for each } l = 0, \cdots, T. \qquad (3)$$

  Here, z_k and $y_k(l)$ is are secrets known to A_k only.

In other words, M-ORAM encrypts each data item with the product of secret keys from all the anonymizers. This is in sharp contrast to C-ORAM that encrypts each data item with the secret key known to the user.

4.1.3 User Initialization

For each user U_j, when it joins the system, it is preloaded by the initialization server with the following secret:

$$w_j = \prod_{k=0}^{m-1} h(s_k, j), \tag{4}$$

where $h(.,.)$ is a public one-way hash function that maps two elements of F_p to another element of the same field, and is known to all users, anonymizers, and the storage server.

4.2 Phase 1: Data Query

Same as C-ORAM, in order for user U_j to obtain a desired data item $D_i = (g_i, d_i)$ from the storage server, M-ORAM executes the *data query* phase in an iterative manner at each nonempty layer l of the storage hierarchy from the top layer $l = 0$ to the bottom layer $l = T$. Each iteration consists of six steps from [Q1] to [Q6].

[Q1] The goal of this step is to compute the encrypted ID of the desired data item. However, different from C-ORAM, which uses the secret key of an individual user as the encryption key, M-ORAM uses the product of all anonymizers' secret keys as the encryption key. Therefore, [Q1] in M-ORAM requires a collaboration between the user and the anonymizers, as shown in Figure 4. It consists of two sub-steps as follows.

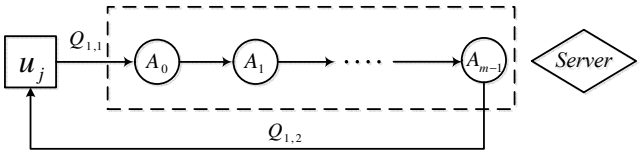

Figure 4: Generation of encrypted data ID in M-ORAM.

[Q1.1] In the first sub-step, user U_j sends the following message to anonymizer A_0:

$$\left\langle U_j, g_i^{r_{j,0} w_j}, (g_i \beta_j)^{w_j}, \beta_j^{r_{j,1}} \right\rangle, \tag{5}$$

where $r_{j,0}$ and $r_{j,1}$ are two nonces randomly picked from $F_p \setminus \{0\}$, β_j is an element randomly picked from group G_p, and w_j is a secret preloaded to U_j.

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

$$\left\langle U_j, \left(g_i^{r_{j,0} w_j}\right)^{\frac{\prod_{t=0}^{k} x_t(l)}{\prod_{t=0}^{k} h(s_t,j)}}, \left((g_i \beta_j)^{w_j}\right)^{\frac{\prod_{t=0}^{k} y_t(l) z_t}{\prod_{t=0}^{k} h(s_t,j)}}, \left(\beta_j^{r_{j,1}}\right)^{\prod_{t=0}^{k} z_t} \right\rangle. \tag{6}$$

Note that $x_k(l)$, $y_k(l)$, z_k, and s_k are secrets preloaded to A_k. After the message has traversed the entire anonymizer chain, it becomes

$$\begin{aligned} &\left\langle U_j, \left(g_i^{r_{j,0} w_j}\right)^{\frac{\prod_{t=0}^{m-1} x_t(l)}{\prod_{t=0}^{m-1} h(s_t,j)}}, \left((g_i \beta_j)^{w_j}\right)^{\frac{\prod_{t=0}^{m-1} y_t(l) z_t}{\prod_{t=0}^{m-1} h(s_t,j)}}, \left(\beta_j^{r_{j,1}}\right)^{\prod_{t=0}^{m-1} z_t} \right\rangle \\ &= \left\langle U_j, \left(g_i^{r_{j,0} w_j}\right)^{\frac{x(l)}{w_j}}, \left((g_i \beta_j)^{w_j}\right)^{\frac{y(l) z}{w_j}}, \left(\beta_j^{r_{j,1}}\right)^{z} \right\rangle \\ &= \left\langle U_j, g_i^{r_{j,0} x(l)}, (g_i \beta_j)^{y(l) z}, \beta_j{}^{r_{j,1} z} \right\rangle, \end{aligned} \tag{7}$$

according to Equations (1), (2), (3), and (4). Then, A_{m-1}

- keeps the $(g_i \beta_j)^{y(l) z}$ locally, which will be used in step [Q5: Data Reply], and
- returns the following message: $\left\langle U_j, g_i^{r_{j,0} x(l)}, \beta_j{}^{r_{j,1} z} \right\rangle$ back to U_j.

Upon receiving the message, U_j can obtain $g_i^{x(l)}$ and $\beta_j{}^{z}$ from $g_i^{r_{j,0} x(l)}$ and $\beta_j{}^{r_{j,1} z}$, respectively, as $r_{j,0}$ and $r_{j,1}$ are its self-generated nonces. Note that $g_i^{x(l)}$ is the ID of the desired data item encrypted with the product of all anonymizers' secret keys, which will be used in [Q2]. $\beta_j{}^{z}$ will be used in [Q5: Data Reply].

[Q2] Based on $g_i^{x(l)}$, the user computes the positions pos_0 and pos_1 of the buckets that may contain the desired data item:

$$(pos_0,\ pos_1) \ \leftarrow \ H_l\left(g_i^{x(l)}\right).$$

This step is the same as that in C-ORAM.

[Q3: Bitmap Retrieval] This step is the same as that in C-ORAM. The goal is to retrieve the bitmap that will be used by the user to decide the buckets to request.

[Q4: Bucket Request] This step is the same as that in C-ORAM. The goal is to allow the user to select the buckets to request based on the retrieved bitmap, and send the request directly to the storage server.

[Q5: Data Reply] In response to the bucket request from U_j, the storage server returns all the data items at the requested buckets to U_j in two sub-steps: [Q5.1: From Server to Anonymizers] and [Q5.2: From Anonymizers to User], as shown in Figure 5.

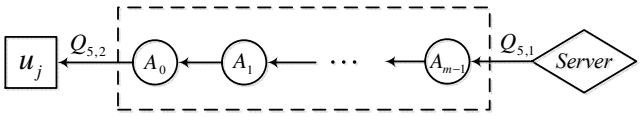

Figure 5: Data reply in M-ORAM.

[Q5.1] The storage server returns each encrypted data item in the requested buckets to anonymizer A_{m-1} in the following format:

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

[Q5.2] A_{m-1} first updates the second component of the pair by multiplying it with $(g_i\beta_j)^{y(l)z}$ (which it has learned at the end of [Q1.2]):

$$\left(g_{i'}^{x(l)}, (g_{i'}^{-z} g_i^z d_{i'} \beta_j^z)^{y(l)}\right). \tag{9}$$

Then, each anonymizer A_k ($k = m-1, \cdots, 0$) updates it and forwards to A_{k-1}:

$$\left(\left(g_{i'}^{x(l)}\right)^{\frac{1}{\prod_{t=k}^{m-1} x_t(l)}}, \left((g_{i'}^{-z} g_i^z d_{i'} \beta_j^z)^{y(l)}\right)^{\frac{\prod_{t=k}^{m-1} h(s_t,j)}{\prod_{t=k}^{m-1} y_t(l)}}\right). \tag{10}$$

Note that $x_k(l)$, $y_k(l)$, and s_k are secrets preloaded to A_k. After the message has traversed the entire anonymizer chain, it becomes

$$\begin{aligned}&\left(\left(g_{i'}^{x(l)}\right)^{\frac{1}{\prod_{t=0}^{m-1} x_t(l)}}, \left((g_{i'}^{-z} g_i^z d_{i'} \beta_j^z)^{y(l)}\right)^{\frac{\prod_{t=0}^{m-1} h(s_t,j)}{\prod_{t=0}^{m-1} y_t(l)}}\right) \\ &= \left(g_{i'}, \left(g_{i'}^{-z} g_i^z d_{i'} \beta_j^z\right)^{w_j}\right),\end{aligned} \tag{11}$$

according to Equations (1), (3), and (4). Then, A_0 returns the data item to U_j.

[Q6: Obtain Desired Data] If the ID of the returned data item matches g_i (meaning that $i = i'$), U_j performs the following computation to the second component of the pair: (i) raising it to the power of $\frac{1}{w_j}$ (where w_j was preloaded to U_j); and (ii) dividing it by β_j^z (which U_j has learned at the end of [Q1.2]). As a result, the second component of the pair has become:

$$\frac{\left(\left(g_{i'}^{-z} g_i^z d_{i'} \beta_j^z\right)^{w_j}\right)^{\frac{1}{w_j}}}{\beta_j^z} = d_i, \tag{12}$$

which is exactly the content of U_j's desired data item. For other returned data items whose ID does not match g_i, they are stored in a local buffer at the user.

4.3 Phase 2: Data Uploading

The *data uploading* phase in M-ORAM also becomes more complicated as it involves the anonymizers. As shown in Figure 6, it now consists of three steps as follows.

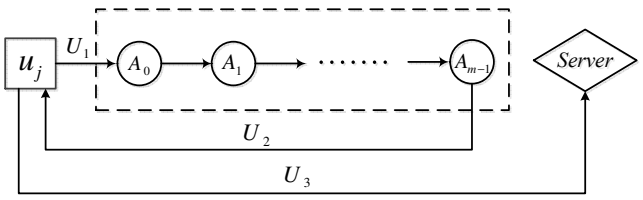

Figure 6: Data uploading in M-ORAM.

[U1] In order to re-encrypt and then upload a data item $D_{i'} = (g_{i'}, d_{i'})$ to the storage server, U_j first picks two elements $r_{j,2}$ and $r_{j,3}$ uniformly at random from $F_q \setminus \{0\}$, and sends the following message to A_0:

$$\left\langle g_{i'}^{x(l)}, (d_{i'} g_{i'}^{-z} g_i^z)^{w_j r_{j,3}}), g_i^{r_{j,2}} \right\rangle. \tag{13}$$

[U2] Upon receiving the message, each anonymizer A_k ($k = 0, \cdots, m-1$) updates it and forwards to A_{k+1}:

$$\begin{aligned}&\left\langle \left(g_{i'}^{x(l)}\right)^{\frac{\prod_{t=0}^{k} x_t^{\text{new}}(l)}{\prod_{t=0}^{k} x_t(l)}}, \left((d_{i'} g_{i'}^{-z} g_i^z)^{w_j r_{j,3}}\right)^{\frac{\prod_{t=0}^{k} y_t^{\text{new}}(l)}{\prod_{t=0}^{k} h(s_t,j)}}, \right. \\ &\left. \left(g_i^{r_{j,2}}\right)^{\prod_{t=0}^{k} y_t^{\text{new}}(l) z_t} \right\rangle,\end{aligned} \tag{14}$$

where $x_k^{\text{new}}(l)$ and $y_k^{\text{new}}(l)$ are the new keys selected by A_k for re-encryption. Note that z_k and s_k are secrets preloaded to A_k. After the message has traversed the entire anonymizer chain, it becomes

$$\begin{aligned}&\left\langle \left(g_{i'}^{x(l)}\right)^{\frac{\prod_{t=0}^{m-1} x_t^{\text{new}}(l)}{\prod_{t=0}^{m-1} x_t(l)}}, \left((d_{i'} g_{i'}^{-z} g_i^z)^{w_j r_{j,3}}\right)^{\frac{\prod_{t=0}^{m-1} y_t^{\text{new}}(l)}{\prod_{t=0}^{m-1} h(s_t,j)}}, \right. \\ &\left. \left(g_i^{r_{j,2}}\right)^{\prod_{t=0}^{m-1} y_t^{\text{new}}(l) z_t} \right\rangle \\ &= \left\langle g_{i'}^{x^{\text{new}}(l)}, (d_{i'} g_{i'}^{-z} g_i^z)^{y^{\text{new}}(l) r_{j,3}}, g_i^{r_{j,2} z y^{\text{new}}(l)} \right\rangle,\end{aligned} \tag{15}$$

which is returned to U_j by A_{m-1}.

[U3] Upon receiving the message, U_j extracts the re-encrypted data ID:

$$g_{i'}^{x^{\text{new}}(l)},$$

and performs the following computation:

$$\left[(d_{i'} g_{i'}^{-z} g_i^z)^{y^{\text{new}}(l) r_{j,3}}\right]^{1/r_{j,3}} \left[g_i^{r_{j,2} z y^{\text{new}}(l)}\right]^{-\frac{1}{r_{j,2}}}$$

to obtain

$$(d_{i'} g_{i'}^{-z})^{y^{\text{new}}(l)},$$

which is the re-encrypted data content. Then, the re-encrypted data item $\left(g_{i'}^{x^{\text{new}}(l)}, (d_{i'} g_{i'}^{-z})^{y^{\text{new}}(l)}\right)$ is uploaded to a temporary *shuffling buffer* at the storage server.

4.4 Phase 3: Data Shuffling

As in C-ORAM, data shuffling is also conducted in the five steps. However, it differs from the data shuffling in C-ORAM in two aspects:

- Shuffling is performed by anonymizers not the user.
- As there are multiple anonymizers, Steps 2, 3 and 4 should be conducted by all the anonymizers collaboratively. Also since anonymizers may be compromised, extra measures should be taken to protect the mapping between data items before and after shuffling from being known to any proper set of all the anonymizers. This way, the mapping is kept secret if at least one anonymizer is not compromised.

The following data shuffling algorithm outlines the differences for data shuffling in M-ORAM from that in C-ORAM

(especially, **S2**, **S3** and **S4**). Each anonymizer A_k will execute **S2**, **S3** and **S4** completely, until all of them finish the execution. In order of simplicity, we denote the data D_i before A_k raises its ID and content to $x_k(l_s)$ and $y_k(l_s)$ as D_i^k, therefore, the initial data D_i can be represented as D_i^0.

Algorithm 2 Data Shuffling in M-ORAM

S1: Determine Shuffling Layer
$l := 0$
$S := S \cup S_0$
while $\phi_l < |S|$ **do**
  $S := S \cup S_{l+1}$
  $l := l + 1$
end while
$l_s := l$
for $k := 0$ **to** $m - 1$ **do**
S2: Data Encryption
  for $D_i^k \in S$ **do**
    A_k.Download(S, D_i^k)
    $D_i^{k+1} := A_k$.Transform($(D_i^k, \frac{x_k(l_s)}{x_k(l)}, \frac{y_k(l_s)}{y_k(l)})$)
    $E_{A_k}(D_i^{k+1}) := A_k$.Encrypt($D_i^{k+1}$, A_k)
    A_k.Upload(S, $E_{A_k}(D_i^{k+1})$)
  end for
S3: Data Sorting
  A_k.Obliviously-Sort(S)
S4: Data Decryption
  for $D_i^{k+1} \in S$ **do**
    A_k.Download(S, $E_{A_k}(D_i^{k+1})$)
    D_i^{k+1} := Decrypt(D_i^{k+1}, A_k)
    A_k.Upload(S, D_i^{k+1})
  end for
end for
S5: Server Locates Data into Buckets
Server.Map(l_s, S)

Specifically, all data items at layers $0, \cdots, l_s$ should be updated through the collaboration of all anonymizers, such that the ID of each data item becomes encrypted by $x(l_s)$ and the content of each data item becomes encrypted by $y(l_s)$. In addition, all the n data items in the shuffling buffer should be randomly permuted.

Therefore, for each anonymizer A_k, it actually does the following:

[S2] A_k first downloads D_i^k from the shuffling buffer and raises the ID part of the data item to power $x_k(l_s)/x_k(l)$ and the content part to power $y_k(l_s)/y_k(l)$ (Notice, after all anonymizers have updated accordingly, D_i will be in form of $(g_i^{x(l_s)}, (g_i^{-z} d_i)^{y(l_s)})$). Then, the updated data item will be encrypted by A_k's private key and uploaded to the shuffling buffer at the storage server. Let denote the total number of data items stored at the shuffling buffer as n.

[S3] A_k performs data shuffling based on D_i^{k+1}'s data ID by adopting Goodrich's randomized shellsort algorithm.

[S4] After data shuffling, A_k removes the private-key encryption on data and uploads the data back to the shuffling buffer.

As a result, since not all of the anonymizers can be compromised, the overall data shuffling is data-oblivious shuffling guaranteed by the non-compromised anonymizers.

5. SECURITY ANALYSIS

This section presents the security analysis of the proposed C-ORAM and M-ORAM schemes.

5.1 Security of C-ORAM

Our proposed C-ORAM follows the existing hierarchical, bucket-based, and hash function-based ORAMs [1], with the following major differences: (i) a balanced mapping mechanism is designed to map data items to buckets during the shuffling process; (ii) as retrieved data items are removed from their buckets, empty buckets may be left at layers, and therefore a bitmap is requested from the storage before retrieving any bucket at a non-empty layer. Hence, it is necessary to show the new mapping mechanism can ensure a negligible bucket overflow probability, and the probability of failure in locating non-empty buckets during the data query process is also negligible.

5.1.1 Probability of Bucket Overflow

Theorem 1. *For any layer l, no buckets on l will overflow with very high probability after data shuffling. To be more specific, $\forall 1 \leq l \leq L$,*

$$\Pr[\text{Buckets overflow on layer } l] \leq O(N^{-\log\log N}). \quad (16)$$

The proof sketch of this theorem is presented at Appendix 1.

5.1.2 Failure Probability in Locating Non-empty Buckets during Data Query

Theorem 2. *Among X buckets arbitrarily selected from layer l, the probability for all the buckets to be empty is at most $e^{-\frac{X}{c}}$, where $c = 8\log\log N \cdot (4\log\log N + 1)$.*

Proof. For each layer $l < T$, the number of buckets at the layer is $n_l = 2^{l+1}\log N \log\log N$. According to the design of C-ORAM, when a shuffling process has just been conducted for the layer, the layer should have at least n_{l-1} data items. After that, the number keeps decreasing until the next time a shuffling process is conducted for the layer. So, the number may reach the minimum value right before a shuffling process begins. The minimum value can be estimated as

$$\frac{n_{l-1}}{4\log\log N + 1}$$

due to the following reasons: (i) the condition to trigger a shuffling process for layer l is that the number of data items at layers $0, \cdots, l-1$ has reached or exceeded n_{l-1}; (ii) the data items at layers $0, \cdots, l-1$ were brought from layer $l, \cdots, T$; (iii) in the worst case, a fraction $\frac{1}{4\log\log N+1}$ of the above data items were brought from layer l, which could happen if $8\log\log N$ data items (in two buckets) from together and only 2 date items (in two buckets) from layer T were brought to layers $0, \cdots, l-1$ after each of the past queries. If all the minimum number of data items at layer

l fill as few buckets as possible according to theorem 1, the number of non-empty buckets is at least

$$\frac{n_{l-1}}{4\log\log N(4\log\log N+1)}$$

Hence, for a randomly selected bucket B_k in layer l, the probability for the bucket to be non-empty is

$$\begin{aligned}&\Pr[B_k \text{ is non-empty}]\\ &\geq \frac{n_{l-1}}{4n_l\log\log N(4\log\log N+1)}\\ &= \frac{1}{8\log\log N\cdot(4\log\log N+1)}\\ &= \frac{1}{c}.\end{aligned} \tag{17}$$

For layer T, if it stores at least N data items initially, the minimum number of data items that should stay at the layer is $\frac{N}{2}$, because all data items should be shuffled to this layer as long as the number of data items at layers $0,\cdots,T-1$ reaches or exceeds n_{T-1} (i.e., $\frac{N}{2}$). Hence, the non-empty probability for an arbitrary bucket B_k at layer T is

$$\begin{aligned}&\Pr[B_k \text{ is non-empty}]\\ &\geq \frac{\frac{N}{2}}{4N\log\log N} = \frac{1}{8\log\log N} \geq \frac{1}{c}.\end{aligned} \tag{18}$$

Therefore, the probability for all X arbitrary buckets at a layer l to be empty is

$$\begin{aligned}&\Pr[\text{all } X \text{ buckets are empty at layer } l]\\ &\leq (1-\frac{1}{c})^X = (1-\frac{1}{c})^{c\cdot\frac{X}{c}} \leq e^{-\frac{X}{c}}.\end{aligned} \tag{19}$$

□

Corollary 3. *The probability for C-ORAM to fail in finding two non-empty buckets to query in steps Q3 is negligible.*

Proof. In the data query phase of C-ORAM, two buckets should be queried at each non-empty layer. In step Q3 of C-ORAM, a bitmap of

$$\min\{2^l, 8\log\log N(4\log\log N+1)\}\log N\log\log N$$

bits should be retrieved. According to Theorem 2, the probability for all the buckets represented by these bits to be empty is negligible, i.e., at most $O(N^{-\log\log N})$. □

5.2 Security of M-ORAM

The purpose of M-ORAM is to protect the access pattern of an innocent user from being revealed to the adversary who has control on a user and some but not all anonymizers. In the following, we first define a game to formally specify the desired property of access pattern protection, based on which, we then present a theorem that quantifies the privacy preservation capability of our proposed scheme and describe the proof of the theorem.

5.2.1 Game Definition: Formal Description of Access Pattern Protection

A game $\mathcal{G}(N,m)$ between a *challenger* (who stands for an innocent user and non-compromised anonymizers) and an *adversary* (who controls a user, the storage server and some anonymizers) is defined as follows.

- *Initialization*: The challenger initializes the storage server, the N exported data items, the chain of m anonymizers, and two users including user U_0 who is innocent and U_1 who is controlled by the adversary.
- *Phase I Queries*: The adversary can make any number of queries of the following types.

  Anonymizer Compromising: The adversary requests to compromise an anonymizer. Then, the information owned by the compromised anonymizer and its behaviors will be controlled by the adversary. However, we require at least one of the m anonymizers not be compromised.

  U_0's data access: The adversary requests U_0 to query a data item, and the related processes of data request, reply, upload, and shuffling are performed accordingly. The adversary may specify the queried data item to be either (i) a random one that has not been queried by U_0, or (ii) a specific one that has already been queried by U_0. However, the adversary cannot specify a data ID for U_0 to query.

  Bucket inspection: The adversary specifies a bucket, and the challenger returns all the data items stored at the bucket.

  U_1's data access: U_1 (controlled by the adversary) appears to query a data item, and the related processes of data request, reply, upload, and shuffling are performed. Before each query, U_1 may request an ID of data item that it has not queried before; in response, a data ID is randomly picked and provided.
- *Selection I*: the adversary requests U_0 to arbitrarily query a data item, denoted as α_0.
- *Phase II Queries*: same as the Phase I. Note that, the adversary may request U_0 to query α_0, which it has queried during the Selection I phase.
- *Selection II*: the adversary requests U_0 to arbitrarily query a data item, denoted as α_1.
- *Phase III Queries*: same as the Phase I. Also note that, the adversary may request U_0 to query α_0 or α_1.
- *Challenge*: The challenger randomly decides a binary bit b, and then requests U_0 to query data item α_b.
- *Phase IV Queries*: same as the Phase III queries, except that the adversary is not allowed to directly specify α_0 or α_1 to query, but is allowed to specify α_b or α_{1-b} to query.

- *Response*: the adversary returns a binary bit b' as a guess of the b chosen by the challenger at the challenge phase.

The adversary wins the game if $b' = b$; otherwise, it loses. The advantage for the adversary to win the game is defined as $|Pr[b = b'] - 1/2|$.

5.2.2 Security Property of Our Scheme

The security property of our proposed complete scheme is formally stated in the following theorem.

Theorem 4. *If there is an algorithm $\mathcal{A}$ that can win the game $\mathcal{G}(p, m)$ with an advantage σ in time t, then an algorithm $\mathcal{B}$ can be developed to solve the following matching Diffie-Hellman problem in group G_p with advantage σ in time $O(t^2)$: Consider a multiplicative cyclic group G_p of order p and generator g. Given g^{a_0}, g^{a_1}, g^c, and $(g^{a_r c}, g^{a_{1-r} c})$, for some a_0, a_1 and c randomly picked from F_p and a binary bit r randomly picked from $\{0, 1\}$, determine r.*

A proof sketch of the theorem is presented at Appendix 2.

6. OVERHEAD ANALYSIS

In this section, we will analyze the communication and storage overhead of the proposed M-ORAM scheme. As M-ORAM involves three parties: users, anonymizers, and server, we analyze the overhead from the following aspects. Recall that there is a total of

$$T + 1 = \lceil \log N - \log(\log N \log\log N) \rceil$$

layers in the storage hierarchy, each layer l has

$$\phi_l = 2^{l+1} \log N \log\log N$$

buckets, and each bucket can store up to $B = 4 \log\log N$ data blocks.

User's Query Overhead ($q_u(N)$): For each query, the user needs to retrieve two buckets from each layer, together with an X-bit string from the bitmap, where X is

$$\min\{2^{l+1}, 16 \log\log N(4 \log\log N + 1)\} \log N \log\log N.$$

In a typical file system, the data block size usually is $64KB$ or $256KB$. This means that, as long as there is a total of $N < 2^{64} \approx 10^{20}$ or $N < 2^{128} \approx 10^{40}$ data items in the system, the X-bit string can be fit into a single data block; hence, the communication overhead for the bitmap retrieval is negligible. Therefore, in such realistic file systems, the query overhead for the user is (in the unit of data block):

$$q_u(N) < 2 \cdot T \cdot B \approx 8 \log N \log\log N. \tag{20}$$

Anonymizer's Query Overhead ($q_A(N)$): For each query, each data item needs to go through every anonymizer. Thus, the query overhead for each anoynmizer is the same with user's query overhead without the bitmap retrieval. Therefore, we have:

$$q_A(N) < q_u(N). \tag{21}$$

User's Shuffling Overhead ($s_u(N)$): In M-ORAM, the user is not responsible for data shuffling. Thus, we have:

$$s_u(N) = 0. \tag{22}$$

Anonymizer's Shuffling Overhead ($s_A(N)$): In M-ORAM, data shuffling at layer l is triggered only if the total number of data items on all higher layers exceeds ϕ_{l-1}. On the other hand, each time when a user accesses the storage server, at most $q_A(N)$ data items will be inserted into the top layer (i.e., layer 0). This means that, data shuffling at layer l is triggered every $\frac{\phi_l}{q_A(N)} = 2^l$ accesses. Moreover, as M-ORAM employs randomized shellsort to perform data shuffling, the overhead for shuffling ϕ_l buckets at layer l is $O(\phi_l \log \phi_l)$. Therefore, the amortized communication overhead for data shuffling at layer l is:

$$s_A^l(N) < O\left(\frac{\phi_l \log \phi_l}{2^l}\right). \tag{23}$$

Overall, the communication overhead incurred by data shuffling has the following upper bound:

$$\begin{aligned} s_A(N) &= \sum_{l=0}^{l} s_A^l(N) < \sum_{l=0}^{T} O\left(\frac{\phi_l \log \phi_l}{2^l}\right) \\ &= O(\log^3 N \log\log N). \end{aligned} \tag{24}$$

User's Cache and Storage Overhead: As explained in the detailed description of M-ORAM in Section 4, each user needs to maintain a $O(\log N \log\log N)$ cache during data retrieval but zero permanent storage.

Server's Storage Overhead: The storage server needs to maintain a hierarchy of buckets, with a size of $O(N \log\log N)$.

Table 6 compares M-ORAM against the best-performance single-user ORAM [7] (to our knowledge).

Table 1: Overhead Comparison

Overhead		Best single-user ORAM	M-ORAM
Query	User	$O(\frac{\log^2 N}{\log\log N})$	$O(\log N \log\log N)$
	Anonymizer	N.A.	$O(\log N \log\log N)$
Shuffling	User	$O(\frac{\log^2 N}{\log\log N})$	N.A.
	Anonymizer	N.A.	$O(\log^3 N \log\log N)$
Client Cache		$O(1)$	$O(\log N \log\log N)$
Client Storage		$O(1)$	$O(1)$
Server Storage		$O(N)$	$O(N \log\log N)$

7. RELATED WORK

Numerous ORAM schemes have been proposed in the past decade to protect a single user's access pattern privacy. One type of ORAM schemes use indices as the major data-locating technique, such as [10, 11]. Indices can be stored at the user side or outsourced to the storage server, which incurs either storage overhead at the user side or communication overhead between the user and server. The scheme

proposed in [10] achieves $O(\log^3 N)$ worst-case communication overhead with constant local storage. The one proposed in [11] achieves asymptotical $O(\log N)$ communication overhead with $O(\sqrt{N})$ local storage, under the assumption that there is a total of $N \leq 2^{34}$ data items.

Another type of ORAM schemes are based on hash functions. Two hash based schemes were proposed by Goldreich and Ostrovsky [1]: a *square-root* scheme with $O(\sqrt{N}\log N)$ communication overhead and $O(\sqrt{N})$ local storage, and a *hierarchical* scheme with $O(\log^3 N)$ communication overhead and constant local storage. Since then, many advanced hash based ORAM schemes have been proposed: a *cuckoo-hash based ORAM* [9], a *Bloom-filter based ORAM* [12], and a few hybrid schemes [3–5, 7] based on hybrid structures of bucket-hash and cuckoo-hash with stashes. In all of the hybrid schemes, a bucket-hash data structure is used at the top of the ORAM and a cuckoo-hash data structure with stash is used at the bottom to ensure the overflow probability during data reshuffling to be negligible. Among all the hash based ORAM schemes, the one proposed in [7] yields the best performance with $O\left(\frac{\log^2 N}{\log\log N}\right)$ worst-case communication overhead and constant local storage.

Unfortunately, none of the above schemes supports multiple users in the system. If they are applied directly to a multi-user system, an adversary user can easily obtain the data access patterns of other users by colluding with the storage server. In contrast, our proposed M-ORAM can handle multiple users (who may not trust each other) by introducing *anonymizers* as a new component into the system. We have proved that M-ORAM can protect a user's data access pattern privacy from other users and the storage server as long as at least one of the anonymizers is not compromised. There have been a few works on multi-user ORAM [6, 13]. However, their studies focused on how to support multiple users to access the shared storage server and deal with the associated concurrency issues, based on the assumption that all users trust each other, which differs significantly from the focus of our study.

8. CONCLUSION AND FUTURE WORK

This paper proposes Multi-user ORAM (M-ORAM), a new type of ORAM system, in which a chain of anonymizers is introduced to act as a common proxy between users and the storage server, and therefore users can be securely isolated from each other and the compromise of some but not all anonymizers can be tolerated. Consequently, the design can protect the data access pattern of each individual user from others, as long as not all anonymizers are compromised. Extensive security and overhead analysis has been conducted to quantify the strength of the scheme in protecting individual user's access pattern and the costs incurred to provide the protection.

In the future, we plan to improve the performance of the current design. Potential directions include (i) optimizing the shuffling algorithm to reduce the data shuffling cost, and (ii) optimizing the design of data mapping mechanism to decrease the bucket size and hence reduce the cost for both query and shuffling.

9. REFERENCES

[1] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious ram. In *JACM'96*. 1996.

[2] GOODRICH, M. T. Randomized shellsort: a simple oblivious sorting algorithm. In *Proc. SODA'10* (2010).

[3] GOODRICH, M. T., AND MITZENMACHER, M. Mapreduce parallel cuckoo hashing and oblivious ram simulations. In *Proc. CoRR'10* (2010).

[4] GOODRICH, M. T., AND MITZENMACHER, M. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Proc. ICALP'11* (2011).

[5] GOODRICH, M. T., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Oblivious ram simulation with efficient worst-case access overhead. In *Proc. CCSW'11* (2011).

[6] GOODRICH, M. T., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proc. SODA'12* (2012).

[7] KUSHILEVITZ, E., LU, S., AND OSTROVSKY, R. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proc. SODA'12* (2012).

[8] MITZENMACHER, M., RICHA, A. W., AND SITARAMAN, R. The power of two random choices: a survey of techniques and results. In *Handbook of Randomized Computing* (2000).

[9] PINKAS, B., AND REINMAN, T. Oblivious ram revisited. In *CRYPTO'10*. 2010.

[10] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Proc. ASIACRYPT'11* (2011).

[11] STEFANOV, E., SHI, E., AND SONG, D. Towards practical oblivious ram. In *Proc. NDSS'12* (2012).

[12] WILLIAMS, P., AND SION, R. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. CCS'08* (2008).

[13] WILLIAMS, P., SION, R., AND TOMESCU, A. Privatefs: a parallel oblivious file system. In *Proc. CCS'12* (2012).

[14] WILLIAMS, P., SION, R., AND TOMESCU, A. Single round access privacy on outsourced storage. In *Proc. CCS'12* (2012).

Appendix 1: Proof Sketch for Theorem 1

Proof. In C-ORAM, the capacity of each layer l is $\phi_l = 2^{l+1} \log N \log\log N$. If we set the bucket size of layer l to be $B_l = 4 \log\log \phi_l$, the bucket overflow probability for layer l can be estimated as:

$$\Pr[\text{Bucket overflow on layer } l] = O(\frac{1}{\phi_l}), \qquad (25)$$

according to the layered induction method adopted in [8]. Hence, when we set the bucket size as

$$B = \max_l\{B_l\} = 4 \log\log \phi_T = 4 \log\log N, \qquad (26)$$

the bucket overflow probability for each layer can be bounded by the maximum value of the overflow probability over all layers. Therefore, we have:

$$\begin{aligned} &\Pr[\text{Bucket overflow for every layer}] \\ \leq &\max_l\{\Pr[\text{Bucket overflow on layer } l]\} \\ = &O(\frac{1}{\phi_0}) \\ = &O(\frac{1}{2 \log N \log\log N}). \end{aligned} \qquad (27)$$

Notice that, the upper the layer is, the higher the overflow probability is. However, in case of overflow happening, two new random hash functions can be selected to re-hash all data on the shuffled layer until there is no bucket overflowed on it, as described in the scheme. To guarantee bucket overflow probability to be negligible in N, at most $\log N$ pairs of hash functions for the shuffled layer need to be selected. In this case, the probability of all $\log N$ pairs resulting in bucket overflow on the selected layer will be:

$$\begin{aligned} &\Pr[\text{Bucket overflows on all } \log N \text{ hash functions}] \\ \leq &O((\frac{1}{2 \log N \log\log N})^{\log N}) \\ = &O(N^{-\log\log N}). \end{aligned} \qquad (28)$$

Notice that, as shown in equation 28, the bucket overflow probability on the selected layer, whichever the layer is, is upper bounded by $O(N^{-\log\log N})$. Different from existing bucket-hash ORAMs, for each non-empty bucket, it can be accessed only once and marked as empty before the layer is shuffled. Therefore, the security flaw pointed out in [7] does not appear in C-ORAM. □

Appendix 2: Proof for Theorem 4

Proof. The proof is conducted in two steps: first, the construction of algorithm $\mathcal{B}$ which leverages the game $\mathcal{G}(p, m)$ to solve the matching Diffie-Hellman problem; second, reasoning about the advantage of $\mathcal{B}$ in solving the matching Diffie-Hellman problem.

Matching Diffie-Hellman Problem (MDHP): *Let g be a group generator with order N. Let $a_0, a_1, c_0, c_1 \in_R \mathbb{Z}_N$ and $b \in_R \{0, 1\}$. Given tuple $(g^{a_0}, g^{a_1}, g^{a_0 c_0}, g^{a_1 c_1})$, to decide the following tuple $(g^{c_b}, g^{c_{1-b}})$, i.e., to find b is a hard problem.*

Notice the MDHP will degrade to Decisional Diffie-Hellman Problem (DDHP) if c_0 and c_1 are the same, however, as MDHP is equivalent to DDHP, we still use MDHP without loss of generality.

In the first step, we show how to construct algorithm $\mathcal{B}$, which is provided with the input of the MDHP and acts as the challenger to play the game $\mathcal{G}(p, m)$ with the adversary algorithm $\mathcal{A}$.

In the initialization phase, $\mathcal{B}$ initializes the storage server by establishing N data items, m anonymizers with restriction that at least one of them cannot be compromised. As indicated above, a_0 and a_1 are two data IDs among the database which are chosen according to the input of the MDHP from $\{0, \cdots, N-1\}$.

At selection I query phase, $\mathcal{A}$ is able to make any number of "legal" queries (including arrogating $\mathcal{B}$ to make queries, asking compromised user to make queries) such that $\mathcal{B}$ needs to respond. Then, at selection I phase, $\mathcal{B}$ launches the query of data item of ID g^{a_0}. Similarly, $\mathcal{A}$ acts similarly in phase II query and $\mathcal{B}$ launches the query of the data item of ID g^{a_1} in selection II phase. Before the challenge phase, $\mathcal{A}$ can still make any number of "legal" queries which are the same as phase I and phase II queries.

At the challenge phase, $\mathcal{B}$ uniform randomly picks up a data ID g^{a_b} to query, where b is randomly picked up from $\{0, 1\}$ by $\mathcal{B}$. After the query by $\mathcal{B}$ was made in the challenge phase, it will send the transcript of the query to data items whose ID is g^{a_b} to $\mathcal{A}$. Now, $\mathcal{A}$ is able to make the last "legal" query phase, phase IV query, with the exception that it is now unable to specify g^{a_0} and g^{a_1} directly as it cannot identify these two data items after encryption ($g^{a_0 c}$ and $g^{a_1 c}$) and data shuffling. However, it is allowed to ask the users to query those two data items, whose IDs are denoted as g^{a_b} and $g^{a_{1-b}}$.

When $\mathcal{A}$ finishes phase IV query, it will need to return its guess on b as b' and return b' back to $\mathcal{B}$, then, this will be used by $\mathcal{B}$ as the answer to the original MDHP.

Notice in the challenge phase, $\mathcal{B}$ cannot obtain g^{a_r} or $g^{a_{1-r}}$ from the provided $g^{a_r c}$ and $g^{a_{1-r} c}$ (also note that only g^c, not c, is unknown by $\mathcal{B}$), we adopt the following trick in the game. Right after the last time when the data item of either ID g^{a_0} or ID g^{a_1} is queried before the challenge phase, the data items associated with these two IDs are both brought into the shuffling buffer. Then, during the follow-up shuffling processes, the anonymizer that is not allowed to be compromised always uses c or kc (where k is known by $\mathcal{B}$) to encrypt data items, which makes the data items associated with IDs g^{a_0} and g^{a_1} to become associated with encrypted IDs g^{a_r} and $g^{a_{1-r}}$, with an unknown matching relation between them. Thus, using $g^{a_r c}$ as encrypted ID in a query will result in the query of the data item of actual ID g^{a_r}. Note that our proposed scheme allows $\mathcal{B}$ to launch a query based on either an actual or encrypted data ID. In the former case, anonymizers are used to facilitate the translation from the actual ID to the encrypted ID; while in the latter case, results obtained from anonymizers are not really used.

In the second step, let's look at the $\mathcal{B}$'s advantage in solving MDHP if $\mathcal{A}$ wins the game $\mathcal{G}(N, m)$ with an advantage of σ in time $O(t)$, After investigate the construction of $\mathcal{B}$, when $b = 0$, $\mathcal{B}$ launches a query for the data item with ID g^{a_0} in the challenge phase. If $\mathcal{A}$ wins the game, it should respond $b' = 0$, which is the correct value of b. When $b = 1$, $\mathcal{B}$ launches a query for the data item with ID g^{a_0} in the challenge phase, accordingly. If $\mathcal{A}$ wins the game, it should respond $b' = 1$, which is also the correct value of b. Therefore, when $\mathcal{A}$ wins the game, $\mathcal{B}$ will also provide a correct answer to the MDHP, and vice versa.

The time complexity of $\mathcal{B}$ can also be analyzed and the result is $O(t^2)$ if $\mathcal{A}$ needs time t to win the game.

□