2014

Implementing a segmentation-based oblivious RAM

Wen Zhao
Iowa State University

Implementing a segmentation-based oblivious RAM

by

Wen Zhao

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

Shashi K. Gadia

Lu Ruan

Iowa State University

Ames, Iowa

2014

DEDICATION

I would like to dedicate this thesis to my girl friend Ting without whose support I would not have been able to complete this work. I would also like to thank my family and friends 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, Dr. Shashi K. Gadia and Dr. Lu Ruan, for their enhancement and contributions to this work.

ABSTRACT

As cloud-based storage becomes more popular, an increasing amount of data has been outsourced to cloud storage. For many businesses, individuals, or even governments, data privacy has become an important concern. The confidentiality of data can be protected by encryption, but this alone may not suffice for ensuring data privacy. Data access patterns can leak a considerable amount of information about the data as well. Even though existing Oblivious RAM (ORAM) constructions are provable solutions to this issue, their performance in practice has been impeded by the high communication and storage overheads incurred. In this thesis, we implemented a Segmentation-based ORAM (S-ORAM) to reduce communication and storage overhead. Experiments have shown that S-ORAM can effectively protect user's privacy as well as achieve low communication and storage overhead.

CHAPTER 1. INTRODUCTION

Cloud storage, such as Dropbox [Dropbox (2007)] and Amazon S3 [Amazon (2006)], are widely used nowadays. It is a model of networked enterprise storage that data is stored in virtual pools of storage, which are generally hosted by third parties. Compared to traditional private storage, cloud storage has two main advantages. First, the price is much cheaper. Users (individuals, institutions, companies or government agencies) only need to pay for the storage they actually use, typically an average of consumption during a month. Second, it is much easier to maintain. Users now can write software or scripts to configure hundreds of storage servers automatically instead of configuring them one by one.

More and more companies and even government agencies choose public cloud for storing data. However, as cloud providers have complete control of users' outsourced data, the security and privacy problems arise and are becoming a big concern.

Encrypting data has been a common practice for data protection. But the sequence of storage locations accessed by the client (i.e., access pattern) could still leak a significant amount of sensitive information, if statistical inference is applied. For example, Islam et al. [Islam et al. (2012)] demonstrated that an adversary can infer as much as 80% of the search queries by observing accesses to an encrypted email repository.

To address this issue, the Oblivious RAM (ORAM) algorithm, first proposed by Goldreich and Ostrovsky [Goldreich and Ostrovsky (1996)], has attracted a lot of attention. The algorithm conceals a client's access pattern to the remote storage by continuously shuffling and re-encrypting data as they are accessed. Even though an adversary can observe the physical storage locations that a client has accessed, the ORAM algorithm ensures that the adversary has negligibly-small probability to learn anything about the client's true access pattern.

Since its proposal, the research community has developed and implemented many ORAM

schemes. Among them, hash-based ORAMs and index-based ORAMs are two main branches. Hash-based ORAMs [Goldreich and Ostrovsky (1996); Goodrich and Mitzenmacher (2010); Goodrich et al. (2012); Goodrich and Mitzenmacher (2011); Goodrich et al. (2011); Kushilevitz et al. (2012); Pinkas and Reinman (2010); Williams and Sion (2008)] utilize hash functions (including ordinary hash functions, cuckoo hash functions, bloom filters, etc.) to determine a data block's storage location when the block is stored or shuffled, and to look up intended data blocks. In comparison, index-based ORAMs [Shi et al. (2011); Stefanov et al. (2013); Stefanov and Shi (2013); Stefanov et al. (2011)] maintain index tables to keep the mapping from data blocks to their storage locations and update them after each query.

ORAM can effectively protect user's privacy in theory. However, most existing ORAM constructions are not practical because of the high communication and/or storage overheads that they may incur. Particularly, hash-based ORAMs require a large extra storage space at the server side in order to deal with hash collisions. Also access pattern privacy usually has to be preserved through frequent data retrievals and complicated data shuffling. Index-based ORAMs are also impractical. Even though they rely on index tables to avoid the above problems in hash-based ORAMs, it is hard to find a way to store the index table in a space-efficient way and to search and update it in a time-efficient way. This limitation has also impeded their applications in practice.

Recently, Zhang et. al. [Jingsheng Zhang (2014)] proposed a novel ORAM scheme, called segmentation-based oblivious RAM (S-ORAM), aiming to bring theoretical ORAM constructions more practical. The design was motivated by the observation that a large-scale storage system usually stores data in blocks and such a block typically has a large size [Stefanov et al. (2011)], but most existing ORAM constructions treat data blocks as atomic units for query and shuffling, and do not factor block size into their designs. In order to make a better use of the large block size, two segment-based techniques have been introduced, namely, piece-wise shuffling and segment-based query, which can efficiently improve the efficiency in query and data shuffling. Data can be perturbed across a larger range of blocks in a limited user-side storage with piece-wise shuffling. In this way, the shuffling efficiency can be improved, and the improvement gets more significant as the block size increases. With segment-based query,

S-ORAM organizes the data storage at the server side as a hierarchy of single-segment and multi-segment layers, and each layer are with an encrypted index block. With these two techniques at the core, together with a few supplementary algorithms for distributing blocks to segments, S-ORAM can accomplish efficient query with only O(log N) communication overhead and a constant user-side storage, while existing ORAM constructions have to use a larger user-side storage to achieve the same level of communication efficiency in query.

The Balanced ORAM (B-ORAM) [Kushilevitz et al. (2012)] and the Path ORAM (P-ORAM) [Stefanov et al. (2013)] are the state-of-the-art hash and index based ORAMs respectively, in terms of both practical and theoretical standards. In terms of communication and storage overheads, S-ORAM outperforms both of them. When they have the same constant-size user-side storage, S-ORAM's communication overhead is 12 to 23 times less than B-ORAM, 60% to 72% less bandwidth than P-ORAM, particularly under practical settings. Meanwhile, S-ORAM consumes 80% less server-side storage than P-ORAM.

This thesis work has implemented the S-ORAM design and delivered a prototype of distributed file system that aims to preserve access pattern for users of the file system. Experiments based on the prototype have also been conducted. The results show that our implemented system can fully protect user's access pattern and more importantly, significantly reduce communication and storage overhead compared to the other ORAM schemes. Particularly, the difference between querying the same block and querying randomly picked block for many times is negligible. The user-side storage remains at a small constant value and the average communication overhead is small.

The rest of the thesis is organized as follows: Chapter 2 presents related works. Chapter 3 introduces problem statement. Chapter 4 describes design of S-ORAM. Chapter 5 details implementation of S-ORAM. The evaluation results are reported in Chapter 6. Finally, Chapter 7 concludes this thesis.

CHAPTER 2. RELATED WORK

2.1 Theoretical ORAM

Oblivious RAM was first proposed by Goldreich and Ostrovsky [Goldreich (1987); Goldreich and Ostrovsky (1996); Ostrovsky (1990)] for the purpose of protecting software from piracy and efficient simulation of programs on oblivious RAMs. Two solutions were elaborated. The first solution, called Square-Root, has high complexity so only served as a preliminary step towards the second solution, the Hierarchical solution. The Hierarchical solution makes any program oblivious by replacing N stored items with $O(NlogN)$ items and replacing a direct access to an item in RAM with $O(log^3N)$ accesses to RAM.

The high-level description of Ostrovskys hierarchical solution can be stated as follows: there is a sequence of buffers whose sizes grow at a geometric rate, and smaller buffers are reshuffled into larger ones as they fill up. In the original work [Ostrovsky (1990)], the buffers were standard hash-tables with sufficiently large buckets, and a technique known as oblivious shuffling was employed for reshuffling. Then, a slightly different and somewhat simpler reshuffling method was proposed [Goldreich and Ostrovsky (1996)].

The efficiency of Oblivious RAM is measured by three main parameters: the amount of local (client) storage, the amount of remote (server) storage, and the (amortized) overhead of reading or writing an element. [Goldreich (1987)] used sub-linear local storage, while [Ostrovsky (1990); Goldreich and Ostrovsky (1996)] both used a constant amount of local storage. Ostrovskys hierarchical solution used $O(nlogn)$ remote storage, and offered two different ways to perform oblivious shuffling that led to either $O(log^4n)$ access overhead with a small hidden constant, or $O(log^3n)$ access overhead with a large hidden constant.

Subsequent works improved upon Ostrovskys hierarchical solution. They can be roughly

classified into two categories based on the data lookup technique used: hash-based ORAMs and index-based ORAMs.

2.2 Hash-based ORAMs

Hash-based ORAMs [Goldreich and Ostrovsky (1996); Goodrich and Mitzenmacher (2010); Goodrich et al. (2012); Goodrich and Mitzenmacher (2011); Goodrich et al. (2011); Kushilevitz et al. (2012); Pinkas and Reinman (2010); Williams and Sion (2008)] use hash functions for data lookup, and facilities such as buckets and stashes are required to deal with hash collisions. To the best of our knowledge, among these ORAMs, the Balanced ORAM (B-ORAM) [Kushilevitz et al. (2012)] achieves the best asymptotical communication efficiency.

In B-ORAM, the server-side storage is organized as a hybrid hierarchy with a total of $\frac{\log N}{\log \log N}$ layers, where each layer consists of $\log N$ equal-size sub-layers. For the top $O(\log \log N)$ layers, the bucket-hash structure [Goldreich and Ostrovsky (1996)] is deployed and the remaining layers are cuckoo-hash structures with a shared stash [Goodrich and Mitzenmacher (2010)]. Since each layer is extended to multiple sub-layers, the shuffling frequency is reduced while the query overhead is increased; a balance is struck between the query and shuffling overheads. The randomized shell-sort [Goodrich (2010)] is selected as the underlying oblivious sorting algorithm for the shuffling process. In theory, the amortized communication overhead of B-ORAM is $O\left(\frac{\log^2 N}{\log \log N}\right)$ blocks per query. In practice, however, the overhead is on the magnitude of $\log^3 N$ due to a large constant hidden in the above big-O notation; particularly, querying one data block may require the user to access at least 1000 data blocks, which may not be acceptable in many practical applications.

2.3 Index-based ORAMs

Index-based ORAMs [Shi et al. (2011); Stefanov et al. (2013); Stefanov and Shi (2013); Stefanov et al. (2011)] use an index structure for data lookup. Therefore, they requires that the user-side storage stores the index table, which is feasible only if the number of data blocks is not quite large. Otherwise it will greatly increase user-side storage. When the user-side

storage cannot afford to store the index, it can outsource the index table to the server in a way similar to storing real data blocks at the cost of increased communication overhead. And the index outsourcing can be done recursively. The Path ORAM (P-ORAM) [Stefanov et al. (2013)] outperforms all other schemes in this category.

In P-ORAM, the server-side storage is organized as a binary tree in which each node contains a constant-size bucket for storing data blocks. Initially, data blocks are randomly stored at leaf nodes, and an index structure is maintained to record the mapping between the IDs of data blocks and the IDs of the leaf nodes storing the blocks. Based on the index, a data query process retrieves all blocks on the path that contains the query target block and then moves the target block to the root node. In addition, a background eviction process is performed after each query process, to gradually evict blocks from the root node to nodes of lower-height so as to avoid or reduce node overflowing. The index can also be outsourced to the server and stored in a similar binary tree. Besides, to keep bucket size constant at each node, a user-side storage whose size is a logarithmic function of the number of data blocks is needed to form a stash. P-ORAM achieves a communication overhead of $O\left(\frac{\log^2 N}{\log(Z/\log N)}\right) \cdot \omega(1)$ blocks per query, where Z is data block size and $\omega(1)$ is a security parameter. Though the communication overhead is considered to be acceptable in practice [Stefanov et al. (2013)], the overhead of server-side storage, which is about $32N$ blocks, may pose as a big cost to the user.

CHAPTER 3. PROBLEM STATEMENT

3.1 System Model

We consider a system composed of a user and a remote storage server, which is similar to existing ORAM constructions [Goldreich and Ostrovsky (1996); Goodrich and Mitzenmacher (2010); Goodrich et al. (2012); Goodrich and Mitzenmacher (2011); Goodrich et al. (2011); Kushilevitz et al. (2012); Pinkas and Reinman (2010); Williams and Sion (2008)]. Initially the user exports a large amount of data to store at the server, and wishes to hide from the server both the content of data and the pattern of his/her accesses to the data. The content of data can be secured with encryption techniques. But new algorithms are needed to protect user's access pattern.

Here we assume that data is stored and accessed in the unit of blocks to simplify the problem. This assumption is practical because file systems usually store data in blocks. We use a unique block id to identify each block. The typical size of a block ranges from 32 KB to 256 KB [Stefanov et al. (2011)]. Let N denote the total number of data blocks user outsourced to the server.

Each request from the user can be one of the following types:

- read a data block D of unique ID i from the storage, denoted as a 3-tuple $(read, i, D)$; or
- write a data block D of unique ID i to the storage, denoted as a 3-tuple $(write, i, D)$.

From the server side, a user's access pattern can be one of the following types:

- retrieve (read) a data block D from a location l at the remote storage, denoted as a 3-tuple $(read, l, D)$; or

- upload (write) a data block D to a location l at the remote storage, denoted as a 3-tuple $(write, l, D)$.

3.2 Threat Model

In the threat model we make three assumptions. First the user is trusted. Operations made by the user before transmitting are considered to be secure. Second network connection between the user and the server, such as data transmission, is assumed to be secure. Techniques such as SSL [Freier et al. (2011)] can effectively achieve this. Third the server is assumed to behave honestly according to the user's request but to be curious. The server may try to figure out the user's access pattern. And our goal is to prevent this happen.

Intuitively, an ORAM system is considered secure if the server learns nothing about the user's data access pattern. The definition of the security of S-ORAM is inherited from the standard security definition of ORAMs [Goldreich and Ostrovsky (1996); Stefanov et al. (2013, 2011)]. It is defined as follows [Jingsheng Zhang (2014)]:

Definition Let $\vec{x} = \langle\ (op_1, i_1, D_1), (op_2, i_2, D_2), \cdots \rangle$ denote a private sequence of the user's intended data requests, where each op is either a $read$ or $write$ operation. Let $A(\vec{x}) = \langle\ (op'_1, l_1, D'_1), (op'_2, l_2, D'_2), \cdots \rangle$ denote the sequence of the user's accesses to the remote storage (observed by the server), in order to accomplish the user's intended data requests. An ORAM system is said to be secure if (i) for any two equal-length private sequences $\vec{x}$ and $\vec{y}$ of the intended data requests, their corresponding observable access sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable; and (ii) the probability that the ORAM system fails to operate is negligibly small, i.e., $O(N^{-\log N})$.

CHAPTER 4. SYSTEM DESIGN

4.1 Overview

The main contribution of S-ORAM is that it makes better use of the large block size to improve the efficiency. This is because a large-scale storage system usually stores data in blocks and such a block typically has a large size. But most existing ORAM constructions treat data blocks as atomic units for query and shuffling, and do not factor block size into their designs, which may increase communication and storage overhead unnecessarily.

Based on the observation, S-ORAM does not treat data blocks as atomic units. Instead, it introduces the concept of *piece* and *segment* to make better use of large-size blocks.

Piece is the unit of data shuffling in S-ORAM while most existing ORAM use block as shuffling unit. A block is divided into *pieces*. The size of a *piece* is much smaller than a block. With the same size of user-side storage, shuffling data in pieces is much more efficient than shuffling in blocks.

Segment is introduced to improve query efficiency. In S-ORAM, the server-side storage is organized in hierarchy with single-segment and multi-segment layers. Each segment contains of an encrypted index block and several encrypted data blocks. The index block maintains the mapping between data block IDs and their locations within the segment. Thus, for each query in a segment, only the index block and the intended block need to be accessed. Multi-segments layers are introduced because one single index block may not be able to store all the mapping in that layer.

4.2 Storage Organization

4.2.1 Data Block Format

In S-ORAM, data are stored in data blocks, which is the unit for user to query. The difference between S-ORAM and other existing ORAMs is that, each block is divided into *pieces*, as shown in Figure 4.1. The size of each *piece* is $z = \log N$ bits long, where N is the total number of user outsourced data blocks. The first piece of a data block contains the ID of it and the remaining pieces store the actual data content.

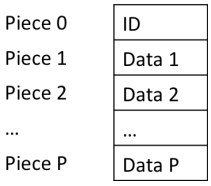

Figure 4.1 Plian-text Block Structure

4.2.2 Data Block Encryption

lain-text data blocks need to be encrypted before exporting to server. In S-ORAM data blocks are encrypted piece by piece with a secret key k, as shown in Equation 4.1 and in Figure 4.2 [Jingsheng Zhang (2014)]:

$$\begin{aligned} &c_{i,0} = E_k(r_i), \text{ where } r_i \text{ is a random number;} \\ &c_{i,1} = E_k(r_i \oplus d_{i,1}); \\ &c_{i,2} = E_k(c_{i,1} \oplus d_{i,2}); \\ &\cdots, \\ &c_{i,P-1} = E_k(c_{i,P-2} \oplus d_{i,P-1}). \end{aligned} \tag{4.1}$$

The encrypted data block (hereafter called data block for brevity) is one more *piece* larger than plain-text data block. The details of encryption method is described in Section 5.5.

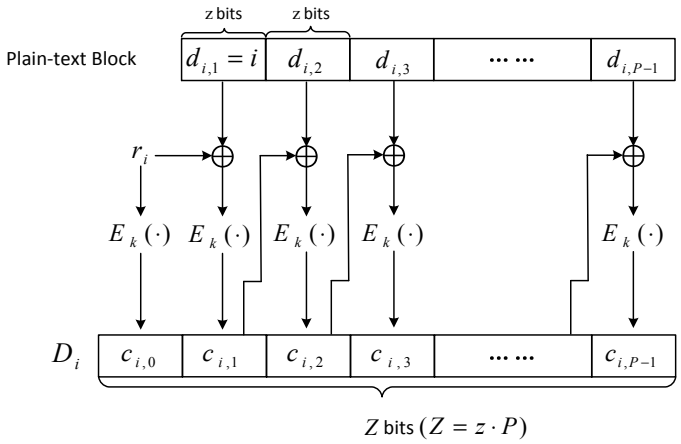

Figure 4.2 Data Block Format in S-ORAM [Jingsheng Zhang (2014)]

4.2.3 User-side Storage

In S-ORAM user-side storage is of constant size. Particularly, it consists of two parts: *permanent storage* and *cache (temporary storage)*. Permanent storage is used to store user's secret information, including three parts:

- *Query Counter.* It keeps track of the number of queries that the user makes. And it triggers data shuffling to achieve obliviousness.
- *Secret Key.* It is used for data block encryption and decryption. The secret key remains the same during the whole procedure.
- *Hash Function.* The user keeps a hash function for each T2-layers. It can map a data block to one of the segments in the layer.

Cache is used for buffering and processing queried data, including encryption, decryption, shuffling and oblivious sorting. In S-ORAM the size of it is fixed to 2 data block size. Compared to cache, the size of permanent storage is much smaller. So when compute user-side storage, permanent storage can be negligible.

4.2.4 Server-side Storage

Data blocks are stored in a hierarchy structure in server as shown in Figure 4.3. The layers are divided into three groups as follows.

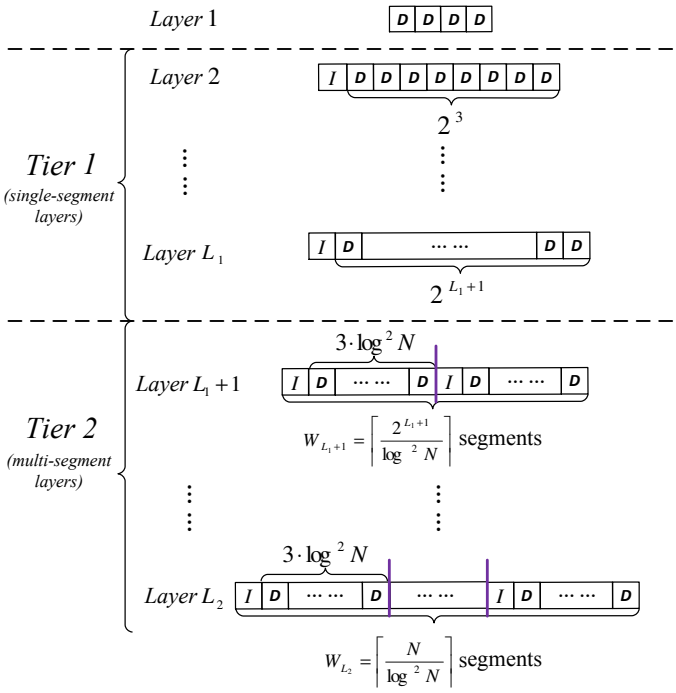

Figure 4.3 Server-side Storage Organization [Jingsheng Zhang (2014)]

(1) *Layer 1.* The top layer is called *layer 1*, which can contain at most 4 data blocks. For each query all blocks in this layer will be downloaded to user-side storage. And after each query they will be uploaded back to server along with one more data block.

(2) *T1 Layers: Single-Segment Layers.* Layers from layer 2 to layer $L_1 = \lfloor 2 \log \log N \rfloor$ belong to T1. L_1 is set to $\lfloor 2 \log \log N \rfloor$ is to let the layers in T2 starts with 2 segments. For each layer in T1, it consists of an encrypted index block I_l and 2^{l+1} data blocks. The index block is of the same size of a data block, and has 2^{l+1} entries. Each entry corresponds to a data block in the segment with three fields: Data ID, Location and Access Bit. Data ID and Location forms a mapping relation from data blocks to its location in that segment. Access Bit is set to be 1 if that position has been accessed before. Otherwise it's 0. At most half of the data blocks are real data blocks as formatted in Figure 4.2, while the rest are dummy blocks each with ID 0

and random content, as illustrated in Figure 4.4.

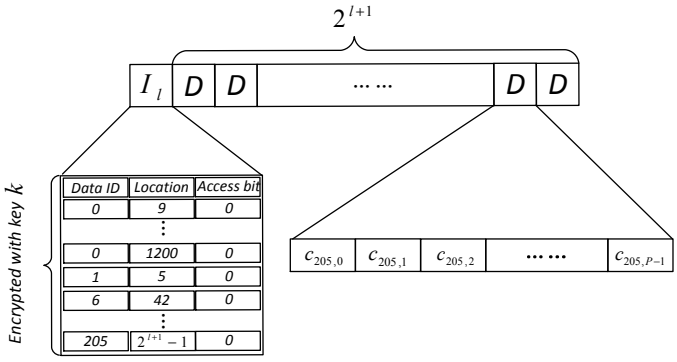

Figure 4.4 T1-layer Structure [Jingsheng Zhang (2014)]

(3) *T2 Layers: Multi-Segment Layers.* Layers from layer $L_1 + 1$ to the bottom layer $L_2 = \log N$ belong to T2. Segments are introduced in T2-layers. Specifically each layer in T2 consists of $W_l = \lceil \frac{2^l}{\log^2 N} \rceil$ segments, each of which is of the same structure as T1-layer, except that each segment consists of $3\log^2 N$ data blocks.

The reason that each segment in T2-layer includes $3\log^2 N$ data blocks is to make sure that, an index with a block size can contain all the blocks information in a segment (regardless whether at a T1-layer or T2-layer). In above storage structure, the index block of a segment has at most $3\log^2 N$ entries. As each entry contains three fields: *ID of the data block* (needing $\log N$ bits), *location of the data block in the segment* (needing $\log(3\log^2 N)$ bits), and *access bit*, an index block needs at most $3\log^2 N[\log N + \log(3\log^2 N) + 1]$ bits. In the existing studies of practical ORAM schemes, $N \leq 2^{36}$ is considered large enough to accommodate most practical applications [Stefanov et al. (2011)]. As the size of an index block is less than 32 KB, it can fit into a typical data block.

4.2.5 Storage Initialization

Before any query or shuffle request sent to the server, storage on both server and client should be initialized. The user initializes the S-ORAM system as follows.

(1) Randomly pick a secret key k in user-side storage.

(2) Generate a one-way hash function $H_{L_2}(\cdot)$ for the bottom layer.

(3) Encrypt N plain-text data blocks into blocks D_i where $i = 1, \cdots, N$ with the secret key k in the format illustrated by Figure 4.2.

(4) $2N$ dummy blocks are randomly generated and encrypted in the same way.

(5) Apply L_2 shuffling algorithm 4.4.2 to these $3N$ data blocks. Upload them to layer L_2 of the server storage.

(6) Upload a dummy block D to the server and let the server know it is a dummy block.

4.3 Data Query

The data query algorithm is as described in Algorithm 1. When the user queries a data block, two blocks are accessed subsequently for each layer, index block and a data block.

More specifically, if the user want to query a data block D_t of ID t, it consists of four phases.

In Phase I, all blocks in layer 1 are retrieved and decrypted, trying to find D_t.

In Phase II, layers in T1 are accessed. Each non-empty layer is accessed top down. For each layer, the index block is first retrieved and decrypted. If D_t can be found in the index block, record the location of D_t, set the corresponding access bit to 1 in index block, re-encrypt and upload it back to server. Retrieve D_t as the user knows its position. If D_t is not found in that layer, randomly pick a dummy block $D_{t'}$ whose access bit is 0, set its access bit to 1 upload the index block back to server. Retrieve $D_{t'}$ and discard it. Each time when a data block is retrieved, the server will use a dummy block to fill in that position.

In Phase III, layers in T2 are accessed. Each non-empty layer is accessed top down as follows. If D_t has not been found before layer l, segment $s = H_l(t)$ of layer l is picked to access. Treat that segment as a layer in T1. Then the procedure is the same as Phase II. If D_t has already been found at a layer prior to layer l, a segment is randomly selected from layer l. Then treat this segment as a layer in T1 and follow the procedure in Phase II.

In Phase IV, the user need to re-encrypt and upload blocks queried from layer 1 back, along with another a data block. If the target block is not found in layer 1, the data block is the target data block. Otherwise the data block is a dummy block.

4.4 Oblivious Shuffling

Data shuffling is used to perturb data block locations to prevent server learning which block before shuffling corresponds to which block after shuffling. It may happen at all layers of the storage hierarchy except layer 1. Data shuffling at layer l ($l = 2, \cdots, L_2 - 1$) is triggered when total number of queries is an odd multiple of 2^l. At this moment, layer l is empty and all layers before layer l are full. During data shuffling at layer l, all data blocks in layers before layer l are re-distributed randomly to layer l. Dummy blocks may be introduced to make layer l full. When the total number of queries is any multiple of N, data shuffling will happen at the bottom layer L_2. It will re-distribute data blocks in the entire storage to fill the bottom layer and clear layers above.

4.4.1 Shuffling a T1-layer l

Each T1-layer contains an index block and a certain number of data blocks, similar to a segment. In S-ORAM shuffling at a T1-layer will call *segment-shuffling algorithm*. *Segment-shuffling algorithm* is shown in Algorithm 2. Here the user-side cache contains three buffers named B_0, B_1, and B_2. Each of them may temporarily store up to $2m^2$ data pieces. m is a system parameter with the value of 2 in my case.

The *segment-shuffling algorithm* includes two phases. In Phase I the first two pieces of all n blocks are shuffled. First pieces can be decrypted using the secret key. And they can be further used to decrypt second pieces. After that, IDs of the blocks are obtained and permuted according to a newly picked permutation function, and then re-encrypted using the key and newly-picked random numbers. After that, the encrypted new random numbers and encrypted shuffled block IDs are uploaded.

In Phase II, the remaining pieces of all n blocks are shuffled. They are retrieved and decrypted level by level. Then shuffle them according to the new permutation function picked in Phase I, re-encrypt them, and upload them back to the server. For example, we can have $(p-1)_{th}$ old order pieces and $(p-1)_{th}$ new order pieces at some time. In order to get p_{th} new order pieces, first retrieve p_{th} old order pieces, and decrypt them with $(p-1)_{th}$ old order

pieces. Then permute them with the permutation, and re-encrypt them with $(p-1)_{th}$ new order pieces. Thus we can get p_{th} new order pieces.

When a T1-layer l $(2 \leq l \leq L_1)$ need to be shuffled, before calling *segment-shuffling algorithm*, the server need to make 4 copies of dummy block in order to make that layer full.

4.4.2 Shuffling a T2-layer l

Before shuffling a T2-layer l, the user need to update the hash function used for it and let the server make $4 \cdot 2^l - w$ copies of the dummy blocks (w is the number of blocks above layer l). Thus there are $4 \cdot 2^l$ data blocks in total to be shuffled. This is to ensure that the following 5 rounds of shuffling can be done successfully.

(1)*Round I: Scanning.* In this round the user retrieve all the blocks in layer l, and label each block with two tags. For each real data block of ID i, use $H_l(i)$ as its first-tag and 0 as its second-tag. All dummy blocks have ∞ as their second-tag. For each first tag value j from 1 to $\lceil \frac{2^l}{\log^2 N} \rceil$, label exactly $3\log^2 N$ dummy blocks with j as their first-tag. And label the rest dummy blocks with ∞ as their first-tag.

(2)*Round II: Oblivious Sorting.*

Based on the tag-tuple labeled in Round I, sort all the blocks obliviously in the non-descending order using the *oblivious data sorting* algorithm presented in Section 4.5. The sorting will first compare first-tag. If two blocks are with the same first-tag, it will compare the second tag. After this round, for each first-tag real data blocks are before dummy blocks.

(3)*Round III: Scanning.* In this round the sorted sequence of blocks is scanned and divided into segments each containing $3\log^2 N$ blocks. For each first-tag j, count $3\log^2 N$ blocks. And after that, change the first-tag of the rest blocks with first-tag j to ∞. Thus there will be exactly $3\log^2 N$ data blocks for each first-tag.

(4)*Round IV: Oblivious Sorting.* In this round it sorts all the blocks again using oblivious sorting. As a result, all the blocks with ∞ as the first-tag will move to the end of the sequence. Then, the redundant blocks are removed.

(5)*Round V: Scanning.* The first four rounds can ensure that the blocks in layer l are distributed correctly. The last round is to rebuild an index block for each segment in layer l.

Note that even though the bottom layer L_2 belongs to T2-layers, there are two main differences between layer L_2 shuffling and T2-layer shuffling. One difference is that when layer L_2 needs to be shuffled, the entire storage shall be shuffled and all blocks from every layer shall participate in data shuffling. Hence, the total number of blocks to be shuffled is $w' = 4 + 2^{2+1} + \cdots + 2^{L_1+1} + 3 \cdot 2^{L_1+1} + \cdots + 3 \cdot 2^{L_2-1} + 3 \cdot 2^{L_2} < 6N$. Second difference is that, in the five rounds, Round I scanning and Round II oblivious sorting are performed on $w' < 6N$ blocks instead of $4 \cdot 2^l$ blocks in T2-layer shuffling. After Round II oblivious sorting, only the first $4N$ blocks participate in Rounds III, IV, and V. The rest of shuffling at layer L_2 are identical to the ones in T2-layer shuffling.

4.5 Oblivious Sort

In S-ORAM, an m-way oblivious sorting scheme based on the m-way sorting algorithm in [Lee and Batcher (1995)] is applied. It sorts data in pieces rather than blocks, which exploits the user cache space more efficiently and thus achieves a better performance than the existing algorithms, particularly when the block size is relatively large (which is common in practice [Stefanov et al. (2011)]).

As shown in Algorithm 3, to sort a set $\mathcal{D}$ of n blocks, the m-way oblivious sorting algorithm works recursively as follows: if $n \leq 2m^2$, a *segment-sorting algorithm* similar to the segment-shuffling algorithm is applied to sort the n blocks at the communication cost of $O(n)$ blocks; otherwise, the n blocks are split into m subsets each of $\frac{n}{m}$ blocks, the m-way oblivious sorting algorithm is applied to sort each of the subsets, and finally a *merging algorithm* is used to merge the sorted subsets into a sorted set of n blocks.

In Algorithm 3, two subroutines are called, which are the segment-sorting algorithm (Algorithm 4) and the merging algorithm (Algorithm 5). The segment-sorting algorithm is based on the segment-shuffling algorithm (Algorithm 2) with the difference that in segment-sorting algorithm pieces are permuted based on their tags value, while segment-shuffling algorithm pieces are randomly permuted. The merging algorithm first divides blocks into m groups recursively until the blocks number of each group is less than $2m^2$, which means it can call Algorithm 3 to sort this group of blocks at one time. After all groups are sorted, merge them together.

Algorithm 1 Query data block D_t of ID t [Jingsheng Zhang (2014)].

$found \leftarrow false$
Retrieve & decrypt blocks in layer 1
if D_t is found in layer 1 **then** $found \leftarrow true$
for each non-empty layer $l \in \{2, \cdots, L_1\}$ **do**
    Retrieve & decrypt I_l – index block of the layer
    if $(found = false \wedge t \in I_l)$ **then**
        Set the access bit of D_t to 1 in I_l
        Re-encrypt & upload I_l
        Retrieve & decrypt D_t
        $found \leftarrow true$
    else
        Randomly pick a dummy $D_{t'}$ with access bit 0
        Set the access bit of $D_{t'}$ to 1 in I_l
        Re-encrypt & upload I_l
        Retrieve & discard $D_{t'}$
    end if
end for
for each non-empty layer $l \in \{L_1 + 1, \cdots, L_2\}$ **do**
    if $(found = false)$ **then**
        $s \leftarrow H_l(t)$
    else
        s is randomly picked from $\{0, \cdots, W_l - 1\}$
    end if
    Retrieve & decrypt I_l^s – index block of segment s
    if $(found = false \wedge t \in I_l^s)$ **then**
        Set the access bit of D_t to 1 in I_l^s
        Re-encrypt & upload I_l^s
        Retrieve & decrypt D_t
        $found \leftarrow true$
    else
        Randomly find a dummy $D_{t'}$ with access bit 0
        Set the access bit of $D_{t'}$ to 1 in I_l^s
        Re-encrypt & upload I_l^s
        Retrieve & discard $D_{t'}$
    end if
end for
if (D_t is found in layer 1) **then**
    Encrypt an extra dummy D in local storage
else
    Re-encrypt D_t in local storage
end if
Upload all blocks in local storage back to layer 1

Algorithm 2 Segment-shuffling of Blocks $(D_{i_1}, \cdots, D_{i_n})$ [Jingsheng Zhang (2014)].

/ Phase I: shuffling first two pieces of all blocks */*
Retrieve $(c_{i_1,0}, \cdots, c_{i_n,0})$ to B_0
Decrypt B_0 to $(r_{i_1,0}, \cdots, r_{i_n,0})$ using k
Retrieve $(c_{i_1,1}, \cdots, c_{i_n,1})$ to B_1
Decrypt B_1 to $(i_1, \cdots, i_n)$ using k and B_0
Store $(i_1, \cdots, i_n)$ in B_2
Pick & store a random permutation in π
Permute B_2 to $(i'_1, \cdots, i'_n)$ according to π
Generate, re-encrypt & upload entries of a new index block based on B_2 and π
for each i'_j in B_2 **do**
    Randomly picks $r'_{i'_j}$
    Encrypt $r'_{i'_j}$ to $c'_{i'_j,0}$ using k, and upload it
    Encrypt i'_j to $c'_{i'_j,1}$ using k and $c'_{i'_j,0}$
end for
Upload B_2 to designated locations
/ Phase II: shuffling remaining pieces of all blocks */*
for each $v \in \{2, \cdots, P-1\}$ **do**
    Retrieve $(c_{i_1,v}, \cdots, c_{i_n,v})$ to B_0
    for each $j \in \{1, \cdots, n\}$ **do**
        Decrypt $c_{i_j,v}$ to $d_{i_j,v}$ using k and $c_{i_j,v-1}$ in B_1
        Replace $c_{i_j,v-1}$ in B_1 by $c_{i_j,v}$ from B_0
        Replace $c_{i_j,v}$ by $d_{i_j,v}$ in B_0
    end for
    Permute B_0 to $(d_{i'_1,v}, \cdots, d_{i'_n,v})$ according to π
    Encrypt $(d_{i'_1,v}, \cdots, d_{i'_n,v})$ in B_0 using k and B_2
    Replace B_2 by B_0
    Upload B_2 to designated locations
end for

Algorithm 3 M-way Oblivious Sorting ($\mathcal{D}$: a set of data blocks) [Jingsheng Zhang (2014)]

if $(|\mathcal{D}| \leq 2m^2)$ **then**
    Apply Algorithm 4 to sort $\mathcal{D}$
else
    Split $\mathcal{D}$ into m equal-size subsets of blocks $\mathcal{D}_0, \cdots, \mathcal{D}_{m-1}$
    for each i $(0 \leq i \leq m-1)$ **do**
        Apply Algorithm 3 to sort $\mathcal{D}_i$
    end for
    Apply Algorithm 5 to merge $\mathcal{D}_0, \cdots, \mathcal{D}_{m-1}$
end if

Algorithm 4 Segment-sorting of Blocks $(D_{i_1}, \cdots, D_{i_n})$ [Jingsheng Zhang (2014)].

1-5: the same as in Algorithm 2
6: Construct a permutation function that sorts B_2 in the non-decreasing order
7: the same as in Algorithm 2
8: blank
9-14: the same as in Algorithm 2
15: **for** each $v \in \{2, \cdots, P\}$ **do**
16-26: the same as in Algorithm 2

Algorithm 5 Merging Sorted-subsets of Blocks $(\mathcal{D}_0, \cdots, \mathcal{D}_{m-1})$ [Jingsheng Zhang (2014)]

/ Regroup blocks */*
$s = |\mathcal{D}_0|$
for each i $(0 \leq i \leq m-1)$ **do**
  for each j $(0 \leq j \leq m-1)$ **do**
    Add $\mathcal{D}_i[j], \mathcal{D}_i[m+j] \cdots, \mathcal{D}_i[s-m+j]$ to $\mathcal{D}'_j$
  end for
end for
/ Recursively merge regrouped blocks */*
for each j $(0 \leq j \leq m-1)$ **do**
  if $|\mathcal{D}'_j| \leq 2m^2$ **then**
    Apply Algorithm 4 to sort $\mathcal{D}'_j$
  else
    Apply Algorithm 5 to merge sort $\mathcal{D}'_j$
  end if
end for
/ Merge sorted blocks */*
for each i $(0 \leq i \leq \frac{s}{m} - 2)$ **do**
  for each j $(0 \leq j \leq m-1)$ **do**
    Add $\mathcal{D}'_j[im], \mathcal{D}'_j[im+1], \cdots, \mathcal{D}'_j[im+2m-1]$ to $\mathcal{D}''_i$
  end for
end for
for each i $(0 \leq i \leq \frac{s}{m} - 1)$ **do**
  Apply Algorithm 4 to sort $\mathcal{D}''_i$
end for

CHAPTER 5. IMPLEMENTATION

5.1 FUSE

5.1.1 Introduction

Filesystem in Userspace (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 only a "bridge" to the actual kernel interfaces. Users just write code that implements file operations like *open()*, *read()*, and *write()*. When the file system is mounted, programs are able to access the data using the standard file operation system calls, which call the afore mentioned code developed by the user.

Released under the terms of the GNU General Public License and the GNU Lesser General Public License, FUSE is free software. This implementation of FUSE is available for Linux, FreeBSD, NetBSD, OpenSolaris, Minix 3, Android and OS X.

FUSE is built upon VFS, same level with Ext2, Ext3 and other real file systems. FUSE is more like a converter. So compared with these traditional file systems, it has two major advantages:

1) Convenience. Traditional file systems are defined on OS kernel level. So in order to apply a new file system, the OS kernel needs to be modified, which is very inefficient. But programming with FUSE is much easier and faster, which significantly reduces workload for writing a new file system.

2) Flexibility. There are very limited libraries and a lot of limitations available when developing for traditional file systems in kernel space. But as FUSE runs in user space level, it can call user level libraries, which are abundant. It can even use HTTPS and FTP protocols

to create remote file systems.

5.1.2 Preliminary: Linux File System

As FUSE provides a library for developers to build new file systems for Linux, it's necessary to have a basic knowledge about Linux file system. A file system is an organization of data and metadata on a storage device. A standard interface is needed because there are many types of file systems and media. The Linux file system interface is implemented as a layered architecture, separating the user interface layer from the file system implementation and the drivers that manipulate the storage devices [Jones (2007)]. For example, let's consider the *read* function call, which allows some number of bytes to be read from a given file descriptor. The function does not need to know file system types, or what particular storage medium upon which the file system is mounted. Yet, when the *read* function is called for an opened file, the data is returned as expected. FUSE provides an interface for developers to override file functions such as *read()* and *write()*.

FUSE works in user space level in the file system architecture. The architecture in Figure 5.1 shows the relationships between the major file system related components in both user space and the kernel. User space contains the applications (FUSE in our implementation) and the GNU C Library (glibc), which provide the user interface for the file system calls. The system call interface acts as a switch, delivering system calls from user space to the appropriate endpoints in kernel space. The Virtual File System (VFS) is the primary interface to the underlying file systems. This component exports a set of interfaces and then abstracts them to the individual file systems, which may behave very differently from one to another. Each individual file system implementation, such as Ext2, JFS, and so on, exports a common set of functions that is used by the VFS. The buffer cache buffers requests between the file systems and the block devices that they manipulate. For example, read and write requests to the underlying device drivers migrate through the buffer cache. This allows the requests to be cached there for faster access, rather than going back out to the physical device. The buffer cache can be managed as a set of least recently used (LRU) lists [Jones (2007)].

There are a common set of objects that implement file system in user space, which consists of

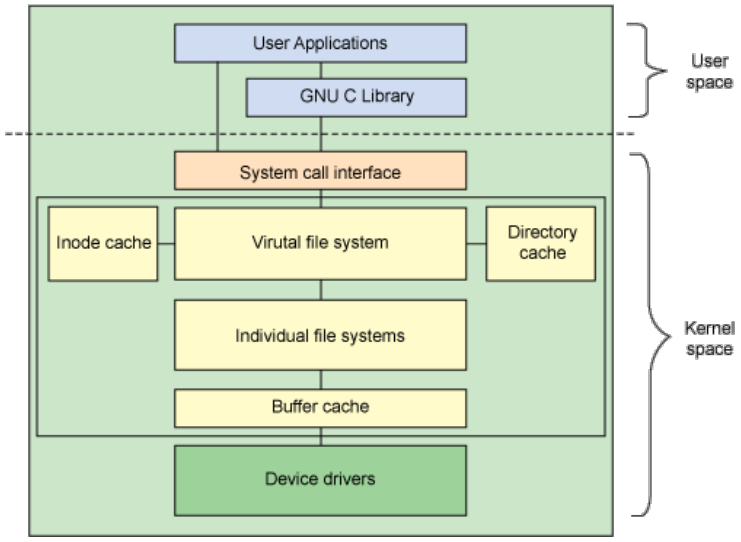

Figure 5.1 Architecture of the Linux Filesystem [Jones (2007)]

superblock, *inode*, *dentry*, and *file* [Jones (2007)]. At the root of each file system is the *superblock*, which describes and maintains state for the file system. Every object that is managed within a file system (file or directory) is represented in Linux as an *inode*. The inode contains all the metadata to manage objects in the file system, including the operations that are possible on it. Another set of structures, called *dentries*, is used to translate between names and inodes, for which a directory cache exists to keep the most-recently used around. The dentries also maintain relationships between directories and files for traversing file systems. Finally, a VFS file represents an open file (keeps states for the open file such as the write offset, and so on).

Among these objects, inode is most widely used in our implementation. The inode represents an object in the file system with a unique identifier. The individual file systems provide methods for translating a filename into a unique inode identifier and then to an inode reference. A portion of the inode structure is shown below along with a couple of the related structures. Each of these structures refers to the individual operations that may be performed on the inode. For example, *inode_operation*s define those operations that operate directly on the inode.

```
struct inode {
    unsigned long                i_ino;
```

```
    off_t                       i_size;
    struct inode                *i_next, *i_prev;
    struct inode_operations     *i_op;
    ...
}
```

5.1.3 Principle of Operation

FUSE includes three modules: user space library, kernel module and mount tools. User space library provides developers with programming interfaces. There are two sets of interfaces, *struct fuse_lowlevel_ops* and *struct fuse_operations*. Developers write their own file system by registering their own functions to these structures. In our implementation *struct fuse_operations* is used. A portion of its structure is shown below. Kernel module (fuse.ko) provides file system framework to support developers' functions, defined in user space. Mount tools are used to mount FUSE file system.

```
struct fuse_operations my_oper = {
  .getattr = my_getattr,
  .open = my_open,
  .read = my_read,
  .destroy = my_destroy,
  ...
};
```

The working process of FUSE is as shown in Figure 5.2. In this figure, example/hello is an executable program linked to FUSE library. /tmp/fuse is mounted directory that will run this file system. The FUSE kernel module and the FUSE library communicate via a special file descriptor which is obtained by opening /dev/fuse. This file can be opened multiple times, and the obtained file descriptor is passed to the mount syscall, to match up the descriptor with the mounted filesystem. When example/hello is executed, a user space file system, which is implemented in example/hello, will be mounted to current system with the mount point

is /tmp/fuse. When user runs ls -l /tmp/fuse, corresponding VFS functions are called, then delivered to FUSE kernel module. And kernel module will call example/hello functions based on a mapping.

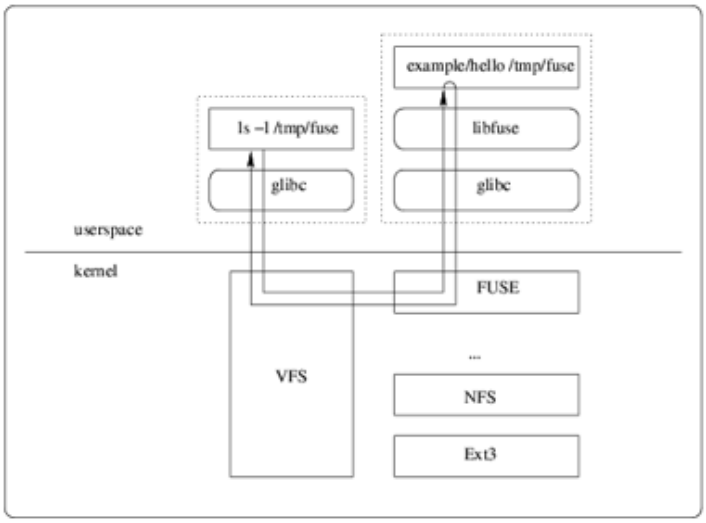

Figure 5.2 Working Process of FUSE [Szeredi (2001)]

5.2 Overview of Our Implementation

In our system model, which is described in Chapter 3, there are user side and server side. As shown in Figure 5.3, there are three parts in our system model: *Client*, *FUSE Server* and *Storage Server*. Among them, *Client* and *FUSE Server* are running locally and are trusted, while *Storage Server* which stores user's data is not trusted. Our goal is to prevent storage server from learning user's access pattern, meanwhile the user can operate normally.

FUSE is already introduced in Section 5.1. The rest of this chapter will cover my implementation with FUSE interfaces, basic data structures used in the system, storage initialization, encryption method, query and shuffling detailed implementation, communication protocol and storage-side buffer.

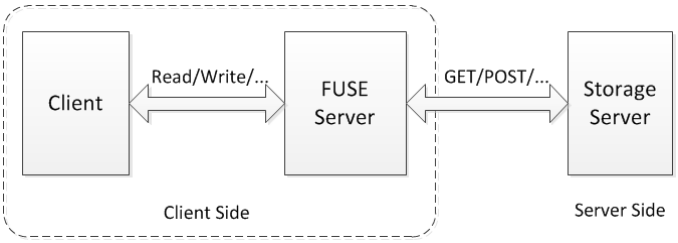

Figure 5.3 System Model

5.3 Interface with FUSE

Based on the description above,the implementation is to write customized file operation functions and register them with FUSE library. This way users can operate on files "locally" in appearance, but actually it is communicating with server using S-ORAM method in the background. The major customized functions are shown as blow. Among all the interfaces provided by FUSE, some of them may not be necessary but they would better be implemented because otherwise once a function is called but it's not defined, the file system will go wrong. Each function returns 0 on success, and returns negative value otherwise. The details of each major function:

```
struct fuse_operations my_oper = {
    .init = my_init,
    .getattr = my_getattr,
    .readdir = my_readdir,
    .read = my_read,
    .write = my_write,
    .destroy = my_destroy,
    ...
};
```

(1) *my_init()*:

```
void *my_init(struct fuse_conn_info *conn){...}
```

This function initializes file system, and is called before other file operation functions. In my implementation it will connect to the server and establish a socket connection. Throughout the lifetime of the file system, it will only use this socket connection with the server.

Also, a log file name is passed to it so that when it is mounted, a log file is created to log activities. It's done by a data structure *struct fuse_context*. It has a field named *void* private_data*, a pointer to arbitrary data stored by the file system. So we can pass some data to the file system before it is mounted, such as parameters user enters when running it. FUSE allows developers store and access data that they defined their own by this way.

(2) *my_getattr()*:

```
int my_getattr(const char *path, struct stat *statbuf){...}
```

This function is to get status of a file or a directory. It passes in a file or directory's path and need to fill in its status to *statbuf*. FUSE does not provide interface to operate on inode directly, instead it allow developers to change file status by providing *struct stat*, which is defined in Linux. Below is its structure.

```
struct stat {
        dev_t     st_dev;         /* ID of device containing file */
        ino_t     st_ino;         /* inode number */
        nlink_t   st_nlink;       /* number of hard links */
        off_t     st_size;        /* total size, in bytes */
    ...
}
```

In our model when the user tries to read or write some file, the file actually does not exist in user's local storage. So when this function is called, we cannot call system call *lstat()*, which can fill in the *statbuf* if the file locates locally. Instead, we need to fill in it manually. As each block size is fixed and total block number is fixed, we can do it manually corresponding to different *path*. If it equals to "\", which is root folder path, fill in total block number. If it equals to a block id, fill in a block size. Other fields in struct stat remain default.

(3) *my_readdir()*:

```
int my_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                    off_t offset, struct fuse_file_info *fi){...}
```

Similar to my_getattr(), given a directory path, we need to fill in *buf* with files information in this directory using *fuse_fill_dir_t filler*. Once user initialize storage at server, block ids and block numbers are known to the user. So we can also fill in the information manually. As a result, when user goes to FUSE mounted directory and run *ls*, all block ids are shown to the user. This process does not need to communicate with server because once server storage is initialized, in our system model block can neither be added deleted. *struct fuse_file_info* keeps information of this directory, no need to change it in this case.

(4) *my_read()*:

```
int my_read(const char *path, char *buf, size_t size,
        off_t offset, struct fuse_file_info *fi){...}
```

Every time a user want to read the content of a file, such as use *cat* command, FUSE will call *my_read()* function. In this procedure client will need to call *query()* function, described in 4.3, which communicate with the server and follows query algorithms in S-ORAM. After the target block is retrieved to user storage, open it locally and fill the content to *buf*.

(5) *my_write()*:

```
int my_write(const char *path, const char *buf, size_t size,
        off_t offset, struct fuse_file_info *fi){...}
```

The procedure of *my_write()* is quite similar to *my_read()*. It also calls *query()* function. The difference is that it overrides target block with changed one locally. So at the end of *query()* when client need to upload local blocks back, new block is uploaded instead of the old one.

(6) *my_destroy()*:

```
void my_destroy(void *userdata){...}
```

This function is called when FUSE file system is unmounted. It will close socket connection with server and FUSE will stop working.

5.4 Data Structure

There are five major data structures in my implementation. They are designed to fit the whole system and are used everywhere.

(1) *PIECE*:

```
typedef struct {
    char data[8];
}PIECE;
```

A block consists of pieces. In theory the size of a piece smaller, the better. But it must be large enough to store a block id and fit encryption method. Because in our implementation DES is chosen as the encryption method, which requires data of at least 64 bits, we set a piece size as 64 bits.

(2) *FIRST_PIECE*:

```
typedef struct {
        int block_id;
        char tag1[3];
        char tag2;
}FIRST_PIECE;
```

The first piece of a data block stores meta data of this block while other pieces store real data. It consists of block id and two tags. In GCC the size of int is 4 bytes. So an int variable can represent up to 2^{32} blocks, which is enough for our system. tag 1 will store segment number. Given $N < 2^{32}$, segments number in a layer is up to 2^{22}. So three bytes, which can represent up to 2^{24} segments, is large enough. tag 2 is an supplement for tag 1 in shuffling. In theory one bit is enough for tag 2 but for encryption and decryption convenience, we pad tag 2 to 1 byte to make *FIRST_PIECE* the same size as other pieces.

(3)*BLOCK*:

```
typedef struct {
        FIRST_PIECE first_piece;
```

```
        PIECE* pieces;
}BLOCK;
```

Each block is a file stored on the server. When a user calls *query()*, it should be returned with a block of this structure. Encrypted block does not have *first_piece* filed and has one more piece than clear block.

(4)*ENTRY*:

```
typedef struct {
        int block_id;
        char pos[3];
        char access;
}ENTRY;
```

One *ENTRY* is one mapping relation from block id to position, along with *access* filed to indicate whether this position has been visited or not. One bit is enough for *access*, but for encryption and decryption convenience, it's padded to 1 byte. Thus one entry will be 8 bytes, same as a piece size.

(5)*INDEX_BLOCK*:

```
typedef struct {
        ENTRY entries[BLOCK_SIZE/PIECE_SIZE-1];
}INDEX_BLOCK;
```

An index block is of the same size of a clear data block. It consists of entries. After encryption its structure is the same as encrypted data block.

5.5 Encryption

In S-ORAM data blocks are encrypted piece by piece with a secret key k, which is mentioned in Section 4.2.2. The encryption method E_k is Data Encryption Standard (DES). We use DES as our encryption method as it can encrypt as low as 8 bytes data at one time, which is exactly a piece size.

The mode of operation we use is Cipher-block chaining (CBC). In CBC mode, each piece of plain text is XORed with the previous cipher text block before being encrypted. This way, each cipher text piece depends on all plain text pieces processed up to that point. To make each block unique, an random vector must be used in the first block. Because in S-ORAM shuffling is on pieces, this mode ensures that the client can sort blocks based on their tag values. Once the client get the first piece of blocks in a segment, it can decrypt the rest pieces in that segment level by level.

The DES algorithm implementation is based on OpenSSL library [Project (1998)]. It implements the basic cryptographic functions and provides various utility functions. DES encryption and decryption can be done with three functions:

- *void DES_set_odd_parity(DES_cblock *Key)*

  This function is to set the parity of the passed key to odd. The key is given by the user and must be 8 bytes long.

- *int DES_set_key_checked(DES_cblock *Key, DES_key_schedule *schedule)*

  It will check that the key passed is of odd parity and is not a week or semi-weak key. If the parity is wrong, then -1 is returned. If the key is a weak key, then -2 is returned. If an error is returned, the key schedule is not generated. The key schedule is an expanded form of the key, used to speed the encryption process.

- *void DES_cfb64_encrypt(const unsigned char *in, unsigned char *out, long length, DES_key_schedule *schedule, DES_cblock *key, int *num, int enc)*

  Encryption and decryption happens in this function. It encrypts or decrypts a single 8 bytes DES_cblock in CFB mode. As we only need to encrypt and decrypt data of 8 bytes, the mode actually does not matter as long as it operates on 64 bits.

  This function transforms the input data, pointed to by input, into the output data, pointed to by the output argument. If the encrypt argument is non-zero, the input is encrypted in to the output using the *key_schedule* specified by the schedule argument, pre-

viously set via *DES_set_key*. If encrypt is zero (*DES_DECRYPT*), the input is decrypted into the output.

5.6 Query

As mentioned in Section 5.3, *query()* function is the bridge between our FUSE Server and S-ORAM design. It is also the only interface provided by S-ORAM implementation. Developers only need to call this function and do not need to consider all the operations happening behind it, including encryption, decryption, shuffling and communication, which makes the system very convenient to use.

The input of *query()* is block id. It will be passed to Algorithm 1. In the implementation of the algorithm there are three key functions.

(1)*INDEX_BLOCK getIndexBlock(int layer, int segment_number, bool* empty)*

When query a block, the user will retrieve an index block from each layer top down. This function is called to retrieve the index block given its segment number and layer number. If such index block does not exist in the server-side storage, it indicates that that layer is empty. The function will set the boolean variable *empty* to true. Otherwise get the encrypted index block from the server, decrypt and return it.

(2)*ENTRY queryInThisLayer(INDEX_BLOCK index_block, int block_id, bool found)*

After get the index block of a segment, the user needs to check whether the target block id can be found in that layer, if it has not been found yet. If yes, return its entry. Otherwise, return an entry recording the information of a dummy block whose access bit is 0.

(3)*void queryBlock(ENTRY entryToQuery, int layer, int segment, INDEX_BLOCK* index_block)* Based on the entry returned by *queryInThisLayer*, this function will retrieve a data block at the position on that layer. Then set the access bit of that position to 1 in the index block. Re-encrypt it and upload it.

5.7 Shuffling

The shuffling part basically follows algorithms described in Section 4.4 but in order to fit our system model, some details need to be specific.

(1) *Query Counter.* The system maintains a global integer variable *query_counter* from the beginning to the end on the client side. Each time the client calls function *query()*, *query_counter* will increase by 1. It's used to trigger shuffling and update hash function. The way it triggers shuffling is explained in Section 5.7. Each T2-layer has a hash function and when it is shuffled the hash function is updated with current *query_counter* [See Section 5.7]. So the client need to maintain the value of *query_counter* for each T2-layer. Each time shuffling a T2-layer, the client need to store current value of *query_counter* for that layer.

(2) *Trigger of Shuffling.* Each time when all layers before layer l is full, shuffling at layer l should happen. This can be done by examining *query_counter*. Based on the observation of server-side storage organization, for any layer except layer 1, say layer l, it need to be shuffled when the l_{th} bit of *query_counter* is 1 and all the bits before it are 0. The code is shown below.

```
int needShuffle(int query_counter){
        int i = 0, x = query_counter;
        while((x != 0) && ((x&1) != 1)){
                i++;
                x = x>>1;}
        if(i>1) return i;
        return 0;
}
```

(3) *Segment Shuffling.* As is described in Section 4.4.1, it is used in T1-layer shuffling and segment shuffling in T2-layer. In our model communication with server is involved. The flow chart is shown in Figure 5.4.

During the process the server maintains an array of block pointers. Before shuffling, the server create an array and store pointers of blocks to be shuffled in it. At first all pointers point at the very beginning of each block. After then because pieces in a block is access sequentially,

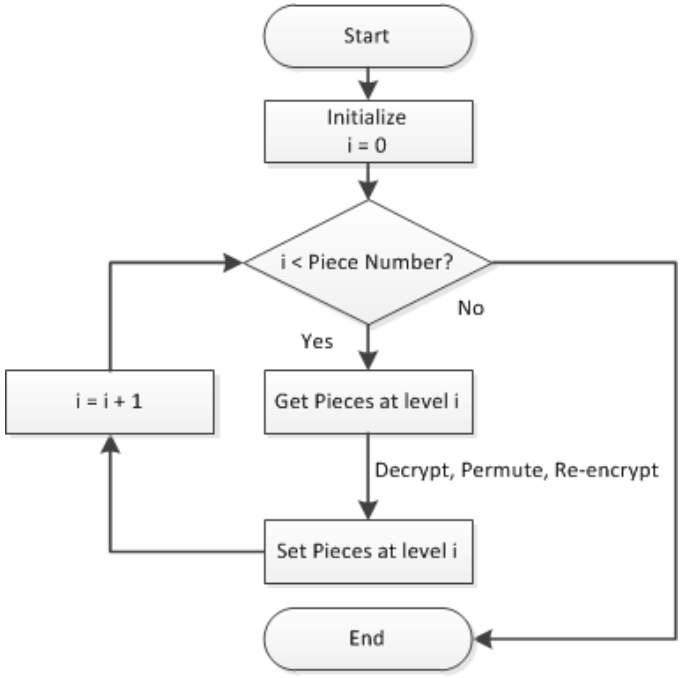

Figure 5.4 Flowchart of Segment Shuffling

we don't need to control the pointers manually. When they point to the end of blocks, the shuffling is also done.

(4)*Hash Funtion* Based on the design of S-ORAM, the value of the hash function in our system need to be different if one of the three variables has changed: block id, layer number and query counter. So these three variables are stored sequentially in a contiguous memory space and then calculate its MD5 result as its hash value.

The MD5 algorithm operates on a 128-bit state, divided into four 32-bit words. These are initialized to certain fixed constants. It then uses each 512-bit message block in turn to modify the state. The processing of a message block consists of four similar stages, termed rounds; each round is composed of 16 similar operations based on a non-linear function, modular addition, and left rotation.

5.8 Communication

The communication between client and server is based on TCP socket. A socket is an end point of communication between two systems on a network. To be more precise, a socket is a combination of IP address and port on one system. So on each system a socket exists for a process interacting with the socket on other system over the network.

The system calls for establishing a connection are different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

The steps involved in establishing a socket on the client side are as follows:

- Create a socket with the socket() system call. The call to the function socket() creates an UN-named 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. Information like IP address of the remote host and its port is bundled up in a structure and a call to function connect() is made which tries to connect this socket with the socket (IP address and port) of the remote host.
- Send and receive data. Once the sockets are connected, the server sends the data on client's socket through client's socket descriptor and client can read it through normal read call on the its socket descriptor.

The steps involved in establishing a socket on the server side are as follows:

- Create a socket with the socket() system call.
- Bind the socket to an address using the bind() system call. This call assigns the details specified in the structure *serv_addr* to the socket created in the step above. The details include, the family/domain (for Internet family of IPv4 addresses we use *AF_INET*), the interface to listen on(in case the system has multiple interfaces to network) and the port on which the server will wait for the client requests to come.

- Listen for connections with the listen() system call. After the call to listen(), this socket becomes a fully functional listening socket.
- Accept a connection with the accept() system call. In the call to accept(), the server is put to sleep and when for an incoming client request, the three way TCP handshake is complete, the function accept() wakes up and returns the socket descriptor representing the client socket.
- Send and receive data.

In our system once a socket connection is established between the client and the server, it will be used for future communication until the client close it. Two categories of functions are defined in this part:

(1) *GET*

The client call functions in this category to get data from the server. Usually one round is enough. The client send a *GET* request along with a few parameters then wait at the socket. The server receives the request and send corresponding data to client. The client will count received data size. Once received data size reaches expected value it stops receiving. After finishing sending data, the server will return to waiting for request on the socket.

(2) *POST*

This category consists of functions for client to post data to server. There are usually two rounds in such a process. In first round The client send a *POST* request along with a few parameters then wait at the socket. Once the server receives it sends *READY* back to the client and wait for data. In the second round the client sends data to server. After server finished receiving, it returns to wait for a new request.

5.9 Server-side Buffer

In server-side storage, data are stored in blocks. But segment shuffling is triggered, we need to get and set the i_{th} pieces of a set of blocks. So a buffer mechanism is needed to operate on a certain level of pieces of a set of blocks. In my implementation, each time before segment

shuffling, the server will open all the blocks in that segment and store the file pointers in an array. Then the following two functions will be called to operate on the pieces.

```
char* getPieceArray(int size){
        PIECE* res = malloc(PIECE_SIZE*size);
        int i;
        for(i=0; i<size; i++){
            fread(&res[i], PIECE_SIZE, 1, FilePointers[i]);}
        return (char*)res;}

void setPieceArray(PIECE* content, int size){
        int i = 0;
        for(i=0; i<size; i++){
            fseek(FilePointers[i], -sizeof(PIECE), SEEK_CUR);
            fwrite(&content[i], sizeof(PIECE), 1, FilePointers[i]);
        }
}
```

getPieceArray() will read a piece-size data from each of the file pointers to form the pieces array buffer. After processing the pieces array, *setPieceArray* is called to set the pieces array back to blocks. Note that during this process piece level does not need to be given as we always do this level by level. And file pointers will move automatically when we read or write from it. We only need to move them a piece size back when we write it. This makes processing pieces array more convenient.

CHAPTER 6. EVALUATION

6.1 Settings

The evaluation is done in a MacBook Air laptop. Two virtual machines are created by Virtual Box with the same configuration. One plays the client and the other plays the server. The specific configurations of the MacBook Air and virtual machines are shown in Table 6.1. In Host-only Mode, two virtual machines can be set in the same internal network so that they can communicate with each other.

Configuration	MacBook Air	Virtual Machine
Operationg System	OS X 10.9.2	Ubuntu 12.04 LTS
CPU	1.8 GHz Intel i5 with 2 processors	1.8 GHz Intel i5 with 1 processor
Storage Spcace	128 GB	20 GB
Memory	4 GB DDR3	1 GB DDR3
Network	-	Host-only Mode
FUSE Version	-	2.9.2

Table 6.1 Experiment Environment

6.2 Security

To prove that our implementation can successfully protect the user's access pattern, we observe blocks access frequency from the server side. In the first experiment the client keeps querying the same block. In the second experiment, the client queries blocks randomly. If the access pattern of the two experiments are the same or very similar, the security of our implementation can be demonstrated. Here, we set the total number of real data blocks to $N = 256$ and querying times $T = 100$, $T = 500$ and $T = 1000$, respectively.

The results show that if we observe the frequency of each layer, there is no difference at all.

The access patterns are exactly the same as shown in Table 6.2 and Table 6.3. So we choose layer 3, layer 5 and layer 7 to see if we can tell any difference on certain layers. The results when querying 1000 times are shown in Figure 6.1, Figure 6.2 and Figure 6.3. We can see that the positions on a certain layer are access randomly. No access pattern is leaked.

Layer	1	2	3	4	5	6	7	8
Frequency when T = 100	150	48	48	48	36	36	0	100
Frequency when T = 500	750	248	248	244	244	244	244	500
Frequency when T = 1000	1500	500	496	496	488	488	488	1000

Table 6.2 Block Access Frequency when Keep Querying Same Block

Layer	1	2	3	4	5	6	7	8
Frequency when T = 100	150	48	48	48	36	36	0	100
Frequency when T = 500	750	248	248	244	244	244	244	500
Frequency when T = 1000	1500	500	496	496	488	488	488	1000

Table 6.3 Block Access Frequency when Keep Querying Random Block

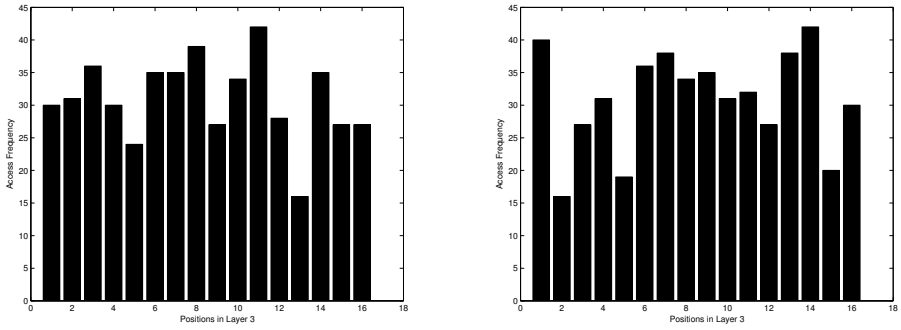

Figure 6.1 Access Pattern in Layer 3 Querying Same and Random Block

6.3 Overhead

The overhead includes user-side storage, communication overhead for query and communication overhead for shuffling. As the permanent user-side storage is much smaller than cache, we only count cache as user-side storage. Because for every N queries the average communication overhead would be the same, we only record overhead when querying N times for different

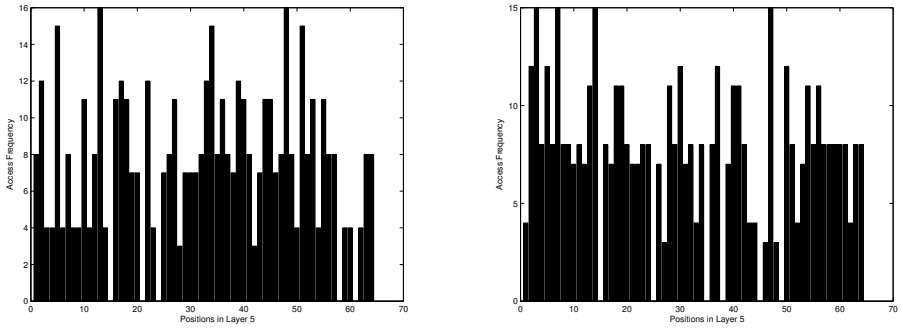

Figure 6.2 Access Pattern in Layer 5 Querying Same and Random Block

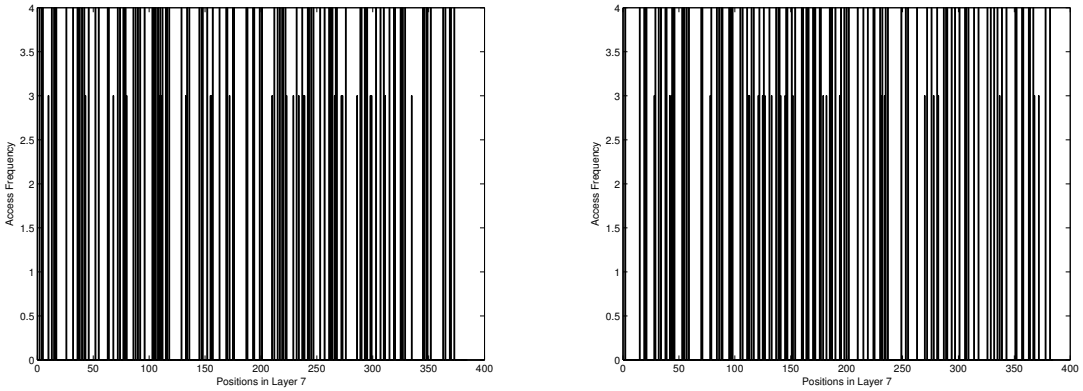

Figure 6.3 Access Pattern in Layer 7 Querying Same and Random Block

N.

The communication overhead results are shown in Table 6.4 and Table 6.5. The results show that the measured communication overhead of my implementation is consistent with theoretical results, in both query and shuffling.

N	Max User-side Storage (# of blocks)	Average Overhead Measured (# of blocks)	Average Overhead In Theory (# of blocks)
64	2	8	$O(\log(N))$
256	2	10	
1024	2	12	
4096	2	14	

Table 6.4 Query Overhead Measured and Theoretical Results

N	Total Shuffling Times	Average Overhead Measured (# of blocks)	Average Overhead in Theory (# of blocks)
64	16	13	$O(\log^2(N))$
256	64	44	
1024	256	81	
4096	1024	147	

Table 6.5 Shuffling Overhead Measured and Theoretical Results

CHAPTER 7. CONCLUSION

Cloud storage has a wide range of applications and is becoming more and more popular. This thesis implements a Segmentation-based Oblivious RAM (S-ORAM) to efficiently protect users' data security and privacy in cloud storage .

Data encryption alone is often not enough to protect the user's privacy in outsourced storage as access pattern can leak sensitive information as well. A typical approach to prevent this from happening is using Oblivious RAM, which allows a client to conceal its access pattern by continuously shuffling and re-encrypting data as they are accessed. However, many existing ORAM constructions consume too much storage and communication bandwidth, which makes them not practical.

In this thesis, we implement a new approach named S-ORAM. Two segment-based techniques, piece-wise shuffling and segment-based query, are used to efficiently improve the efficiency in query and data shuffling.

Evaluation have been conducted to test the security and the performance of my implementation. The results show that my implementation of S-ORAM can fully protect clients' access pattern. It only requires constant user-side storage and the communication overhead is on the same complexity level as the theoretical result.

BIBLIOGRAPHY

Amazon (2006). http://aws.amazon.com/s3/. In *Amazon S3*.

Dropbox (2007). http://www.dropbox.com/. In *Dropbox*.

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

Goldreich, O. (1987). Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM.

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

Goodrich, M. T. (2010). Randomized shellsort: a simple oblivious sorting algorithm. In *Proc. SODA*.

Goodrich, M. T. and Mitzenmacher, M. (2010). Mapreduce parallel cuckoo hashing and oblivious RAM simulations. In *Proc. CoRR*.

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

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

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

Islam, M., Kuzu, M., and Kantarcioglu, M. (2012). Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Proc. NDSS*.

Jingsheng Zhang, Wensheng Zhang, D. Q. (2014). S-ORAM: A Segmentation-based Oblivious RAM. In *ACM Symposium on Information, Computer and Communications Security*.

Jones, M. T. (2007). Architecte of the linux file system components. Online; accessed 20-April-2014.

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

Lee, D.-L. and Batcher, K. E. (1995). A multiway merge sorting network. *IEEE Transactions on Parallel and Distributed Systems*, 6(2).

Ostrovsky, R. (1990). Efficient computation on oblivious rams. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 514–523. ACM.

Pinkas, B. and Reinman, T. (2010). Oblivious RAM revisited. In *Proc. CRYPTO*.

Project, T. O. (1998). An open-source implementation of the ssl and tls protocols. Online; accessed 20-April-2014.

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

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

Stefanov, E., Shi, E., and Song, D. (2011). Towards practical oblivious RAM. In *Proc. NDSS*.

Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S. (2013). Path ORAM: an extremely simple oblivious RAM protocol. In *Proc. CCS*.

Szeredi, M. (2001). Path of a filesystem call. Online; accessed 23-April-2014.

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