2014

Implementation of a multiuser customized oblivious RAM

Priyangika Rumesh Piyasinghe
Iowa State University

Implementation of a multiuser customized oblivious RAM

by

Priyangika Rumesh Piyasinghe

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Wensheng Zhang, Major Professor

Leslie Miller

Ying Cai

Iowa State University

Ames, Iowa

2014

DEDICATION

I would like to dedicate this thesis to Dr. Wensheng Zhang without whose support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Wensheng Zhang for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Les Miller and Dr. Ying Cai.

ABSTRACT

Outsourcing data to cloud storage has become popular. Example systems such as Google Drive, Amazon S3 and Microsoft Azure are affordable and convenient, and provide scalable storage space. However, since the data management is left to third party, users no longer have physical control of their sensitive data, which raises new challenges in terms of data privacy. Data encryption provides confidentiality, but encryption alone is not enough since information may be leaked through the pattern in which users access the data. In this thesis, we implemented a Customized ORAM(C-ORAM) system that allows oblivious access to remotely-stored data in multi-user scenario. Experiments have shown that C-ORAM can effectively protect user's privacy as well as achieve low communication overhead at individual users.

CHAPTER 1. INTRODUCTION

The popularity of storing data in the "cloud" has increased in recent past because of its cost effectiveness compared to traditional storage systems. In cloud environment, the clients pay only for the resources that they use, and they are more reliable due to the redundancy provided by replications presented in the cloud server. When outsourcing sensitive data such as financial and health records, it becomes a problem of data privacy as users are giving control of their data to third party where in this case it is the cloud provider. The primary way to maintain confidentiality of user's data is to encrypt them using a key, which is only known to the particular user. However, data encryption does not fully guarantee the privacy since the patterns of data access could leak considerable information about the stored data.

From clients' data access patterns, a server can monitor their queries and perform it's own traffic analysis.Remote server can learn the regular accessing patterns of data, and try to relate it to other client information gained by the third party channels. For example, suppose there is always certain stock exchange action take place after query sequence of q1, q2 and q3 from the client. In this case, a curious server can learn about the content of the queries. It can also predict what will happen next when similar sequence of queries appear, even though the data that query is encrypted [Pinkas et. al. 1].

Furthermore, it is also possible to analyze the significance of different areas in the storage, by keeping track of how frequent the same data items access by the client. A malicious server with substantial but limited power, may try apply it's resources to decrypt only the data items which are frequently accessed by the user.

In addition, the server can draw conclusions about associations between queries by considering the users' data access patterns. For an example, suppose a company outsources its data and the employees are accessing those data time to time. By observing and comparing the data

access patterns of the employees, the storage provider can decide what kind of access policy that company follows since in general, higher privilege users have access to larger portion of data than the lower privilege users.

In the context of hiding users' data access patterns from the curious server, Goldreich and Ostrovsky introduced Oblivious RAM (ORAM) architecture [2] which uses repeated encryption and shuffling of outsourced data. The basic idea of hierarchical solution proposed by Ostrovsky can be stated like this: there are sequences of layers which are filled by the data elements at each querying process, and smaller layers are shuffled into larger layers as they fill up. It is required that shuffling also need to be oblivious to the untrusted party. In order to protect the data content, a private key encryption is used when the retrived elements are written back to untrusted RAM. After that, many ORAM constructions have been proposed to protect the users' data access patterns privacy with some restrictions [2-6]. One of the fundamental restrictions in most of such ORAMs, is the assumption of single user access to the remote storage. However, in reality there can be multiple users who access to the remote storage simultaneously.

In C-ORAM proposed by Zhang et. al. [7] there are two entities in client side:multiple users who trust each other, and shared agent. In general we can assume they belong to same organization. C-ORAM support two kind of operations, data query and data shuffling. Querying a data item from remote storage is done by the users while shuffling done by the shared agent.The motivation to develope a system like this is to detach the overhead incured in shuffling phase from the client.The efficiency of an ORAM is measured by amount of local storage, and amount of communication overhead for querying and the shuffling. Compared to the single-user ORAM with the best-known performance [6] C-ORAM archive a lower communication overhead per query by the user $O(logNloglogN)$, a higher communication overhead for data shuffling $O(log^3NloglogN)$ which is handled by shared agent, and a moderately increased local cache at the user O(logN log logN).

In this study, we implement C-ORAM which works with a FUSE-based distributed file system. Experimental results show that users access pattern can be preserved in that there is no difference between querying one same element and randomly picked element for multiple

times. The overhead results for querying and shuffling measured from the evaluation process have close match to the theoretical results.

The rest of this thesis is organized as follows: In Chapter 2, related works are presented. Chapter 3 outlines the problem statement in more detailed. Chapter 4 describes the design of C-ORAM scheme and Chapter 5 includes the detailed implementation of C-ORAM. Chapter 6 includes the results of evaluation and discussion. Finally, Chapter 7 concludes the thesis with some future work suggestions.

CHAPTER 2. RELATED WORK

2.1 Oblivious RAM

Oblivious RAM was first examined as theoretical method by Goldreich and Ostrovsky [8,9,10] for protecting a software from piracy. In that context the processor is trusted while the memory is not. Goldreich and Ostrovsky in [9] prove that oblivious RAM (ORAM) simulation using an outsourced data requires an overhead of at least logN, for a RAM memory of size N. When client side has only a constant size storage, they show how client capable of achieving an overhead of $O(\sqrt{N}logN)$, using a scheme called the "square-root" solution, and with $O(N)$ storage at server. After that with a more complex scheme, they also show how client capable of achieving an overhead of $O(log^3N)$ with $O(NlogN)$ storage at the server, using a scheme called the "hierarchical" solution. Apart from suggesting a hierarchical solution with $O(log^3N)$ amortized cost, Goldreich and Ostrovsky [9] also proposed an ORAM scheme with lower bound amortized cost for client at least $O(logN)$ (for ORAM capacity of N). In 2010, Beame and Machmouchi [11] improved the lower bound to $O(logNloglogN)$.

The application of above mentioned ORAM solutions were not that straight forward. Because those approaches contain several complications and hidden constant factors that make these solutions not practical for real-world use in the context of privacy protection in outsourced data management. Some other works have been done base on Goldreich and Ostrovsky hierarchical solution which can be denoted as index based and hash based ORAM schemes by considering their lookup mechanisms.

As an index based ORAM, Stefanov et al. [12] proposed an ORAM scheme which has reduced worst-case bound for data access. Because of the lower overhead than theoretical ORAMs this is more suitable for real world application. With the ORAM scheme they show

that client can achieve an amortized overhead of $O(logn)$ and worst-case performance $O(\sqrt{N})$, with $O(\epsilon N)$ storage on the client, for a constant $0 < \epsilon < 1$. It is also able to achieve amortized overhead of $O(logN)$ and similar worst-case performance, with a client side storage of $O(\sqrt{N})$. The scheme of [13] provides a tree-based construction that uses poly-logarithmic $O(NlogN)$ server storage and incurs $O(log^2N)$ overhead on each access when the client has access to $O(\sqrt{N})$ local storage.

Another index based ORAM (Path-ORAM)[14] proposed makes $O(logN)$ accesses to the server. It is performance wise better than all other ORAMs which are index based. It uses a position map and stash which are both stored at the client side. The size of position map used is n while the position map is size of $log(N)$. With $O(logN)$ client side Path-ORAM achieves $O(log^2N)$ amortized overhead. The oblivious simulations described above consider a single-client scenario where all accesses, including read-only accesses, are processed sequentially. Extending these solutions to support parallel access is important if we consider multi user scenario. The works of Stefanov and Shi [15] and Williams et al. [16] allow parallel access. The clients access oblivious storage of [15] via a load balancer that is responsible for scheduling client requests.

As a hash based ORAM, Williams and Sion[17] proposed one with a constant overhead by using computation power of the server. With $O(\sqrt{N})$ client side storage they achieved an expected amortized time overhead of $O(log^2N)$. Williams and Sion propose another construction with $O(\sqrt{N})$ client-side storage, that achieves $O(logNloglogN)$ amortized cost [18].Pinkas and Reinman proposed a hash based ORAM [1] construction that achieves $O(logN^2)$ overhead with $O(1)$ client-side storage.Goodrich and Mitzenmacher [19] show that overhead of $O(log^2N)$ in an ORAM simulation can be achieved , with high probability, for a client with constant sized local memory, and $O(logN)$, for a client with $O(N^\epsilon)$ memory, for a constant $\epsilon > 0$. Kushilevitz et al. [20] also show that one can achieve an overhead of $O(log^2N/loglogN)$ in an ORAM simulation, with high probability, for a client with constant-sized local memory.

CHAPTER 3. PROBLEM STATEMENT

3.1 System Model

As shown in Figure 3.1, multiple users, who trust each other, share N data items, which are exported to an un-trusted remote storage server. The users share a trusted local agent, which has limited storage resources(Figure 3.1). The ORAM system involves two types of operations: data query and data shuffling. A user can query the remote storage server by sending requests directly to the server and processing replies from the server; the shared agent can perform data shuffling for the users.

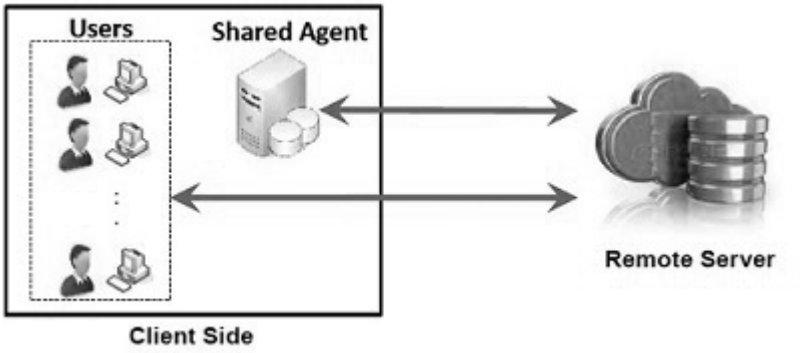


Figure 3.1 C-ORAM Design

3.1.1 Definition of data Item

Let Fp be a finite field of p distinct elements, where p is a large prime number. Let Gp be a multiplicative, cyclic group with also p distinct elements. Hence, for any element $g \in Gp$, elements $g^0, g^1, g^2, ..., g^{p-1}$ should all belong to G.

Each data item, denoted as Di, consists of two components: a unique data ID and the data content that is a sequence of elements of Gp. 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.

For the rest of this work, each data item Di is represented as (gi, di), where $gi \in Gp$ is the data ID and $di \in Gp$ is the data content.

3.1.2 Basic functionalities provided by C-ORAM

From the viewpoint of client side:

- $read(data, pos)$ to read data at physical address pos.
- $write(data, pos)$ to write data from physical address pos.

From the viewpoint of server side:

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

3.2 Threat Model

In the threat model several assumptions are made. First, the users are trusted. The keys, used for encryption, exchange between the users are considered to be secure. Second, communication channel between users and the server is secure. Techniques such as SSL [Freier et al 21] can effectively achieve this. Third, the server is assumed to be curious but not malicious. That means server do whatever the operations mentioned in 3.1 correctly on behalf of the client, but at the same time it may try to figure out the pattern in which client access the data. This work follows standard security definition of ORAM scheme. According to the definition, ORAM system is considered secure if the server cannot pick up anything about the user's data access pattern which is formally defined in[12] as:

Definition: Let $\overrightarrow{y} := ((op_1, u_1, data_1), (op_2, u_2, data_2), ..., (op_M, u_M, data_M))$ denote a data request sequence of length M, where each op_i denotes a $read(u_i)$ or a $write(u_i; data)$ operation. Specifically, u_i denotes the identifier of the block being read or written, and datai denotes the data being written. Let $A(\overrightarrow{y})$ denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests $\overrightarrow{y}$. An O-RAM construction is said to be secure if for any two data request sequences $\overrightarrow{y}$ and $\overrightarrow{z}$ of the same length, their access patterns $A(\overrightarrow{y})$ and $A(\overrightarrow{z})$ are computationally indistinguishable by anyone but the client.

CHAPTER 4. SYSTEM DESIGN

4.1 Overview

C-ORAM differs from previously introduced bucket ORAM schemes mainly due to the way how it selects the buckets. A hash function associated with each layer select two possible locations for a given data item. The "locate" operation of data item into bucket is probabilistic, which means data items are always inserted into the bucket with less number of items. This guarantees there is less chance in overflowing a bucket. Following sections present system initialization, data query, uploading and shuffling which are the three phases of operations in C-ORAM(Figure 4.2)

4.2 System Initialization

C-ORAM organizes the storage as a hierarchy of buckets with the following properties;

- $T + 1$ layers, where $T = [logN - log(logNloglogN)] - 1$.
- ϕ_l buckets in each layer, where $\phi_l = 2^{l+1}logNloglogN$. Bottom layer of the hierarchy ($LayerT$) with at least N buckets.
- Public hash function H_l for each layer, which maps each element of group G_p to two integers uniformly at random between 0 and ϕ_{l-1}.
- $4loglogN$ data items in each bucket.
- A counter, which keep track of the elements in each bucket.

At system initialization, the user is preloaded with two keys $x(l)$ and $y(l)$ for each *layer* l. $x(l)$ is for encrypting data id; $y(l)$ is for encrypting data content. Initially, the user encrypts and exports all N data items to the bottom layer (T) of storage hierarchy at storage server. It is done by computing $g_i^{x(T)}$, $d_i^{y(T)}$ for each data item i and exporting to one of the two buckets generated by the hash function associated with layer T.

4.3 Data Query

To query a data item, C-ORAM executes the data query phase in iterations for each each nonempty *layerl* of the storage hierarchy from the top *layerl* = 0 to the bottom layer $l = T$. To query an item $D_i = (g_i, d_i)$ iteration of 6 steps are carried out.

- Step1: The user computes the encrypted ID $g_i^{x(l)}$
- Step2: The user computes the positions pos0 and pos1 of the buckets that may contain the desired data item (pos0, pos1) $\leftarrow H_l(g_i^{x(l)})$
- Step3: The user retrieve bitmap of the layers which indicate whether a bucket is empty or not. If the position need to query is empty, user pick non-empty bucket from that layer.
- Step4: Bucket request. If the bucket D_i has already been found at a layer higher than l,then from following layers user randomly picks a nonempty positions. Otherwise user picks two non-empty buckets that computed from hash function.
- Step5: Storage server returns all the encrypted data items at the requested buckets directly to the user.
- Step6: User uses keys $x(l)$ and $y(l)$ to decrypt ID and content of return data. If desired item found, perform necessary action to the data content; otherwise, the data item is stored temporarily in a local cache at the user.

4.4 Data Upload

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. The user picks a new pair of keys $x(l)$ and $y(l)$ randomly 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.

4.5 Data Shuffling

Data shuffling is done as shown in Figure 4.1.

Algorithm 1 Data Shufling in C-ORAM

S1: Determine Shuffling layer

1: $l := 0$
2: $S := S \cup S_0$
3: **while** $\phi_l < |S|$ **do**
4: $\quad S := S \cup S_{l+1}$
5: $\quad l := l + 1$
6: **end while**
7: $l_s := l$

S2: Data Encryption

8: **for** $D_i \in S$ **do**
9: $\quad$ user.Download(S, D_i)
10: $\quad D'_i :=$ user.Transform$\left(D_i, \frac{x(l_s)}{x(l)}, \frac{y(l_s)}{y(l)}\right)$
11: $\quad E_u(D'_i) :=$ user.Encrypt(D'_i, u)
12: $\quad$ user.Upload$(S, E_u(D'_i))$
13: **end for**

S3: Data Sorting

14: user.Obliviously-Sort(S)

S4: Data Decryption

15: **for** $D'_i \in S$ **do**
16: $\quad$ user.Download$(S, E_u(D'_i))$
17: $\quad D'_i :=$ user.Decrypt(D'_i, u)
18: $\quad$ user.Upload(S, D'_i)
19: **end for**

S5: Server Locates Data into Buckets

20: Server.Map(l_s, S)

Figure 4.1 Algorithm 1 : Data Shuffling in C-ORAM [Zhang et. al. 7]

[S1] Determine the layer for shuffling. 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 $0, ..., l_{s-1}$ 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 $0, ..., l_s$ is less than the total number of buckets at layer l_s.

[S2] All data items at layers $0, ..., l_s$ update such that the ID of each data item becomes encrypted by $x(l_s)$ and the content of each data item becomes encrypted by $y(l_s)$.

First download all data items, encrypt with users private key and upload the new items back to shuffling buffer. The new data items will be as $[D'_i = (g_i^{x(l)}, d_i^{x(l)})]$

[S3] For the $|S|$ data items stored on the shuffling buffer, the user performs data-oblivious sorting, based on the new data ID.

[S4] After the end of data shuffling, the user further scans all data in the data buffer to remove the private-key encryption.

[S5] Storage server moves the data items at the shuffling buffer to the buckets at layer l_s using $logN$ random hash functions $H_{l_s}^{\theta}(g_i^{x(l_s)})$ where $\theta \epsilon \{1, ..., logN\}$.

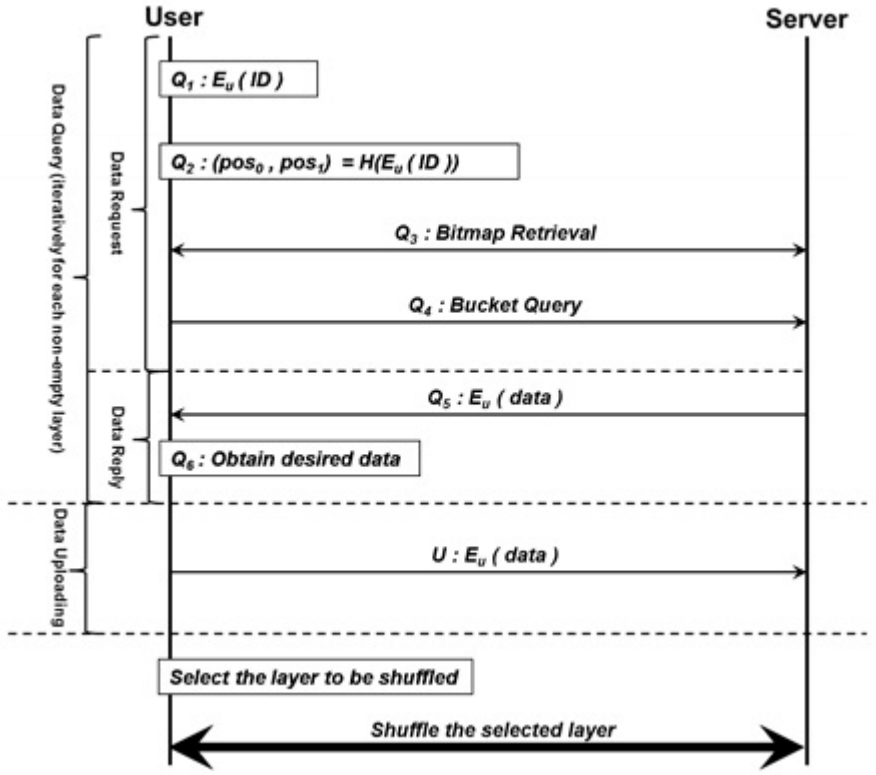


Figure 4.2 Three phases of operations in C-ORAM [Zhang et. al. 7]

CHAPTER 5. IMPLEMENTATION

We implemented the C-ORAM in a user-space distributed file system. With the distributed file system, we can make it easy for users to access and manage files which are physically distributed across various areas of the disk, or across a network which may be multiple disks or multiple computer systems.

5.1 FUSE

Our user-space distributed system is built based on FUSE(File System in User Space). FUSE is an operating system mechanism for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module provides a "bridge" to the actual kernel interfaces.

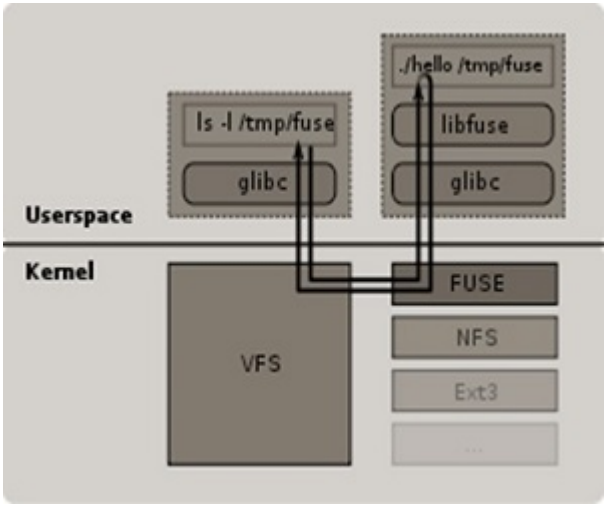


Figure 5.1 Path of a filesystem call [Szeredi et. al. 22]

FUSE provide simple more uniform API function to interact with file system (Table 5.1).Be-

cause of the file system is running in the user space it is easy to debug and failures will not result in crashing the machine like kernel space file systems.

Table 5.1 API Functions

	filel	**directory**
create	mknod	mkdir
remove	unlink	rmdir
read	read	readdir
write	write	
misc	open, truncate	

5.1.1 FUSE Basic Functions

According to FUSE documentation [23], many FUSE functions provide two ways to identify the file being operated. The default option that always available is the "path" argument, which is the full pathname (relative to the file system root) of the file in request. However, due to the expensiveness of pathname lookup, FUSE sometimes provides "file handle" in the "$fuse\_file\_info$" structure as alternative option.

When using FUSE to implement a new file system, the first step is to set the fields in *open*, *create*, and *opendir* functions so that the other functions can use them. Although, the file handle in new FUSE file system implementation is a pointer to a useful data structure, it can be used as either an index into an array or a hash key, or anything else that is useful. Following functions are some of the key function need to be modified when implementing a new file system using FUSE.

$getattr(constchar * path, structstat * stbuf)$

This function gets called every time whether the operation is for directory or a file. Therefore it needs to be implemented first. For the given pathname, this fills in the elements of the "stat" structure.

$readdir(constchar*path, void*buf, fuse\_fill\_dir\_tfiller, off\_toffset, structfuse\_file\_info* fi)$

FUSE provides a mechanism to store the entries in a directory structure and the basic mechanism is the creation of data and calling a FUSE-supplied function to place it back in the structure.

The $filter()$ is one of the function called by the $readdir()$. The goal of this function is to insert directory entries into the directory structure which is called as $buf.filler()$. The function prototype is given as:

$intfuse\_fill\_dir\_t(void*buf, constchar*name, conststructstat*stbuf, off\_toff);$

It is important to note that the $readdir()$ uses $filler()$ in a simple way as possible to just copy the related directorys filename into the mounted directory.

$open(constchar*path, structfuse\_file\_info*fi)$

This uses to open a file. To handle the files need to allocate any necessary structures need to set the path using fi→fh which is described as absolute path above. Moreover, fi includes some additional fields that might be useful to an advanced files system.

$read(constchar*path, char*buf, size\_tsize, off\_toffset, structfuse\_file\_info*fi)$

This function first fill the buffer buf with the $size$(in bytes) of the given files and the beginning of offset(in bytes) to the file. Then, it returns either the number of bytes being transferred or 0 if the offset was at/beyond the end of the file. This function is required for any kind of filesystem as reading and writing are essential file opearations.

$release(constchar*path, structfuse\_file\_info*fi)$

After FUSE completed all the operations with the given file, the function $release()$ is called in order to free any temporarily allocated data structures. There exists one $release$ per $open$. Therefore, if some function carried out at the open time of the file, by the release time, operation needs to be completed.

By default, FUSE is multithreaded. That is useful when there are multiple clients like

in our ORAM scheme.Multithreading lets one client to proceed even if other client stop the communication. However there are also drawbacks of this.One is, it introduces the possibility of race conditions. Also it makes debugging harder.

5.2 Overview of Implementation

To implement single virtual user (client) and remote storage (server) socket program which communication over the network in TCP/IP model is used (Figure 5.2). When system starts single connection between client and server establish and stay as long as client disconnect from the system.

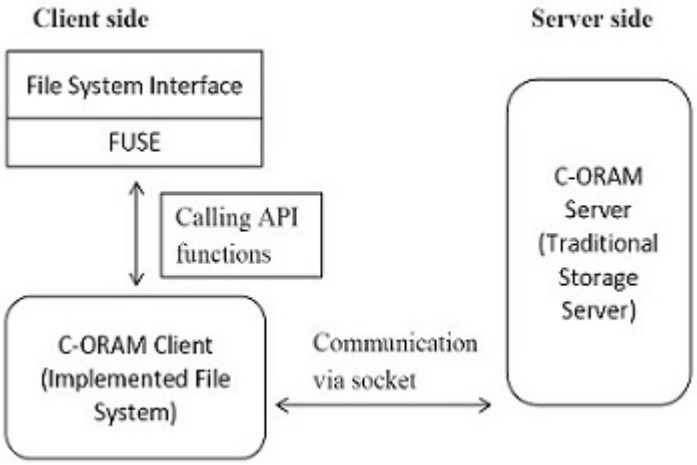


Figure 5.2 Implemented model structure

Following sections talks about the interaction with fuse, data structures used, querying, shuffling and the communication between client and the storage.

5.3 Interaction with the fuse

Modifications to basic functions of FUSE mentioned in section 5.1 are done in the implementation in order to carry out the query process of the C-ORAM scheme. Query process get executed at the *cus_open*() shown in Figure 5.3 which eventually result in shuffling process mentioned in the latter part of this section.

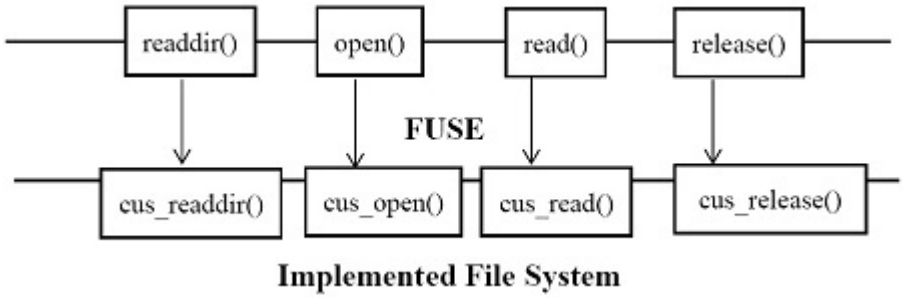


Figure 5.3 Interaction with FUSE by new file system

5.4 Data Structures

According to design, data items fall into buckets generated by hash function and there is a limit of items IDs that a bucket can hold. To facilitate this architecture layers implemented as linked list of buckets nodes (*Bucket*) and for each layer starting bucket and number of buckets stored at a separate structure called Layer.

```
typedef struct Bucket
{
int index
int item* // array to hold itemIDst
int item_count
struct Bucket *next
}bucket

typedef struct Layer
{
bucket *st
int num_bucket
}layer
```

Figure 5.4 Data Structures

In order to find particular data ID falls into which bucket MD5 hash function is used as shown in Figure 5.5. The encryption and decryption operations used in the implementation, all are integer operations which are described by the system design in the chapter 3.

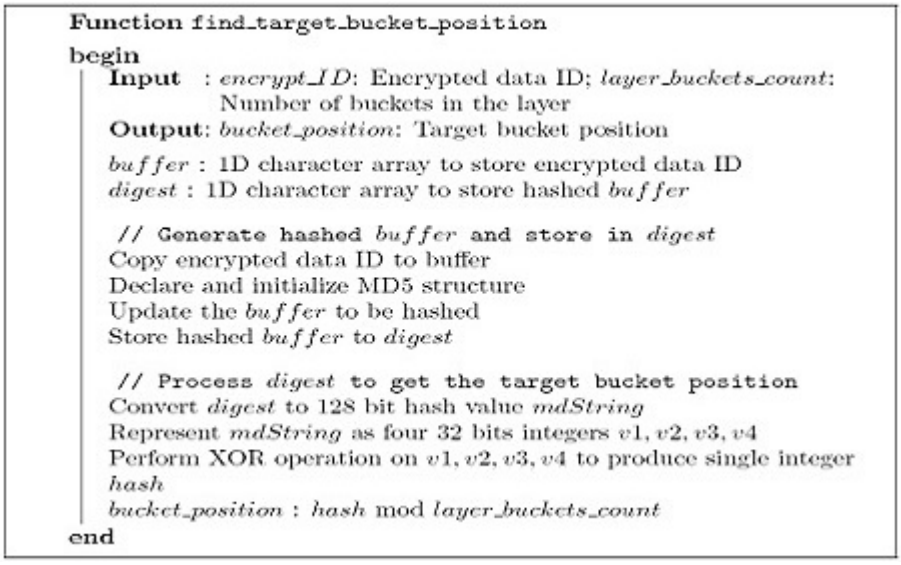


Figure 5.5 Function: find_target_bucket_position

5.5 Query

In the implementation query function is called at fuse open file command. Then from each layer buckets are queried. If the condition for shuffling is triggered at data uploading then shuffling is done which include downloading data from layers, re-encryption, oblivious sort and upload back to targeted layer. All these operations carried out via the socket established at the beginning of fuse execution.

Query function consists of downloading all buckets from *layer*0 and downloading two bucket from each non empty subsequent layers which generated by our hash function associated with each layer. *data_id* takes as the input to the query function. To download all the items in particular layer, first ids stored at that layer need to be find. For that *down_layer* function is called by passing starting bucket of that layer. This function stores all ids (which are references to actual files) to an array and return to query function. Then query function will query one by one id from remote storage.

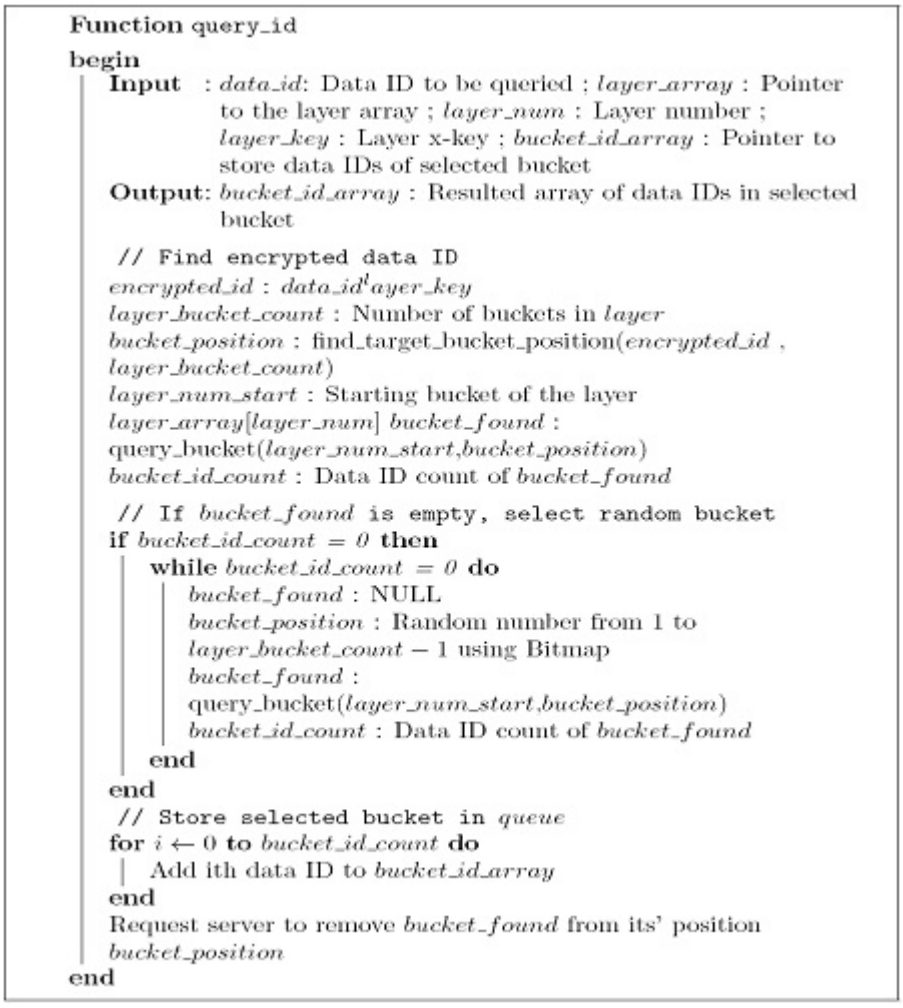
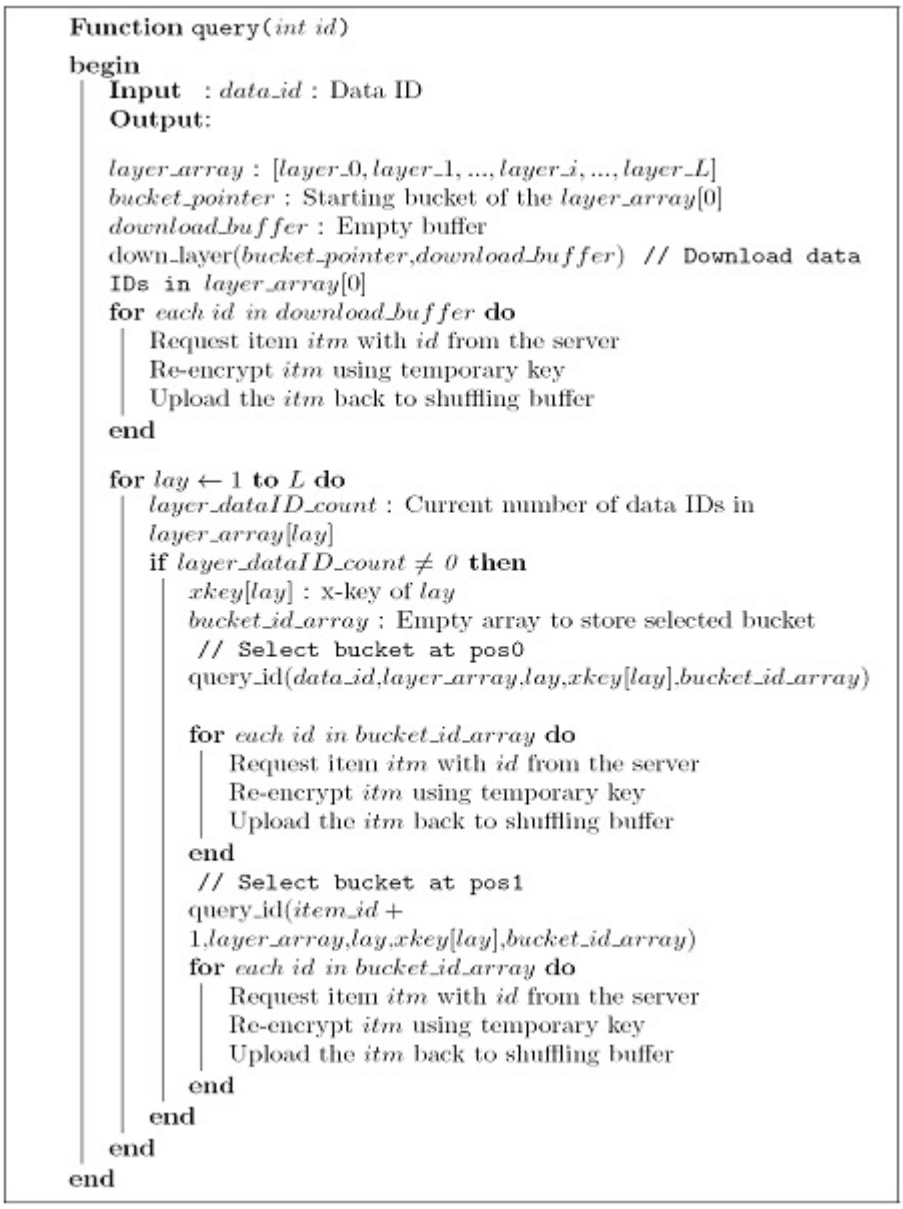


Figure 5.6 Function : query_id

To query all the items in one of the two positions generated by hash function *query_id* function is used. If particular bucket selected in a layer is empty then random bucket queried from that layer. This function takes *xkey* associated with layer in addition to id in order to compute bucket position by calling *find_target_bucket_position*. This also returns all ids as array to query function which eventually gets queried from remote storage. After querying ids on bucket, that bucket needs to be emptied.



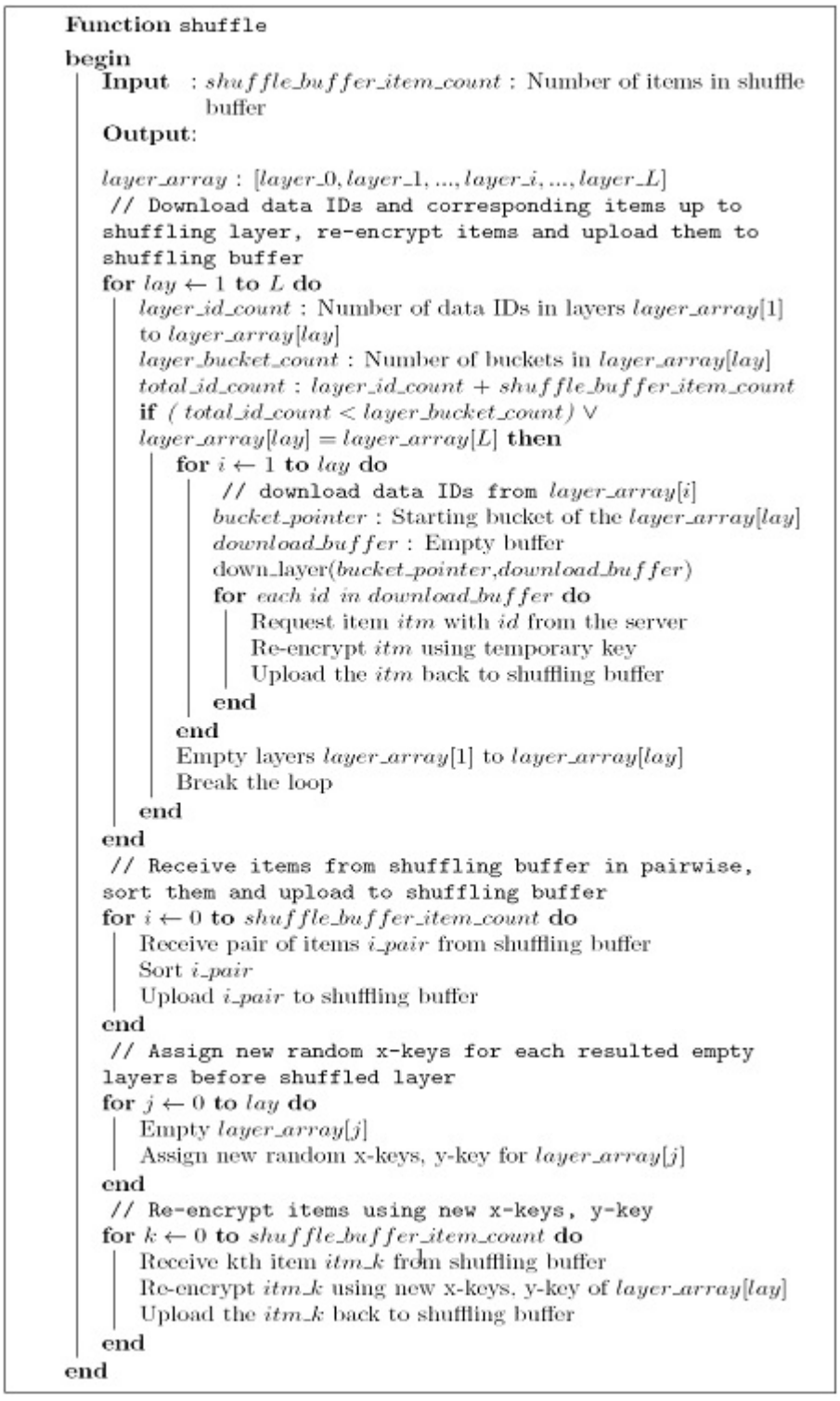
```
Function query(int id)
begin
    Input  : data_id : Data ID
    Output:

    layer_array : [layer_0, layer_1, ..., layer_i, ..., layer_L]
    bucket_pointer : Starting bucket of the layer_array[0]
    download_buffer : Empty buffer
    down_layer(bucket_pointer,download_buffer)  // Download data
    IDs in layer_array[0]
    for each id in download_buffer do
        Request item itm with id from the server
        Re-encrypt itm using temporary key
        Upload the itm back to shuffling buffer
    end

    for lay ← 1 to L do
        layer_dataID_count : Current number of data IDs in
        layer_array[lay]
        if layer_dataID_count ≠ 0 then
            xkey[lay] : x-key of lay
            bucket_id_array : Empty array to store selected bucket
             // Select bucket at pos0
            query_id(data_id,layer_array,lay,xkey[lay],bucket_id_array)

            for each id in bucket_id_array do
                Request item itm with id from the server
                Re-encrypt itm using temporary key
                Upload the itm back to shuffling buffer
            end
             // Select bucket at pos1
            query_id(item_id +
            1,layer_array,lay,xkey[lay],bucket_id_array)
            for each id in bucket_id_array do
                Request item itm with id from the server
                Re-encrypt itm using temporary key
                Upload the itm back to shuffling buffer
            end
        end
    end
end
```

Figure 5.7　Function : query

5.6 Shuffling

Shuffling consist of several steps as shown in shuffling Algorithm1. At the time of data uploading, if total of items in shuffling buffer and $layer0$ exceeds number of buckets in $layer0$ shuffling is triggered. It is done by keeping track of the number of element get queried at query phase and compare it with number of buckets at $layer0$ for given N.

If shuffling is needed, next step is finding to which layer shuffled data can be inserted. For that each layers item count is taken and check whether it can hold all items from layers above that (including the items in shuffling buffer). After that downloading and uploading take place via the socket connection established at the beginning. During the process re-encryption and sorting take place in order to make shuffling oblivious to storage server.



```
Function shuffle
begin
    Input  : shuffle_buffer_item_count : Number of items in shuffle
             buffer
    Output:

    layer_array : [layer_0, layer_1, ..., layer_i, ..., layer_L]
     // Download data IDs and corresponding items up to
    shuffling layer, re-encrypt items and upload them to
    shuffling buffer
    for lay ← 1 to L do
        layer_id_count : Number of data IDs in layers layer_array[1]
        to layer_array[lay]
        layer_bucket_count : Number of buckets in layer_array[lay]
        total_id_count : layer_id_count + shuffle_buffer_item_count
        if ( total_id_count < layer_bucket_count) ∨
        layer_array[lay] = layer_array[L] then
            for i ← 1 to lay do
                 // download data IDs from layer_array[i]
                bucket_pointer : Starting bucket of the layer_array[lay]
                download_buffer : Empty buffer
                down_layer(bucket_pointer,download_buffer)
                for each id in download_buffer do
                    Request item itm with id from the server
                    Re-encrypt itm using temporary key
                    Upload the itm back to shuffling buffer
                end
            end
            Empty layers layer_array[1] to layer_array[lay]
            Break the loop
        end
    end
     // Receive items from shuffling buffer in pairwise,
    sort them and upload to shuffling buffer
    for i ← 0 to shuffle_buffer_item_count do
        Receive pair of items i_pair from shuffling buffer
        Sort i_pair
        Upload i_pair to shuffling buffer
    end
     // Assign new random x-keys for each resulted empty
    layers before shuffled layer
    for j ← 0 to lay do
        Empty layer_array[j]
        Assign new random x-keys, y-key for layer_array[j]
    end
     // Re-encrypt items using new x-keys, y-key
    for k ← 0 to shuffle_buffer_item_count do
        Receive kth item itm_k from shuffling buffer
        Re-encrypt itm_k using new x-keys, y-key of layer_array[lay]
        Upload the itm_k back to shuffling buffer
    end
end
```

Figure 5.8 Function : shuffle

5.7 Sort

For oblivious sort considering limited client space available batchers sort is used. Figure 5.9 shows how pair wise compare and exchange works in batchers sort for 8 elements.

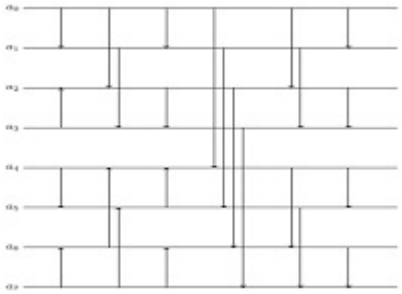


Figure 5.9 Pair-wise compare and exchange of eight elements

By comparing odd and even positions in the list that need to be sorted, batchers sort makes half of elements in list to be in ascending order while rest of the elements in descending order as shown in example in Figure 5.10.

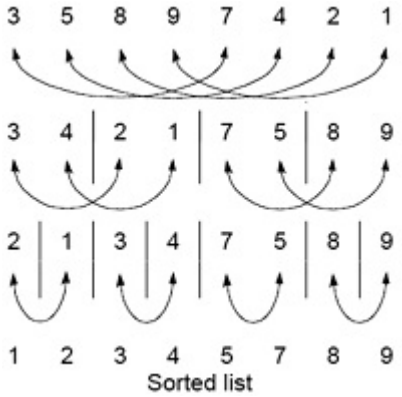
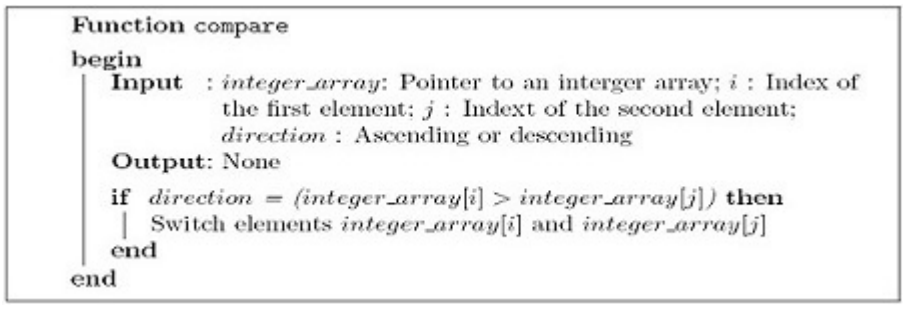


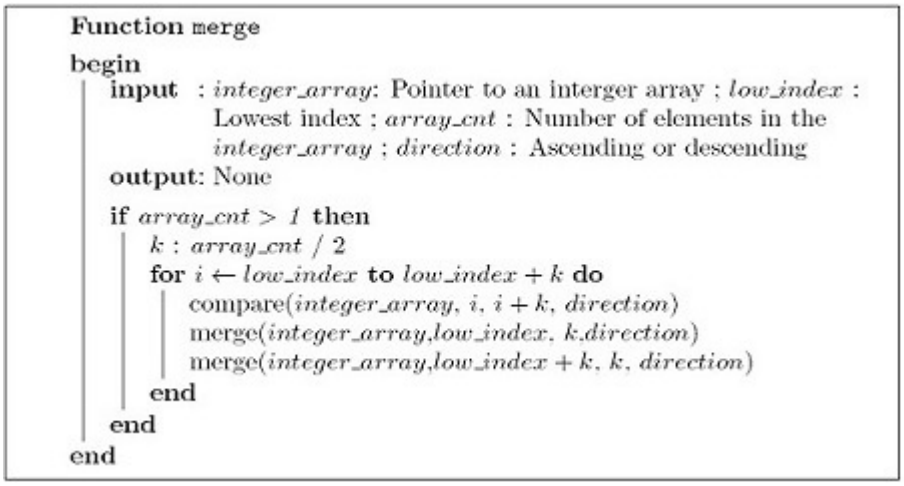
Figure 5.10 Intermediate step in Batchers sort of eight elements

Then compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right recursively to get the sorted list. Implementation of batchers sort algorithm is shown in following Figures 5.11, 5.12 and 5.13 as a whole. From the algorithm comparison part is carried by the client side.



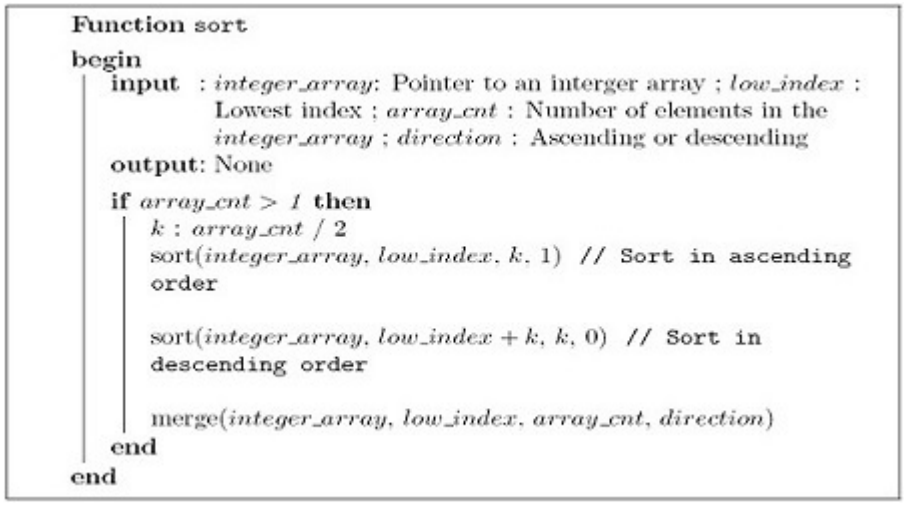
```
Function compare
begin
    Input  : integer_array: Pointer to an interger array; i : Index of
             the first element; j : Indext of the second element;
             direction : Ascending or descending
    Output: None
    if direction = (integer_array[i] > integer_array[j]) then
        Switch elements integer_array[i] and integer_array[j]
    end
end
```

Figure 5.11 Function : compare



```
Function merge
begin
    input  : integer_array: Pointer to an interger array ; low_index :
             Lowest index ; array_cnt : Number of elements in the
             integer_array ; direction : Ascending or descending
    output: None
    if array_cnt > 1 then
        k : array_cnt / 2
        for i ← low_index to low_index + k do
            compare(integer_array, i, i + k, direction)
            merge(integer_array,low_index, k,direction)
            merge(integer_array,low_index + k, k, direction)
        end
    end
end
```

Figure 5.12 Function : merge



```
Function sort
begin
    input  : integer_array: Pointer to an interger array ; low_index :
             Lowest index ; array_cnt : Number of elements in the
             integer_array ; direction : Ascending or descending
    output: None
    if array_cnt > 1 then
        k : array_cnt / 2
        sort(integer_array, low_index, k, 1)  // Sort in ascending
        order

        sort(integer_array, low_index + k, k, 0)  // Sort in
        descending order

        merge(integer_array, low_index, array_cnt, direction)
    end
end
```

Figure 5.13 Function : sort

5.8 Communication

Communication process between client and server take place is shown Figure 5.14. Here read() and write() operations are receiving and sending out the items between client and server. In the implementation single thread is used assuming communication is serialized which mean at a given time only one of sending or receiving operations will happen. The connection established by the client with the server eventually ends when client close the connection.

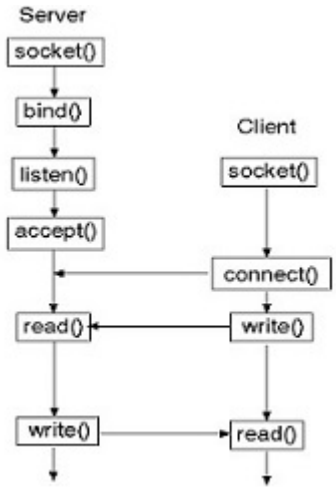


Figure 5.14 Communication process between client and server

Client communication with server work as follows:

- Create a socket with the *socket*() system call. The call to the function *socket*() creates socket inside the kernel and returns an integer known as socket descriptor.
- Connect the socket to the address of the server using the *connect*() system call. Connect operation waiting for successful connection establishment. After successful connection, the application is allowed to start sending data. The address, which is taken as an argument, contains the remote participants address.
- Send and receive data via the connection. The server sends the data on client's socket through client's socket descriptor which read by the client by its own socket descriptor.

Server communication with client work as follows:

- Create a socket with the *socket*() system call.
- Bind the socket to an address using the *bind*() system call. This operation assigns a socket to an address. binds the newly created socket to the specified address. This is the network address of the server. Address includes IP address, TCP port number of the server. After *bind*(), socket can accept connections to other hosts.
- Listen for connections with the *listen*() system call. This operation prepares the socket for incoming connections. In *listen*(), it is define that how many number of pending connections that can be queued up at any one time on the specified socket.
- Accept a connection with the *accept*() system call. This works as passive open operation because it is a blocking the process and wait until a remote participant has established.
- Send and receive data via the connection.

CHAPTER 6. EVALUATION

6.1 Setup

For the evaluation two virtual machines created in a Dell laptop was used. One virtual machine configured as server by allocating more resources (memory, storage etc) while the other virtual machine setup as the client with less resources. The configuration settings are shown in Table 6.1.

Table 6.1 Experiment testbed

Settings	**Virtual machine1(client)**	**Virtual machine2 (server)**
Operating System	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS
CPU	1.8 GHz Intel i5 with 1 processor	1.8 GHz Intel i5 with 1 processor
Storage	20 GB	80 GB
Memory	1GB	4GB
Fuse Version	2.9.3	-

6.2 Access Pattern

In order to prove server cannot distinguish between two quires, results were obtained for querying same and random item for multiple times. The total number of items are set to N=256. Both same item and random item were queried for 200,500 and 1000 times respectively. Figure 6.1, 6.2 and 6.3 show how many times each bucket gets accessed in particular layer (bucket access frequency). The results shows access patterns for querying an item are pretty much the same.

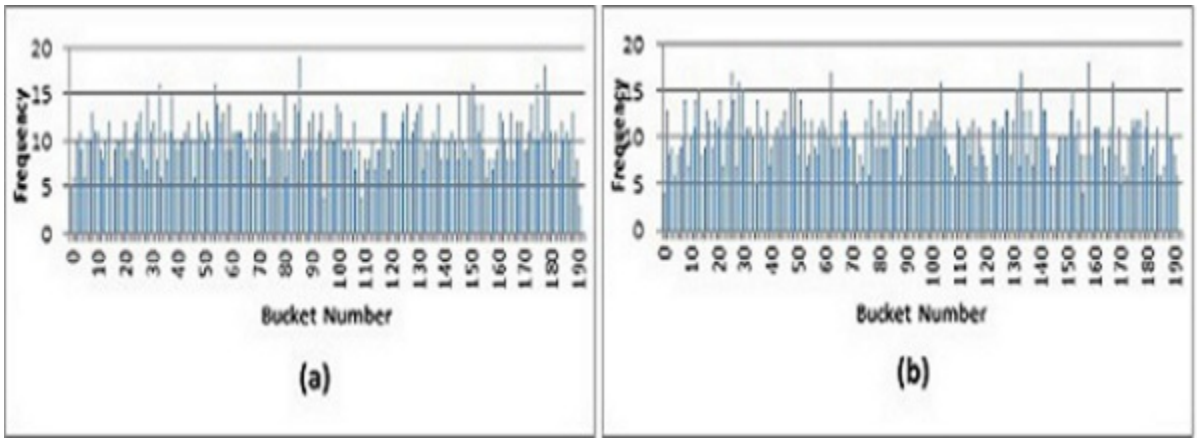


Figure 6.1 Bucket access frequency for querying in *Layer*3 - 1000 times (a) Same item (b) Random item

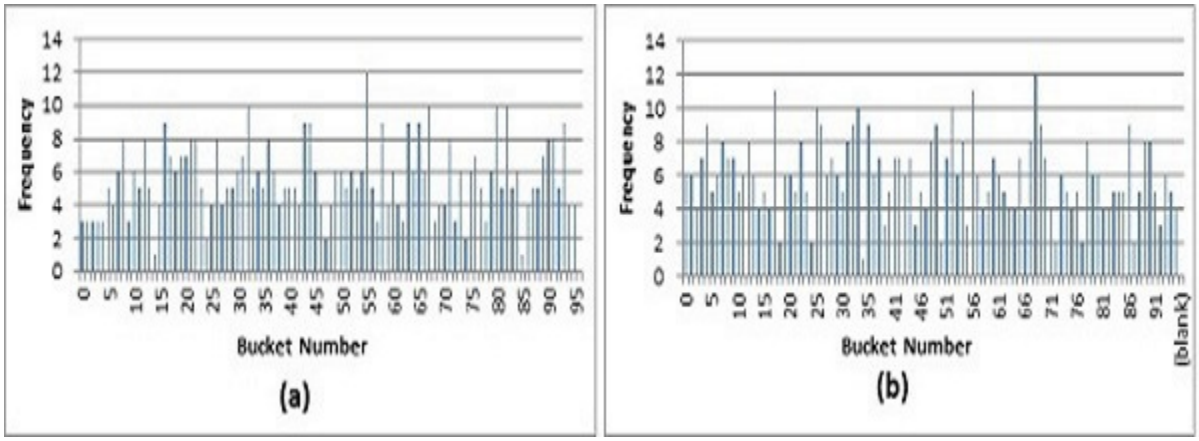


Figure 6.2 Bucket access frequency for querying in *Layer*2 - 500 times (a) Same item (b) Random item

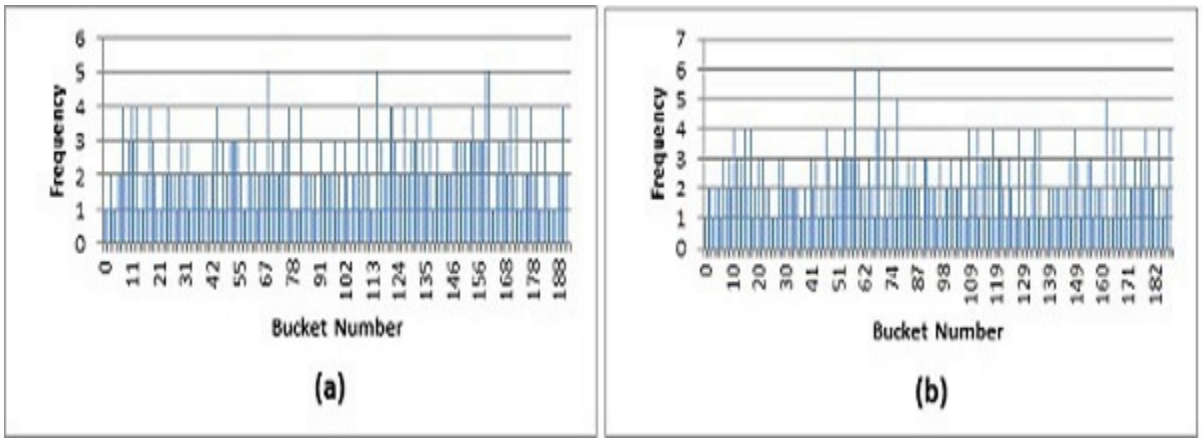


Figure 6.3 Bucket access frequency for querying in *Layer*3 - 200 times (a) Same item (b) Random item

Table 6.2 and 6.3 shows how many times each layer gets queried for different number of items (N). This also shows that number of times each layer gets queried is nearly the same for both the same item and random item cases.

Table 6.2 Access time for each layer for same item query

	query time	**layers**			
		2	3	4	5
N=256	200	107	200		
	500	269	500		
	1000	554	1000		
N=512	200	107	100	200	
	500	280	274	500	
	1000	564	555	1000	
N=1024	200	116	94	200	
	500	293	221	500	
	1000	575	427	1000	
N=2048	200	116	87	53	200
	500	287	225	254	500
	1000	590	478	472	1000

Table 6.3 Access time for each layer for random item query

	query time	**layers**			
		2	3	4	5
N=256	200	110	200		
	500	268	500		
	1000	555	1000		
N=512	200	106	99	200	
	500	281	274	500	
	1000	564	556	1000	
N=1024	200	118	93	200	
	500	291	218	500	
	1000	574	427	1000	
N=2048	200	115	86	52	200
	500	289	223	252	500
	1000	589	476	472	1000

6.3 Overhead

The communication overhead includes both query and shuffling overheads. Results for these are shown in Table 6.4 and 6.5. According to the results, measured communication overhead of implementation is consistent with theoretical results.

Table 6.4 Query overhead

N	**Theory**	**Experiment(100 times)**	**Average items**
128	19.65	2284	22.84
256	24	2700	27
512	28.53	3231	32.31
1024	33.22	3724	37.24
2048	38.05	4325	43.25

Table 6.5 Shuffle overhead

N	**Theory**	**Experiment(shuffle times)**	**Average items**
128	962.92	14	1024
256	1536	27	1475.56
512	2310.88	55	1710.74
1024	3321.93	109	3327.10
2048	4604.5	214	3465.60

CHAPTER 7. CONCLUSION

Cloud base storage popularity has increased in recent past because of its cost effectiveness compared to traditional storage systems. In this work we implemented a multi user customized Oblivious RAM (C-ORAM) to efficiently protect users' data security and privacy in cloud storage. Data encryption does not fully guarantee the users privacy in cloud environment because the patterns of data access could leak considerable information about that stored data.

Oblivious RAM techniques suggested by previous studies as a method of protecting users access pattern from a curios server. It uses continuous data re-encryption and shuffling in order to conceal users access pattern. Most of the suggested ORAM schemes assume a single user to cloud storage. However, in practice it not the case. In C-ORAM we assume mutually trusted multi user scenario instead of traditional single user.

Evaluation of the system in terms of access pattern preservation and performance shows that C-ORAM can fully protect users access pattern. C-ORAM needs constant user-side storage and the overhead results for query and shuffling in the implementation has a closer match to the theoretical results.

BIBLIOGRAPHY

[1] Pinkas, B. and Reinman, T. (2010). Oblivious ram revisited. *Advances in Cryptology (CRYPTO'10).*

[2] Goodrich, M. T., and Mitzenmacher, M. (2010). Mapreduce parallel cuckoo hashing and oblivious ram simulations. *In Proc. CoRR10.*

[3] Goodrich, M. T., and Mitzenmacher, M. (2011). Privacy-preserving access of outsourced data via oblivious ram simulation. *In Proc. ICALP11.*

[4] Goodrich, M. T., Mitzenmacher, M.,Ohrimenko, O., and Tamassia, R. (2011). Oblivious ram simulation with efficient worst-case access overhead. *In Proc. CCSW11.*

[5] Goodrich, M. T., Mitzenmacher, M.,Ohrimenko, O., and Tamassia, R. (2012). Oblivious ram simulation with efficient worst-case access overhead. *In Proc. ACM-SIAM Symp. On Discrete Algorithms (SODA12).*

[6] Kushilevitz, E., Lu, S., and Ostrovsky, R. (2012). On the (in)security of hash-based oblivious ram and a new balancing scheme. *In Proc. SODA12.*

[7] Zhang Jinsheng, Zhang Wensheng, Daji Qiao(2014). A Multi-user Oblivious RAM for Outsourced Data.

[8] Goldreich, O. (1987). Towards a theory of software protection and simulation by oblivious rams. *In Proc. STOC'87.*

[9] Goldreich, O. and Ostrovsky, R. (1996). Software protection and simulation on oblivious rams. *Journal of J. ACM.*

[10] Ostrovsky, R. (1990). Efficient computation on oblivious rams. *In Proc. STOC'90.*

[11] Beame, P. and Machmouchi, W. (2010). Making rams oblivious requires superlogarithmic overhead. *In Proc. Electronic Colloquium on Computational Complexity (ECCC).*

[12] Stefanov, E., Shi, E., and Song, D (2012). Towards Practical Oblivious RAM. *In Proc. Network and Distributed System Security Symposium, NDSS'12.*

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

[14] Stefanov, E., Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S (2013). Path ORAM: An extremely simple oblivious ram protocol. *In Proc. ACM SIGSAC Conference on Computer and Communications Security, CCS '13.*

[15] Stefanov, E., and Shi, E., (2013). ObliviStore: High Performance Oblivious Cloud Storage. *In Proc. IEEE Symposium on Security and Privacy.*

[16] Williams, P., Sion, R., and Tomescu, A. (2012). PrivateFS: a parallel oblivious fille system. *In Proc. ACM conference on Computer and Communications Security, CCS '12.*

[17] Williams, P., and Sion, R. (2012). Single round access privacy on outsourced storage. *In Proc. ACM conference on Computer and Communications Security, CCS '12.*

[18] Williams, P., Sion, R., and Carbunar, B. (2008). Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. *In Proc. CCS08.*

[19] Goodrich, M. T., Mitzenmacher, M. (2011). Privacy-preserving access of outsourced data via oblivious RAM simulation. *In Proc. Int. Colloq. on Automata, Languages and Programming (ICALP).*

[20] Kushilevitz, E., Lu, S., and Ostrovsky, R. (2012). On the (in)security of hash-based oblivious RAM and a new balancing scheme. *In Proc. ACM-SIAM Symp. On Discrete Algorithms (SODA'12).*

[21] Freier, A. O., Karlton, P., and Kocher, P. C. (2011). The secure sockets layer (SSL) protocol version 3.0. In RFC 6101.

[22] Szeredi, M. (2001). Path of a filesystem call. Online; accessed 12-July-2014.

[23] Kuenning, G. (2010), Fuse documentation. Online; accessed 12-July-2014.

[24] Pfeiffer, J. J. (2012), Fuse tutorial. Online; accessed 12-July-2014.