Title	行列型のストレージ構造を使用したORAM(Oblivious Random Access Machine)の帯域幅コストの改善について
Author(s)	Sumongkayothin, Karin
Citation	
Issue Date	2017-09
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/14827
Rights	
Description	Supervisor:宮地　充子，情報科学研究科，博士

Improving bandwidth cost of Oblivious Random Access Machine by using matrix based storage structure

Karin SUMONGKAYOTHIN

Japan Advanced Institute of Science and Technology

Doctoral Dissertation

Improving bandwidth cost of Oblivious Random Access Machine by using matrix based storage structure

Karin SUMONGKAYOTHIN

Supervisor: Atsuko Miyaji

School of Information Science
Japan Advanced Institute of Science and Technology

Degree conferment: September, 2017

Abstract

With the internet technological advances of today, cloud computing is used as a shared pool of computer processing and data for computers and other devices. Since a cloud server is semi-trusted, data stored on the server may be secretly investigated for some benefit of the service provider. The cryptography algorithm such as encryption is widely used to ensure the curious server cannot investigate any confidential information of a client. Although the encryption can protect the confidential content from investigating, the curious server still can gain the benefit from other sources of information. Client's behaviour of accessing information is one of the examples which can be investigated. Occasionally, the access pattern or location of data accessed in the storage may leak the information which tells the user's personality.

Oblivious Random Access Machine (ORAM) is known as the algorithm used to hide the clients access pattern from a trusted but curious storage server. Instead of working like a general client-server, ORAM client generates both uploads (write) and download (read) operations whether it wants to download or upload the information. The reason for doing so is to make every access pattern look similar whether upload or download is being performed. In addition, rather than transferring only data of interest, ORAM client transfers group of data in order to hide a data of interest among the other data that are transferred together. According to the processes mentioned earlier, ORAM algorithm incurs of increasing communication overhead, storage overhead, and computation overhead of the system compared to the normal client-server operation. Therefore, the ORAM research focuses on improving the efficiency of the algorithm so that it is able to work as close as the normal system while still maintaining the level of security as the ORAM should be.

Each different ORAM type has a different design goal, and the protocol used for accessing the information is slightly different depending on the ORAM data structure. However, the common goal for every ORAM researcher is to create the ORAM which is well functional in the practical solution. Typically, the complexity of the ORAM operation is inversely proportional to the size of space requirement on a client. The ORAM which requires the constant storage space of client (e.g. [1] and [2]) needs some extra complex operations to create the security properties, while only few simple operations are enough to a create a secure operation for the ORAM which can provide more extra space on the client (e.g. [3] and [4]). Although an extra space which is required on the client can reduce the complexity of ORAM operation, it does not very beneficial to improve bandwidth consumption spent during the transmission procedure. Since the bandwidth overhead of existing ORAM construction varies according to the ORAM size, bandwidth overhead is another important aspect to be considered besides the operation complexity and storage overhead. In addition, the large storage capacity and high-performance CPUs are generally available at affordable prices. Therefore, the problem of large storage capacity requirement and high complexity of operation is not as important as the use of large amounts of bandwidth for operation. This research focuses on designing ORAM construction which consumes less bandwidth consumption than the other existing schemes while it is still secure under low operation complexity and small storage space usage on the client. To achieve the lower bandwidth consumption than the other ORAM schemes, the new ORAM data structure format should be introduced since the lower bound of bandwidth consumption is based on

the ORAM data structure format. In this research, matrix data structure format is used as a basic structure to design the new ORAM construction. It has been named as *Matrix based ORAM*.

Matrix based ORAM is a novel ORAM construction that is implemented based on matrix data structure. Two versions of matrix based ORAM will be introduced in this thesis: Matrix ORAM (M-ORAM) and Recursive Matrix ORAM (RM-ORAM). These two constructions are intended for different uses. M-ORAM is used when the system needs to maximise the efficiency of data transmission while RM-ORAM is suggested to be used when the client has very limited storage capacity. M-ORAM is the first matrix based ORAM construction which has bandwidth overhead dependent from the size of ORAM. Rather than depending on the size of ORAM as other existing ORAM schemes; bandwidth of M-ORAM varies by the height of matrix data structure, thereby allowing M-ORAM to have constant bandwidth cost for any size of ORAM. In addition, M-ORAM uses very simple operation to do an oblivious transfer. It is one of light-weight ORAM schemes that ever have been presented. RM-ORAM, on the other hand, uses slightly complex operations to do the same purpose as M-ORAM. Since RM-ORAM is designed to reduce the use of storage space on the client, it needs some extra operations for transmitting a data to achieve the same security level as M-ORAM. RM-ORAM significantly reduces the client storage usage by using recursion while the computational and bandwidth overhead are slightly increased as a trade-off. However, it can achieve better overall asymptotic performance compared with other existing recursive ORAM schemes.

With our two proposed ORAM constructions and their experimental results shown by this thesis, it is evident that the ORAM research has gradually shifted from the theoretical research into practice. It seems likely to be effectively deployed for today's technology.

Keywords: Oblivious Random Machine, Cloud security, Information security, Secure access protocol, Applied cryptography.

Acknowledgments

I would like to express my sincere gratitude to the persons who played a significant role in my PhD journey. The first person is Professor Atsuko Miyaji, my supervisor from Japan Advanced Institute of Science and Technology (JAIST), who has been an excellent caregiver since I started my PhD program. With her rich ideas and wide expertise in the field of security, it makes this work possible and succeeded magnificently. The second is Associated Processor Chuhua Su, my co-supervisor. Without his encouragement and suggestion by leading me to the feasible direction, it is unlikely that this research will be completed within the time-frame. Last but not least, the person who is my sub-theme supervisor and former supervisor from Sirindhorn International Institute of Technology(SIIT), Dr. Steven Gordon. With his critical and valuable advice, I can pass through the most difficult time and further improve the quality of this dissertation.

I also would like to express my gratitude to my theses examination committees: Professor Mineo Kaneko, Professor Eiichiro Fujisaki, Associate Professor Omote Kazumasa, and Associate Professor Komwut Wipusitwarakun for their valuable comments and suggestion on this thesis.

I would like to thank all of Miyaji's lab members who provide consistency supporting throughout my stay in Japan. I am much grateful to my beloved family and my wife Chantanat Sumongkayothin for their encouragement and always being supportive beside me throughout my time in the doctoral program.

I would like to thank Japan Advance Institute of Science and Technology (JAIST), Sirindhorn International Institute of Technology(SIIT), and Thailand's National Electronics and Computer Technology Center (NECTEC) for their scholarship support.

Contents

"This dissertation was prepared according to the curriculum for the Collaborative Education Program organized by Japan Advanced Institute of Science and Technology and Sirindhorn International Institute of Technology, Thammasat University."

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard
CBC	Cipher Block Chaining
ECB	Electric Code Book
GCM	Glois/Counter Mode
IV	Initialization Vector
M-ORAM	Matrix based Oblivious Random Access Machine
ORAM	Oblivious Random Access Machine
PRF	Pseudo Random Function
RM-ORAM	Recursive Matrix based Oblivious Random Access Machine

Chapter 1

Introduction

1.1 Cloud Computing and Its Security Consideration

Cloud computing has many benefits; however, raises significant privacy issues. The resources of computation and data are shared among unknown clients. In addition, trusted but curious server can investigate client's activity and data to gain the profit. The client may encrypt the data before uploading to the server to ensure the other clients and server cannot read the data. The cryptographic primitive known as searchable encryption (SE) is suggested for efficient retrieval of data of interest without decrypting. Although many SE algorithms were introduced [5–8], it is still possible for an adversary to observe and analyse the patterns of data accesses between client and server [11, 12]. However by observing the client's activity such as transmission period, access pattern, or address which has been accessed in order to whether read or write data, those fragments of data can possibly yield the sensitive information of client when they are collated and analysed. Oblivious RAM (ORAM) [1] is an approach to hide the client's activity from an untrustworthy investigation. The aim is to allow a client to access the data without the server knowing: whether the client is performing a download or upload; which data the client intends to access; and if a sequence of accesses is the same or different from an observed previous sequence. To generalise an approach of ORAM, data in ORAM system is arranged in the form of *block*. The client actually reads multiple blocks every time it wants to access one data of interest and then writes some number of blocks back to a server. The number of reads and writes is the same for every access. Furthermore, the block having been written to a server is re-encrypted by new secret key to ensure the server cannot do a statistical analysis on client's access pattern. Therefore, we can consider the general security requirements of ORAM are as follows:

- **The relationship between a data and its address cannot be observed:**
  A frequency of access the specific data by client may reveal the significance information. In the normal system, once data has been uploaded to the server, it is stored in the address having been designed by a server until it will be removed. Since the data and its address are tied, the server can easily measure the frequency of accessing any data. In ORAM system once a data has been downloaded; it is re-encrypted by new secret key, and then uploaded to a random address on the server. To be more precise, the information in ORAM is not bound to any addresses, and it is randomly relocated after it has been accessed. In the existing ORAM schemes such as [1] and [2], a data which have been downloaded are re-encrypted by new secret key then the oblivious shuffling and oblivious

merging operation are used to relocate the data. On the other hand, [4] uses random operation to change some parameters which cause of shifting the data to another location. Since the ciphertext of encrypted data is always changed and it is randomly relocated to somewhere within ORAM, the statistical analysis of access frequency on address information is futile.

- **The updated and non-updated data are indistinguishable:**
  In the normal system, data is usually encrypted by one secret key which is only known by the data owner. However, the adversary can recognise the data transformation via the result of encryption. Since the encryption of non-updated data with same secret key always produces the same ciphertext, adversary can realise that the data has been updated if the ciphertext looks different. In ORAM system, the downloaded data is decrypted and re-encrypted by a different secret key. Therefore, whether or not the content of data is changed, the re-encrypted data always looks different from before it was downloaded. Hence, the other party members excepted the data owner can not identify the change in data. The random key re-encrypted is used in every ORAM scheme in the purpose is to conceal the identity of information.

- **The data interested by client and other downloaded data are indistinguishable:**
  Generally, only data of interest is accessed in the normal system. In ORAM system not only a data of interest is accessed, some other non-interested data also must be accessed during an access operation for hiding the data of interest. In current existing ORAM schemes, the number of downloaded data per access request is equal to the height of ORAM data structure, which one among of those data is the data interested by a client. From the perspective of an adversary, it sees a group of data has been downloaded by the client. Although only content of data of interest has been updated, every downloaded data is re-encrypted by a new secret key which makes their ciphertext look different from once they were downloaded. Hence, the adversary cannot separate a data of interest from the other downloaded data.

- **The difference of two access sequences with the same length are indistinguishable from random bit string:**
  Two access sequences can be distinguished by two factors: the number of downloads/uploads, and the addresses which have been accessed by a client. The difference of two access sequences possible leaks the identity of client if it is not well designed. To correct this issue, the same pattern and number of downloads and uploads are generated for every access request to ensuring the access pattern does not reveal the identity of client. Furthermore, the addresses of accessed data are randomly changed after finish access operation. Therefore, although two access sequence is different, this distinction is indistinguishable from a random bit string.

1.2 Motivation of Designing ORAM Construction

In general, ORAM algorithm incurs of higher bandwidth cost, computation complexity, and extra storage requirement than usual system. To retrieve one data of interest , ORAM client has to generate multiple downloads and uploads for retrieving multiple blocks from a server which causes an extra bandwidth cost. Some downloaded blocks are dummy data used for

dummy access by client and it cause of waste of the space usage in server storage. In addition, to achieve the indistinguishable access pattern property, client has to reserve some storage space for processing some extra operations such as shuffling and merging for data location obfuscation. Despite many ORAM schemes being designed [3, 13–41], achieving a satisfactory performance trade-off remains a challenge. The key performance objectives for designing algorithm and data structure of ORAM are:

- Minimize client storage space requirement.
- Maximize server storage usage efficiency.
- Minimize bandwidth cost.
- Minimize client computational complexity.

1.3 Our Contribution

One aspect of ORAM's inefficiency is the bandwidth cost. To hide access patterns, ORAM requires a client to download/upload multiple blocks, even though it is interested in either reading or writing just one of those blocks. In the existing ORAM schemes, the number of blocks to access depends on the total number of blocks that the server may store. In other words, the bandwidth usage is increasing when the size of server storage is growing. This research introduces two new ORAM structures based on a matrix data structure, called Matrix ORAM (M-ORAM) and Recursive Matrix ORAM (RM-ORAM). The design of the matrix data structure allows keeping the bandwidth cost independent of the number of blocks stored on the server, thereby reducing bandwidth cost compared to other similar schemes.

M-ORAM designing focuses on minimising bandwidth usage between client and server. Unlike any existing schemes, M-ORAM server storage is built upon a matrix data structure which allows keeping bandwidth usage constant even though the size of ORAM is varied. In addition, M-ORAM's read and write operation are performed using a simple random permutation function without any complex operations such as shuffling, sorting and merging. However, M-ORAM requires an extra space on a client to collect the address mapping. Therefore M-ORAM is applicable for the devices which has sufficient storage space.

RM-ORAM is an altered construction of M-ORAM which is particularly designed for the small storage device. The majority of address mappings are kept on the server which significantly reduces space requirement on client's storage. However, RM-ORAM requires multiple rounds of access before data of interest can be downloaded. Hence, it incurs of slightly higher bandwidth and computational overhead than M-ORAM. Therefore, to achieve the aim of designing M-ORAM and RM-ORAM, our major contributions are:

- **The design construction and operation of M-ORAM and RM-ORAM**
  We present the detailed design of operation and storage data structure for two ORAM constructions: M-ORAM and RM-ORAM. Both constructions have slightly difference operation and storage structure which are particularly designed for the different purposes of usage.

- **Optimise bandwidth cost and computational complexity**
  We introduce the first matrix based ORAM structure where the bandwidth cost is logarithmic growth of the size of RM-ORAM storage; while in M-ORAM, the bandwidth cost

is independent of ORAM size. Therefore, they can achieve the improved bandwidth cost with any size of ORAM compared with the existing schemes. Regarding the computational complexity, instead of using expensive operation, the simple random permutation function is used to create random behaviour of download/upload operation for both M-ORAM and RM-ORAM. Therefore, the asymptotic cost of computational complexity is lower than the other existing ORAM schemes.

- **Maximize and efficient storage usage**
  In ORAM construction, there are two types of data stored on the server: a dummy data and a real data. Different from the real data, the dummy data (in short, dummy) is the data which tells there is no real data in the block. Different ORAM schemes use the dummy in different manners. In [2], when the real data is writing to either left or right child node, the dummy will be written to another child node for hiding the real information. In [4], the dummy will be written to a server when there is no any real data can fit into the node of accessing path. In [1], the dummy will be downloaded if data of interest is not found after searching. Hence, server storage can not be used full capacity in the existing ORAM construction. For M-ORAM and RM-ORAM, the dummy is represented as the empty block when the number of real data blocks is smaller than the reserved space. Since dummy of M-ORAM and RM-ORAM can be replaced by the real data when it is required, M-ORAM and RM-ORAM can achieve 100% of server storage usage.

- **Theoretical performance analysis**
  We give the theoretical performance models of both M-ORAM and RM-ORAM constructions and use Path ORAM as a reference for comparison. In addition, we give a proof of the appropriate parameters (eg. height, width, stash size, etc.) used for constructing M-ORAM and RM-ORAM in order to function effectively.

- **Experimental analysis**
  To provide further insights into M-ORAM and RM-ORAM performance and security which are not given by the theoretical analysis, we have implemented both constructions and provide experimental analysis of the bandwidth cost, client storage usage, and randomization over download/upload operation.

1.4 Organization

The preliminaries regarding cryptography primitive, knowledge, and tools for an ORAM design are introduced in Chapter 2. The examples of existing ORAM construction, the operation, pros and cons, and their comparison are briefly discussed in Chapter 3. The details of conceptual design, operation, and security analysis of novel ORAM construction called "Matrix based ORAM" (M-ORAM) are introduced in Chapther 4. Chapter 5 introduces the detailed descriptions of the alternate version of Matrix based ORAM for constrained storage space device, called "Recursive Matrix based ORAM" (RM-ORAM). In Chapther 6 the experimental results of M-ORAM and RM-ORAM are given in three aspects: communication overhead, computational complexity, and storage usage efficiency; then the suggested value of parameter used for constructing M-ORAM and RM-ORAM are given in Section 6.1.4 and Section 6.2.4, respectively. Finally, Chapter 7 concludes with the future research direction.

Chapter 2

Preliminary Knowledge for ORAM Construction and Operation

As mentioned in Chapter 1, there are the necessary extra operations applied in a normal system to make it satisfy the security condition required by ORAM problem. For the sake of reader, this chapter presents a concept of ORAM, definition and cryptographic primitive used throughout this thesis.

2.1 Oblivious Random Access Machine

The Oblivious Random Access Machine (ORAM) or sometimes known as Oblivious Random Access Machine is an access operation executed on the particular format of logical storage data structure. It is beneficial for hiding access pattern from the other untrusted members. Nowadays, to secure the information from the other untrusted members, it relies on the strong cryptographic infrastructure the such as public-private key algorithm and symmetric key algorithm. However, those mentioned algorithms still reveal some extra information from the statistical analysis which may be classified as the critical information. The information such as access pattern, the frequency of attempts to access the information, and the data's address; possibly reveal the behaviour of user which can lead to the other attack vectors.

Therefore, the ORAM has to provide the 4 majors properties. First, the number and sequence of reads and writes must be symmetric for every access operation, whether that access operation is for reading or writing the information. The behind reason is that the every access operation must have indistinguishable pattern from the other. Second, the address which has been accessed during access operation must be unpredictably changed to either same or new address to prevent the statistical analysis. Third, the identity of interest information must be blinded from untrusted members perspective, therefore the multiple bogus information are downloaded along with the actual information. The final property is the content of access sequences must be seen as the random bit string.

To kindly note that the ORAM does not provide the feature to protect or hide the content of information but offers the ability to hide access pattern when the user attempts to access the information from the untrusted server. Therefore, if the user needs to secure the content of information, the strong encryption algorithm is necessary.

The very first ORAM construction was introduced by Goldreich *et.al* [1]. It was designed as the software protection which the accessing to memory's address during the execution is not protected and may reveal the essential properties of the program. With the same manner,

accessing information on the cloud storage is as the program assesses to local the memory in the past. The software is like the client and the memory is like the cloud storage server. Hence, the most modern ORAM constructions are designed for the data identity protection on semi-trusted cloud storage.

The opened discussion of ORAM's performance which still is secured by mentioned four properties can be stated into 3 essential issues as follows:

Minimize the bandwidth cost: The ORAM has to downloaded multiple information for downloading the single data of interest by the reason to hide a real data among the other dummy data. In addition, it also has to do reads/writes back and forth in order to create the symmetric access operation. It precise that the bandwidth cost that is produced by ORAM is therefore at least 4 times of the bandwidth produced by the normal system.

Minimize the computational overhead: Due to the extra operations and the extra downloaded information, the computational overhead of the ORAM is worse than the normal system. It is even worse when the security properties are applied to create the ORAM's construction. Therefore, the balance point of security and computational overhead in the ORAM system is an important discussion topic.

Minimize the client's storage space requirement: The ORAM requires an extra space on a client to temporarily contain data that has been downloaded during the access operation. Generally, the space requirement is varied according to the different ORAM constructions. In addition, the small size space requirement is much more extra operation needed in the system. Hence, there is a trade-off between the computational complexity and the client's storage space requirement of the ORAM.

To get to the best answer to the mentioned issues above, various ORAM constructions have been introduced. ORAM construction can generally be divided by the server storage structure: either a hierarchical structure [3, 13–20] where each layer is independent of each other; or a tree–structure [2, 4, 28, 38] where nodes in neighboring layers have a relation of child and parent. Although sometimes ORAM uses the same data structure, a different operation to access the data interested by a client is used in order to get the better performance. However, every ORAM construction has some common operations to keep its access operation indistinguishable from random access: random re-encryption and pseudo-random access. Both operations need some preliminary knowledge of cryptographic primitive and security definition as a tool to well explain their work process. The details will be explained in Section 2.2 and Section 2.3.

2.2 Random Re-encryption Operation

During access operation, multiple blocks must be downloaded and uploaded back and forth from ORAM server. Some of downloaded blocks are possible to be uploaded back to the server within the same period of operation. Since the encryption is a deterministic function, the input with same value will generate the same value of output. Hence, the non-changed data can be appreciable by a server if the client does not change a secret key for encryption.

Random re-encryption is a function which encrypts the input plaintext with random secret key whether or not the content is changed from the original. The key generated by this function

must be well random and not duplication within a particular number of accesses. Pseudo-random function and block cipher are used to construct random re-encryption function. The pseudo-random function is used to generate a new secret key, while block cipher is used to encrypt an input data with the new secret key generated by the pseudo-random function. The detailed definition of both functions is described in the following sections.

2.2.1 Pseudo-Random Function

Pseudo-random functions are deterministic function and computable in polynomial time which returns the output that is indistinguishable from random sequences. The input of function may be arbitrary but the output must always look completely random.

Definition 1. *suppose there are functions $A(x)$ which has two inputs: key K and data X , where $K \in \{0,1\}^k$ and $X \in \{0,1\}^n$:*

$$A : X \times K \rightarrow \{0,1\}^m \tag{2.1}$$

And random function $B(x)$:

$$B : X \rightarrow \{0,1\}^m \tag{2.2}$$

Suppose Z is the algorithm on the oracle, used for guessing the right answer when it get the same inputs as $A(x)$ and $B(x)$. Such that $A(x)$ is the pseudo-random function iff:

$$|[Pr[Z^{A(x)} = 1] - [Pr[Z^{B(x)} = 1]| \leq \epsilon \tag{2.3}$$

2.2.2 Block Cipher

Block cipher is suitable than stream cipher to be an encryption algorithm for ORAM construction with two reasons. First, every data collected in the ORAM server is arranged in block format with the same size. The encrypted data produced by block cipher is also in the units of block which plain-text is split into n blocks with the equal size before encrypted by block cipher algorithm to be n encrypted data blocks (see Figure 2.1). Stream cipher, on the other hand, the size of ciphertext is in Bits or Bytes instead of blocks format. The ciphertext of stream cipher is eventually arranged in block format before being stored to ORAM server, which the extra operation is required. Second, stream cipher algorithm does not provide integrity protection or authentication but block cipher does. The ORAM server is counted as semi-trustful which can arbitrarily change any data stored in its storage. Hence the data integrity protection and data authentication are necessary.

Advanced Encryption Standard (AES), also known as Rijndael is a block cipher encryption algorithm which is used in many ORAM constructions. It has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. The same key is used in both of encryption and decryption. The different key size means the different number of rounds of operation. which 10, 12, and 14 rounds is for 128, 192, and 256 bits key, respectively. The original input key is expanded to be multiple *RoundKeys* which are equal to the number of rounds of operation as shown in Figure 2.2a. To encrypt or decrypt information, AES uses several rounds of operation which RoundKey_i is used in round i. Each round consists of several steps of substitution, transposition (shifting), and mixing operation as illustrated in Figure 2.2b. Although there are several attacks

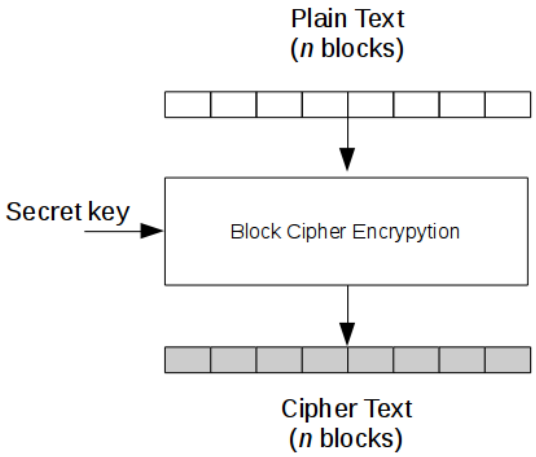
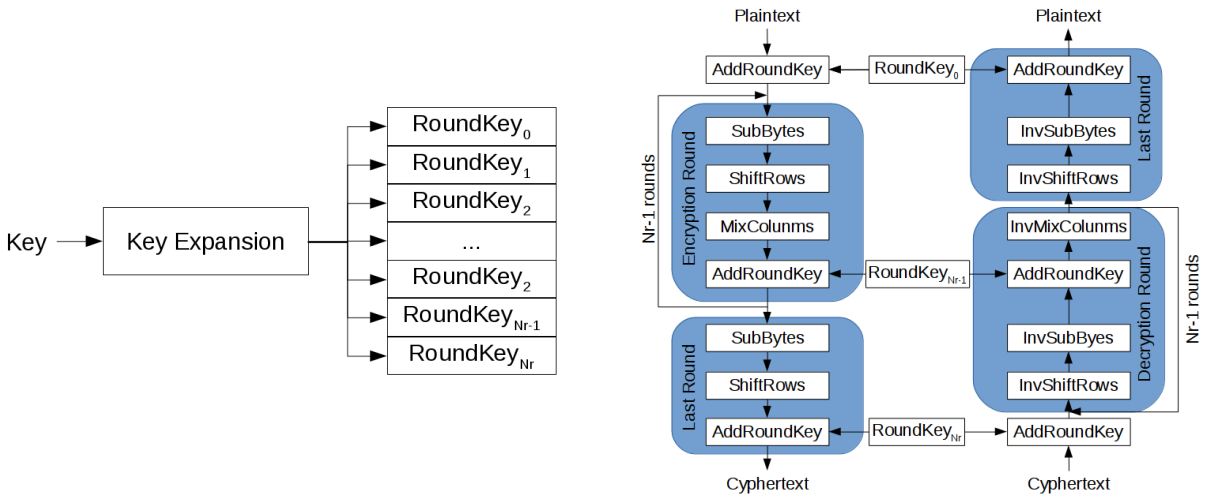


Figure 2.1: Block Cipher Encryption



(a) AES Key Expansion

(b) AES Operation

Figure 2.2: Advanced Encryption Standard (AES)

introduced to against AES [42–47], there are no known practical attacks which are successful in breaking this encryption algorithm.

AES has several modes of operation as described in [48]. Each mode of operation has pros and cons as follows:

Electric Code Book (ECB): ECB is the simplest mode of operation which encrypts every data block with the same secret key. Every data block can be encrypted in parallel which makes ECB is the fastest mode of operation. However, ECB is terribly insecure with a large amount of duplicated information. Since we cannot guarantee that how many duplicate data blocks are stored in ORAM server, ECB is inappropriate to be used in ORAM.

Cipher Block Chaining (CBC): In CBC every encryption uses the same secret key as of ECB. However, to prevent the same issue as occurred in ECB, the input plaintext will be XORed with the ciphertext from previous encryption. The first block of plaintext is XORed with a

random number called initialization vector (IV) and its output ciphertext will be used to XOR with the second block of plaintext and so on. Since the ciphertext is chained from previous ciphertext and IV has been known, the decryption can be done in parallel. The bad point of this mode of operation is the encryption cannot do in parallel since it requires the previous output result to alter the input of a current operation. Furthermore, this mode does not has integrity and authenticity checks. Therefore, it weak against to malleability attacks (e.g. chosen cipher-text attack or chosen plaintext attack). As an ORAM server is semi-trustful and it can arbitrarily change the content kept in its storage. The integrity protection and authentication are necessary to prevent a server altering the information. Therefore we look for another operating mode that supports integrity protection and authentication.

Glois/Counter Mode (GCM): Galois/Counter Mode or in short GCM is a mode of operation of block cipher which provides an authenticated encryption (AE) property on encrypted data. AES-GCM is an altered counter mode of AES which includes GMAC or known as authentication tag for authentication. Data sent to the receiver of AES-GCM consists of four types: authentication data, initialization vector (IV), encrypted data, and authentication tag. Authentication data and initialization vector are a non-encrypted data and publicly known to anyone who can access this information while the authentication tag is calculated from the Galois multiplication and addition of the last block of authentication data, the last block of encrypted data, and the bit length of these two blocks. Authentication tag is a sensitive information which is changed with high probability when authentication data or encrypted data has been modified. By using Galois field multiplication, addition, and a secret key known by both sender and receiver to calculate the authentication tag, AES-GCM decryption will notice if the result is not equal to a received authentication tag. Since AES-GCM supports every feature required by ORAM (e.g. authentication encryption and integrity proctection.), it is used as a mode of AES operation of our research.

2.3 Pseudo Random Access Operation

As the security requirement of ORAM, the accesses requested by a client must not reveal any information which can help the server to identify the client's target data. In other words, the probability of data of interest chosen from the ORAM of the different accesses must be indistinguishable from a server perspective. Therefore, some mathematical definitions and statistical hypothesis test are used as a tool to state and prove this property of ORAM construction. This section, the mathematical definitions and statistical modelling for testing the random behaviour of reading and writing data of interest used throughout the thesis are introduced.

2.3.1 Security Parameter

The security parameter is of the input size of cryptographic algorithm which usually represents in bit length. It expresses the probability that the adversary can break the security. Generally, the security parameter is inversely proportional to the probability value of successful attack by adversary. In other words, if security parameter is large enough, the probability of the security to be broken by an adversary will be negligible.

2.3.2 Negligible Function

In non-perfect secrecy schemes, all of them are insecure if adversary has an unbounded computational power. Therefore, the definition of security against the adversary who has bounded computational power is required. Suppose the adversary's power is bounded to run in polynomial time. There is an algorithm to compute input length n to get an output of polynomial n, where n is the security parameter; and it can be broken by only brute force attacking. This algorithm is insecure if the advisory can guess the correct answer in polynomial time. On the other hand, if the overall success rate of guessing is always less than $\frac{1}{poly(x)}$ where *poly(x)* is every possible polynomial function, the algorithm is secure.

Definition 2. *where $\forall c \in \mathbb{N}$, $\exists n_0 \in \mathbb{N}$, and $\forall n \geq n_0$. A function $\epsilon(n)$, in short ϵ is the negligible function iff:*

$$\epsilon < \frac{1}{n^c} \tag{2.4}$$

2.3.3 Computational Indistinguishability

Computational indistinguishability is the property of the system which is seeing the limited samples of two distributions and cannot separate two apart. In the computational complexity theory, the two series of random distribution are computationally indistinguishable when the differentiation of the probability to guess the right input in polynomial time for those series are equal or less than the negligible function.

Definition 3. *suppose X_n and Y_n are the series of random distribution length n. An $A(x)$ is the function, giving input length n to get output $\{0, 1\}$ in polynomial time. The X_n and Y_n are computational indistinguishable iff:*

$$|[Pr[A(X_n) = 1] - [Pr[A(Y_n) = 1]| \leq \epsilon \tag{2.5}$$

2.3.4 Chi-Squared Test for Randomness

The idea behind the statistical testing of randomness is a comparing an expected value of the reference distribution against the value of the test subject with the same statistical test. Chi-square (χ^2) test is known as the test for discrete distribution of a large number of samples. Chi-squire test is most suitable for testing the random access on the ORAM since the sample of random accesses experiment is the logical address accessed by a client during access operation and it is a vast range. Chi-square test is a statistical hypothesis testing which uses a chi-square as a reference distribution. Two hypotheses are set up which are a Null hypothesis (H_0) and Alternative hypothesis (H_A). H_0 denotes the sequence of samples being tested is random while H_A negates H_0 which denotes the sequence of samples is not random. The result of testing is classified to be either H_0 or H_A according to the p-value assisted by chi-square distribution table. By using the calculation result of chi-square value (Equation 2.6) together with degree of free dome (Equation 2.7) of sequence of samples, the approximate p-value of experiment can be found from chi-square distribution table.

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{2.6}$$

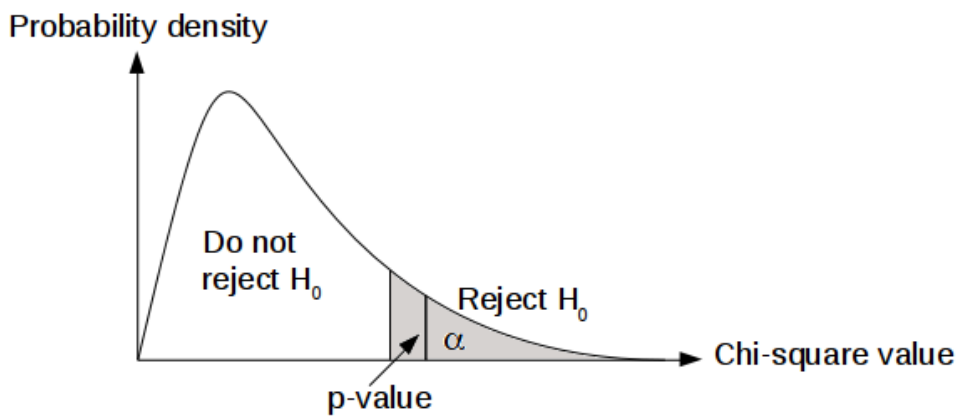


Figure 2.3: p-value which do not reject null hypothesis (H_0)

$$df = n - 1 \tag{2.7}$$

where χ^2 is Pearson's cumulative test statistic, which asymptotically approaches a chi-square distribution, O_i is the number of observations of the group i, E_i is the theoretical frequency of group i, and n is the number of groups of samples.

If the p-value is greater than the significant level (α), the sequence of samples is likely random with regard to H_0 otherwise it is not random (see Figure 2.3). According to *National Institute of Standards and Technology(NIST)* [49], 0.01 is the value of significant level used for measuring.

Chapter 3

Literature and Related work

This chapter describes the fundamental knowledge of existing ORAM construction which will help the reader understand how the ORAM operates. Generally, the existing ORAM constructions are categorised into two types according to the data structure: Hierarchical based ORAM and Binary tree based ORAM. The first ORAM scheme is introduced in the format of hierarchical construction or sometimes known as pyramid construction. The binary tree ORAM is a second generation which is introduced to break the boundary of performance limited by a hierarchical data structure. The data that is stored in both ORAM constructions are in the block format which has an identical size for the same type of block. ORAM has an exact size as a number of blocks stored on a server rather than Bits or Bytes. Although Hierarchical based ORAM and Binary tree based ORAM are similar in some respects of conceptual design to achieve the same goal of security, the operation is different in the details due to their data structure. The detailed construction and operation of hierarchical ORAM and binary tree ORAM are respectively described in Section 3.1 and Section 3.2.

3.1 Hierarchical based ORAM

The original ORAM was introduced by Goldreich *et.al* [1] in a hierarchical data structure (or pyramid data structure). Hierarchical ORAM consists of multiple levels. The most top level has a smallest size while the lowest level is the largest as illustrated in Figure 3.1. Each level contains groups of blocks called *bucket* which every bucket contains the same number of blocks. Although there are many hierarchical ORAM schemes have been introduced since the first construction was presented, they have a very similar operation for retrieving the block of interest. Every bucket of level 1 and one bucket of the rest levels are scanned. A bucket to be scanned of each level 2 to h is carefully chosen to ensure that it seems to be random access. After the block of interest has been retrieved, it is relocated to the top level of the data structure. To prevent the buffer overflow, content from level 1 must be moved down to the next level after a certain number of operations. Nonetheless, the process of randomly choosing a bucket and relocating the block is different in each design. This section four models of hierarchical ORAM are described in details.

3.1.1 Goldreich and Ostrovsky ORAM

The original ORAM introduced by Goldreich and Ostrovsky is usually referred as GO-ORAM model. The data structure of GO-ORAM is illustrated as Figure 3.2. The concept behind the

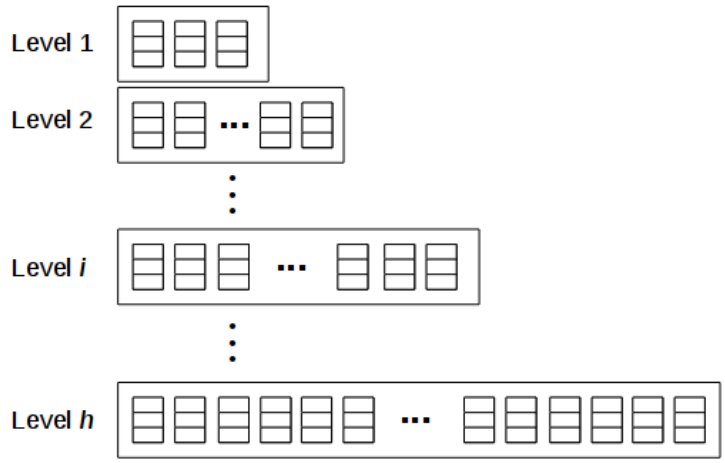


Figure 3.1: Hierarchical ORAM data structure

design is the information which is frequently accessed must be in the buffer which is frequently shuffled. In addition, the buffer which is frequently accessed must have smaller size than the buffer which is not often accessed. Each level is a hash table. Level i where $1 \leq i \leq h-1$ is indexed by random number referred as s_i, and it contains 4^i buckets which are arranged according to the hash value. Level h, on the other hand, it has $4n$ buckets where n is the size of input in the units of block. Each bucket has size $O(\log n)$ blocks. The hash value of each level is computed from $h_{s_i}(v) \mod 4^i$ where v is a virtual address (logical address) of information. In the other words, the information at level i which has virtual address v will be stored in the first empty block of a bucket number $h_{s_i}(v) \mod 4^i$.

To retrieve the data of interest all buckets of level 1 are scanned. If data of interest was found, a random bucket of each rest level is chosen to be scanned. If not, the bucket number $h_{s_2}(v) \mod 4^2$ of level 2 is scanned. If the data of interest is found at level 2, a random bucket of each rest level will be chosen to be scanned. If it is still not found. the process applied for level 2 is used in level 3 and so on. Since the address of data of interest is revealed to a server after retrieving process, data of interest must be relocated by saving to the first empty bucket of buffer level 1. However, after 4 accesses, the level 1 buffer may become full. Therefore, the content from level 1 must be moved to a bigger buffer before it overflows. In Go-ORAM, the content of level i will be moved to level $i+1$ after passing 4^{i-1} accesses. Every block of level i is mixed with the blocks of level $i+1$ then shuffled and saved within level $i+1$ buffer. After the mixing process, the level i buffer is turned to be empty. To do so, the client of GO-ORAM has to perform the extra processes called oblivious sorting and oblivious shuffling. The oblivious sorting and oblivious shuffling are used to hide the empty bucket from server's perspective during the mixing operation.

Suppose client needs to combine the buffer level i and $i+1$ which are respectively referred as A and B. The client creates temporary buffer called C on a server and copies every element from A and B to C. An empty block on C will be marked with 0 and the non-empty block will be marked with a positive number. One dummy data is added to each empty bucket to make sure that the empty buckets are indistinguishable from the other non-empty buckets. A new s is chosen to create the new hash function then a client does online scanning of each bucket of C. For each non-empty block, it is assigned with a tag that corresponds to the new hash function. In addition, for each i from 1 to $4n$ of a dummy block it is assigned with tag value i. After finish scanning process, the oblivious sorting is executed on C. As a result, the blocks

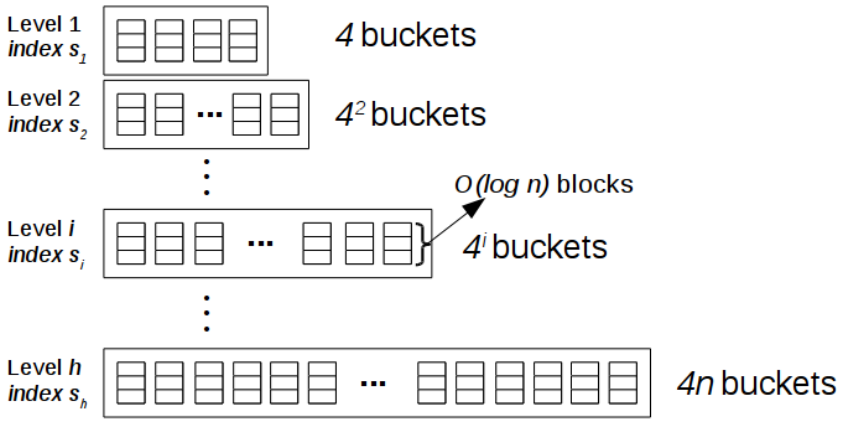


Figure 3.2: Goldreich and Ostrovsky ORAM data structure

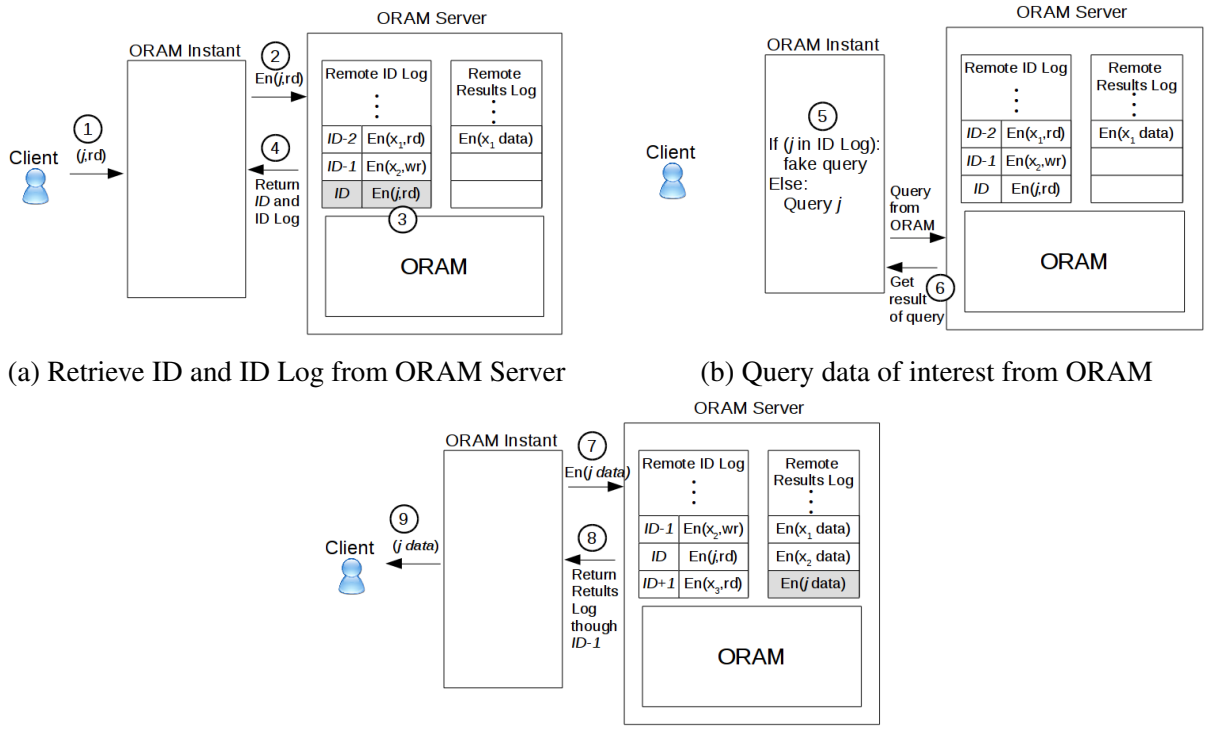
are arranged into two groups: empty block tagged with 0 and non-empty block tagged with an integer number. The client does another oblivious sorting for grouping element with the same tag into the same bucket then copy every element back to B. At the final step, the client does online scanning on B to remove the dummy blocks contained in every bucket.

The disadvantages of GO-ORAM are: most operations are performed on the server side which requires more bandwidth, and it must have empty buckets within ORAM which is about 60% of overall ORAM spaces. Nevertheless, this construction requires only one block space on a client to contain the block of interest. Since most operations are performed as "online", GO-ORAM has $O(\log^3 N)$ asymptotic bandwidth cost with $O(1)$ client storage requirement.

3.1.2 Parallel De-amortized ORAM

Parallel De-amortized ORAM or PD-ORAM [19] was introduced as an additional application of GO-ORAM. The additional modules and protocols are added to improve the efficiency of querying data. One module is introduced: ORAM instant; and two buffers: Remote ID log and Remote Result log, are added. The ORAM instant works as an intermediate between client and ORAM server since the client cannot directly communicate with a server. The aim behind this design is to provide the simultaneous unique queries according to the request of clients. Figure 3.3 shows how ORAM instant operates. Suppose client requests to read j from ORAM. It therefore sends information (j, rd) to ORAM instant. The information (j, rd) means "*read* data j" from ORAM. On the other hand, if it is wr instead of rd, it means *write* data j. Once ORAM instant receives the command from a client, it encrypts that command then appends to a buffer called *Remote ID Log* on the ORAM server as the aid of the sequence number in Figure 3.3a. After the encrypted command was appended, ORAM server sends the IDs together with their corresponding encrypted content back to ORAM instant. After ORAM instant received the content of Remote ID Log, it checks to see whether or not the required data is being queried. If it is, the dummy query for random data is generated. If not, ORAM instant retrieves content of j from ORAM (see Figure 3.3b). After the content of j was retrieved, it is appended to Remote Result Log in ORAM server. ORAM server sends the information within Remote Result Log back to an ORAM instant then the ORAM instant returns decrypted content of j back to a client. Based on past descriptions, ORAM instant can produce simultaneous queries instead of having to finish one by one. Hence it also can handle multiple requests from clients at the same time.

Besides of introducing the new module, ORAM instant; PD-ORAM also presents the new



(a) Retrieve ID and ID Log from ORAM Server

(b) Query data of interest from ORAM

(c) Save retrieved data content in Results Log

Figure 3.3: PD-ORAM retrieving operation

solution when two consecutive levels of ORAM have to be merged, called *Level-based deamortized ORAM*. It provides the new feature which the data can be queried even though it is being merged with another buffer level. The deamortization means reducing the worst case query cost to average case. To do so, the special read-only buffer is used. Suppose the original ORAM contains information as the Figure 3.4. Since a level 3 buffer is full, it needs to be merged with level 4 buffer. Rather than using the normal process as GO-ORAM, PD-ORAM creates a read-only copied version of buffers which will be merged then empties a level 3. Since the query can do on the copy version buffer, the merging and shuffling can be processed on a buffer level 4 in the meantime. Therefore, PD-ORAM construction can reduce the worse cast cost of the query to an average amortize cost. PD-ORAM requires $O(\log^2 N)$ client storage requirement and $O(\log N)$ bandwidth cost.

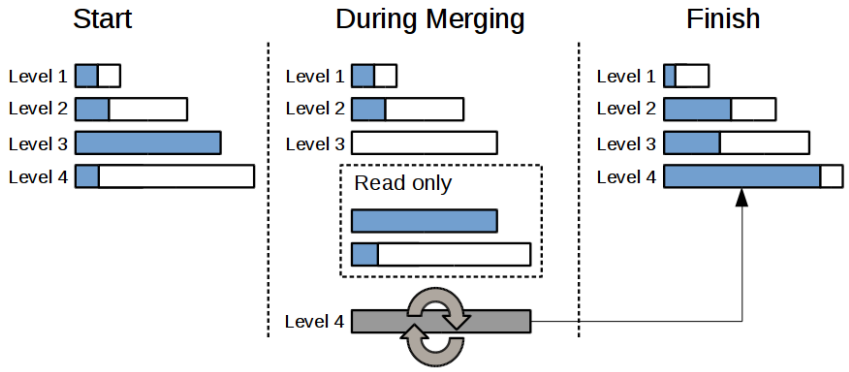


Figure 3.4: PD-ORAM's merging and shuffling operation

3.1.3 SSS-ORAM

The partition ORAM or SSS-ORAM was introduced by Stefanov *et al.* [3]. The SSS-ORAM is designed by using hierarchical ORAM as its data structure. Instead of using single ORAM, ORAM of SSS-ORAM is separated into several partitions to reduce the number of blocks accessed and shuffled during read/write operations. Suppose the total capacity of ORAM on the server is N blocks (Figure 3.5), there are $\sqrt{N}$ partitions of ORAM which each contains about $\sqrt{N}$ blocks. On the client side, there are three types of storage: cache slot, shuffling buffer, and position map. Cache slot uses to temperately contain block downloaded from ORAM which the number of slots is equal to the number of partitions. The blocks within cache slot p will be uploaded to partition p during eviction operation where $1 \leq p \leq \sqrt{N}$. Shuffling buffer is $\sqrt{N}$ blocks buffer and will be used when the two levels of partitioned ORAM are needed to be merged into next bigger level. Position map contains the partition number where the data blocks reside. There are 2 types of operation performed by SSS-ORAM which are the operation for access information and the operation for data eviction from cache slot to a partitioned ORAM. An access operation consists of reads and writes, while an eviction operation consists of 2 different types called Piggyback eviction and Background eviction. The piggyback eviction is executed as a regular operation for every access request. Suppose a data block was retrieved from partition p, a piggyback eviction evicts one data block from cache slot p then writes to partition p at that time. On the other hand, a background eviction is executed independently from data access request. It is fixed proportionally to the data access rate. In the other words, the eviction rate of ν means that there are expected ν number of background evictions are attempted with every data access request.

Once a client wants to access (either read or write) a data block on the ORAM server (see Figure 3.6), the partition number of data of interest is retrieved from a position map. If it is in the cache slot, the client retrieves it from a cache slot then generates the dummy read operation and sends to a server. Otherwise, the client will retrieve the information from a server according to the partition number that has been retrieved. After the block of interest was downloaded, it

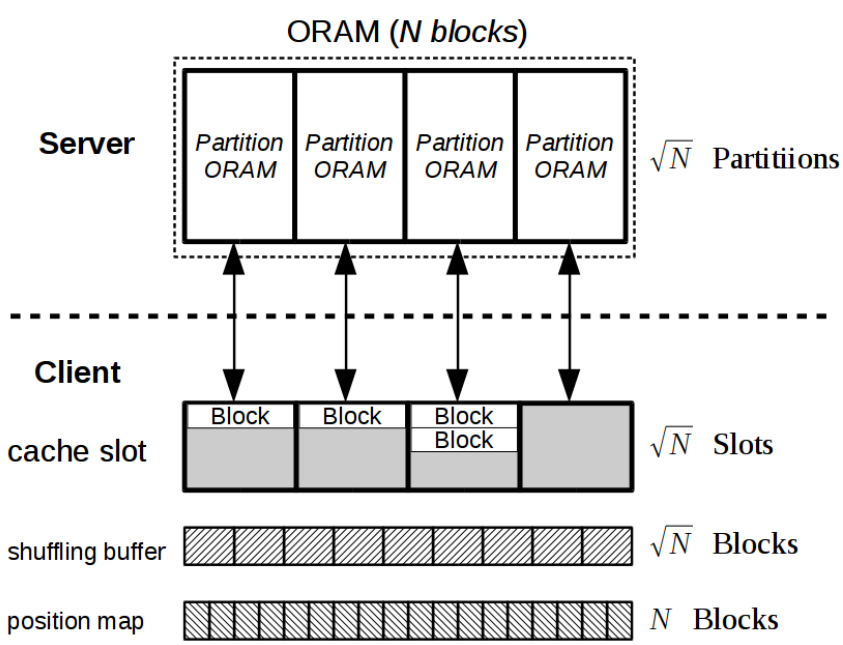
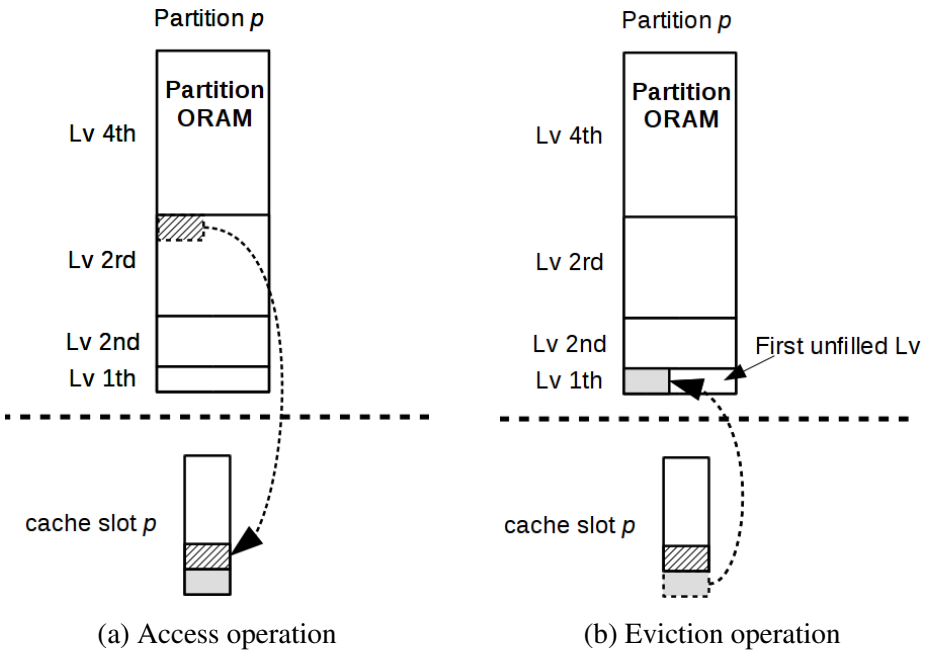


Figure 3.5: SSS-ORAM storage structure



(a) Access operation (b) Eviction operation

Figure 3.6: SSS-ORAM access and eviction operation

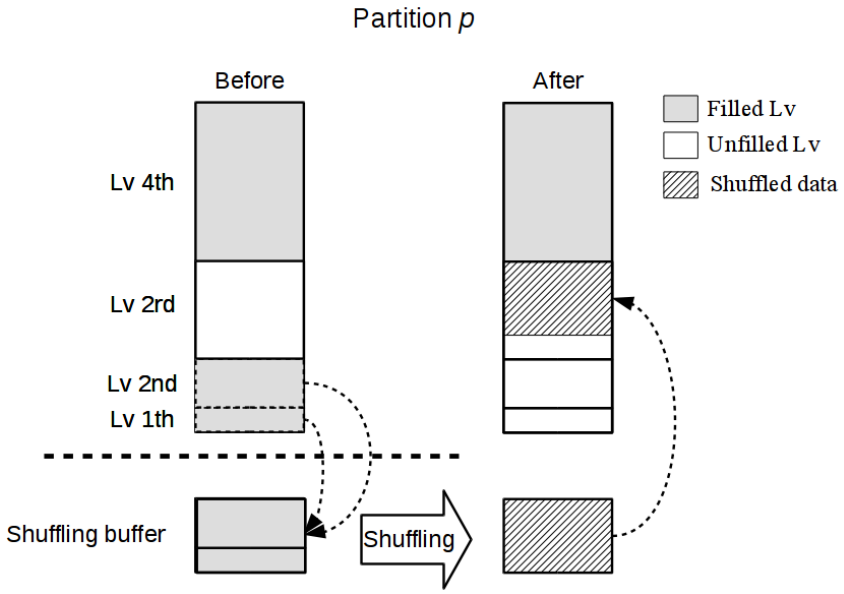


Figure 3.7: SSS-ORAM shuffling operation

is randomly appended to one of cache slots within the client. If the client needs to update the information (write operation), old content will be replaced by the updated content at this point. The appended block is located in the cache slot until it is selected to be uploaded by background eviction or piggyback eviction process. As an eviction process is executed; if it is a piggyback eviction, the most top block within the cache slot associating with a partition that was accessed will be uploaded to ORAM server. Otherwise, the most top block from random cache slot will be chosen to be uploaded if it is a background eviction. The uploaded block goes to the first unfilled level of a partitioned ORAM.

Since SSS-ORAM uses GO-ORAM as its ORAM storage, some ORAM levels will be full

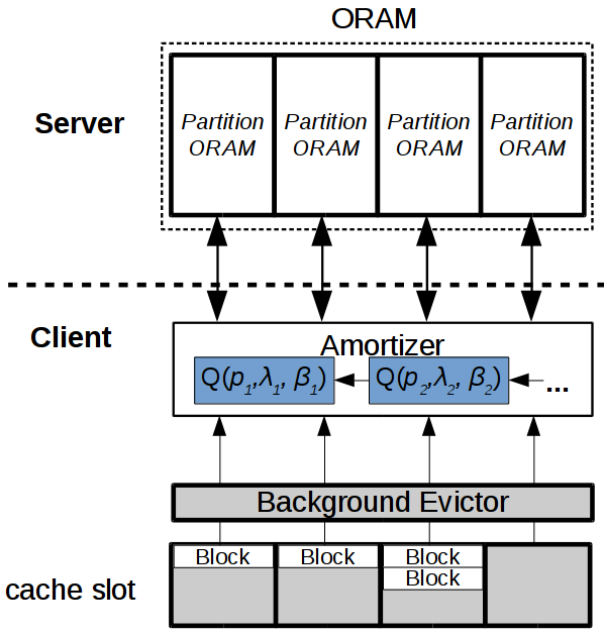


Figure 3.8: SSS-ORAM Amortizer

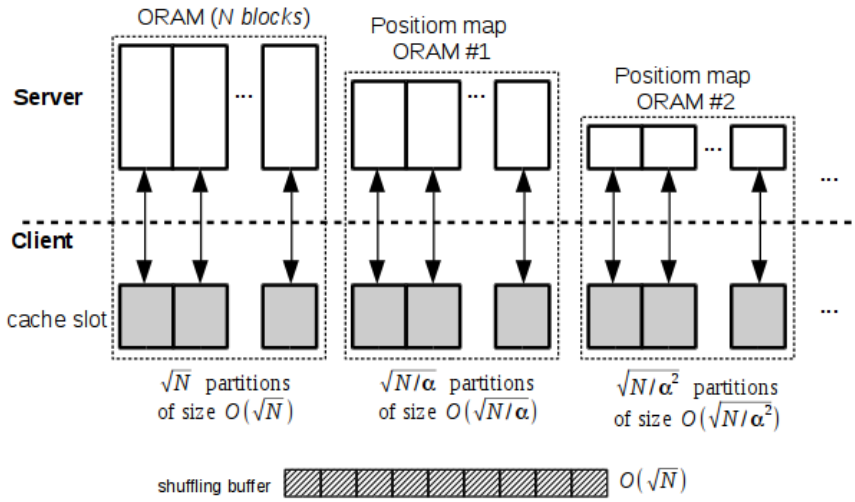


Figure 3.9: Recursive SSS-ORAM

at some point. Therefore, a possible filled level must be merged and saved to the next bigger level. To do so, the blocks from consecutively filled levels of the same partition are downloaded to shuffling buffer as shown in Figure 3.7. The block locations are permuted then uploaded to the first unfilled level of the same partition. As all elements from filled level are downloaded and uploaded to another bigger level, all of that levels are marked as unfilled. This scheme costs an $O(\sqrt{N})$ worst cast bandwidth cost because sometimes it needs to reshuffle $O(\sqrt{N})$ blocks. To reduce the worst cast cost to an average cost, the amortizer is introduced as an additional module for SSS-ORAM.

The amortizer of SSS-ORAM is a queue management module merging the existing job with the new job that is being operated in the same partition. Let $Q(p, \lambda, \beta)$ represents a process of reshuffling of level 1 to λ of partition p then writing the blocks in β to partition p on a server,

and Figure 3.8 illustrates the details of amortizer. An amortizer is a FIFO list of jobs which a job at the head of the queue will be processed first and the job at the tail will be the last one to be processed. Once the new job occurs, it is added at the tail. Merging will happen when the different jobs require shuffling the same partition. Suppose there are two different jobs referred as $Q(p, \lambda, \beta)$ and $Q(p, \lambda', \beta')$. These two jobs require to access and shuffle the same partition p. The amortizer merges two jobs to be $Q(p, \lambda'', \beta'')$ where $\lambda'' = max(\lambda, \lambda')$ and $\beta'' = \beta \cup \beta'$, and saves it to the older job queue. By using amortizer, SSS-ORAM can reduce the worst cast cost of bandwidth from $O(\sqrt{N})$ to $O(\log N)$.

In addition, an alternate construction called recursive SSS-ORAM also was introduced to reduce a storage requirement for position map from $O(\sqrt{N})$ to $O(1)$. Instead of storing all position map on a client, most of the position map is stored on the server within the buffer called position map ORAM. Position map ORAM is guaranteed to be a constant factor smaller than the ORAM. As illustrated in Figure 3.9, the size of position map ORAM is reduced by a constant factor according to the number of levels of recursion. Hence, after some number of recursions, the size of position map stored on the client is reduced to $O(1)$. However, it still requires $O(\sqrt{N})$ client storage for shuffling buffer and cache slot. In addition, the amortize bandwidth cost also increase from $O(\log N)$ to $O(\log^2 N)$ in recursive construction.

3.1.4 Burst ORAM

Bust ORAM [14] is designed based on SSS-ORAM by adding the process and resource management module into an ORAM client. The design aims to optimise the response time between two consecutive requests of retrieving a data of interest. To do an optimisation, three solutions are introduced: process scheduling, block transfer compression, and level caching on the client. In burst ORAM a block transferred between a client and server referred as *IO*. There are two majority types of IO affecting to an overall performance: online-IO and offline-IO. Online-IO is the number of blocks transferred before an access operation is satisfied by ORAM security requirement. On the other hand, offline-IO refers to the number of blocks transferred for an extra operation in order to guarantee the privacy of future access (e.g. shuffling and merging operation).

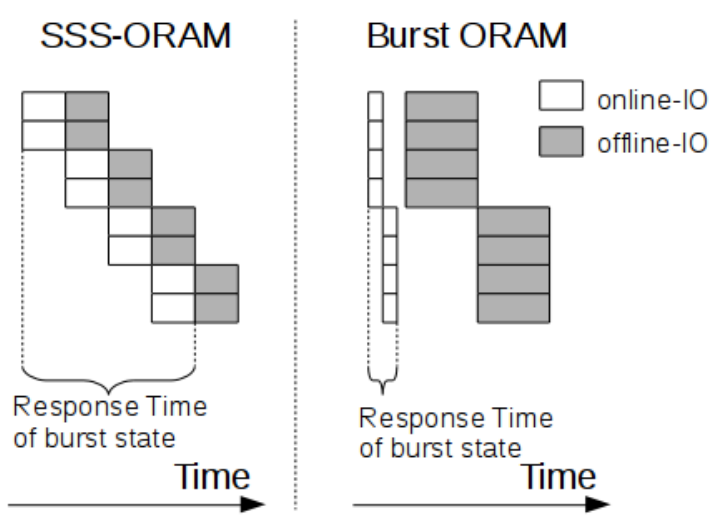
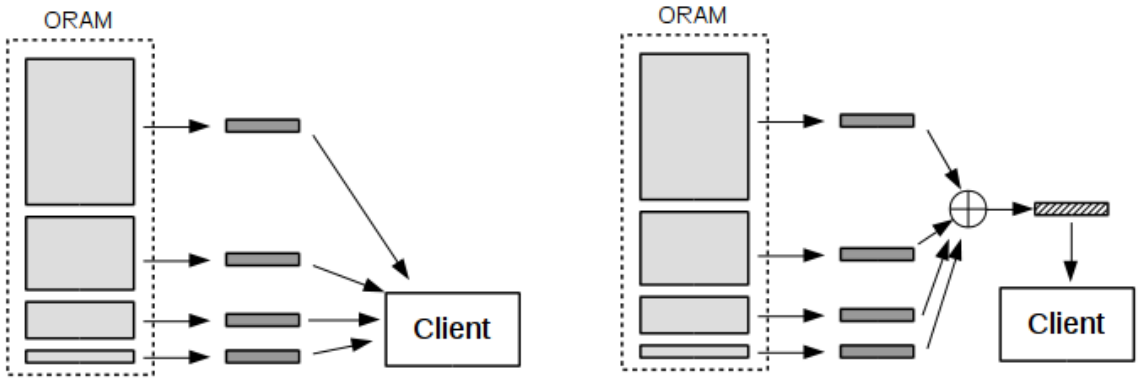


Figure 3.10: Process scheduling comparison between SSS-ORAM and Burst ORAM



(a) Normal process (b) XOR compression

Figure 3.11: Normal process and XOR compression of sending selected blocks to client

Recall from the three optimise solutions, process scheduling is used to prioritise the online-IO and offline-IO to optimise the processing time. Semaphores algorithm is used to control the flow of operation. As illustrated in Figure 3.10, rather than the pair of online-IO and offline-IO are performed one by one as of SSS-ORAM, burst ORAM delays the offline-IO to until the idle periods which is after complete every online-IO in the queue. The online-IO of burst ORAM dramatically consumes less bandwidth cost since the block transfer compression named as XOR compression is applied. Although XOR compression causes an offline-IO to slightly consume more bandwidth than SSS-ORAM, the overall bandwidth consumption of burst ORAM is still lower.

XOR compression is the technique that allows ORAM server to combine the blocks downloaded during access operation into a single block. As there is only one block containing the data of interest, the rest can be any block stored in the different ORAM levels. However, those blocks must contain the dummy content which is known by a client. To compress the blocks being retrieved to a single block, ORAM server XOR every block together. Since a client knows the dummy content and also knows the encryption key of every data, it is very easy to extract the data of interest from a compressed content. By using the XOR compression technique, burst ORAM can reduce the online-IO from $O(\log N)$ to $O(1)$ as illustrated in Figure 3.11.

Level caching is another feature which is introduced in burst ORAM. By spending space to cache the smallest level of each partition, the client can eliminate the shuffling operation of that level which reduces the offline-IO of the system. However, caching means increasing space requirement on the client. According to the experimental result from [14], although burst ORAM has overall practical bandwidth overhead greater than SSS-ORAM, its response time of operation is significantly smaller than SSS-ORAM.

3.2 Binary tree based ORAM

Hierarchical based ORAM is limited by it data structure to maintain the privacy of download and upload data of interest. Since each level of hierarchical needs to be moved to next bigger level before it overflows, it causes an entire block of that level must be accessed and mixed with the blocks stored in the next level. It is hard to reduce the worst cast bandwidth cost to under $O(N \log N)$ while still maintain client storage requirement at $O(1)$. The new ORAM data structure is introduced base on binary tree format to achieve the lower worst case bandwidth

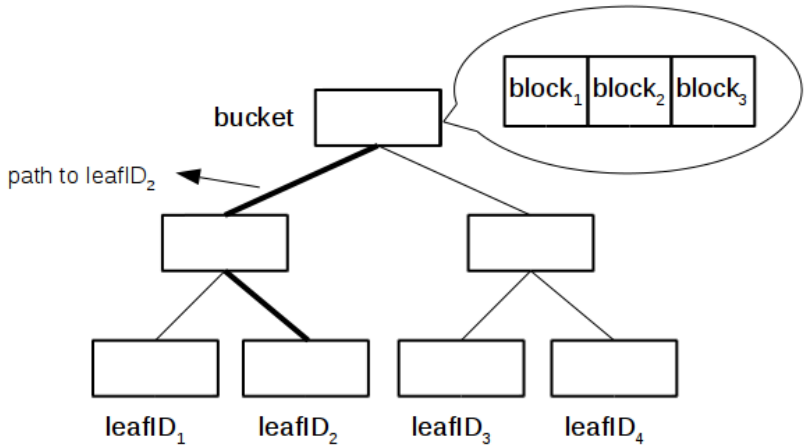


Figure 3.12: Binary tree ORAM

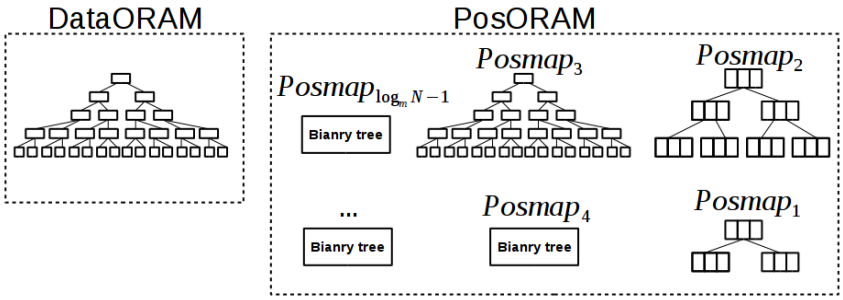


Figure 3.13: Recursive Binary tree ORAM

cost than hierarchical ORAM construction. Binary tree based ORAM is first proposed by Shi *et al.* [2] which ORAM is constructed in the perfect binary tree format as illustrated in Figure 3.12. Each node is referred as *bucket* which consists of many blocks inside. The most top level is called *root level* and the bottom level is called *leaf level*. Each bucket in the leaf level has a unique identification number called *leafID*. By the reason of there is only one path from root node to a leaf node of binary tree data structure, leafID is therefore used to determine the buckets accessed through from root to leaf node. To be more precise, the bold line of Figure 3.12 connects the bucket which associate with $leafID_2$. This bold line is called "path of $leafID_2$", and every bucket through this path will be accessed during access operation when $leafID_2$" is referred.

Some of binary tree ORAM constructions also provide their recursive version such as [2] and [4]. ORAM of position map is constructed in the binary tree format as well as the ORAM which contains the data as illustrated in Figure 3.13. DataORAM represents binary tree ORAM containing data of client while PosORAM represents a group of binary tree ORAM called Posmap which contains position map stored on the server. Most of the position map is stored on the server which each points to a block within either another Posmap or DataORAM. The size of consecutive Posmap is factored by some constant value (m) which Posmap_{i+1} is bigger than Posmap_i. Posmap_0 is kept on the client. Therefore, Posmap_0 has a constant size ($O(1)$) for every recursive binary ORAM. The number or Posmap is equal to the number of levels of recursion which equals to $\lceil \log_m N \rceil$, and recursive operation of binary tree ORAM is same as of SSS-ORAM.

This section three models of binary tree ORAM are given as an example to well understand how they operate. The details of their operation and construction are described in the following sections.

3.2.1 Shi Binary tree ORAM

Shi binary tree ORAM (or in short Shi ORAM) is the first ORAM represented in binary tree data structure format. It can achieve the worst case bandwidth cost equal to $O(\log^3 N)$. Each data block in the bucket is associated with a leafID which is used to verify the path where the data block resides. The storage of Shi ORAM can be categorised into ORAM on the server and position map on the client. In non-recursive version, with N data blocks ORAM, the position map needs N blocks to contain the leafID of each data block within an ORAM. To do an access operation, a client has to check the leafID of block of interest from position map. A path of bucket being accessed is created from the leafID which has been retrieved. Every bucket along to the path is scanned from leaf to root level by every block in the bucket will be read and written. If the block having read is not a data of interest, the same content encrypted by new secret key will be written. On the other hand, if it is the data of interest, the content is remembered by a client then the encrypted dummy is written rather than the real data. Read and write operation will be proceeded until the last block of the path although the data of interest is already found in the middle of scanning. After data of interest was retrieved, its new associated leafID is randomly assigned then updated in the position map. This data is re-encrypted by a new secret key then uploaded to the first empty block of root bucket. As the same manner of hierarchical ORAM, the root bucket will eventually overflow if the blocks inside the bucket are not evicted to the bucket of next lower level.

The eviction operation of Shi ORAM is different from hierarchical ORAM. The eviction process occurs every access operation. Instead of blocks of entire level are moved down and mixed up with the blocks of its next lower level as hierarchical ORAM, only 2 buckets of each level are randomly chosen. Each block is pushed down to either left or right child bucket (see Figure 3.14) depending on its leafID. If the block is pushed down to right child bucket, the dummy content is written to the left child bucket and vice versa. It produces the equal probability of moving real data block to left or right child.

Since Shi ORAM is constructed in binary format and requires only 2 buckets per each level to be shuffled, it requires $O(\log^2 N)$ accesses during eviction operation with the low probability that any bucket will be overflowed. Furthermore, in Shi ORAM recursive version, the bandwidth cost is multiplied by logarithmic of N since it needs $\lceil \log_m N \rceil$ rounds of recursion. Therefore, the bandwidth cost of Shi ORAM is $O(\log^2 N)$ and $O(\log^3 N)$ in normal construction and recursive construction, respectively; while it requires $O(N)$ blocks on a client for normal construction and $O(1)$ blocks for recursive construction.

3.2.2 SE-ORAM

SE-ORAM [25] stands for "Storage-Efficient Oblivious RAM". The goal of this design is to reduce the storage overhead in the ORAM server. According to Shi binary tree construction, dummy blocks are used to maintain the oblivious moving of data block with high probability during eviction operation. Suppose Shi ORAM contains n real data blocks, it requires $n \log n$ dummy blocks as the storage overhead cost. SE-ORAM also requires dummy blocks for its eviction operation; however, the number of required dummy blocks is equal to the number of

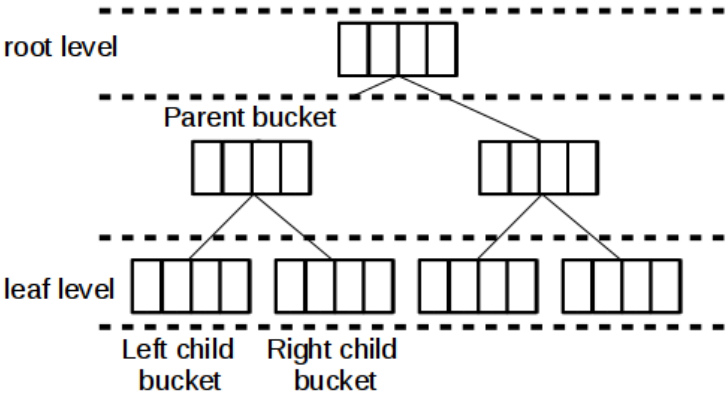


Figure 3.14: Shi binary tree ORAM structure

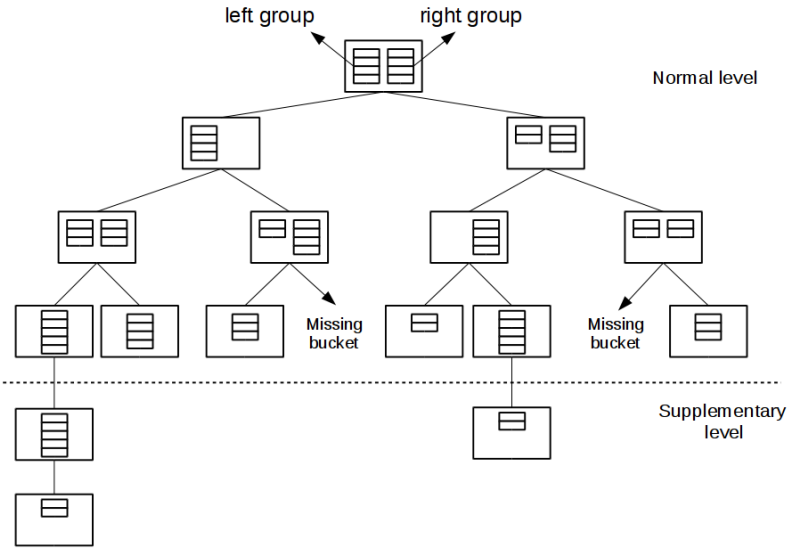


Figure 3.15: SE-ORAM structure

real data blocks cached on the client. In other words, there is zero overhead cost of storage usage on SE-ORAM server. SE-ORAM is a binary tree based ORAM construction. Although it is a balanced tree at the very beginning, it is not necessarily arranged in the format of a balanced tree after passing through several eviction operations as illustrated in Figure 3.15. Some buckets might be missing from the binary tree and some supplementary buckets might be appended to the existing leaf bucket. The blocks within each bucket are divided into two groups with equal size referred as *left group* and *right group*. The client contains 2 types of buffer called cache buffer and position map buffer. Cache buffer is used to temporarily store data block remained from eviction operation. Position map buffer is used the same way as Shi ORAM which keeps the leafID of each data block stored on the server.

To retrieve the data of interest, a client first looks in a cache buffer. If there is, data is retrieved from the cache. Otherwise, the client retrieves a leafID of data from position map buffer. One of two solutions for retrieving data block is used depending upon the current binary tree format. If the leaf bucket is currently in the tree, all buckets included supplementary buckets are accessed. Otherwise, all buckets within the longest path which has most overlap buckets

will be accessed. The scanning starts from the farthest bucket from the root, and all blocks within the bucket are downloaded to a client. If the block of interest is among the downloaded blocks, the client keeps this block then re-encrypts the rest of blocks and uploads them back to a server. If the block of interest is not in there, the client selects an arbitrary block from the group of retrieved blocks and keeps it instead. After retrieving, the bucket will be removed from tree construction if it is empty. For simplicity, B_i and B_a are denoted to a block of interest and an arbitrarily selected block, respectively. After all blocks from the farthest bucket were completely uploaded back to a server, all buckets of the next upper bucket are downloaded. If B_i is among the downloaded blocks, B_i is kept by a client, then the rest of blocks together with B_a are re-encrypted and uploaded back to a server. Otherwise, all blocks are re-encrypted then uploaded to a server. This process continues until all buckets along the selected path are accessed.

To upload the data of interest back to a server, a random path is selected. The new leafID of that path is assigned to B_i and updated to the position map buffer. All blocks within the path are accessed, downloaded, and mixed with B_i to create an oblivious upload. Those blocks are re-encrypted then uploaded back to a server. Since all blocks are rearranged, some blocks may be moved from their original location. However, the movement should ensure that the uploaded blocks are stored in the bucket which can lead to its leafID. The block that cannot fit in the buckets will remain in the cache buffer. The access starts from root level towards leaf level. Every block within the root bucket is downloaded to the client. B_i is added to either left group or right group according to its associated leafID then the next bucket which is being accessed is randomly chosen. The next bucket to be accessed is a child bucket of the current bucket. The number of blocks on the left and right group of a current bucket affects the random selection of the child bucket. The child bucket on the same side of a group which has more number of blocks has more chance to be chosen than another. A block of the chosen group is randomly chosen and moved toward to the next bucket which will be accessed. Let B_e be the chosen block to be evicted of current tree level and B'_e will be the chosen block to be evicted of next tree level. There are 4 possible cases to move data from current bucket to its next lower bucket as follows:
Case1: If B_e is dummy block, $B'_e = B_e$.
Case2: If B_e is real data block and there is at least one block in the selected group, a block from selected group is randomly chosen as B'_e and it is replaced by B_e.
Case3: If B_e is real data block and there is no data block but only dummy blocks contain in the selected group, selcted dummy block is replaced by B_e and B'_e = dummy block.
Case4: If B_e is real data block and there is no any blocks contained in the selected group, dummy block is generated to be B'_e, and B_e is stored in the cache buffer on the client.

If the bucket at leaf level is full, the supplementary bucket will be created and logically connected to the leaf bucket. In addition, if the supplementary bucket connecting with leaf bucket is also full, the new supplementary bucket will be created and connected to the current supplementary bucket and so on. After this process finish, every block of the bucket being accessed is re-encrypted and uploaded back to the server. This procedure will continue until no bucket can be selected. Since each bucket of SE-ORAM contains $O(\log N)$ data blocks, the communication overhead is $O(\log^2 N)$. Although SE-ORAM has communication overhead equal to Shi ORAM, it has client storage overhead greater than Shi ORAM. SE-ORAM requires $O(\log N)$ client storage while Shi ORAM requires only $O(1)$. However, considering the overhead cost spending on server storage for oblivious shuffling operation, SE-ORAM has zero overhead for an extra block while Shi ORAM has to spend $O(N \log N)$ blocks for keeping dummy information.

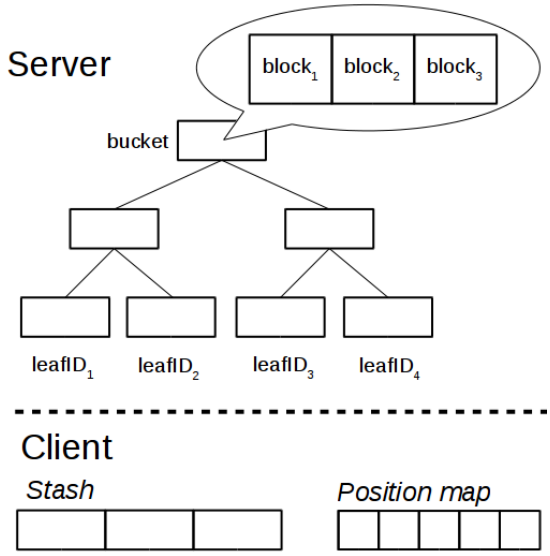
3.2.3 Path ORAM

Path ORAM [4] is a binary tree based ORAM which is adapted from Shi ORAM. Its construction is similar as Shi ORAM as a balanced binary tree, and leafID is also used for tracking the path which contains the data of interest. However, the process of retrieving a data of interest from ORAM server is different. Shuffling and evicting data block from a level to another level of Shi ORAM are required to secure the privacy of information stored on the ORAM server. It incurs of large communication and calculation overhead. Contrary to Shi ORAM, Path ORAM can achieve the security conditions of ORAM by using only simple download and upload operation rather than using extra operations such as of Shi ORAM. Therefore, Path ORAM has extremely low communication and calculation overhead compared with other ORAM schemes.

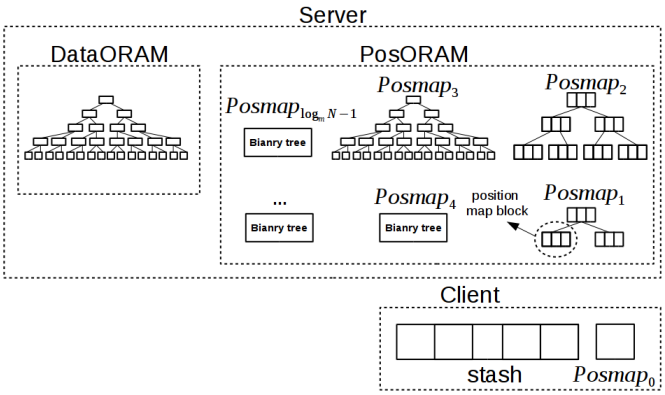
Path ORAM is introduced in 2 versions: non-recursive and recursive construction. In non-recursive Path ORAM, it consists of three types of storage: stash, position map, and ORAM; as demonstrated in Figure 3.16. Stash is used to temporarily store downloaded data block during an access operation. Position map stores leafID of data block kept in the ORAM. ORAM stores the data blocks owned by a client. While the recursive version, an extra ORAM for keeping position map is added to the server. Therefore, to clarify of reading, the ORAM keeping data block and position map block are respectively referred as *DataORAM* and *PosORAM*. PosORAM consists of multiple groups of position map block arranged in binary tree format referred as $Posmap_i$, where $i = \{x \in \mathbb{Z}^+ | x \geq 1\}$. $Posmap_0$ is stored on the client which actually works like a position map of non-recursive version. Rather than using to store N leafIDs of N data blocks as non-recursive version, Posmap_o of recursive Path ORAM stores only one position map block which contains m leafIDs of m position map blocks stored within Posmap_1. By reason of every position map block has a same size which contains m leafIDs, Posmap_1 contains m^2 leafIDs of m^2 position map blocks within Posmap_2 and so on. As there are N data blocks stored in DataORAM and size of consecutive Posmap is multiplied by m, the system needs $r = \lceil \log_m N \rceil - 1$ rounds to retrieve the leafID of data block from PosORAM. Hence, there are roughly $\lceil \log_m N \rceil - 1$ Posmap stored in PosORAM.

The method of retrieving the data of interest of two Path ORAM versions is slightly different. For ease of understanding, the operation of two constructions is described in two following sections by supposing N data blocks stored in a binary tree ORAM of height h.

Access operation of Non-recursive Path ORAM The two basic operations of download/upload data from/to a server are treated as a single *access operation* in Path ORAM. The idea is that whenever the client wishes to access data of interest either read or write/update, it must actually download multiple blocks and then upload multiple blocks. Every block to be uploaded is re-encrypted by random secret key to ensuring that the uploaded content is different from when it was downloaded. To make it easy to explain, a simplified Path ORAM with one block per bucket is used as an example as illustrated in Figure 3.17. Suppose in the beginning of access operation, there are 7 blocks stored in the ORAM which have data ID from a through g which each is associated with one of four leafID $\in \{l_1, l_2, l_3, l_4\}$, and there is a block f remaining in the stash. Each block is associated with leafID as shown in Figure 3.17a. It is actually that the leafID is stored in the position map. However, for clarifying the explanation, leafID is displayed next to its associated data ID. Let b be the data interested by a client. Since block b is associated with l_2, the access path from root bucket to leaf bucket l_2 is created by a client which is illustrated as a bold line. Every block along the path is downloaded and kept within a stash as shown in Figure 3.17b. Only leafID of data of interest is re-chosen by random from $\{l_1, l_2, l_3, l_4\}$.



(a) Non-recursive Path ORAM



(b) Recursive Path ORAM

Figure 3.16: Path ORAM storage structure

Suppose l_4 was chosen and associated with block b as Figure 3.17c, the new leafID is updated to a position map. At this point, the content within b can be updated if it is necessary. At the final state of access operation (Figure 3.17d), a client tries to upload the data stored in stash back to a server at the same path which was previously accessed. Block by block will be uploaded to the suitable bucket according to its leafID. The block can be only uploaded to the bucket which is in a path from root bucket to the bucket marked by its associated leafID. The upload starts from leaf bucket through to the root bucket. The block will be checked one-by-one to determine the suitability for the bucket which is being accessed. If there is no suitable block, the dummy block is generated then uploaded. There might be some blocks left in stash after finish access operation (block b in this case). These block will be remained in the stash and have a chance to be uploaded again when the next access operation is established.

Notice that, the shuffling and merging data block between the different buckets of Path ORAM is automatically done during searching the suitable block to be uploaded to the bucket. Therefore, Path ORAM does not require any extra shuffling or merging operation as the other existing ORAM schemes.

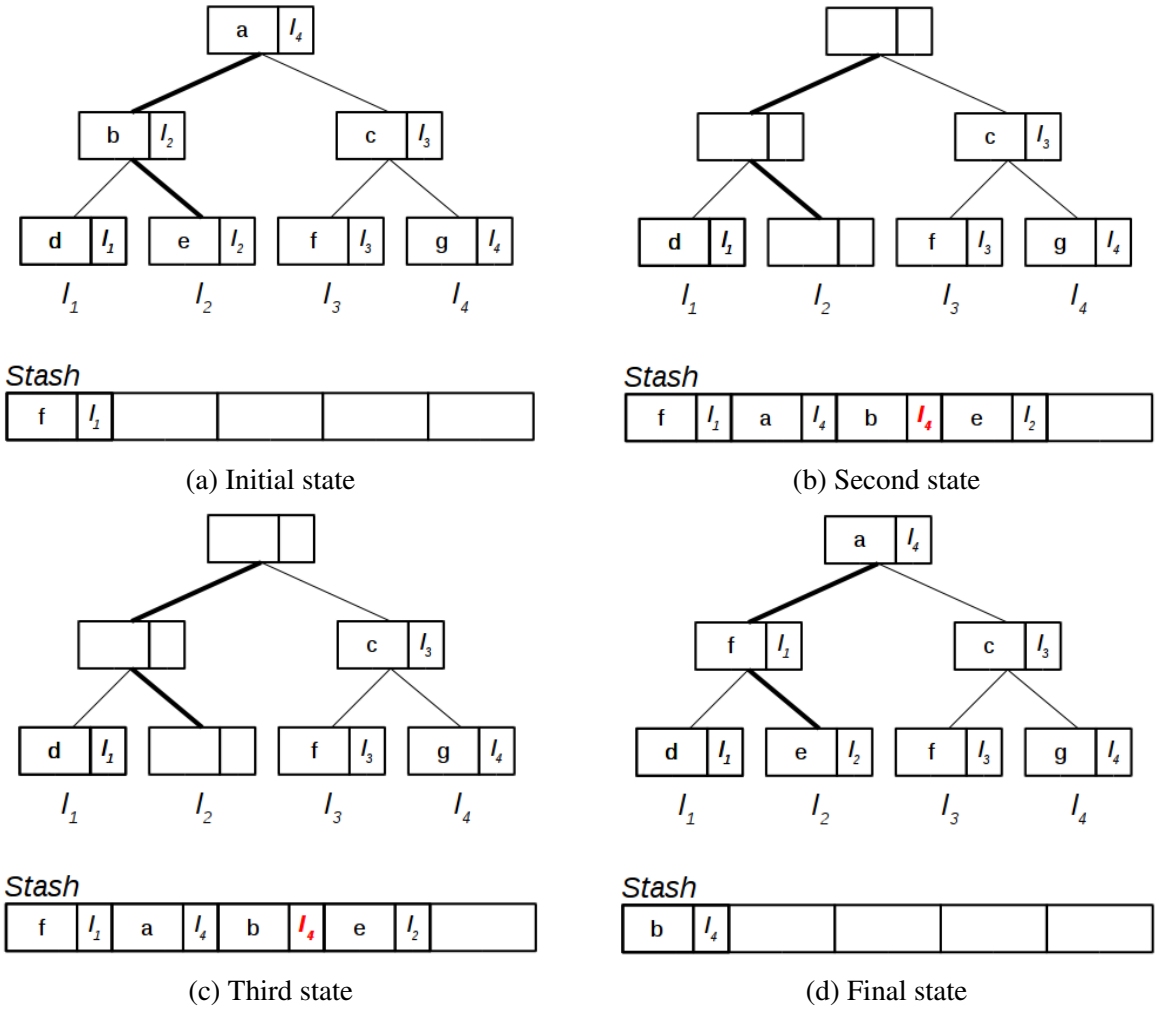


Figure 3.17: Path ORAM access operation

Access operation of Recursive Path ORAM Access operation of recursive Path ORAM is slightly different from the non-recursive version. By the fact that most of the position map blocks are stored on the server rather than on the client as the normal version, therefore an extra ORAM called PosORAM is added to the system. As described in the beginning of Section 3.2.3, an access operation consist of $\lceil \log_m N \rceil$ downloads and uploads. It requires $\lceil \log_m N \rceil$ downloads to retrieve the data of interest from Data ORAM and requires $\lceil \log_m N \rceil$ uploads to complete the oblivious access operation. The group of elements from each download is stored individually in the stash and it will be uploaded back to the Posmap that it was retrieved. The download/upload procedure of each binary tree ORAM (Posmap and DataORAM) is the same as it is processed in non-recursive version. The leafID of interested data in Posmap_{i+1} is retrieved from Posmap_i one-by-one until leafID of data of interest is retrieved from Posmap_r, where $r = \lceil \log_m N \rceil - 1$.

By the fact that Path ORAM relies on the binary tree construction, the non-recursive Path ORAM incurs $O(\log N)$ bandwidth cost. Furthermore, in recursive Path ORAM, it needs $\lceil \log_m N \rceil$ downloads/uploads to retrieve the data of interest from DataORAM. Hence, it incurs $O(\log^2 N)$ bandwidth cost for recursive Path ORAM. According to the proof which has been given in [4], both construction requires $O(\log N) \cdot \omega(1)$ client storage with probability that stash will overflow is $N^{-\omega(1)}$.

3.3 Summary of Overhead Cost and Existing ORAM's Limitation

Both of hierarchical and binary tree based ORAM have a different procedure to generate the oblivious access pattern seen by the server. As using the different procedures, different ORAM algorithms incur of different overheads spent in their system. Table 3.1 shows the overhead cost of different ORAM constructions in three aspects: bandwidth cost, client storage overhead, and server storage overhead. Bandwidth cost and client storage overhead are detailed in asymptotic cost while server storage overhead is shown as a requirement of dummy block stored on the server. Both ORAM constructions have a common limitation which is an ORAM data structure since hierarchical and binary tree ORAM has to access at least one bucket on each level of their logical storage construction in order to keep every access indistinguishable with high probability. Each bucket of hierarchical ORAM is designated to contain $O(\log N)$ blocks, and binary tree ORAM has $O(\log N)$ buckets along the path which has to be accessed. Therefore, the existing ORAM algorithm has $\Omega(\log N)$ asymptotic bandwidth cost. Furthermore, for the sake of completeness of ORAM operation, dummy data is required to be used one way or another. It is the cause of wasted space on the ORAM server. To break the limitations, there are two possible solutions: design the new ORAM data structure format which can achieve the bound of overhead cost lower than $O(\log N)$, or introduce the new algorithm used under the existing ORAM data structure which can achieve better performance. The first solution is designated to be our research with two following reasons.

First, the bandwidth cost is difficult to make lower than $O(\log N)$ under the existing ORAM data structure format. Hierarchical based ORAM is designed to have a bucket with logarithmic of N capacity. It requires at least $O(\log N)$ bandwidth cost to download the data of interest. For Binary tree ORAM, the number of buckets accessed during access operation is equal to the height of construction. The height of binary tree is logarithmic of the number of nodes, therefore it has to spend at least $O(\log N)$ for transfer the information.

Second, dummy information is difficult to be removed from the ORAM construction, since

Table 3.1: Performance comparison of existing ORAM schemes

Scheme	Client Storage Overhead	Bandwidth Cost	Server Storage Overhead
Hierarchical Structure			
GO-ORAM [1]	$O(1)$	$O(\log^3 N)$	require dummy
SSS-ORAM [3]	$O(N)$	$O(\log N)$	require dummy
Recurisve SSS-ORAM [3]	$O(\sqrt{N})$	$O(\log^2 N)$	require dummy
PD-ORAM [19]	$O(\log^2 N)$	$O(\log N)$	require dummy
Burst ORAM [14]	$O(N)$	$O(\log N)$	require dummy
Tree Structure			
Shi-ORAM [2]	$O(N)$	$O(\log^2 N)$	require dummy
Recursive Shi-ORAM [2]	$O(1)$	$O(\log^3 N)$	require dummy
Path-ORAM [4]	$O(N)$	$O(\log N)$	require dummy
Recursive Path-ORAM [4]	$O(\log N) \cdot \omega(1)$	$O(\log^2 N)$	require dummy
SE-ORAM [25]	$O(N)$	$O(\log N)$	require dummy (small number)

it is necessary for hiding the data of interest from a server perspective. It is actually that the dummy is not necessary to be a meaningless information. The other non-interested data can also be used as a dummy when a client retrieves a data of interest. However, the dummy in existing ORAM is an arbitrary information without any meaning. Hierarchical ORAM uses the dummy to make an empty bucket indistinguishable from the non-empty bucket during an oblivious sorting process. For binary tree ORAM, since every data block stored in ORAM is associated with leaf-ID, it is possible that some data blocks cannot fit in the path which is being accessed. Therefore, the meaningless dummy information must be used in order to make an ORAM work properly. With the various reasons mentioned, the meaningless dummy information is indispensable for both existing ORAM construction.

According to the two reasons mentioned above, it is difficult to reduce the overhead cost if we still rely on the old ORAM data structure. Therefore, the new ORAM data structure and its operation are introduced as our contribution. Two constructions called matrix based ORAM (in short M-ORAM) and recursive matrix based ORAM (in short RM-ORAM) are introduced which can achieve the asymptotic bandwidth cost at $O(1)$ and $O(\log N)$, respectively. In addition, both constructions use non-interested data as a dummy information during retrieving data of interest. Since the meaningless dummy information is not necessary for M-ORAM and RM-ORAM, both construction has zero overhead on server storage. The detailed operation and construction of M-ORAM and RM-ORAM are described respectively in Chapter 4 and Chapter 5.

Chapter 4

Matrix based ORAM

Matrix based ORAM [9], in short M-ORAM is an ORAM built upon a matrix data structure. The design of the matrix data structure allows this ORAM construction to keep the bandwidth cost independent of the number of blocks stored on the ORAM server, thereby reducing bandwidth cost compared to other ORAM schemes. In addition, M-ORAM performs reads and writes by using a simple pseudo-random function. Without any complex operations such as shuffling, sorting and merging; M-ORAM can achieve less computational complexity compared with other ORAM schemes. In this chapter, M-ORAM's storage structure, operations and security analysis are presented in Section 4.1, Section 4.2, and Section 4.3, respectively. For the sake of writing, the word client and server are used instead of ORAM client and ORAM server, respectively. In addition, the notation using in this chapter is denoted as Table 4.1.

Table 4.1: Notation of parameter used by M-ORAM

Parameter	Description
N	Size of ORAM in units of block.
H	Height of ORAM logical structure in units of block.
W	Width (length) of each stash in units of block.
L	Number of blocks in the history list .
d	Data content.
c	Size of counter of each information for secret key generator.
K	Encryption/Decryption key.
ID	Data identifier.
$stash$	Storage for downloaded data during access operation.
(x, y)	Coordination of rows and columns in matrix structure.
$PRF()$	Pseudo-Random Function.

4.1 M-ORAM Storage Structure

The storage structure in M-ORAM system can be categorised into: server storage structure and client storage structure as illustrated in Figure 4.1. The server storage is arranged in matrix format referred as ORAM. Client storage can be categorised into 3 types: stash, position map, and history list. Data stored in the storage is arranged in units of block and the block size may differ in the different types of storage. For a clearer understanding of the reader, the block of ORAM and position map is referred as the data block and position map block, respectively.

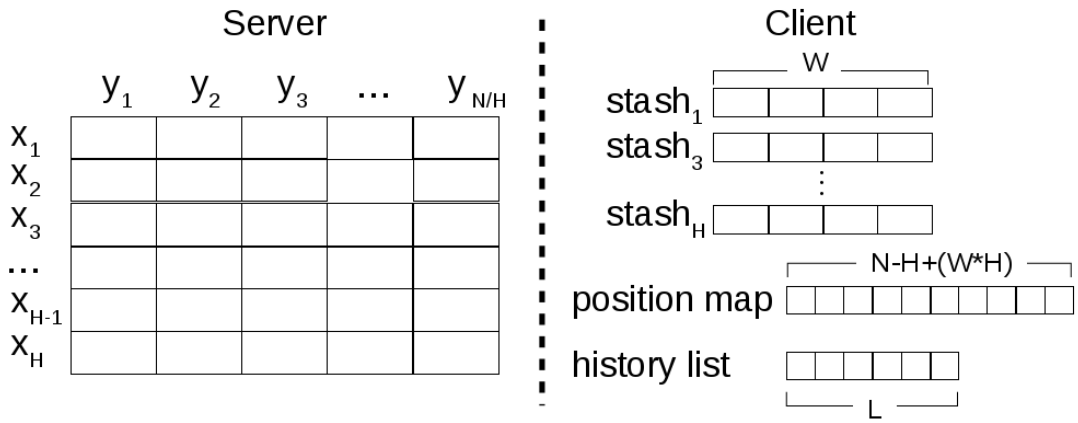


Figure 4.1: M-ORAM storage structure

4.1.1 Server Storage Structure

The matrix format of ORAM aims for controlling bandwidth cost spent during ORAM access operation. In other ORAM schemes, the bandwidth cost depends on the total number of blocks stored on the server. For example, Path-ORAM [4] has bandwidth cost $O(\log N)$ for N blocks of ORAM. To go further than $O(\log N)$ amortized bandwidth cost, we do so by introducing a matrix structure for server storage which is illustrated in Figure 4.1. The physical addresses of server storage are mapped to the set of logical addresses in the matrix format, rows (height) x_i and columns (width) y_j with $1 \leq i \leq H$ and $1 \leq j \leq \frac{H}{N}$, where $i, j \in \mathbb{Z}$. Those logical addresses are stored in a client and will be accessed when the client needs to retrieve data from ORAM. The motivation for the matrix data structure is that the bandwidth cost depends on its height, but it is independent of the width. Therefore, we can keep the bandwidth cost constant for any size of ORAM by varying the width of the structure.

4.1.2 Client Storage Structure

There are three types of storage within a client. Two from three is similarly as other ORAM schemes which are: *stash* and *position map*. Another is *history list* which is a special storage of M-ORAM containing the list of data identifiers which have been uploaded to a server. Stash is used to store the blocks downloaded during access operation while position map contains the logical mapping addresses of data to the blocks in ORAM. M-ORAM has multiple stashes and the number of stashes is equal to the number of rows of ORAM. It is referred as $stash_i$ where i is the matrix rows number. The number of blocks contained in stashes is equal to $H \cdot W$ while the number of blocks position map and history list are $N - H + (W \cdot H)$ and L, respectively. The block of each stash has an identical size and equal to the block size of ORAM. On the other hand, the block size of position map and history list is smaller compared with the block of ORAM.

4.1.3 M-ORAM Block Structure

Block structure of ORAM, stash, position map and history list is illustrated as Figure 4.2. In ORAM, data block consists of two tuples: integrity checking value (ICV) and data content. Integrity checking value is generated from AES-GCM encryption which exists for preventing

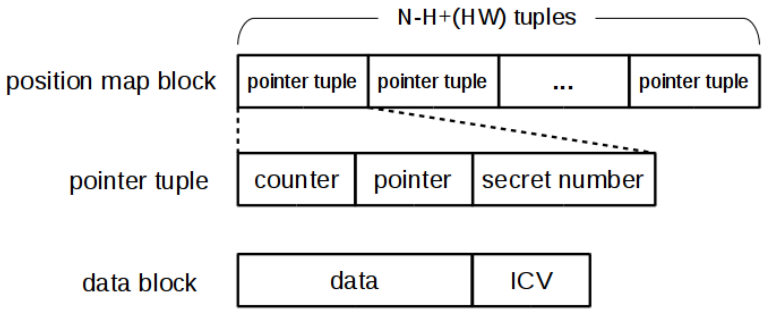


Figure 4.2: M-ORAM storage structure

the data tampering. Position map consists of $N - H + (W \cdot H)$ pointer tuples for N data blocks in the ORAM and $(W \cdot H) - H$ data blocks in stash. Each pointer tuple consists of three parts: counter, the logical address of data in ORAM called pointer, and secret number. The counter is an initialization vector (IV) for block encryption. It starts as a random number and increased by one when its corresponding data is downloaded. The pointer is a coordinate of row (x) and column (y) of matrix data structure which represents the location of data block on the server. Secret number is a random number generated by the client and will be changed in a particular period of time. These three elements are used to generate the secret key to encrypt the data before uploading to a server. The history list is L tuples of data identifier which have been uploaded in the past. It is used for the purpose of creating indistinguishable access sequences.

4.2 M-ORAM Access Operation

As mentioned at the beginning of this chapter, the bandwidth cost of M-ORAM depends on a height rather than the size of ORAM. It gives an ability to control a system bandwidth by keeping the height as a constant value. We use the same concept as Path-ORAM by hiding the data of interest among other real data (rather than dummy data as used by the hierarchical based structure and Shi's binary tree ORAM). Hence, we can maximise the efficiency of storage usage to 100% for containing the client's information. Another advantage of the matrix based structure is the independence of block in the different levels of hierarchy. Therefore, the client can freely choose a block from each hierarchical level, which makes M-ORAM's operations secure with respect to indistinguishable access of information. An access operation consists of two operations: download and upload operation which the details of download and upload operation are illustrated in Algorithm 1,2, and 3; and the definitions for the functions and parameters used by algorithms are as follows:

Definition 4. *The parameter used in Algorithm 1 and 2 are referred as following table:*

Function Name	Description
$\{i\}$	Set of element i.
d^*	Updated data content.
o	Number of data blocks randomly chosen from previous access operation.
l	Number of data blocks randomly chosen from list of history list.
n	Number of data blocks which have never been accessed.
$(x, y)_d$	Coordinate of data of interested in ORAM.
$(x, y)_{old}$	Coordinate of data selected from previous operation in ORAM.
$(x, y)_{hist}$	Coordinate of data selected from history list in ORAM.
$(x, y)_{new}$	Coordinate of new selected data in ORAM.
$\{(x, y)_{update}\}$	Updated coordinate of downloaded data.
ID_{all}	ID of all data in ORAM.
ID_{prev}	ID of previous download operation.
ID_{old}	Random ID chosen from ID_{prev}.
ID_{hist}	Random ID chosen the from history list.
ID_{new}	ID of data excluded $\{ID_{old}\}$ and $\{ID_{prev}\}$.

Definition 5. *ReadPositionMap($\{ID\}$) is a function for retrieving the coordinates of set of identifiers from position map.*
Function name: *ReadPositionMap()*
Input: $\{ID\}$
Output: $\{(x, y)\}$

Definition 6. *Stash((x, y)) is a function for retrieving a data from stash by using data coordination retrieve from position map as an input.*
Function name: *Stash()*
Input: (x, y)
Output: *data*

Definition 7. *SelectBlocks(ID,$\{ID_{prev}\}$) is a function for retrieving a set of coordinates of selected data identifiers: some identifiers of data from previous access operation (ID_{old}), some data identifiers from history list (ID_{hist}), and some data identifiers which are not included in history list and not be accessed from previous access operation, named as new data (ID_{new}). The input values of this function are identifier of data of interest, and set of identifiers of data accessed in previous operation. The return value is $\{(x, y)_1, (x, y)_2, \ldots, (x, y)_{(H-1)}\}$ which x of coordinate i is not equal to x of coordinate j, and both of them are not equal to x of coordinate of data of interest, where $i, j \in \{m \mid 1 \leq m \leq H - 1 \in \mathbb{Z}\}$ and $i \neq j$.*
Function name: *SelectBlocks()*
Input: *ID*
Output: $\{(x, y)_{old}\}, \{(x, y)_{hist}\}, \{(x, y)_{new}\}$, *where* $|\{(x, y)_{old}\}, \{(x, y)_{hist}\}, \{(x, y)_{new}\}| = H - 1$

Definition 8. *ReadORAM((x, y)) is a function for retrieving a set of data from ORAM by using a set of data coordinations retrieved from position map as an input.*
Function name: *ReadORAM()*
Input: (x, y)
Output: *data*

Definition 9. *RndPutStash($\{d\}$) is a function used to add a set of downloaded data to stash where each downloaded data is added uniquely to each row of stash. The input of this function*

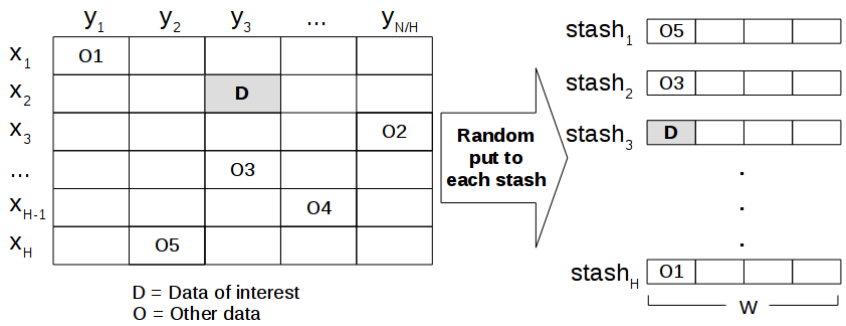
is a set of data contents downloaded from ORAM while the output is a set of updated locations which points to somewhere in stash.
Function name: *RndPutStash()*
Input: $\{d\}$
Output: $\{(x, y)_{update}\}$

Definition 10. *UpdatePositionMap(*$\{(x, y)\}, \{ID\}$*) is a function used to update coordinate of data in position map. The input is a set of data coordinates and a set of identifiers accessed in previous access operation. There is no output of this function.*
Function name: *UpdatePositionMap()*
Input: $\{(x, y)\}$, $\{ID_{prev}\}$
Output: *none*

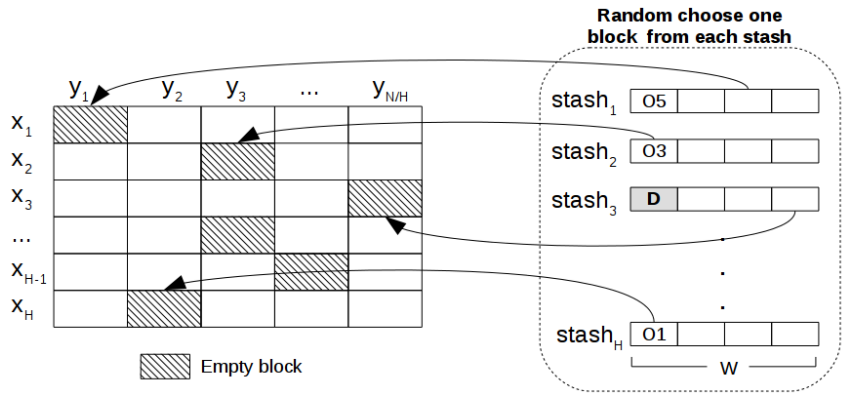
Definition 11. *RndSelect(*$\{ID\} \setminus \{ID_{exp}\}$*, n) is a function for random retrieving the n data identifiers from the specific set of data identifiers. It takes two inputs: a set of data identifiers (*$\{ID\}$*) excepted some elements in the set (*$\{ID_{exp}\}$*), and the number of elements which will be chosen (n). It produces two outputs: a set of chosen data identifiers and a set of chosen data addresses. Every chosen data identifier must not from the same row and must not from the same row of elements in a set of excepted data identifiers.*
Function name: *RndSelect()*
Input: $\{ID\} \setminus \{ID_{exp}\}$, n
Output: $(\{ID_{chs}\})$, $\{(x_{chs}, y_{chs})\}$

Definition 12. *ReplaceHist(*$\{ID\}$*) is a function for setting the status of chosen data being uploaded during upload operation. A status of uploaded data which is set as 'never has been accessed' will be turned to 'has been accessed' while the status of oldest ID in equal numbers will be set to 'never has been accessed'. The input of this function is a set of identifiers of data which are chosen to be upload, and there is no output from this function.*
Function name: *ReplaceHist()*
Input: $\{ID_{prev}\}$
Output: *none*

Definition 13. *WriteORAM(*$\{(x, y)\}$,$\{d\}$*) is a function for uploading a set of data to ORAM by using a set of data coordinations retrieved and set of selected data contents as the inputs.*
Function name: *WriteORAM()*
Input: $\{(x, y)\}$,$\{d\}$,
Output: *none*



(a) Download Operation



(b) Upload operation

Figure 4.3: M-ORAM Download and Upload Operations

4.2.1 Download Operation

In M-ORAM whenever the client wishes to download or upload a data, it must actually download multiple blocks and then upload multiple blocks back and forth from a server. One of the downloaded blocks must be the data interested by a client, whereas the set of uploaded blocks is not necessarily the same set of the previously downloaded blocks.

The download operation (Algorithm 1) will start whenever a client wants to request a data from a server. The pointer of data of interest is retrieved from position map block which corresponds to data of interests' identifier referred as *ID*. If the pointer points to stash (line 3), a client does a local access to a data of interest. If it is not (line 8), the download operation is executed. To download, a client chooses H data block to be downloaded which one among H data block is data of interest. Other $H - 1$ data block are randomly chosen from three groups: o data block from the previous operation, l data block from the list in the history list, and n data block which have never been accessed. Each selected data locates in a unique row of ORAM. In other words, a data block will be accessed in each row of ORAM (see Figure 4.3a). After the client finished choosing data blocks (line 9), the set of data block coordinates is used to download data from ORAM (line 10). Then each of the H downloaded data will be randomly added to H unique stashes (line 14). After data has been added to stash, client updates the new data location to its position map (line 15). With this method, the location in X–axis of every data is changed and independent from its original position.

Algorithm 1 Download Operation

1: **Input:** $ID, d^*, \{ID_{prev}\}$
2: $(x, y)_d \leftarrow$ ReadPositionMap(ID)
3: **if** $(x, y)_d$ in stash **then**
4: **if** updata **then**
5: Stash($(x, y)_d$) $\leftarrow d^*$
6: **end if**
7: $d \leftarrow$ Stash($(x, y)_d$)
8: **else**
9: $\{(x, y)_{old}\}, \{(x, y)_{hist}\}, \{(x, y)_{new}\} \leftarrow$ SelectBlocks(ID,$\{ID_{prev}\}$)
10: $d, \{d_{other}\} \leftarrow$ ReadORAM($(x, y)_d, \{(x, y)_{old}\}, \{(x, y)_{hist}\}, \{(x, y)_{new}\}$)
11: **if** updata **then**
12: $d \leftarrow d^*$
13: **end if**
14: $\{(x, y)_{update}\} \leftarrow$ RndPutStash($d, \{d_{other}\}$)
15: UpdatePositionMap($\{(x, y)_{update}\}$)
16: **end if**
17: **return** d

Algorithm 2 SelectBlocks()

1: **Input:** ID, $\{ID_{prev}\}$
2: **if** $ID \in \{ID_{prev}\}$ **then**
3: $\{ID_{old}\}, \{(x_{old}, y_{old})\} \leftarrow$ RndSelect($\{ID_{prev}\} \setminus ID, o - 1$)
4: **else**
5: $\{ID_{old}\} \leftarrow$ RndSelect($\{ID_{prev}\}, o$)
6: **end if**
7: **if** $ID \in \{\{ID_{hist\_list}\} \setminus \{ID_{prev}\}\}$ **then**
8: $\{ID_{hist}\} \leftarrow$ RndSelect($\{ID_{hist\_list}\} \setminus \{\{ID_{prev}\} \uplus ID\}, l - 1$)
9: **else**
10: $\{ID_{hist}\} \leftarrow$ RndSelect($\{ID_{hist\_list}\} \setminus \{ID_{prev}\}, l$)
11: **end if**
12: **if** $ID \in \{\{ID_{all}\} \setminus \{\{ID_{hist\_list}\} \uplus \{ID_{prev}\}\}\}$ **then**
13: $\{ID_{new}\} \leftarrow$ RndSelect($\{ID_{all}\} \setminus \{\{ID_{prev}\} \uplus \{ID_{hist\_list}\} \uplus ID\}, h - o - l - 1$)
14: **else**
15: $\{ID_{new}\} \leftarrow$ RndSelect($\{ID_{all}\} \setminus \{\{ID_{prev}\} \uplus \{ID_{hist\_list}\}\}, h - o - l$)
16: **end if**
17: $\{(x, y)_{old}\}, \{(x, y)_{hist}\}, \{(x, y)_{new}\} \leftarrow$ PositionMap($\{ID_{old}\} \uplus \{ID_{hist}\} \uplus \{ID_{new}\}$)
18: **return** $\{(x, y)_{old}\}, \{(x, y)_{hist}\}, \{(x, y)_{new}\}$

4.2.2 Upload Operation

The randomization in the Y–axis will take place during the upload operation as illustrated in Figure 4.3b. A data will be selected from each stash uniformly at random, and the identifiers of chosen data are set as ID_{prev} (line 2 of Algorithm 3). As a block is selected from the entire $stash_i$, an empty block may be selected. In this case dummy data is uploaded. The identifiers of chosen data are sent to update in history list (line 3) and then the new coordinates of chosen

Algorithm 3 Upload Operation

1: **Input:** none
2: $\{ID_{prev}\}, \{x, y\}_{upload}, \{d_{upload}\} \leftarrow$ RndFromStash()
3: ReplaceHist($\{ID_{prev}\}$)
4: UpdatePosionMap($\{x, y\}_{upload}$)
5: WriteORAM($\{\{x, y\}_{upload}, \{d_{upload}\}$)
6: **return** $\{ID_{prev}\}$

data in position map are updated. Finally, all selected data are uploaded to ORAM according to their new location.

4.3 M-ORAM Security Analysis

In this section, we state the security requirements of ORAM and then explain why M-ORAM achieves the requirements. We can generally summarise the security requirements of the ORAM problem as follows:

1. Server cannot observe the relationship between the data and its location.
2. Server cannot distinguish between updated and non-updated data when it is downloaded then uploaded to the server.
3. Server cannot differentiate the data of interest of different access operations.
4. Two access sequences with the same length are computationally indistinguishable from the polynomial-time adversary.

To achieve first and second requirement, a content of data before download and after upload must looks difference. One of the solutions is re-encrypting the content by the a different secret key. These two requirements are covered by random re-encryption method which the details and security proof of random re-encryption are provided in Section 4.3.1. For the third and fourth requirement, the access pattern generated by a client must look random from a server's perspective. The security analysis of random access pattern over M-ORAM is given in Section 4.3.2.

4.3.1 Random Re-encryption

M-ORAM uses AES-GCM as an encryption method to protect the original content of every data block. Counter is used as an IV and encryption key is randomly generated to encrypt a data. To achieve ORAM's security requirements, the different secret keys must be used for encrypting the same data block for each upload. To generate the secret key used by a client, strong pseudo-random function (*PRF*) is used. Every time the client has data to be uploaded; data identifier (*ID*), common secret number, and a counter in the corresponding pointer tuple are used as input of *PRF* to generate the new secret key. To ensure that the different secret keys will be used for the different data blocks with high probability, the unique *ID* of each data block is used as one of the inputs of PRF. In addition, the counter is used as second input to guarantee that the same secret key is not applied twice in a row of accessing the same data block. The common secret

number is used as the third input of *PRF* to make the key generator more flexible. Since *ID* is unique for each data block, and the size of counter is fixed, the secret key will be reused when the counter is rolled back to its original value. By changing the common secret number before the counter value is rolled back, it can prevent the key reusing with high probability.

To calculate the time period (number of uploads) for changing the common secret number, a size of counter and the probability of choosing data to be uploaded are taken into consideration. Let a size of counter be c bits and a client focus repeatedly requests one specific data without changing the content. The probability which the same secret key will be applied to the same data is according to Theorem 1.

Theorem 1. *Let T denotes the time period (number of uploads) to change a common secret number. Suppose a counter has size c bits and stash width is equal to W blocks. $P(X = t)$ is the probability that client spends t trails until first success to retrieve data of interest from stash with the probability of success p. Therefore, the expected period of time to change a common secret number is:*

$$T = c \cdot \sum_{t=1}^{\infty} \frac{t}{W}(1 - \frac{1}{W})^{t-1} \tag{4.1}$$

Proof. Since *PRF* is a deterministic function, the same input value must be used to generate the same output. The c bits counter must take c uploads before the same value of counter will return. In addition, the probability of successfully picking a specific data from stash to be uploaded is $\frac{1}{W}$. From the expected value of geometric distribution, the equation of expected value of x failure attempts until getting the first success with a probability of success p is:

$$\mathbb{E}(X = x) = \sum_{x=1}^{\infty} xp(1 - p)^{x-1}$$

Since the system needs to have exactly c successes to retrieve data of interest from stash when the same secret key is being reused, the expected value of number of uploads is:

$$T = c \cdot \sum_{t=1}^{\infty} \frac{t}{W}(1 - \frac{1}{W})^{t-1}$$

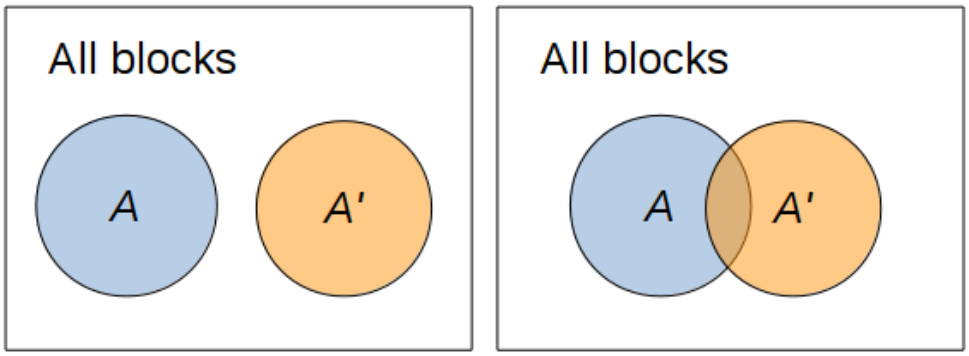
□

4.3.2 Randomization Over Access Pattern

In M-ORAM, a client downloads H blocks although only one block is interested. The other blocks are selected with the column chosen uniformly at random from each row which does not contain a data of interest, one block per one row. Therefore, the access sequence seen by the server is a set of block locations rather than *ID* which is used by a client to identify a data. To generate the access sequence which is seen to be random, some data blocks from the past accesses referred as *old block*, are required to be downloaded in current access operation. If this wasn't the case, if the client however consecutively accesses a different set of blocks, then the set of blocks accessed would be distinct which against the ORAM's security requirement. The chosen old blocks are from two group of data blocks: the data block from previous access and the data block from past accesses until two times ago. In this section, the reason for choosing the block from previous access and past access until two times ago are described, then proof of randomization over M-ORAM's access pattern are given.

Choosing the block from previous access

According to the explanation in the beginning of this section, M-ORAM requires to download some blocks which were accessed in the previous access operation, otherwise, the differentiation of two accesses will reveal an extra information to a server. Suppose there are two access request A and A' generated by a client to access the data blocks on a server. There are two possible patterns of blocks which can be accessed by A and A' as illustrated in Figure 4.4. From the perspective of server, if the blocks have been accessed by A and A' are as Figure 4.4a, the server can recognise that the data interested by client of two access request are a different information. To ensure the server cannot recognise, some accessed blocks from previous access must be accessed by current access operation as illustrated in Figure 4.4b. Since the data identifier of data which have been uploaded can be represented as the block locations which have been accessed; the list of data identifiers contained in the storage named *old list* is used for the same purpose. The proper number of blocks from previous access which have to be chosen for current operation will be described in details in Section 6.2.4.



(a) A and A' access different blocks (b) A and A' access some of same blocks

Figure 4.4: Two possible patterns of block can be accessed by A and A'

Choosing blocks from the past accesses

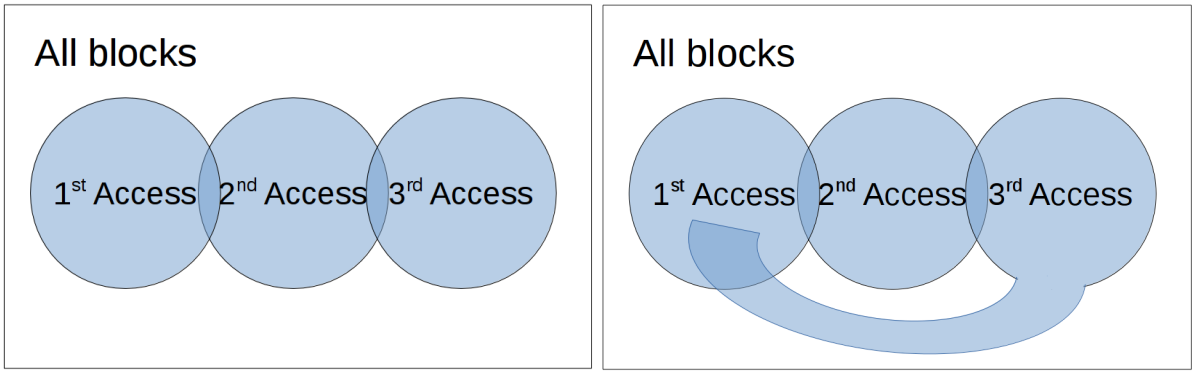
In some specific circumstances, the different access sequences can be distinguishable. The vulnerability occurs when the blocks which have been accessed are requested to be downloaded again in the short period of time. For example, let two sequences of accesses observed by a server be A' and A''. Suppose A' and A'' consist of sequent of reading d_1, d_2, and d_3; and d_1, d_2, and d_1, respectively. Therefore, A' and A'' can be represented as follows:

$$A' = \text{Read}(d_1), \text{Read}(d_2), \text{Read}(d_3)$$

$$A'' = \text{Read}(d_1), \text{Read}(d_2), \text{Read}(d_1)$$

Suppose d_1 was uploaded back to a server after finish the first access operation, and it has not be chosen as the old blocks of second access. There are two possible patterns of the blocks which can be accessed during third access as illustrated in Figure 4.5. Considering the third access, the probability that d_1 will be chosen to be read during the third request of A' is $(h - o)/N$. On the other hand, as d_1 is requested to be read by a client on the third access of A'', the probability that d_1 will be read is 1. Since the two probability values are obviously distinguishable, a server can determine the block of interest of a client with high probability.

However, the client is a person who controls the request of access in ORAM system. Therefore, a client can design of how often the same information will be repeatably requested which leads to two solutions that can solve this problem.



(a) No block from first access (b) Some blocks from first access

Figure 4.5: Two possible patterns of block can be accessed during third access

1. **Does not request same block twice more often than one time of $N/(h-o)$ accesses:** To make the selection looks like a uniform distribution, the client has to limit the number of accesses to the block that has been accessed. Therefore, from the example given in the beginning of this section, the number of accesses to those blocks should be approximately 1 time of $N/(h-o)$ accesses.

2. **Random selecting from the history list:** Another solution to keep A' indistinguishable from A'' is to create the *history list* which is a list containing the identifier of data blocks which have been uploaded to the server. Client randomly chooses l blocks from the history list as additional blocks for each access operation. To do so client starts with *'warm-up access'* at the very beginning of its first communication with a server. This warm-up access is dummy accesses for constructing the history list before starting the real communication. It occurs only once when a new client joins the system. By the fact that if a new list is continually added, the number of data IDs in the history list will be eventually equal to N. Therefore, oldest l IDs in the history list will be removed and replaced by the set of recent uploaded IDs after reaching the particular number of accesses. This number varies according to how much frequency that client wants to access the same block location. However, the number of accesses before replacing is bounded by $N/(h-o)$ to make sure that the probability of accessing the oldest set of blocks in the history list is not greater than $(h-o)/N$.

Considering those two solutions, the first solution is limited by the frequency of accessing the information. A client has to control the average number of accesses not more than 1 of $N/(h-o)$ which is quite impractical. On the other hand, the second solution is more flexible and more practical. The trade-off is client has to do warm-up access to construct the history list. However, the warm-up access will be performed one and only one on new communication which is negligible for long-term communication.

Proof of Randomization Over M-ORAM Access Pattern

Suppose A is an access sequence seen by a server consists of i accesses (either upload or download), $adr_j[data_j]$ is a set of addresses revealed to server during j^{th} access where $j \in \{1 \leq x \leq i \mid x \in \mathbb{Z}\}$. It can be illustrated as Equation 4.2.

$$A = (adr_i[data_i], adr_{i-1}[data_{i-1}], \ldots, adr_1[data_1]) \tag{4.2}$$

By the fact that there are three possible different patterns of two series of accesses A and A' generated by a client, which can be described as in Lemma 1, Lemma 2, and Lemma 3. To show the random access pattern of M-ORAM is secure, we analyse case-by-case of these three lemmas.

Lemma 1. *Different access sequences on the same set of data blocks of M-ORAM are indistinguishable from the adversary who has limited computational power to polynomial time computation.*

Proof. According to Section 4.3.1, the content of accessed block is indistinguishable from a random string by randomised encryption whether or not the content is changed. In addition, since the content within the data blocks is randomly changed according to M-ORAM access operation, the server cannot identify the relationship between the content and the data block. Although A and A' access the same set of data blocks, the uploaded content of both access sequences does not necessary to be the same. Therefore A and A' is indistinguishable by the polynomial time adversary. □

Lemma 2. *Different access sequences on completely different data blocks of M-ORAM are indistinguishable from the adversary who has a computational power bounded to polynomial time.*

Proof. Suppose two access sequences A and A' have length i and access on different data blocks. According to the Section 4.2, o blocks are selected from the previous operation to be accessed in current operation to make sure that the consecutive operation does not access a completely different set of data blocks. Furthermore, a set of data blocks which has been downloaded is randomly replaced by data within stash which may or may not be the same set of previous data. Hence, although A and A' access on different data blocks, it does not mean that client try to access two different groups of data. Therefore, Lemma 2 holds by the same proof of Lemma 1. □

Lemma 3. *Different sequences of accesses on the some (but not all) of the same blocks of RM-ORAM are indistinguishable from the adversary who has limited computational power to polynomial time computation.*

Proof. Two access sequences can be distinguished if the frequency of accessing data blocks which have been accessed is different. However, by varying the size of history list, a client can control the probability of ID listed in the history list that will be accessed. High probability means the ID is possible to be accessed more often although it is not a data of interest. Suppose the probability of data of interest among of chosen l data blocks from history list size L is l/L. It means data of interest possibly be selected every L/l accesses. By maintaining this probability to be greater than or equal to l/L; although two sequences have different frequencies of accessing data of interest, they look similar from server's perspective. Hence, the Lemma 3 holds when the content of data blocks is indistinguishable by randomised encryption. □

Theorem 2. *The randomness of M-ORAM's access pattern is secure from the adversary who has limited computational power to polynomial time computation.*

Proof. Suppose A and A' are two different access patterns of client. B_A and $B_{A'}$ are the set of blocks accessed by client of A and A', respectively; and $B_A, B_{A'} \in B_N$ where B_N is a set of every block in ORAM. By case analysis, there are three possible cases of blocks accessed by two different access sequences which are $B_A = B_{A'}$, $B_A \cap B_{A'} = \emptyset$, and $(B_A \cap B_{A'} \neq \emptyset) \wedge (B_A \neq B_{A'})$. For the case $B_A = B_{A'}$, it can be covered by Lemma 1; case $B_A \cap B_{A'} = \emptyset$ can be covered by Lemma 2; and for the last case $(B_A \cap B_{A'} \neq \emptyset) \wedge (B_A \neq B_{A'})$, it can be covered by Lemma 3. Therefore, any sequences of access pattern are secure under the advisory whose computational power is bounded by polynomial time. □

Chapter 5

Recursive Matrix based ORAM

Many different ORAM algorithms have been designed. Despite all offering privacy, they make trade-offs in performance, where a key metric is bandwidth cost. Downloading/uploading multiple blocks in order to access just a single block leads to significant bandwidth cost. However, that comes at the expense of increased storage space on clients and/or computational overheads. In particular, most ORAM algorithms store a position map on the client which records the addresses of each data item on the server. This has $O(N)$ client storage requirements, where N is the maximum number of data blocks on the server. A general technique to reduce the storage requirements (at the expense of increased bandwidth cost) is recursion: store the position map on the server in ORAM, and a second, smaller position map to that is stored on the client. In this chapter, we present a detailed design of *Recursive M-ORAM (RM-ORAM)* [10], and provide performance and security analysis of this construction. For the sake of writing, the word client and server are used instead of ORAM client and ORAM server, respectively. In addition, the notation using in this chapter is denoted as Table 5.1.

Table 5.1: Notation of parameter used by RM-ORAM

Parameter	Description
DataORAM	ORAM which contains data block.
PosORAM	ORAM which contains position map block.
StashData	Stash which contains data block downloaded between download/upload.
StashPath	Stash which contains position map block downloaded between download/upload.
StashPos	Stash of Path ORAM which contains position map block downloaded between download/upload.
N	Size of DataORAM (block unit).
N'	Size of PosORAM (block unit).
$Posmap_i$	A group of position map blocks which is accessed at level i of recursion.
ID_i	Identifier of data i.
d_i	Content of data i.
p_i	Content of position map i.
b_i	Logical address of block i on the server.
h	Number of data blocks which are being downloaded uploaded per level of recursion.
m	Number of pointer tuples within each of position map block.
r	Number of levels of recursion.

Table 5.1: Notation of parameter used by RM-ORAM

Parameter	Description
o	Number of old blocks which will be chosen for next access operation.
l	Number of blocks from history list which will be chosen for next access operation.
s	Size of StashData after finish download operation.
c	Size of counter.

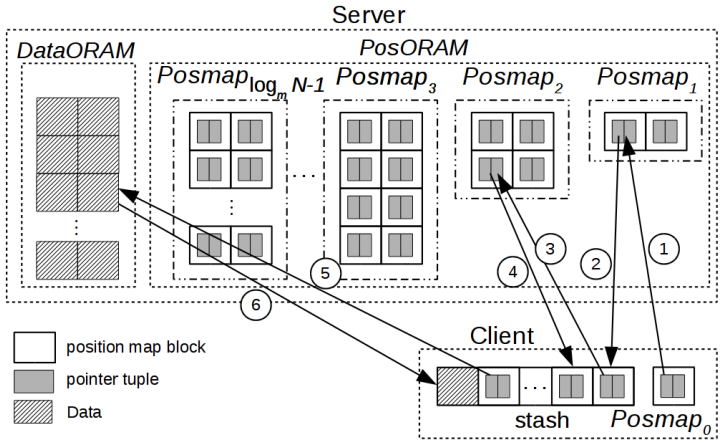
5.1 General Concept of Recursive ORAM Construction

Recursive ORAM first appears in [3]. It significantly reduces the size of position map on a client to $O(1)$ while it incurs logarithmic growth of bandwidth overhead according to the size of ORAM. A non-recursive ORAM construction has to keep the position map on the client which is a major limitation. To overcome this problem, a recursive ORAM construction stores most of the position map on the server. Information of recursive ORAM can be categorized to *data block* and *position map block* which is allocated in *DataORAM* and *PosORAM*, respectively. The number of blocks in DataORAM and PosORAM are referred to N and N', respectively and may have a different size. PosORAM can be separated into multiple groups of position map blocks called $Posmap_i$ where i is the number of groups which is equal to the number of levels of recursion minus one.

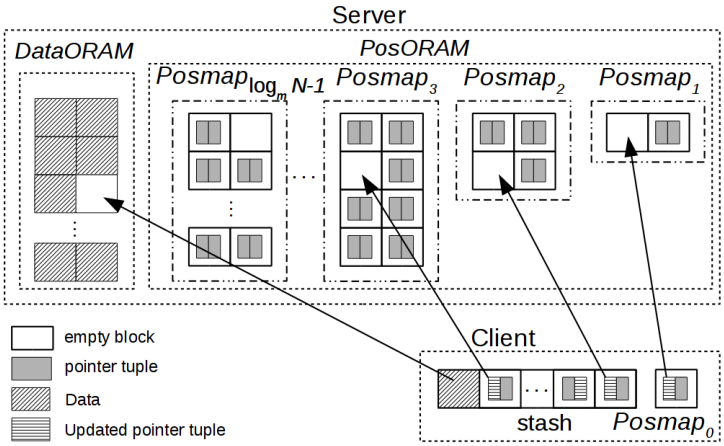
Recursive operation is explained with the aid of Figure 5.1, and sequence of numbers in Figure 5.1a represents the order of downloading. Let each position map block contains m tuples, called *pointer tuple*. Each pointer tuple contains a pointer and the pointers of current level are an injective mapping to position map blocks of the next level of recursion. To download data of interest from DataORAM, the client looks for the associated pointer in $Posmap_i$ which points to a position map block located somewhere in $Posmap_{i+1}$ then downloads that block to the client. The operation will start from $Posmap_0$ and repeatedly proceed until the data of interest is downloaded from DataORAM. After finishing the download, the client uploads blocks back to the server. The procedure of uploading information depends on which ORAM algorithm is used.

5.2 RM-ORAM Storage Structure

Recursive ORAM is designed to reduce the number of position map blocks stored on the client, which the different ORAM constructions have the different solutions to achieve the result. Recursive Path ORAM is a binary tree based ORAM which the groups of data blocks and position map blocks are arranged in binary tree data structure. Each node of binary tree stores m pointers, where each of these pointers points to a block in either DataORAM or PosORAM. Each access from a client to server is performed using the normal Path ORAM access operations. However, Path ORAM and M-ORAM are designed on the different constructions, the model of recursive ORAM used by Path ORAM cannot be applied directly to M-ORAM. The details of RM-ORAM's server storage, client storage, and block structure are discussed in Section 5.2.1, 5.2.2, and 5.2.3, respectively.



(a) Download Operation



(b) Upload operation

Figure 5.1: General recursive operation for ORAM

5.2.1 Server Storage Structure

RM-ORAM server storage (Figure 5.2) consists of two types of ORAM *DataORAM* and *PosORAM*. DataORAM and PosORAM may differ by block size and number of blocks which is known by a server. Same as other recursive ORAM constructions, DataORAM and PosORAM of RM-ORAM are used to store data block and position map block, respectively. To make the server harder identify an identity of information, we propose RM-ORAM with a single PosORAM that integrates every Posmap on the server into one. A PosORAM containing Posmap$_1$ through to Posmap$_{\log_m N-1}$ of Figure 5.1 which position map blocks of those Posmap are randomly distributed within PosORAM. Therefore, size of PosORAM is $\sum_{i=1}^{r-1} m^i$ blocks. Hence, the server cannot distinguish the blocks from different recursion levels as happening on Path ORAM. Moreover, by relying on a key strength feature of M-ORAM, it allows the number of downloaded blocks per access request is independent of ORAM size.

5.2.2 Client Storage Structure

There are four types of storage on a client: *StashData*, *StashPath*, *Posmap*$_0$ and history list. StashPath and StashData are used to store downloaded position map blocks and data blocks, respectively. Every storage excepted StashPath has a constant size while StashPath has a size

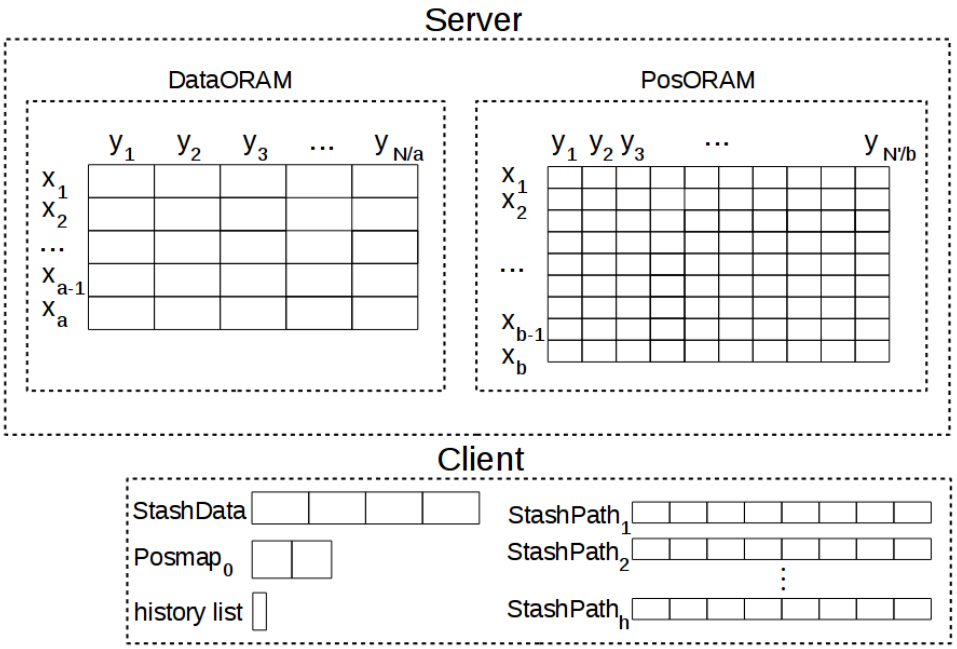


Figure 5.2: RM-ORAM storage

equal to $h \log N$. The size of StashData is greater than the number of data blocks downloaded during an access operation while the sum of the total size of StashPath is equal to the number of downloaded position map blocks. StashPath is divided into h paths referred as StashPath$_i$ where h is equal to the number of blocks downloaded per level of recursion. Posmap$_0$ contains pointer tuples which have a pointer pointing to the blocks in PosORAM and size of Posmap$_0$ must be smaller than PosORAM. History list is the list of blocks having been accessed by a client, and each bit of history list represents the status of a block having been accessed (e.g. a block has been accessed is 1, else is 0).

5.2.3 RM-ORAM Block Structure

Different from M-ORAM, there are two types of the block stored on a server of RM-ORAM system: data block and position map block. Figure 5.3 illustrates the details of block in each type. Each position map block contains m tuples referred as *pointer tuple*, and each tuple consists of four parts: counter, index, pointer, and secret number. Index tells what are the *IDs* related to the pointer of position map block. Counter is used as *IV* for AES-GCM encryption. Secret number is a random number generated by a client which will be changed in a period of time. In addition, index, counter, and secret number are also used as the input of pseudo-random function (*PRF*) to generate an encryption key in the random encryption process. Data block consists of two sections: encrypted data content (or in short data) and integrity checking value (*ICV*). Integrity checking value is generated from AES-GCM encryption which exists for preventing the data tampering.

5.3 RM-ORAM Access Operation

An access operation of RM-ORAM is slightly different from original M-ORAM. Since most of the position map blocks are stored in a server, a client cannot directly download data of

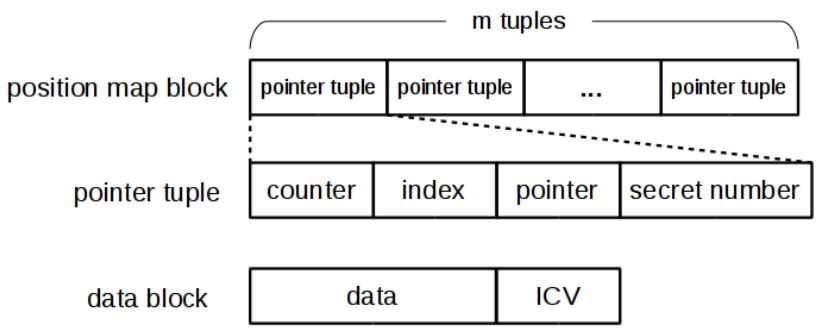


Figure 5.3: Details of data block and position map block

interested. The client has to repeatably access PosORAM until the address of data of interest in DataORAM is downloaded. Then an upload operation is executed to create a symmetric pattern on every accesses. In this section, Algorithm 4 and 5 illustrate the details of download operation while Algorithm 6 illustrates the details of upload operation with the aid of Figure 5.4. The definition of functions and parameters used by algorithms are as follows:

Definition 14. *The parameter used in Algorithm 4, 5, and 6 are referred as following table:*

Function Name	Description
$\{i\}$	Set of element i.
d^*	Updated data content.
d_{ID}	Content of data which has unique identifier *ID*.
d_{ID_i}	Content of position map block at i^{th} level of recursion which leads to data of interest.
d_{oth_i}	Content of position map block at i^{th} level of recursion which leads to other data.
d_{oth}	Content of other data.
d_{rnd}	Content of random chosen data.
b_{ID_i}	Address of position map block or data block of data of interest of at i^{th} level of recursion.
b_{oth_i}	Address of position map block or data block of other data at i^{th} level of recursion.
b_{drnd}	Random address of random chosen data block from StashData.
b_{prnd}	Random address of position map block in StashPath.
o	Number of data blocks randomly chosen from previous access operation.
l	Number of data blocks randomly chosen from list of history list.
n	Number of data blocks which have never been accessed.
ID_{all}	ID of all data in ORAM.
ID_{prev}	ID of previous download operation.
ID_{old}	Random ID chosen from ID_{prev}.
ID_{hist}	Random ID chosen from history list.
ID_{new}	ID of data excluded $\{ID_{old}\}$ and $\{ID_{prev}\}$.

Definition 15. *ReadStashData(*ID*) is a function for retrieving a data content which is corresponding to* ID *from StashData.*

Function name: *ReadStashData()*
Input: $\{ID\}$
Output: d_{ID}

Definition 16. *UpdateStashData(ID, d) is a function for updating a data content d of* ID *in StashData. There is nothing returned from this function.*
Function name: *ReadStashData()*
Input: ID, d
Output: *none*

Definition 17. *SelectBlocks(ID,{ID_{prev}) is a function for retrieving a set of selected data identifiers which some identifiers of data from previous access operation (ID_{old}), some identifiers of data from history list (ID_{hist}), and some identifiers of new data (ID_{new}). The input value of this function is an identifier of data of interest and a set of identifiers of data from previous access operation.*
Function name: *SelectBlocks()*
Input: *ID,* $\{ID_{prev}\}$
Output: $\{ID_{old}\}, \{ID_{hist}\}, \{ID_{new}\}$

Definition 18. *RndSelect($\{ID\} \setminus \{ID_{exp}\}$, n) is a function for random retrieving the n data identifiers from specific set of data identifiers. It takes two inputs: a set of data identifiers ($\{ID\}$) excepted some elements in the set ($\{ID_{exp}\}$), and number of elements which will be chosen (n). It produces a set of chosen data identifiers as an output.*
Function name: *RndSelect()*
Input: $\{ID\} \setminus \{ID_{exp}\}$, n
Output: $\{ID_{chs}\}$

Definition 19. *ReadPos*($\{b\}, \{ID\}$) *is a function for retrieving a set of pointers from PosORAM according to a set of data identifiers and their corresponding address. There are two inputs to this function which are a set of data identifiers and a set of block addresses. This function returns two sets of information which are set of block addresses of next level of recursion and set of contents contained in input block addresses.*
Function name: *ReadPos()*
Input: $\{b_i\}, \{ID\}$
Output: $\{d_i\}, \{b_{i+1}\}$

Definition 20. *ReplaceHist($\{ID\}$) is a function for setting the status of chosen data that are uploaded during upload operation. A status of uploaded data which is set as 'never has been accessed' will be turned to 'has been accessed' while the status of oldest ID in equal numbers will be set to 'never has been accessed'. The input of this function is a set of identifiers of data which are chosen to be upload, and there is no output from this function.*
Function name: *ReplaceHist()*
Input: $\{ID_{prev}\}$
Output: *none*

Definition 21. *A set of position map blocks which belongs to each path created during access operation (see. Figure 5.4a) is stored in one of StashPaths. AppendStashPath($\{d\}$) is used to append each downloaded position map blocks of each recursion level to their respective StashPath. The input of this function is a set of position map blocks and it does not generate an*

output.
Function name: *AppendStashPath()*
Input: $\{d\}$
Output: *none*

Definition 22. *AppendStashData($\{d\}$) is a function used to store the content of downloaded data blocks in StashData. A set of downloaded contents is an input and there is no output of this function.*
Function name: *AppendStashData()*
Input: $\{d\}$
Output: *none*

Definition 23. *ReadData($\{b\}$) is a function for reading data block from DataORAM. It takes an input as a set of data addresses which is selected to read. It gives a set of contents that corresponds to the input addresses as a return value.*
Function name: *ReadData()*
Input: $\{b\}$
Output: $\{d\}$

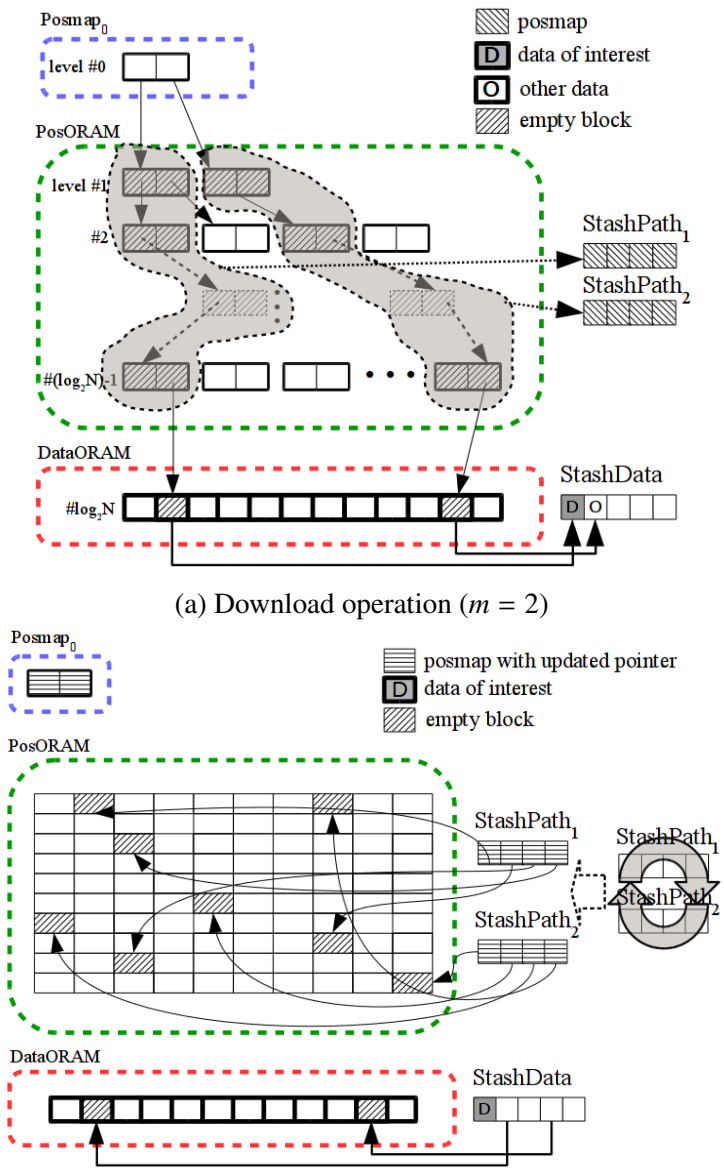
Definition 24. *RndRetrievStashData(h) is used for randomly selecting the data block from StashData to be uploaded. The number of elements to be chosen is used as input and it returns the set of data and its corresponding identifier as an output. In addition, the set of output IDs is used as ID_{prev} for next download operation.*
Function name: *RndRetrievStashData()*
Input: *number of elements to be chosen (h)*
Output: $\{ID_{prev}\},\{d\}$

Definition 25. *After choosing the blocks to be uploaded, the new corresponding addresses are updated via RndAssignNewPointer() function. RndAssignNewPointer() updates the new addresses to $Posmap_0$ and StashPaths before uploading information. The return values of this function is a set of updated addresses of data blocks and position map blocks.*
Function name: *RndAssignNewPointer()*
Input: $Posmap_0$, $\{StashPaths\}$
Output: $\{b_{drnd}\},\{b_{prnd}\}$

Definition 26. *WriteData($\{d\}$,$\{b\}$) is used to upload selected data blocks to DataORAM. Set of data contents and its address are used as the inputs, and there is no output returned from this function.*
Function name: *WriteData()*
Input: $\{d\},\{b\}$
Output: *none*

Definition 27. *WritePos($\{d\}$,$\{b\}$) is used to upload position map blocks in StashPath to PosORAM. Set of data contents and its address are used as the inputs, and there is no output returned from this function.*
Function name: *WriteData()*
Input: $\{d\},\{b\}$
Output: *none*

Definition 28. *ClearStashPath() is used to empty every StashPath.*
Function name: *ClearStashPath()*
Input: *none*
Output: *none*



(a) Download operation ($m = 2$)

(b) Upload operation

Figure 5.4: RM-ORAM's access operation

5.3.1 Download Operation

Once a client wants to access (either download or upload) a block of data, it starts by checking the data blocks remained in StashData as shown on line 1-7 of Algorithm 4. If data of interest is in StashData, a client performs a local access. If it is not, client randomly chooses $h - 1$ other data and a data of interest to be downloaded. Among of $h - 1$ other data, there are o data blocks selected from the data blocks of previous uploads, and l data blocks (that are not data

Algorithm 4 RM-ORAM's download operation

1: **Input:** ID, d^*
2: **if** ID in StashData **then**
3: $d_{ID} \leftarrow$ ReadStashData(ID)
4: **if** update **then**
5: $d_{ID} \leftarrow d^*$
6: UpdateStashData(ID, d^*)
7: **end if**
8: **else**
9: $\{ID_{old}\}, \{ID_{hist}\}, \{ID_{new}\} \leftarrow$ SelectBlocks(ID,$\{ID_{prev}\}$)
10: ReplaceHist($\{ID_{new}\}$)
11: **for** $i \in \{r \mid 0 \leq r \leq \lceil \log_m N \rceil - 1\}$ **do**
12: $d_{ID\_i}, b_{ID\_(i+1)} \leftarrow$ ReadPos(b_{ID_i}, ID)
13: $\{d_{oth\_i}\}, \{b_{oth\_(i+1)}\} \leftarrow$ ReadPos($\{b_{oth\_i}\}, \{ID_{new}\} \uplus \{ID_{hist\_list}\} \uplus \{ID_{old}\}$)
14: **if** $i < \lceil \log_m N \rceil - 1$ **then**
15: AppendStashPath($d_{ID\_i}, \{d_{oth\_i}\}$)
16: **end if**
17: **end for**
18: $d_{ID} \leftarrow$ ReadData($b_{int\_\lceil \log_m N \rceil}$)
19: $\{d_{oth}\} \leftarrow$ReadData($\{b_{oth\_\lceil \log_m N \rceil}\}$)
20: **if** update **then**
21: $d_{ID} \leftarrow d^*$
22: **end if**
23: AppendStashData($d_{ID}, \{d_{oth}\}$)
24: **end if**
25: **return** d_{ID}

blocks of previous uploads) selected from the history list. The client is therefore looking into all corresponding pointers of data blocks being downloaded in Posmap_0 (line 9), then the oldest blocks in history list are replaced by the same number of selected new blocks (line 10). These pointers will point to Posmap_1 within PosORAM, then client downloads the corresponding position map blocks to StashPath. Position map blocks downloaded per level of recursion are split to h StashPaths (line 14-16). Client repeats the same procedures by searching for the corresponding pointers from downloaded position map blocks and ultimately can identify the location of other h position map blocks in Posmap_2. After downloads, the downloaded position map block is stored next to its previous position map block in the same StashPath. Recursion is carried out until the pointers of h data blocks in DataORAM are retrieved (line 18-19). Finally, the selected data blocks are downloaded to StashData (line 23).

5.3.2 Upload Operation

At the final state of access operation, the same number of downloaded blocks are uploaded to the server (Algorithm 6). Client randomly chooses h data blocks from StashData (line 1). Since the size of StashData is greater than h blocks, there is no guarantee that the selected blocks are the same blocks of the previous download. The addresses having been accessed of DataORAM and PosORAM from download operation are randomly assigned to selected data blocks and

Algorithm 5 SelectBlocks()

1: **Input:** ID, $\{ID_{prev}\}$
2: **if** $ID \in \{ID_{prev}\}$ **then**
3: $\{ID_{old}\} \leftarrow \text{RndSelect}(\{ID_{prev}\} \setminus ID, o-1)$
4: **else**
5: $\{ID_{old}\} \leftarrow \text{RndSelect}(\{ID_{prev}\}, o)$
6: **end if**
7: **if** $ID \in \{\{ID_{hist\_list}\} \setminus \{ID_{prev}\}\}$ **then**
8: $\{ID_{hist}\} \leftarrow \text{RndSelect}(\{ID_{hist\_list}\} \setminus \{\{ID_{prev}\} \uplus ID\}, l-1)$
9: **else**
10: $\{ID_{hist}\} \leftarrow \text{RndSelect}(\{ID_{hist\_list}\} \setminus \{ID_{prev}\}, l)$
11: **end if**
12: **if** $ID \in \{\{ID_{all}\} \setminus \{\{ID_{hist\_list}\} \uplus \{ID_{prev}\}\}\}$ **then**
13: $\{ID_{new}\} \leftarrow \text{RndSelect}(\{ID_{all}\} \setminus \{\{ID_{prev}\} \uplus \{ID_{hist\_list}\} \uplus ID\}, h-o-l-1)$
14: **else**
15: $\{ID_{new}\} \leftarrow \text{RndSelect}(\{ID_{all}\} \setminus \{\{ID_{prev}\} \uplus \{ID_{hist\_list}\}\}, h-o-l)$
16: **end if**
17: **return** $\{ID_{old}\}, \{ID_{hist}\}, \{ID_{new}\}$

Algorithm 6 RM-ORAM's upload operation

1: $\{ID_{prev}\}, \{d_{rnd}\} \leftarrow \text{RndRetrievStashData}(h)$
2: $\{b_{drnd}\}, \{b_{prnd}\} \leftarrow \text{RndAssignNewPointer}(\{\text{StashPath}\}, \text{Posmap}_0)$
3: $\text{WriteData}(\{d_{rnd}\}, \{b_{drnd}\})$
4: $\text{WritePos}(\{\text{data in StashPath}\}, \{b_{prnd}\})$
5: ClearStashPath()

position map blocks, respectively. Pointer in selected position map blocks is set to point to the new corresponding address (line 2). Finally, all of selected data blocks and position map blocks are uploaded to the DataORAM and PosORAM, respectively according to their new location (line 3-4).

5.4 RM-ORAM Security Analysis

In this section, we state the security requirements of ORAM and then explain why RM-ORAM achieves the requirements. The general security requirements of ORAM are:

1. Server cannot observe the relationship between the data and its address.
2. Server cannot distinguish between updated and non-updated data when it is written back to the server.
3. Server cannot differentiate the data of interest of different access operations.
4. Randomness of the sequence of accesses request is secure under polynomial-time adversary.

To achieve first and second requirement, a content of data before download and after upload must looks difference. One of the solutions is re-encrypting the content by a different secret

key. These two requirements are covered by the security proof in Section 5.4.1 which gives a discussion about the probability of using the same secret key on the same content of RM-ORAM construction. For the third and fourth requirement, the access pattern generated by a client must look random from a server's perspective. To strengthen the security analysis of random access pattern over RM-ORAM in Section 5.4.2, we start by introducing the appropriate number of blocks selected from previous access operation in Section 5.4.2. RM-ORAM requires downloading some blocks which were accessed in the previous access operation. If this wasn't the case, if the client however consecutively accesses a different set of blocks, then the set of blocks accessed would be distinct. Then the history list is introduced in Section 5.4.2 to cover the issue of distinguishable access pattern on a specific circumstance.

5.4.1 Random Re-encryption

Client encrypts each data item d_i with a generated secret key $k_{(i,t)}$ before uploading to the server. Encrypting the same data item with the same key multiple times is undesirable because the encryption of non-update content will reveal the identity of information to the server. Therefore, $k_{(i,t)}$ must not equal to $k_{(i,t+1)}$. RM-ORAM use AES-GCM encryption algorithm same as M-ORAM and a counter is also used as an *IV*. However, the inputs used to generate an encryption key are slightly different between two construction. For data block, the inputs used to generate encryption key are same as M-ORAM. On the other hand for position map block, rather than using *ID*, RM-ORAM uses the index of each position map block as one of the inputs of *PRF*.

To generate the secret key for position map block the *index*, *counter*, and shared *common secret number* are used as the inputs of a strong pseudo-random function (PRF). Index and counter are the information within the pointer tuple as shown in Figure 5.3. Index tells what data block is relevant to its pointer which is always updated during access operation according to which data block is associated with the pointer. Counter is a random number which will be increased whenever its corresponding block (either data block or position map block) is accessed. The generated secret key is used for both decrypting and encrypting the block pointed by the pointer. The common secret number is a parameter that client keeps as a secret, and used to make the key generator to be more flexible. Since PRF is deterministic and there is a possibility that same value of index and counter will be applied to PRF. Therefore, this common secret number has to be changed in some period of time. To accurate the changing period, we measure it as the number of executing upload operations.

Let a size of counter be c bits, s is the size of StashData after download operation, $counter \in \{i \mid 0 < i < c, i \in \mathbb{Z}\}$, h is the number of data blocks downloaded during access operation, and a client focus repeatedly requests one specific data without changing the content. The time period of changing the value of common secret number is according to Theorem 3.

Theorem 3. *Let T denotes the time period (number of upload operations) to change a common secret number. Suppose a counter size is c bits and StashData size is equal to s blocks. $P(X = t)$ is the probability that client spends t trails until at least one data block is assigned to the same block as it was during download operation with the probability of success p. Therefore, the expected period of time to change a common secret number is:*

$$T = c \cdot \sum_{t=1}^{\infty} tp(1-p)^{t-1} \tag{5.1}$$

where $p = \frac{(s-h)!}{s!} \cdot \sum_{i=1}^{h} \frac{(-1)^{i+1} \cdot h!}{i!}$

Proof. Since *PRF* is the deterministic function, the same input value must be used to generate the same output. The c bits counter must take c uploads before the same value of counter will return. In addition, the probability of successfully picking a specific data from stash to be uploaded is $\frac{1}{W}$. From the expected value of the geometric distribution, the equation of expected value of x failure attempts until getting the first success with a probability of success p is:

$$\mathbb{E}(X = x) = \sum_{x=1}^{\infty} xp(1-p)^{x-1}$$

Since encryption function is deterministic, the same encrypted data will be generated when the same data content is encrypted by same secret key. Furthermore, the same secret key will be generated by PRF if every input has the same value. As an index of the last level of recursion relates to only one data block of DataORAM, if at least one of downloaded data blocks is selected to be uploaded to its previous location, at least one position map block contains the same value of an index. For now, let counter does not change over the time. The number of solutions which at least one data block is assigned to the same block as it was during the download operation is

$$\sum_{i=1}^{h} (-1)^{i+1} \cdot C(h,i) \cdot (h-i)!$$

Let *S1DW* be the event of the same h data blocks are chosen from StashData size s, and at least one data block is assigned to the same block as it was during the download operation. Therefore, the probability of *S1DW* is:

$$\begin{aligned}\Pr(S1DW) &= \frac{(s-h)! \cdot h!}{s!} \cdot \frac{\sum_{i=1}^{h} \frac{(-1)^{i+1} \cdot h!}{i!}}{h!} \\ &= \frac{(s-h)!}{s!} \cdot \sum_{i=1}^{h} \frac{(-1)^{i+1} \cdot h!}{i!}\end{aligned}$$

Hence, the expected value of number of uploads is:

$$T = \sum_{t=1}^{\infty} tp(1-p)^{t-1}$$

where $p = \Pr(S1DW)$

However, in fact a counter is changed over the time when a data block is uploaded to a server. Suppose counter is size c bits, the system needs to have exactly c successes to retrieve data of interest from stash when the same secret key is being reused, the expected value of number of uploads is:

$$T = c \cdot \sum_{t=1}^{\infty} tp(1-p)^{t-1}$$

where $p = \Pr(S1DW)$. Hence, the expected number of upload operations before the common secret number has to be changed is hold as Theorem 3. □

5.4.2 Randomization Over Access Pattern

Slightly different from M-ORAM, there are two types of ORAM on RM-ORAM server: DataORAM and PosORAM while M-ORAM has only DataORAM, which makes the access pattern of M-ORAM and RM-ORAM are different. However, RM-ORAM is an altered version of M-ORAM construction. It uses the same concept to produce the indistinguishable access pattern between two different accesses. To generate the access sequence which is seen to be random, some data blocks from the past accesses referred as *old block*, are required to be downloaded in current access operation. If this wasn't the case, if the client however consecutively accesses a different set of blocks, then the set of blocks accessed would be distinct which against the ORAM's security requirement. The chosen old blocks are from two group of data blocks: the data block from previous access and the data block from past accesses until the access two times ago. In this section, the reason for choosing the block from previous access and past access until two times ago are described, then proof of randomization over RM-ORAM's access pattern are given.

Choosing the block from previous access

To create indistinguishable access pattern, some blocks which were accessed in the previous access operation must be chosen to be downloaded during the current operation. Since position map block contains the pointer pointing to another blocks either position map block or data block, choosing some previously downloaded data blocks means choosing some downloaded position map blocks of the previous access operation. Therefore, we consider only determining the number of data blocks selected from previous access operation. As RM-ORAM construction is inherited from M-ORAM, the solution of choosing a block from previous access is same as having been described in Section 4.3.2.

Choosing blocks from the past accesses

Same as M-ORAM in some specific circumstances, the different sequences of accesses can be distinguishable. The vulnerability occurs when the blocks which have been accessed are requested to be downloaded again in a short period of time. For example, let two sequences of accesses observed by a server be A' and A''. Suppose A' and A'' consist of sequent of reading d_1, d_2, and d_3; and d_1, d_2, and d_1, respectively. Therefore, A' and A'' can be represented as follows:

$$A' = \text{Read}(d_1), \text{Read}(d_2), \text{Read}(d_3)$$

$$A'' = \text{Read}(d_1), \text{Read}(d_2), \text{Read}(d_1)$$

Suppose d_1 was uploaded back to a server after finish the first access operation, and it has not be chosen as old blocks of the second access. Therefore, the probability that d_1 will be chosen to be read during the third request of A' is $(h - o)/N$. On the other hand, as d_1 is requested to be read by a client on the third access of A'', the probability that d_1 will be read is 1. Since the two probability values are obviously distinguishable, the server can determine with high probability which block is interested. However, a client is the person who controls the request of access in ORAM system. Therefore, the client can design of how often the same information will be repeatably requested which leads to two solutions that can be used to solve this problem.

1. **Does not request same data more often than one time of $N/(h - o)$ accesses:** To make the selection looks like a uniform distribution, a client has to limit the number of

accesses to the block that has been accessed. Therefore, from the example given in the beginning of this section, the number of accesses to those blocks should be approximately 1 time of $N/(h-o)$ accesses.

2. **Random selecting from the history list:** Another solution to keep A' indistinguishable from A'' is to create the *history list*, and client randomly chooses l blocks from this list as additional blocks for each access operation. To do so client starts with *'warm-up access'* at the very beginning of its first communication with a server. This warm-up access is dummy accesses for constructing the history list before starting the real communication. It occurs only once when a new client joins the system. By the fact that if a new list is continually added, the number of blocks in the history list will be eventually equal to N. Therefore, oldest l blocks in the history list will be removed and be replaced by the set of recently accessed blocks after reaching a specified number of accesses. This number varies according to how much frequency that client wants to access the same block location. However, the number of accesses before replacing is bounded by $N/(h-o)$ to make sure that the probability of accessing the oldest set of blocks in the history list is not greater than $(h-o)/N$.

Considering those two solutions, the first solution is limited by the frequency of accessing the information. Client has to control the average number of accesses not more than 1 of $N/(h-o)$ which is quite impractical. On the other hand, the second solution is more flexible and more practical. The trade-off is client has to do warm-up access to construct the history list. However, the warm-up method will be performed one and only one on new communication which is negligible for long-term communication.

Proof of Randomization Over RM-ORAM's Access Pattern

Recall from Section 5.2, the block size of PosORAM and DataORAM may be different and it is known by a server. Therefore, once a client does an access operation, a server knows which type of ORAM is being accessed. The randomness of access pattern is therefore considered in two paths which are the randomness of access pattern over DataORAM and PosORAM.

Suppose A is an access sequence seen by a server consists of i accesses (either upload or download), $adr_j[data_j]$ is a set of addresses revealed to server during j^{th} access where $j \in \{1 \leq x \leq i \mid x \in \mathbb{Z}\}$. It can be illustrated as Equation 5.2.

$$A = (adr_i[data_i], adr_{i-1}[data_{i-1}], \ldots, adr_1[data_1]) \tag{5.2}$$

Since DataORAM and PosORAM are different and known by the server, the set of addresses that has been accessed during j^{th} access can be defined as:

$$adr_j[data_j] = AddrData_j \uplus AddrPos_j \tag{5.3}$$

where $AddrData_j$ and $AddrPos_j$ are the set of addresses that has been accessed on DataORAM and PosORAM, respectively during j^{th} access. Therefore, the series of access requests A from Equation 5.2 can be separated to A_{data} and A_{pos} which are an access sequence range i on DataORAM and PosORAM, respectively. We can define the A_{data} and A_{pos} are as following equations:

$$A_{data} = (AddrData_i, AddrData_{i-1}, \ldots, AddrData_1) \tag{5.4}$$

$$A_{pos} = (AddrPos_i, AddrPos_{i-1}, \ldots, AddrPos_1) \tag{5.5}$$

By the fact that there are three possible patterns of series of accesses generated by a client, they can be described as in Lemma 4, Lemma 5, and Lemma 6. To show the random access pattern of RM-ORAM is secure, we do the proof by case analysis of these three lemmas.

Lemma 4. *Different sequences of accesses on the same set of data blocks of RM-ORAM are indistinguishable from the adversary who has limited computational power to polynomial time computation.*

Proof. According to the Section 3, whether or not the content is changed, the content is re-encrypted by random secret key. Therefore, the series of accessed blocks is indistinguishable from a random string. In addition, since the content of data blocks of each access is randomly changed according to RM-ORAM access operation, the server cannot identify the relationship between the content and the data block. Although A_{data} and A'_{data} access the same set of data blocks, the uploaded content of both access sequences does not necessary to be the same. Therefore A_{data} and A'_{data} is indistinguishable by the polynomial time adversary. □

Lemma 5. *Different sequences of accesses on the different data blocks of RM-ORAM are indistinguishable from the adversary who has limited computational power to polynomial time computation.*

Proof. Suppose there are two access sequences A_{data} and A'_{data}, these two sequences have length i and access on the different data blocks. According to the Section 5.4.2, o blocks are selected from the previous operation to be accessed in current operation to make sure that consecutive operation does not access on a different set of data blocks. Furthermore, a set of data blocks which has been downloaded is randomly replaced by data within StashData which may or may not be the same set of previous data. Hence, although A_{data} and A'_{data} access on different data blocks, it does not mean that client try to access two different groups of data. Therefore, Lemma 5 holds by the same proof of Lemma 4. □

Lemma 6. *Different sequences of accesses on the some (but not all) of the same blocks of RM-ORAM are indistinguishable from the adversary who has limited computational power to polynomial time computation.*

Proof. As a discussion that has been given in Section 5.4.2, two access sequences can be distinguished if the frequency of accessing data blocks which have been accessed is different. However, by varying the size of history list, a client can control the probability of accessing the block listed in the history list. High probability means the block is possible to be accessed more often although it is not a data of interest. Suppose the probability of data of interest is among of chosen l data blocks from history list size L is l/L. It means data of interest possibly be selected every L/l accesses. By maintaining this probability to be greater than or equal to l/L; although two sequences have different frequencies of accessing data of interest, they look similar from server's perspective. Hence, the Lemma 6 holds when the content of data blocks is indistinguishable by randomised encryption. □

Lemma 7. *Different sequences of accesses on position map blocks of RM-ORAM are indistinguishable from the adversary who has limited computational power to polynomial time computation.*

Proof. Choosing the position map blocks to be accessed is related with what data blocks are going to be downloaded. Therefore, when l data blocks from history list are chosen, the $r \times l$ blocks from the position map blocks that have been accessed are also chosen but the order may be different. With the same manner of proving in Lemma 4, 5 and 6, the differentiation of A_{pos} and A'_{pos} is indistinguishable. □

Theorem 4. *The randomness of RM-ORAM's access pattern is secure from the adversary who has limited computational power to polynomial time computation.*

Proof. Suppose A and A' are two different access patterns of client. B_A and $B_{A'}$ is the set of blocks accessed by client of A and A', respectively; and $B_A, B_{A'} \in B_N$ where B_N is a set of every block in ORAM. By case analysis, there are three possible cases of blocks accessed by two different access sequences which are $B_A = B_{A'}$, $B_A \cap B_{A'} = \emptyset$, and $(B_A \cap B_{A'} \neq \emptyset) \wedge (B_A \neq B_{A'})$. For the case $B_A = B_{A'}$, it can be covered by Lemma 4 and 7; case $B_A \cap B_{A'} = \emptyset$ can be covered by Lemma 5 and 7; and for the last case $(B_A \cap B_{A'} \neq \emptyset) \wedge (B_A \neq B_{A'})$, it can be covered by Lemma 6 and 7. Therefore, any sequences of access pattern are secure under the advisory whose computational power is bounded by polynomial time. □

Chapter 6

M-ORAM and RM-ORAM Performance Analysis

The aim of designing M-ORAM is to decrease the bandwidth cost when a client accesses data on a server while the design of RM-ORAM is aimed to be used on the client which has limited storage capacity. With the aim of different applications, the different methods are used to inspect the performance. Therefore the details of performance analysis of M-ORAM and RM-ORAM will be given in separate section. For clarity in the analysis, Path ORAM is used as a benchmark in comparing, since it is one of famous ORAM constructions and its operation is similar to M-ORAM. However, as the recursive ORAM has slightly different operations from normal ORAM construction, the same type of construction must be paired for comparison. In this chapter, the performance analysis and suggested parameters (e.g. size of stash and number of blocks downloaded per access operation) used of M-ORAM and R-ORAM are discussed in Section 6.1 and Section 6.2. However, according to the parameter used in Chapter 4 and Chapter 5 are slightly different. To clarify of reading the parameter used in this chapter are referred as in Table 6.1. To strengthen our analysis, M-ORAM, RM-ORAM and Path ORAM were implemented for the experiment. The constructions were implemented in Python and performed on a computer running Windows7 64 bit with Intel i3 CPU running at 3.3 GHz with 8 Gigabyte memory. These additional experimental results provide the insights information which is hard to be covered by theoretical analysis such as the behaviour of data random relocation or stash usage.

Table 6.1: Notation of parameter used for experimental analysis

Parameter	Description
data block	Block which contains actual data of client.
position map block	Block which contains position map data.
DataORAM	ORAM which contains data block.
PosORAM	ORAM which contains position map block.
StashData	Stash which contains data block downloaded between download/upload. For M-ORAM, StashData consists of h stashes with same width w, and they are arranged as $h \times w$ matrix format. On the other hand, there is only one stash contained in StashData.
StashPath	Stash which contains position map block downloaded between download/upload.

Table 6.1: Notation of parameter used for experimental analysis

Parameter	Description
StashPos	Stash of Path ORAM which contains position map block downloaded between download/upload.
N	Size of DataORAM (block unit).
N'	Size of PosORAM (block unit).
h	Number of data blocks which are being downloaded uploaded per level of recursion.
w	Size of StashData.
m	Number of pointer tuples within each of position map block.
r	Number of levels of recursion.
n	Number of new blocks chosen from DataORAM.
o	Number of old blocks which will be chosen for next access operation.
l	Number of blocks from history list which will be chosen for next access operation.
s	Size of StashData after finish download operation.
c	Size of counter.

6.1 M-ORAM Performance Analysis

The key aim of M-ORAM is to reduce the bandwidth cost when the client accesses data on the server. To optimise the bandwidth cost, M-ORAM is designed to have a constant bandwidth cost although the size of DataORAM is increasing. We analyse the performance of M-ORAM in three aspects: communication overhead, computational overhead, and storage usage. Each of these aspects will be discussed in details in Section 6.1.1, Section 6.1.2, and Section 6.1.3, respectively.

6.1.1 Communication Overhead (Bandwidth Cost)

M-ORAM is designed to have $O(1)$ asymptotic bandwidth cost once an access operation has been executed. In the other word, it consumes a constant bandwidth although the size of DataORAM is changed. Using DataORAM with height h, a client must download h blocks and upload h blocks for every access operation. Rather than increasing h when DataORAM size is growing as other ORAM schemes, w of DataORAM is increasing while h is kept as a constant for M-ORAM. Therefore, the number of downloaded blocks is independently from the DataORAM size which is a key strength of M-ORAM. However, there are limits on the number of downloaded blocks. This limitation impacts on security of the random access pattern which has been discussed in Section 4.3.2 of Chapter 4. It causes the height of DataORAM must be at least 5 blocks ($o = 2, l = 1, n = 1$). Assuming these limits are considered, we claim that the bandwidth cost of the M-ORAM construction is independent of ORAM size and bounded by $O(1)$. As asymptotic bandwidth cost of other existing ORAM schemes is polylogarithmic (e.g. $O(\log N)$ of Path ORAM), M-ORAM consumes less bandwidth cost than other existing schemes.

Figure 6.1 shows the bandwidth cost (per access operation) of Path ORAM and M-ORAM

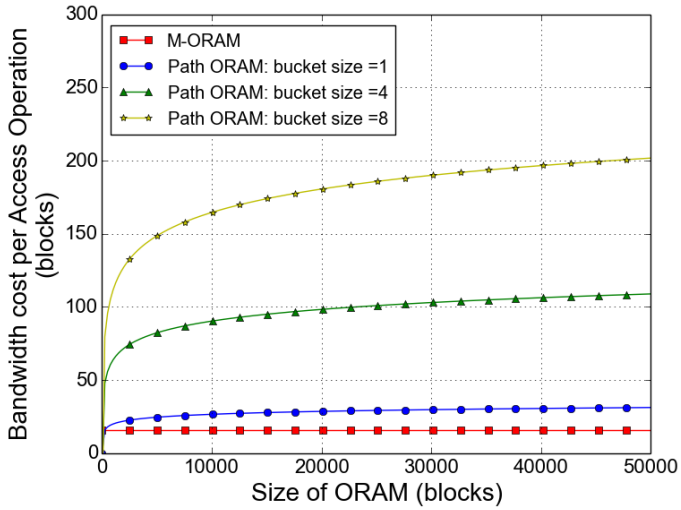


Figure 6.1: Bandwidth cost with height = 8

for different DataORAM sizes in block units. Height of DataORAM of M-ORAM is set to 8 while for Path ORAM is varied according to the height of binary tree. In Path ORAM, there are two methods to increase the size of DataORAM: increasing the number of blocks within a bucket, or increasing the height of the DataORAM. However, both incur of increasing the bandwidth cost. With Path ORAM as the DataORAM size increases, the path length increases, leading to increasing bandwidth cost. Besides, although Path ORAM increases the number of blocks in the bucket rather than expanding the path length, it requires downloading every bucket within the path which is also causing of more bandwidth consuming. On the other hand, M-ORAM has a fixed number of accessed blocks per access operation due to the constant height of the storage structure. Although the size of ORAM is increasing, the bandwidth cost in our M-ORAM remains unchanged.

6.1.2 Computational Overhead

For analysing the computation overhead, we measure it in terms of *time complexity* of the operation. According to the Section 4.2 of Chapter 4, the main operations to be performed in M-ORAM and their costs are:

- **Pseudo random number generation:** We suppose that the PRNG which is used in our construction is efficient. Therefore it has cost $O(1)$ time complexity.
- **Look up address in position map:** It costs $O(1)$ because position map is an array of tuples which are sorted by ID of data. Therefore, the client can directly retrieve information from specific tuple if it knows the ID number.
- **Randomly write h blocks to StashData:**. h blocks are uniquely written at random to h stashes within StashData. The random numbers is bounded by h, where h is a constant height of ORAM construction. Therefore, it has cost $O(1)$.
- **Randomly choose h blocks from StashData:**. The range of random numbers is bounded by height times by width of StashData. According to those two value are constant, the random number is bounded by a constant number. Therefore, it has cost $O(1)$.

- **Updating the data in position map:** Due to h blocks are relocated per each operation (either download or upload operation) and the number of operations executed per access operation is constant, the computational complexity for updating the block location in position map is $O(1)$.
- **Randomly choose some of the old blocks during download operation:** To achieve ORAM security requirements, random o from h blocks of the previous operation are chosen. Since o and h are a constant value, the computational cost of randomly choosing the certain number of elements from a finite set has cost $O(1)$.
- **Randomly choose some blocks from a list of history list:** According to the computational cost of randomly choosing the certain number of objects from finite set is $O(1)$, the computation overhead of random choosing l blocks from the history list which is a finite set is also $O(1)$, where l is constant.
- **Randomly choose new blocks during download operation:** Range of random numbers is bounded by $h - o - l$, where o, l and h are constant. Therefore, it has cost $O(1)$.

Total computational cost is the sum of the above operations, which gives the time complexity (TC) of RM-ORAM operation is:

$$TC = O(1) \tag{6.1}$$

6.1.3 Storage Usage

In M-ORAM, the stash usage is bounded by two controllable parameters: height of DataORAM and the width of stash. Those two parameters are influenced by the behaviour of data blocks relocation after upload. To measure the block relocation behaviour, the probability of block uploaded to its previous location of it was downloaded is taken into consideration. This probability is referred as *probability of duplication*. The lower probability means more chance of data block relocated to somewhere else rather than its previous location. Suppose h data blocks downloaded/uploaded of each access operation and w is the width of StashData in blocks unit. The probability of duplication can be described as equation of Theorem 5

Theorem 5. *Let* DUP *denotes the event of one data block will be uploaded to its previous location before download. Suppose there are h stashes within StashData and each stash width is equal to w blocks. The probability of DUP is:*

$$Pr(DUP) = \frac{1}{h \times w} \tag{6.2}$$

Proof. According to the detailed operation which has been described in Section 4.2 of Chapter 4, each of h downloaded data blocks is uniquely stored in h stashes, and h data blocks are randomly selected from each stash to be uploaded. Suppose a data block has been retrieved from coordinate (i, j). Since there are h stashes and each stash has w blocks, the event DUP will happen when the data block is stored in $stash_i$ and then chosen from $stash_i$ to be uploaded. Therefore, the probability of duplication is $1/(h \times w)$ which is held by Theorem 5. □

Figure 6.2 shows the plots of $Pr(DUP)$ with the different heights of DataORAM and different widths of stash. Increasing width of stash causes the chance of data block written to its

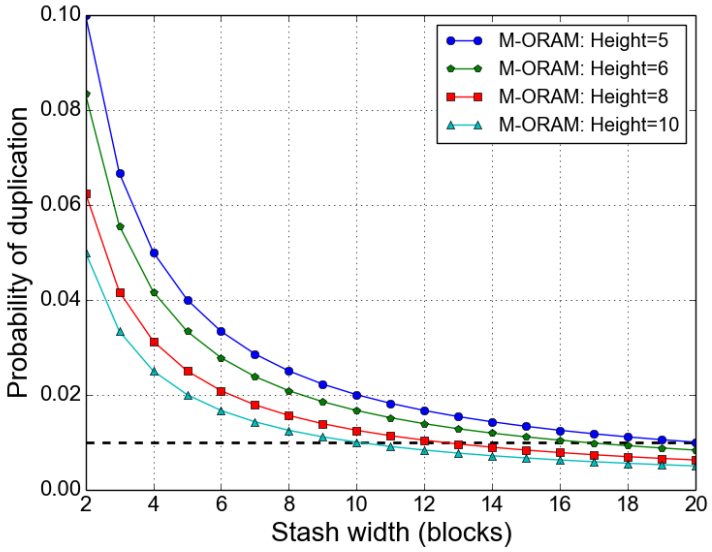


Figure 6.2: Probability of duplication in M-ORAM

previous location is decreased. Therefore, the width of StashData impacts to the data block relocation and it should be large enough to allow the movement of data block look similar to a random movement. Another important feature of our M-ORAM design is that the stash will never overflow. As the number of downloaded blocks is equal to the number of blocks uploaded to the server for every access operation, stashes always have empty spaces for the upcoming blocks downloaded from next access operation.

6.1.4 Suggested Parameter Value for M-ORAM

To accurate the suggested parameters used by M-ORAM, the experiment done on Windows7 64 bit with Intel i3 CPU running at 3.3 GHz with 8 Gigabyte memory. Both of M-ORAM and Path ORAM are implemented by Python. There are two significant parameters affecting the performance and security of M-ORAM which are the height of DataORAM (h) and width of StashData (w). Changing h impacts on system's communication overhead. Changing w is impacts on storage overhead on a client. Moreover, by changing h and w, it affects to the randomness of data block relocation in DataORAM. According to Section 4.3.2, three types of data block are downloaded during access operation to keep two different access sequences indistinguishable. Since h downloaded data blocks consist of o blocks from old list, l blocks from history list, and n blocks from the list of data which is not included in old list and history list; the number of o, l, and n should be well defined before the suggested value of h and w are given.

Suggested number of data blocks chosen from the old list

To accurately determine the proper number of blocks from previous access which have to be chosen for current operation, we compare our construction with Path ORAM. We first define o as the number of blocks in current operation which were also accessed in the previous operation, and $\bar{o}$ is the average number of o which will be accessed during an access operation.

Theorem 6. *The average number of* old blocks *accessed across two Path ORAM is 2 when height of binary tree tends to infinity.*

$$\lim_{H\to\infty} \bar{o}_{path\_oram} = 2 \tag{6.3}$$

Proof. Figure 6.3, shows the binary-tree structure of Path ORAM with height H. It is used for explaining the proof. Suppose that set of nodes chosen during the previous access operation is represented as the grey circles in the binary tree. There are 2^{H-1} possible paths in total from a leaf node to a root in binary tree. The next access operation downloads one of all of the possible paths. As the node of interest are assigned with random leaf ID according to Path ORAM's operation, and the next path is chosen uniformly at random. There is only one possible path that could be chosen out of 2^{H-1} that results in H duplicate nodes (i.e. the exact same path as the previous operation). If the exact same path is not chosen, then there are 2^{i-1} possible paths that result in $H - i$ duplicate nodes, where $i \in \{1, 2, \ldots, H-1\}$. Therefore, the average number of duplicate nodes/blocks is:

$$\begin{aligned}\bar{o}_{path\_oram} &= \frac{1}{2^{H-1}}H + \sum_{i=1}^{H-1} \frac{2^{i-1}}{2^{H-1}}(H-i) \\ &= \frac{\sum_{i=0}^{H} 2^i}{2^{H-1}}\end{aligned} \tag{6.4}$$

Considering as a geometric series, as H tends to infinity, the $\bar{o}_{path\_oram}$ tends to 2. □

Therefore, to achieve the same or better security level of Path ORAM, the number of o_{rm_oram} having to be accessed must be equal or greater than 2.

Suggested number of data blocks chosen from the history list

Regarding the appropriate number of l chosen from history list, it defines as Theorem 7.

Theorem 7. *The appropriate number of blocks chosen from history list, l when $N \gg h$ is:*

$$1 \le l \le \frac{(h-o)}{2} \tag{6.5}$$

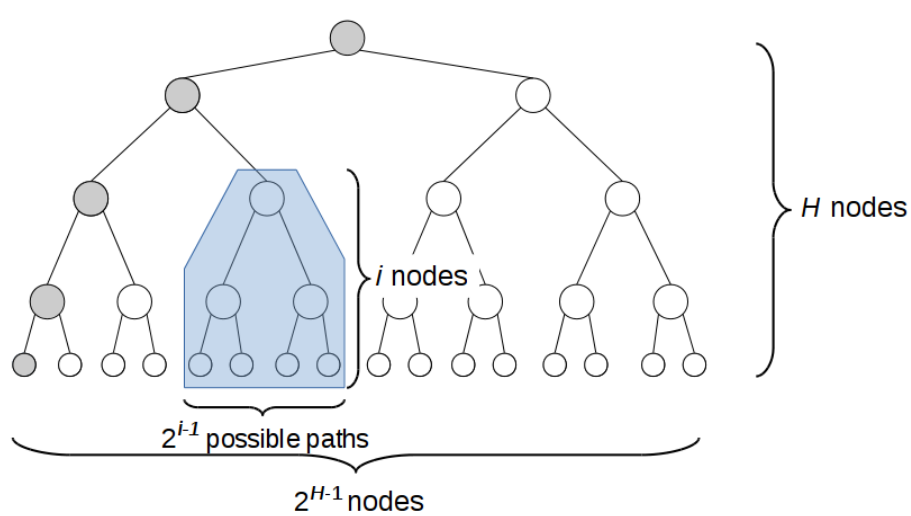


Figure 6.3: Possible path of $H - i$ overlapped nodes

Proof. Suppose h blocks are chosen to be accessed in each access request. There are o blocks come from previous access and l blocks are chosen from the history list. Hence, the number of blocks which have never been accessed since $N/(h-o)$ accesses ago is $h-l-o$ blocks, and it is referred as *new list*. Recall that the $h-l-o$ lists in the history list will be replaced by the new lists after at most $(N/(h-o))^{th}$ access. Therefore, the number of blocks which are listed in the history list is:

$$\text{number of block list} \leq \left(\frac{N}{(h-o)} \times (h-l-o)\right) \tag{6.6}$$

According to the security issue that has been discussed at beginning of this section, by considering the frequency of accesses, the frequency of accessing the history list must be equal to or more often than accessing the new list. Therefore, to define the upper bound of l, the maximum number of blocks in the list is taken into consideration. At a maximum number of blocks listed in the history list, the average frequency of accessing the history list is equal to the new list. In the other words, the probability of data of interest comes from the chosen blocks from history list is equal to the probability of data of interest comes from the chosen blocks from the new list. Suppose that $N \gg h > o$, from the Equation 6.6 we can derive the possible solution of l as following equation:

$$\begin{aligned} \frac{N}{2} &= \left(\frac{N}{(h-o)} \times (h-l-o)\right) \\ l &= \frac{(h-o)}{2} \end{aligned} \tag{6.7}$$

However, in fact, the probability of data of interest comes from the chosen blocks from history list depends on a ratio of l and the number of lists in the history list. Therefore, the number of l holds by Equation 6.5. □

Suggested appropriate value of *h* and *w* of M-ORAM

Since, the minimum value of o and l is 2 and 1 according to Theorem 6 and Theorem 7, respectively; the minimum value of n is therefore 1 as $h = o + l + n$. Hence, at least 4 blocks must be downloaded in each access operation. To figure out the appropriate value of w, we focus on the w which makes the data block relocation seem to be random from a server point of view. To measure the randomness, we use the standard from *National Institute of Standards and Technology (NIST)* to measure the experimental results. The experiment was done by focusing on the movement of one data block on DataORAM for four million access operations. We use chi-square (χ^2) test for measuring a randomness. According to NIST, the significant level (α) > 0.01 means the sequence of samples is random. Figure 6.4 shows *p-value* of data block coordinate of four million access operations. We test on three different sizes of DataORAM: 1200, 2400, and 3000 blocks, with $h = 4$. The result shows that when the width of Stash-Data is greater than 7 blocks, the movement of data block is seen to be random from server's perspective

Now, 4 blocks of h and 7 blocks of w are the minimum requirements for constructing M-ORAM. We do more further by comparing our experimental results with Path ORAM to find the most suitable parameter value. The experimental results of M-ORAM stash usage at *PR(DUP)* = 0.01 are used to compare with stash usage of Path ORAM at the same height. In both Path ORAM and M-ORAM, the stash size is the main difference with respect to client

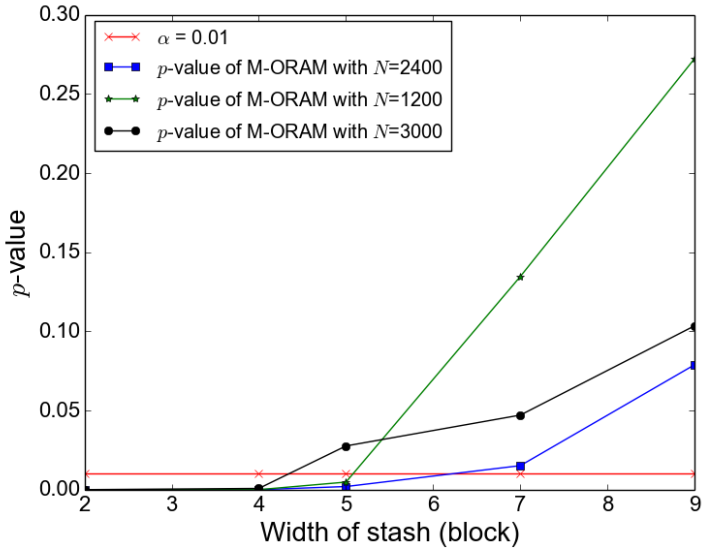
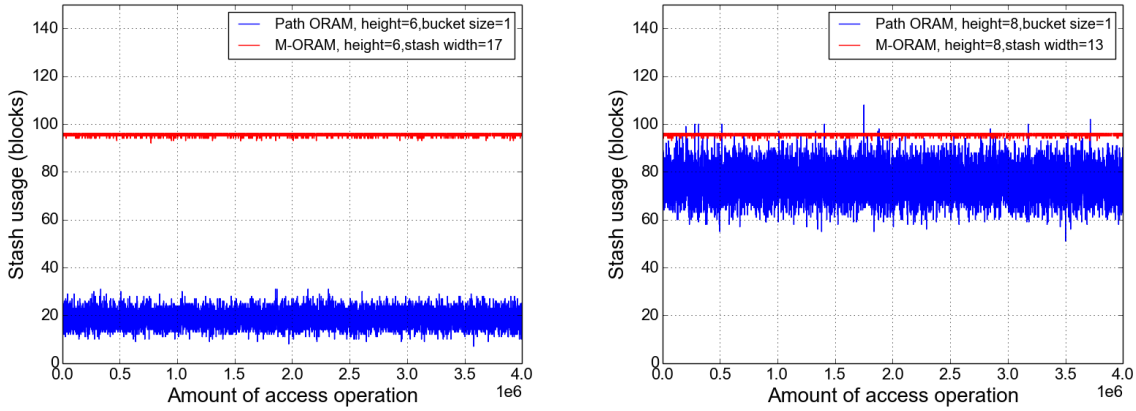


Figure 6.4: *p-value* χ^2 test over varied size of w with $h = 4$

storage requirements. Path ORAM has a single stash while M-ORAM has H stashes, all of the same width. Figures 6.5 shows the total stash usage for two selected experiments, one with DataORAM height of 6 and the other 8, for 4 million access operations. As $PR(DUP) = 0.01$, w will be 17 and 13 for h equals to 6 and 8, respectively, according to Equation 6.2. Figures 6.5a compares the stash usage of Path ORAM and M-ORAM with equivalent sized server storage. The upper bound of stash usage in M-ORAM is 96 blocks under 0.01 probability of duplication with stash width equal to 17; on the other hand, Path ORAM uses only 34 blocks, significantly better than M-ORAM. However with a larger height (Figure 6.5b), although the average stash usage of Path ORAM is still lower than M-ORAM, the maximum stash usage exceeds M-ORAM. By the fact that the maximum stash usage should be the allocated storage to avoid overflow, M-ORAM requires smaller reserved space of stash than Path ORAM. Since the minimum requirement of h and w is 4 and 7, DataORAM height equals to 8 and StashData width equal to 13 are in acceptable range. Therefore, $h = 8$ and $w = 13$ are the suggested values for M-ORAM construction.

6.2 RM-ORAM Performance Analysis

In this section, the performance of RM-ORAM is analysed in three major aspects: storage efficiency, communication overhead, and the client's computational overhead. A key aim of designing RM-ORAM is to use in constrained devices, therefore we focus on maximising the efficiency of storage usage without significantly increasing the other two aspects. Analysis of bandwidth cost on RM-ORAM system is provided in Section 6.2.1. Analysis on computational overhead is given in Section 6.2.2 while analysis of efficient storage usage is discussed in Section 6.2.3.



(a) DataORAM height = 6

(b) DataORAM height = 8

Figure 6.5: Stash usage of M-ORAM and Path ORAM

6.2.1 Communication Overhead (Bandwidth Cost)

In RM-ORAM the new client has to construct a history list from very beginning of communication to a server. It does so by generating dummy accesses to a server called warm-up communication. However, once the history list is already created, the warm-up communication is no longer necessary. Therefore, the bandwidth cost of dummy accesses can be ignored for long term communication. In RM-ORAM the bandwidth spent for real communication depends on 2 parameters: h and r. A client must download and upload h data blocks plus $h{\cdot}(r-1)$ position map blocks for an access operation. The number of levels of recursion (r) can calculate as shown in Lemma 8, and the number of blocks accessed per upload/download operation is h. Therefore, the asymptotic bandwidth cost of RM-ORAM is $O(\log N)$ as described in Theorem 8.

Lemma 8. *The number of levels of recursion of RM-ORAM construction is:*

$$r = \lceil \log_m N \rceil \tag{6.8}$$

Proof. Suppose Posmap_0 has m pointer tuples containing a pointer. Each pointer points to position map blocks in Posmap_1. As there are m pointer tuples per position map block, the total number of position map blocks of Posmap_1 pointed by the pointers is m. With the reason that every position map block has m pointer tuples, the number of pointers contained in Posmap_1 is m^2. Those m^2 pointers will point to m^2 position map blocks in Posmap_2 which contain other m^3 pointer tuples and so on. In the other words, the number of pointers grows exponentially by a factor of m per each level of recursion. We know that the last Posmap must contain at least N pointer tuples or N/m position map blocks to cover N data blocks in DataORAM. Suppose it needs r levels of recursion to reach the DataORAM, the last Posmap will contain m^r pointer tuples. Therefore, $m^r = N$ and it can be derived to be Equation 6.8. □

Theorem 8. *Let BW refers to bandwidth cost of RM-ORAM:*

$$BW = O(\log N) \tag{6.9}$$

Proof. As h and m are a constant and independent of the size of ORAM, and there are two parameters: a number of levels of recursion and a position map block's index header which are

varied by a number of data blocks in DataORAM. However, the index header is very small compared with a size of position map block, the only parameter varied by the size of DataORAM is a number of levels of recursion. Therefore the asymptotic bandwidth of RM-ORAM depends on r and bounded by $O(\log N)$ according to Lemma 8. □

6.2.2 Computational Overhead

For analysing the computation overhead, we measure it in terms of *time complexity* of the operation. According to the Section 5.3 of Chapter 5, the main operations to be performed in RM-ORAM and their costs are:

- **Pseudo random number generation:** We suppose that the PRNG which is used in our construction is efficient. Therefore it has cost $O(1)$ time complexity.
- **Searching for particular information in StashData:** It costs $O(1)$ because $stash_{data}$ has a constant size.
- **Randomly choose h blocks from StashData:**. The range of random numbers is bounded by s, where s is a constant number of elements within StashData. Therefore, it has cost $O(1)$.
- **Randomly assign the new address:** This operation causes the client to randomly select the new address for the chosen h blocks in StashData, and $h \log_m N$ blocks of position map (StashPath and Posmap_0). Therefore, the time complexity of this operation is bounded by $O(\log N)$.
- **Updating the pointer in position map block:** This operation causes the client to access $h \log_m N$ blocks for updating their content. Therefore, it has cost $O(\log N)$.
- **Randomly choose some of the old blocks during download operation:** Random o from h blocks of the previous upload are chosen. Because o is some constant value, the computational cost of choosing the random constant number of elements from a finite set has cost $O(1)$.
- **Randomly choose some of the blocks from a list of history list:** According to the computational cost of randomly choosing constant number of objects from finite set is $O(1)$, the computation overhead of random choosing l blocks from the history list which is a finite set is also $O(1)$.
- **Randomly choose new blocks during download operation:** Range of random numbers is bounded by $h - o - l$, where o, l and h are constant. Therefore, it has cost $O(1)$.

Total computational cost is the sum of the above operations, which gives the time complexity (TC) of RM-ORAM operation is:

$$TC = O(\log N) \tag{6.10}$$

6.2.3 Storage Usage

The efficiency of storage usage in ORAM is one of important performance aspect. Decreasing of storage space reservation means increasing the number of operations to ensure that the construction can achieve all of ORAM's security requirements. Generally, the storage of ORAM system can be categorised to be a storage on a server and a storage on a client as it has been described in Section 5.2. Therefore, the efficiency analysis of this section is done on the use of storage on a client and server.

Client Storage Usage Efficiency

In M-ORAM, the client stores N logical addresses of data block in position map and downloaded data content in a stash. Stash has a constant size while the size of position map varies according to the size of ORAM. Therefore, $O(N)$ blocks are reserved within a client. In RM-ORAM it requires only $O(\log N)$ blocks on a client which consists of four types of storage: StashPath, StashData, Posmap_0 and history list. The number of reserved bits on the client for history list is varied according to the number of blocks in ORAM. However, the history list requires only one bit for a block to show that the block has been accessed in the past accesses. Therefore the space requirement for the history list is small and it can be overlooked when compared with other types of storage. The number of reserved blocks on the client for StashData and Posmap_0 is constant while StashPath is varied according to the number of levels of recursion. As the number of levels of recursion is $\lceil \log_m N \rceil$, it costs $O(\log N)$ reserved blocks within a client.

Server Storage Usage Efficiency

Some ORAM constructions store both data and dummy information on the server (e.g. [1] and [2]). The storage of dummy information is necessary to preserve the security requirement of uploading/downloading the same number of blocks. However, it may reduce the space available for storing data on the server. In both M-ORAM and RM-ORAM, dummy information is not used, and therefore the DataORAM can store 100% data.

In RM-ORAM, besides DataORAM, PosORAM is also stored on the server which contains some unused spaces in some cases. Each position map block contained in PosORAM has a constant number of pointers which each pointer points to the position map block of the next level of recursion. Every recursive ORAM construction shares the same position map block structure but different Posmap constructions. For example, RM-ORAM uses matrix based construction and Path ORAM uses binary tree based construction for Posmap. In RM-ORAM if $N \in \{m^i \mid i \in \mathbb{Z}^+\}$, there are no unused spaces in PosORAM. On the other hand, if it is not, there are some pointer tuples that are not used. For Path ORAM, Posmaps are constructed based on a binary tree data structure. As the number of nodes in binary tree is $2^h - 1$ where $h \in \mathbb{Z}^+ \backslash \{1\}$, it will have unused space in PosORAM when $1 < \frac{m \cdot (2^{h_i} - 1)}{(2^{h_{i+1}} - 1)} < 2$, where h_i is height of binary tree Posmap of ith level of recursion and $h_{i+1} > h_i$. Since RM-ORAM and recursive Path ORAM have same r when they have same number of m and N (Page 9 of [4] and Lemma 8), we show in Theorem 9 that the number non-used space in PosORAM of RM-ORAM is equal to or less than the number of non-used space in PosORAM of Path ORAM when both constructions share the same r.

Lemma 9. *For $x_{i+1}, x_i, m \in \mathbb{Z}^+$, $x_i < x_{i+1}$, and $m \geq 2$. There are some x_i which:*

$$m \cdot x_i \leq x_{i+1} < m \cdot (x_i + 1) \tag{6.11}$$

Lemma 10. *For $y_{i+1}, y_i, m \in \mathbb{Z}^+$, $2^{y_i} < 2^{y_{i+1}}$, and $m \geq 2$. There are some y_i which:*

$$m \cdot (2^{y_i} - 1) \leq 2^{y_{i+1}} - 1 < m \cdot (2^{y_i+1} - 1) \tag{6.12}$$

Theorem 9. *Let the number of blocks of binary tree's Posmap and the number of blocks of RM-ORAM's Posmap at level i of recursion be $|BPos_i|$ and $|RPos_i|$, respectively. The number of non-used pointer tuples in PosORAM of binary tree ORAM is equal to or greater than the number of non-used pointer tuples in PosORAM of RM-ORAM.*

$$\sum_{i=1}^{r-1}(m \cdot |BPos_i| - |BPos_{i+1}|) \geq \sum_{i=1}^{r-1}(m \cdot |RPos_i| - |RPos_{i+1}|) \tag{6.13}$$

Proof. Suppose there are x_a blocks (nodes in binary tree) in Posmap_a, and each position map block has m pointer tuples. Lemma 9 and Lemma 10 represent the range of increasing number of position map blocks in Posmap_i to cover every position map block in Posmap_{i+1}, and let $x_{i+1} = z = 2^{y_{i+1}} - 1$. We can derive this equation to:

$$m \geq m \cdot (x_i + 1) - z > 0 \tag{6.14}$$

$$m \cdot 2^{y_i} \geq m \cdot (2^{y_i+1} - 1) - z > 0 \tag{6.15}$$

As z represents $|\text{BPos}_{i+1}|$ and $|\text{RPos}_{i+1}|$, while $m \cdot (x_i + 1)$ and $m \cdot (2^{y_i} - 1)$ is $m \cdot |\text{RPos}_i|$ and $m \cdot |\text{BPos}_i|$, respectively; Theorem 9 holds when Equation 6.14 and Equation 6.15 are true. □

6.2.4 Suggested Parameter Value for RM-ORAM

Same as M-ORAM, h and w of RM-ORAM are the significant parameters affecting the overall performance of a system. Decreasing or increasing of h and w will result in a change of bandwidth overhead, client storage usage, and random behaviour of block relocation on a server. However, there is one more important parameter of RM-ORAM which is m, the number of pointer tuples in a position map block. The bandwidth spent for downloading information also depends upon a value of m. To accurately define the appropriate value of those three parameters, the experiment is conducted and the results are compared with the recursive version of Path ORAM. RM-ORAM and recursive Path ORAM are implemented by Python and performed on a computer running Windows7 64 bit with Intel i3 CPU running at 3.3 GHz with 8 Gigabyte memory.

Since RM-ORAM inherits the construction from M-ORAM, the number of blocks from previous access (o) which has to be accessed in current operation should be 2 or more as described in Theorem 6. In addition, according to Theorem 7 the number of chosen blocks from history list(l) should be more than or equal to 1. As the set of blocks to be downloaded must include at least one new block (n), h must be greater than or equal to 4 since $h = o + l + n$.

As our aim is to provide equal or better performance than Path ORAM, we consider h that makes RM-ORAM consumes less bandwidth cost than Path ORAM when both constructions share the same number of N and m. By the fact that, RM-ORAM has to download constant h data blocks per access request while Path ORAM has to download $\log N$ data blocks. With a large number of N, the bandwidth cost of downloading data blocks of Path ORAM will exceed RM-ORAM. For now let's suppose RM-ORAM and Path ORAM download h data blocks, therefore the overall bandwidth cost of the both systems is only varied by the number of position map blocks which are downloaded.

Theorem 10. *RM-ORAM can achieve less number of downloaded position map blocks compared with Path ORAM when:*

$$h \cdot (r-1) \leq \left\lceil \log\left(\frac{N}{m}+1\right)\right\rceil + \sum_{i=1}^{r-2}(H_i) \tag{6.16}$$

where $H_i = \left\lceil \log\left(\frac{2^{(H_{i+1})}-1}{m}+1\right)\right\rceil$, $r = \lceil \log_m N \rceil$

Proof. RM-ORAM and Recursive Path ORAM share the same number of levels of recursion (Page 9 of [4]). RM-ORAM's client has to download h position map blocks per each level of recursion, therefore the total number of position map blocks is $h \cdot (r-1)$. For recursive Path ORAM, the client accesses to a different Posmap for each level of recursion. At each level, the number of position map blocks that client has to download is equal to a height of binary tree of this level. According to a construction of recursive Path ORAM, the height of binary tree at level i is $\left\lceil \log\left(\frac{2^{(H_{i+1})}-1}{m}+1\right)\right\rceil$ and the height of last binary tree (level $\lceil log_m N \rceil - 1$) is $\left\lceil \log\left(\frac{N}{m}+1\right)\right\rceil$. Suppose RM-ORAM and recursive Path ORAM have same N and m, RM-ORAM will achieve equal to or less than a number of downloaded position map blocks of recursive Path ORAM when:

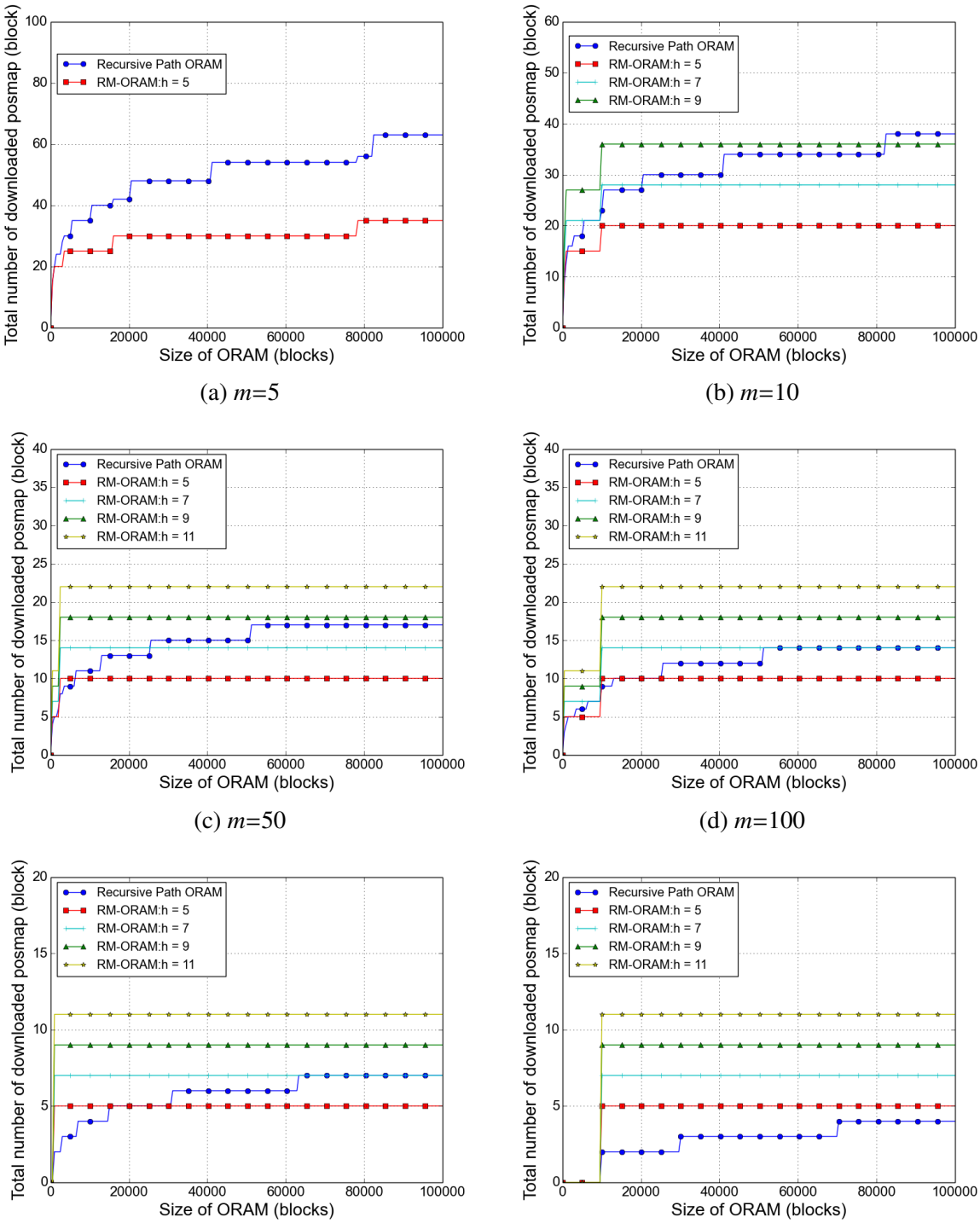
$$h \cdot (r-1) \leq \sum_{i=1}^{r-1}(H_i)$$

$$h \cdot (r-1) \leq \left\lceil \log\left(\frac{N}{m}+1\right)\right\rceil + \sum_{i=1}^{r-2}(H_i) \tag{6.17}$$

where H_i is the height of binary tree ORAM of each level of recursion. □

Equation 6.16 is used with varying values of N, m and h to compare the number of downloaded position map blocks for RM-ORAM and recursive Path ORAM in Figure 6.6. The experiment focuses on the range of h from 5 through 11 which is an acceptable range according to h must equal to or greater than 4. The experiment is tested on varied m and N and it returns the number of position map blocks downloaded per access operation as a result. Even though Path ORAM can achieve the less number of downloaded position map blocks than RM-ORAM when m is equal to or greater than 10000, RM-ORAM has an advantage over recursive Path ORAM when m is equal to or less than 1000 as our focus. Furthermore, at $h = 5$, the difference of the total number of position map blocks downloaded per access request between RM-ORAM and recursive Path ORAM is only 1 block, while the difference of the total number of downloaded data blocks between RM-ORAM and recursive Path ORAM is equal to or greater than 8. Therefore, the total number of blocks downloaded by recursive Path ORAM exceeds RM-ORAM. According to the implementation of RM-ORAM from the experimental results, RM-ORAM needs DataORAM size more than 50,000 blocks to have better bandwidth overhead for every recommended height of construction at $m \geq 100$. On the other hand, it requires around only 20,000 blocks size of DataORAM at $m = 50$ to beat PathORAM. Considering that RM-ORAM is designed to work on systems with limited resource, it is therefore possible to have a small size of DataORAM. Therefore, $4 \leq h \leq 7$ and $m \leq 50$ are the recommended values.

To define the appropriate value of StashData size (w), the randomness of data block relocation behaviour is taken into account. In addition to storing the downloaded data block, StashData data structure is beneficial for random selection of data to be uploaded. In other



(a) m=5 (b) m=10

(c) m=50 (d) m=100

(e) m=1000 (f) m=10000

Figure 6.6: Number of position map blocks downloaded per access request on different m

words, the size of StashData impacts to the data relocation and it should be large enough to allow the movement of data block look similar to a random relocation. To do the experiment, we focus on the moving of a specific information (e.g. data ID_1) after it has been consecutively accessed (downloaded then uploaded) for 1,000,000 times. The block locations (b_i) that data ID_1 has been stored are recorded as an experimental result dataset. The experiment is run over

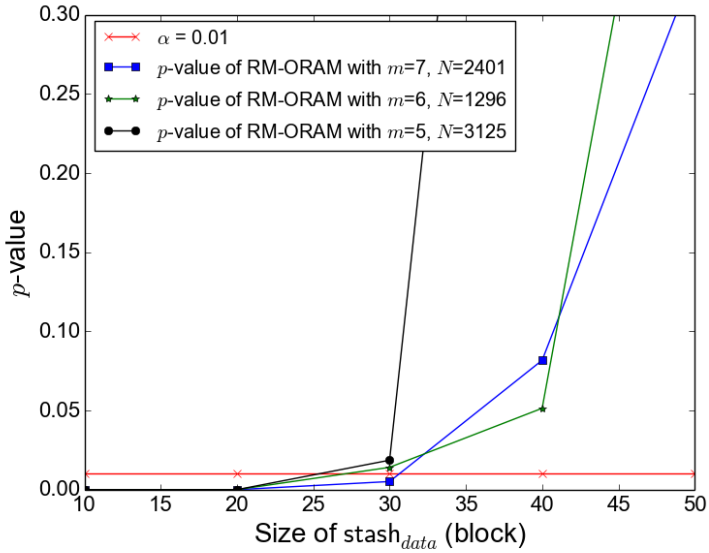


Figure 6.7: p-value of χ^2 test over varied size of StashData

m is equal to 5, 6, and 7 with $h = 6$ which is one of the suggested values of h. We use chi-square (χ^2) for testing the randomness of three result datasets of different sizes of DataORAM: 3125, 1296 and 2401 blocks with m equal to 5, 6, and 7, respectively. We use the standard from *National Institute of Standards and Technology (NIST)* [49] to measure the randomness of the experimental result. According to NIST, the significant level (α) > 0.01 means the sequence of sample is random. Figure 6.7 illustrates the p-value from χ^2 test of the different StashData sizes. After $w = 30 + h$ the movement of a data block can be considered as uniform random. Therefore, the suggested value of w is greater than 30 blocks.

To affirm the appropriate value of h, w and m, an another experiment is conducted. The maximum space required to store position map block and data block of RM-ORAM and Path ORAM are measured. The value of parameter: w, h, m used by RM-ORAM are chosen from the suggested range of each parameter. Figure 6.8 illustrates the maximum number of blocks stored in stash. Figure 6.8a, 6.8b, and 6.8c shows the maximum number of position map blocks kept in StashPath of RM-ORAM and StashPos of Path ORAM, while Figure 6.8d shows the comparison of maximum number of data blocks kept in StashData of both Path ORAM and RM-ORAM.

Recall from Path ORAM operation, leaf-ID is used to identify a path that contains data of interest, and it is randomly changed before an upload operation. There is a possibility that the data (whether it is position map block or data block) cannot be uploaded back to the path when its leaf ID has been changed. Those data left within the stash cause stash size growing. As the experimental results in Figure 6.8a to 6.8c, the space requirement for Path ORAM' stash grows significantly faster than RM-ORAM of $5 \leq h \leq 7$ with $5 \leq m \leq 7$ when size of DataORAM is increased. The growing size of StashData of Path ORAM in Figure 6.8d is the same trend as the growth of its StashPos. On the other hand, the size of StashData required in RM-ORAM is not impacted by the growth of DataORAM as a strong feature of M-ORAM family. As the value of parameters used in this experiment is in the suggested ranges which are: $w = 37$, $5 \leq h \leq 7$, and $5 \leq m \leq 7$, the result shows that RM-ORAM is more efficient than Path ORAM in every performance aspect. Therefore, the recommended parameter value for constructing RM-MORAM is: $w > 30$, $4 \leq h \leq 7$, and $m \leq 50$.

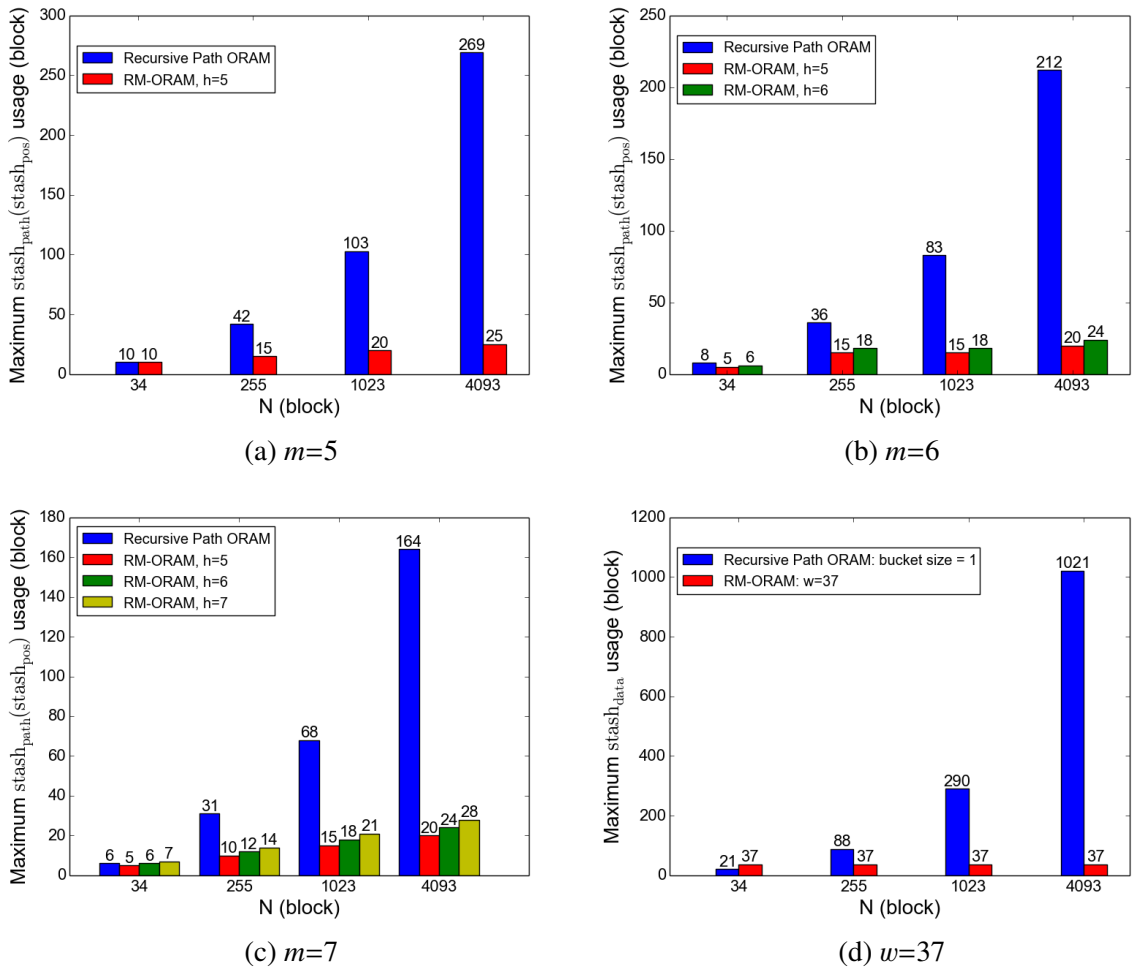


Figure 6.8: Maximum stash usage of Recursive Path ORAM and RM-ORAM with different m

Chapter 7

Conclusion and Future work

In this chapter, the conclusions of this thesis are outlined. We first start with Section 7.1 which recaps the purposes of ORAM research in order to remind the reader of the aim of the research. Then the discussions of research implications are given in Section 7.2, following by the limitation of research together with a suggestion for the future work in Section 7.3 and 7.4, respectively. The chapter ends with the conclusion in Section 7.5 which is given regarding the problems stated in ORAM research area.

7.1 Purpose of this research

The purpose of this research focused on reducing the overhead cost generated by ORAM algorithm in order to construct an ORAM system which tends to have similar performance as a normal system. By the fact that, in ORAM, the extra operations have to be added to prevent the leaking an extra source of information which may exploit to break the system. It is impossible not to have any additional costs when extra operations have been applied. Hence, the most likely way is to design the ORAM which generates an extra overhead as less as possible. Recall from the Chapter 1, the significant performance in ORAM system can be considered into bandwidth usage, computational complexity, and amount extra space usage in a storage. More or less of performance is dependent upon the ORAM data structure and how data is retrieved from the ORAM. Nonetheless, since the retrieval pattern is also limited by the data structure of ORAM, ORAM's structural has a higher impact on the performance than other factors. Therefore, our research is focusing on designing the new ORAM data structure which overcomes the performances limited by other ORAM existing schemes.

When considering all three performance aspects of ORAM, it is found that the efficiency of communication is the most important. Since nowadays the large storage and high-performance CPUs are generally available at affordable price, improving the bandwidth cost spent during transfer operation is first priority focused on our research. The second priority is a storage space usage. Although large storage is a common device for a personal computer, the smart devices such as IoT, smart appliance, or smartphone have very limited storage capacity. Similar to the personal computer, the smart device has to transfer the data back and forth with a server which sometimes the information is more confidential than an information sent by a common personal computer (e.g. medical information). Since the number of smart devices connected to cloud tends to increase year by year [50], designing the ORAM which is usable by the constrained storage space device is another purpose of our research.

In summary, the purpose of this research focused on improving two performance aspects

of ORAM: bandwidth of system and storage space usage on the client. These improving were introduced in two versions of novel ORAM structure called M-ORAM and RM-ORAM which their details were discussed in Chapter 4 and Chapter 5, respectively.

7.2 Research Implications

Generally, the information stored in ORAM is divided into a small piece of blocks with the same size. Each block may be or may not be related depending on the data structure of ORAM. Furthermore, the number of blocks according to the height of the structure is corresponding to the size of ORAM. As the number of blocks transferred per request is equal to the number of blocks according to the ORAM height, the matrix data structure seems more appropriate to be used than existing ORAM structures since the bandwidth is a factor that we want to control. As mentioned in Section 7.1, the way to retrieve data is relevant to how data in ORAM is arranged. Therefore, the algorithm of retrieving data is particularly designed for each ORAM construction. In this research, the two novel ORAM constructions called M-ORAM and RM-ORAM along with the procedures of how to retrieve data from ORAM server have been proposed. The two constructions are served for the different purposes. M-ORAM and RM-ORAM are slightly different in design. M-ORAM focuses on improving the efficiency of data transmission, while RM-ORAM aims of reducing the storage space usage on ORAM client.

According to the theoretical performance analysis having been discussed in Chapter 6, we can summarise the performance of well-known ORAM construction as Table 7.1. In order to achieve the ORAM security requirements, M-ORAM requires $O(1)$ for both bandwidth cost and computational complexity for any sizes of ORAM to proceed the oblivious transfer between ORAM client and ORAM server. As the first purpose of our research is to improve the bandwidth performance while still maintaining a behaviour of oblivious transfer, M-ORAM is clearly better than other well-known ORAM constructions. Yet, M-ORAM still not meet our second purpose since it requires $O(N)$ blocks in client storage to keep the necessary information. In fact, $O(N)$ is actually an $i \cdot N$, where i is a constant positive real number which less than 1; but it is still too far to be accepted for a limited-sized storage device. Hence, another ORAM

Table 7.1: Performance Comparison of Different ORAM Schemes

Scheme	Client Storage (#Block)	Bandwidth Cost (#Block)	Computational Overhead
Hierarchical Structure			
GO-ORAM [1]	$O(1)$	$O(\log^3 N)$	$O(\log^3(N))$
SSS-ORAM [3]	$O(N)$	$O(\log N)$	$O(\log^2(N))$
Recurisve SSS-ORAM [3]	$O(\sqrt{N})$	$O(\log^2 N)$	$O(\log^3(N))$
Tree Structure			
Tree-ORAM [2]	$O(N)$	$O(\log^2 N)$	$O(\log^3(N))$
Recursive Tree-ORAM [2]	$O(1)$	$O(\log^3 N)$	$O(\log^4(N))$
Path ORAM [4]	$O(N)$	$O(\log N)$	$O(\log^2(N))$
Recursive Path ORAM [4]	$O(\log N) \cdot \omega(1)$	$O(\log^2(N))$	$O(\log^3 N)$
Matrix Structure			
M-ORAM	$O(N)$	$O(1)$	$O(1)$
Recursive M-ORAM	$O(\log N)$	$O(\log N)$	$O(\log N)$

construction, RM-ORAM is proposed to be used when the limited-sized storage device is used as an ORAM client. The bandwidth overhead and computational overhead of RM-ORAM is somewhat growth compared with M-ORAM. Nonetheless, the space requirement on the client is dramatically reduced to $O(\log N)$. In addition to the lower the overhead of bandwidth and storage usage, Matrix based ORAM computation performance is also enhanced compared to other ORAM constructions. The other ORAM constructions require a polylogarithmic order to operate the oblivious transfer, while M-ORAM and RM-ORAM require only constant and logarithmic computational overhead to do the same purpose.

7.3 Limitation of the research

This research is limited under two conditions: ORAM structural design restriction and limitation of the experiment. In matrix based ORAM structural design, the delay of operations was not taken into account. Indeed, the delay of operations such as internal delay caused by operating system or network propagation delay can lead to the attack vector called the side-channel attack. Nonetheless, the delay of operations is an uncontrollable factor and beyond the scope of our research. Regarding the limitations of the experiment, the experiments were conducted on personal computers via the local network rather than a cloud server. Since the performance metrics of interest are the number of data blocks transferred and stored on the client, it provides no different results when the experiments are tested on either the personal computer or high-performance cloud server.

7.4 Recommendations for the future research

There are the rooms to improve the current version of matrix based ORAM. By the fact that, the current matrix based ORAM was designed on the premise that the data is in the format of raw text string. But if in the case of calculable data, there may be better processing methods to obtain the same result of current solution, since matrix computation is efficient in parallel computing (e.g. matrix multiplication). In addition, Private Information Retrieval (PIR) or Homomorphic Encryption (HE) may be used to reduce the number of blocks transferred during access operation; therefore it is possible that the performance of bandwidth of M-ORAM can be improved over the current version. However, using PIR or HE is the causes of raising the computational complexity of the system; hence how to apply those solutions in M-ORAM is still a question to keep looking.

7.5 Conclusion

M-ORAM and RM-ORAM use a matrix data structure to provide the same security as other ORAM schemes while it can achieve lower bandwidth cost and client computational overhead. In the normal construction (M-ORAM), it gives $O(1)$ for both bandwidth and client computational overhead with $O(N)$ client storage. The recursive construction, on the other hand, it gives $O(\log N)$ for all of bandwidth overhead, client computational overhead, and client storage. Due to the different construction, M-ORAM and RM-ORAM use slightly different operations to retrieve a data of interest from ORAM server. By the proof conducted by this research, up

until now, M-ORAM has been known as lightweight ORAM algorithm which has the closest performance to normal client-server construction.

Appendix A

Path ORAM Implementation

Oblivious RAM (ORAM) was first introduced by Goodrich and Ostrovsky [1] with the aim of hiding access patterns between CPU and RAM. In the recent year, ORAM is seen as beneficial for cloud computing, specifically for hiding access patterns from client to server. A major limitation of current cloud-based storage applications is that servers that store client's data may learn clients' confidential information. The client encrypting the data before uploading to the server is not sufficient to protect the confidentiality: it has been shown that the sequence of server storage locations read/written by the client (i.e. access pattern) can reveal valuable information to the server [11, 12]. ORAM therefore hides the access patterns by having the client re-encrypt and sort the data.

There are various ORAM algorithms existing today [3, 13–20]. The major differences are the operation at the client and server. The data structure used by the server to store data can be either hierarchical (or pyramid) or binary-tree structure. For the client, the protocol for communications between client and server, the encryption/decryption and sorting algorithms may vary. The different ORAM algorithms lead to several key design trade-offs: the amount of storage used on the client, the communication overhead between client and server, and the complexity of algorithms on the client. A major drawback of ORAM in practice is the communication overhead between client and server. That is, to access a single logical block of data with ORAM, a client needs to access many physical blocks to hide the access patterns. Given the client's N blocks of data stored in the server, Path ORAM has a communication overhead of $O(\log N)$. To achieve this the client must encrypt/re-encrypt data before uploading and also may have significant local storage requirements.

Path ORAM is one of well known binary tree ORAM constructions which was first introduced by Stefamov et al. [4]. Nonetheless, the source code of Path ORAM is not publicly provided, only its algorithm and general construction are. Therefore, we have to construct source code of Path ORAM from scratch included the details of data format (e.g. position map block and data block). We therefore present a design of a specific instance of Path ORAM, and then using a Python implementation, present experimental analysis results. Our contributions can be summarized as follows:

- We propose a data format for both the server and client storage to implement the Path-ORAM efficiently.
- We analyze the probability of uploading a block back to the same location it was downloaded from (i.e. duplication) for different height H binary-tree heights.

- We make a qualitative comparison of AES for encryption in both XTS and GCM modes to show the different effects using different encryption mode.
- We also provide an analysis of average number of blocks stored in client with different height H binary-tree heights.

We will show due to duplication, AES-GCM is an appropriate re-encryption method for Path ORAM. Also, the client storage can be slightly higher than the upper bound presented in [4] in our practical implementation. The rest of the appendix is organized as follows: In Section A.1 we provide an overviewt of Path ORAM, and then present our detailed design in Section A.2. In Section A.3, the analysis results based on our experiment will be illustrated. Then the conclusion will be given in Section A.4.

A.1 Overview of Path ORAM

Path ORAM [4] is based on Shi's binary-tree structure for ORAM [2]. However it reduces the large communication and calculation overhead incurred by the shuffling and eviction algorithms of Shi by using a different way to store data on the server and random re-encryption for every upload, while still satisfying the required security properties of ORAM. Section A.1.1 describes the storage approach of Path ORAM and Section A.1.2 summaries the protocol operation for reading/writing data from/to the server.

A.1.1 Server and Client Storage

In Path ORAM the server stores data in a binary-tree of height H, with nodes called *buckets* (see Figure A.1). Each bucket stores Z blocks of data; each block is B bytes; the total number of blocks that the client can store on the server is N; the total number of blocks that the server must store is $Z \cdot N$. Buckets are always contains Z blocks: if the *real* data does not fill a bucket, then it is padded with *dummy* blocks.

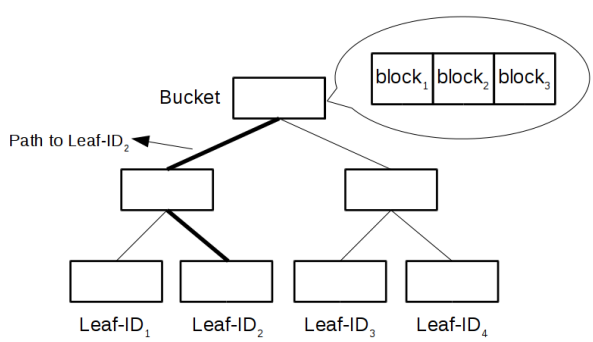


Figure A.1: Binary-tree storage structure

All blocks of data have a leaf-ID assigned to them (the assignment is presented in the next section). The leaf-ID is associated with the bucket position: a block with leaf-ID l must be stored in a bucket on a path from root to bucket with leaf-ID l. A block of data is in fact the concatenation of the data content and the leaf-ID.

The client maintains two data structures. A position map maintains a mapping of a data block to a leaf-ID. For example in Figure A.2(a), block a has leaf-ID l_2. Also, the client has

a *stash* that temporarily stores blocks that have been downloaded from the server but currently cannot be written back to the server. The use of the client data structures is described in Section A.1.2. Note however that Path ORAM does not specify the data format for the client storage; we discuss that in Section A.2.

A.1.2 Read/Write Operation

The two basic operations of reading data from the server and writing data to the server are treated as a single *access* operation in Path ORAM. The idea is that whenever the client wishes to read or write data, it must actually read multiple blocks and then write multiple blocks. This, combined with sorting and encrypting the blocks before they are written, provides the security properties of ORAM: hiding the access patterns.

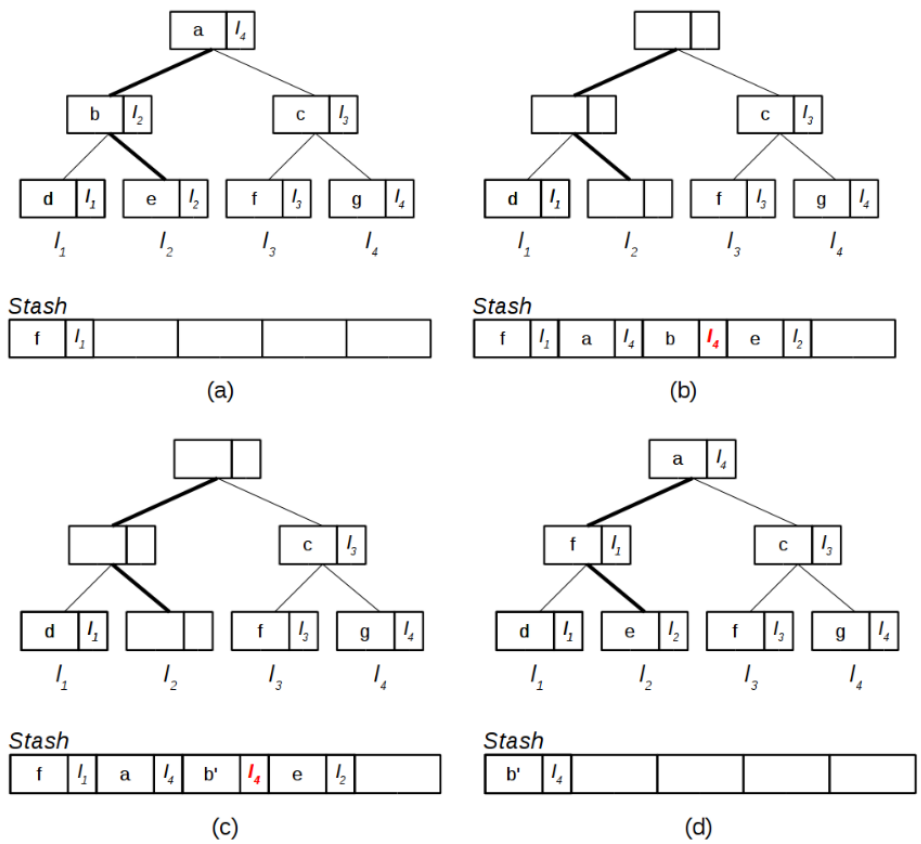


Figure A.2: Path ORAM download/upload process

To explain the access operation we will use a simplified Path ORAM with one block per bucket ($Z = 1$), as illustrated in Figure A.2. Consider the access operation applies on a block b, with current leaf-ID l_2. The client must read the entire path containing the block, e.g. the path from root to leaf l_2 in Figure A.2(a). All blocks are stored in the client stash. For the block of interest, a new leaf-ID is randomly selected (e.g. l_4). If the access is a write operation, then block b is updated inside the stash. The last step is to write blocks back to the same path on the server. The blocks are selected from the stash to first fill the leaf bucket, and then its parent bucket, up until the root. Recall the requirement that the block must be on a path that leads from root to leaf with ID matching the block leaf-ID. If there are no blocks in the stash that

meet this requirement, then a bucket is filled with dummy blocks. The result is that blocks may be leftover in the stash after the path is written.

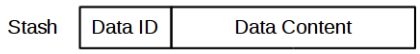
A.2 Our Detailed Design

Several design details are not presented in [4]: the data formats for storage, the encryption/decryption method, local operations, initial operation and main operation at the client. Each of these may impact on performance of Path ORAM. We therefore present design solutions in this section, then in the subsequent sections provide an experimental analysis of these solutions.

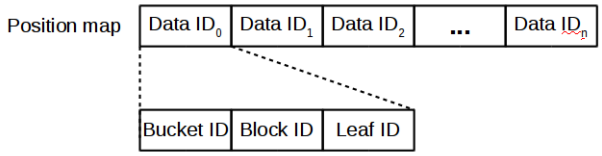
A.2.1 Data Types and Formats

In our design we specify the format for storing three important pieces of information: block, stash and position map (summarised in Figure A.3). These are not specified in the original Path ORAM. Each of these make use of general data types, which we introduce first.

(a) Data format in Binary-tree storage



(b) Data format in stash buffer



(c) Data format in position map buffer

Figure A.3: Data formats

Data type In our design we categorize information into: data content, data-ID, block-ID, and bucket-ID. Data content is the data the client wants to store on the server. When stored on the server it is encrypted; on the client the data content is stored as plaintext to reduce unnecessary calculation overheads. Data-ID is added to our design in the purpose of identifying information within the stash. The data-ID, block-ID and bucket-ID are unique positive integers give to each data content, block, and bucket, respectively. An exception is that a block-ID can *NULL* or a negative integer when the information is a new information or it is being stored in the stash.

Format of block in binary-tree storage In binary-tree storage on the server, each block contains the client's information. This information consists of two paths which are encrypted data content and initial vector (IV). The IV is added with the purpose of making encrypted content look different for every upload operation via a key that is stored as a secret within client. The encryption algorithms are discussed in Section A.2.2.

Format of stash buffer After an entire path is downloaded to client's stash, each data content is decrypted, the IV removed and data-ID added as an index. The data-ID allows the client to identify data content.

Format of position map The position map is a consecutive array which contains the details of block location that the data is associated with. The data-ID is used as the array index, e.g. data content with data-ID 4 is stored in array element with index 4. Each element stores bucket-ID, block-ID and leaf-ID for the data content. The bucket-ID is used to identify the physical address of the bucket that stores the data content on the server. The block-ID is an offset address of the bucket-ID, giving the exact location of data content on the server. The leaf-ID is used to construct the path before downloading the information.

A.2.2 Encryption/Decryption Method

As each bucket is read during the access operation it is decrypted; it is then encrypted before it is written back to the server. Path ORAM does not however specify the encryption algorithm to use. The selection of algorithm may impact on both client storage requirements (e.g. necessary to store different keys, IVs, parameters) and computational complexity.

ORAM has the characteristic that is very beneficial for encryption method through fixed block size, under the condition that we use only single secret key for all elements and every uploaded information must have a fresh look in the server's perspective. It is so precise that stream cipher and some of block cipher modes can possibly reveal the pattern of encrypted content to adversary. Moreover if we do not authenticate the encrypted message, adversaries can arbitrary tweak information which will lead to other types of attacking. At the very beginning, we are interested in XTS-AES mode. Due to the ORAM's bucket structure mimics from storage's sector which each sector has it owns unique address. XTS-AES modes somehow, may give an answer for none extra value adding for security purpose. Unfortunately, in the Section A.3.1, we show that the amount of duplicated location after uploading is high and XTS-AES uses sector-ID as the first AES key and sector's block-ID as the second key. Hence, if we use XTS-AES as the encryption method, the encrypted content will still be the same as before it was downloaded if it is uploaded to the same block location. Moreover, XTS-AES itself does not have authentication tags and hence any ciphertext will be ultimately decrypted as some plaintext even it was modified by adversary.

AES-GCM is the most appropriate solution for our implementation as it meets both requirements of confidentiality and integrity. However, the downside is the cost of storing additional values such as IV (Initialization Vector) and ICV (Integrity Check Value) for each block information.

A.2.3 Local Read/Write Operation

The original Path ORAM does not consider how to read content that is already in the stash. We propose using the client stash as a cache for local read/write operations. That is, when the client wants to read data which is currently in the stash, rather than accessing the server the client directly reads from the stash. This reduces commuication overhead.

During read/write operation, client will first look for requested information in stash buffer. If the information is found in the stash, read/write operation will be done locally to reduce the number of messages transmitted. By looking for the block-ID of requested content in position

map, a negative value means the information is being stored within stash buffer instead of server storage. The client will search through stash buffer by using data-ID and then read or update that information.

The effectiveness of using the stash as a cache depends on the stash size. Section A.3 gives experimental results on the stash size for different bucket sizes and tree heights.

A.2.4 Initialization of Path ORAM

In the original paper, every block in server's storage will be filled by random encrypted dummy at initial state. However, the details of the initialisation operation are not presented. Hence in this section, we present the algorithm for initialisation Path ORAM, which is summarised in Algorithm 7.

At initial state, client needs to specify the block amount that will be used to store the information. The client will ask the server for free space of $N \cdot Z \cdot B$ bytes. Once client receives all free spaces addresses, client will construct the virtual binary-tree structure by those addresses. Bucket-ID will be associated with those addresses and position map will be created by random assignment to each data-ID without duplication. At this point, block-ID is not yet assigned in the position map where it will be assigned later during read/update operation. Hence every block-ID in position map will be filled by *NULL*. In addition, client needs to fill up every block of storage by *dummy information* so that the information which will be uploaded in future process becomes indistinguishable from server's perspective.

Algorithm 7 Initial Operation

1: **Input:** FreeSpace(Bytes)
2: **Output:** Position Map, data-ID linked list database
3: $ListOfAddress \leftarrow$ RequestFreeSpace($FreeSpace$)
4: $LeafIDs \leftarrow$ CreateTree($ListOfAddress$)
5: $BucketIDs \leftarrow$ CreateTree($ListOfAddress$)
6: $BlockIDs \leftarrow$ CreateTree($ListOfAddress$)
7: $PositionMap \leftarrow$ CreatePosMap($BucketIDs, LeafIDs$)
8: $LinkedList \leftarrow$ CreateLinkedList()
9: **return** $PositionMap, LinkedList$

A.2.5 Main Operation of Path ORAM

From now on we will call read/write operation as access request operation. In this operation, client will first retrieves input information's data-ID by checking in the database lookup table. If it does not appear in its database, new data-ID will be generated and added to database. After data-ID has been retrieved, client is looking for more details in position map to get bucket-ID, block-ID and leaf-ID. Access request will be locally operated in such block-ID is negative value otherwise *Path* of downloading will be created by bucket-ID and leaf-ID then the download operation is invoked. The operation will be operated as usual until the decrypted information will be stored into the stash. Difference from the traditional Path ORAM, data-ID will be concatenated with its content as a tag to distinct information from each other before storing. This tag is removed when this information is going to be uploaded to the server. During upload operation, information will be re-encrypted with AES-GCM method by fresh regenerated IV.

The encrypted information will be attached by IV together with ICV and then uploaded to server. The operation is summarised in Algorithms 8 and 9.

Algorithm 8 Split Content

```
Input: content
Output: dataIDs
if content > blocksize then
    blockcontent[n] ← SplitContent(content)
else
    blockcontent[0] ← content   #size=BlockkSize-IV-ICV
end if
if NewContent then
    for i ∈ n do
        dataIDs, LinkedList ← AssID(blockcontent[n])
    end for
else
    dataIDs ← LinkedList
end if
return dataIDs
```

A.3 Experimental Analysis

In this section, we will analyze our implementation in various dimensions in purpose of gathering information for our future work which includes the block location duplication before downloading and after uploading, number of download/upload per request, and relation of size of stash usage with increasing binary-tree's level and number block per bucket.

We implement Path ORAM in Python, running experiments on an Intel core i5 CPU with 2 GB of memory, using 20,000 string datasets from UCI Machine Learning Repository [51] for data. The default values used are : 512 byte block size, 4 blocks per bucket and 10 level binary-tree.

A.3.1 Duplication location

Stefanov illustrates the Path ORAM using random re-encryption with the purpose of differentiating the downloading and uploading content even when they are the same information. By this notion, there are two possible solution to achieve that purpose. First, client has to change secret key for every access request which means that every block must have its individual secret key. The second method is using random initial vector for encrypting. By using this method, every block can share the same secret key and client does not need to change its secret key for every access request. As we provided pros and cons of our two prospect methods, AES-GCM and XTS-AES in Section A.2.2, we need to investigate the percentage of duplicating when upload the information to the same location that it was recently downloaded. For our experiment, we run read/write operation for 7 days with more than 10TB send/receive information for each different height of binary-tree.

In Figure A.4, it shows the percentage of duplication that associates with height of binary-tree storage which was calculated by the ratio between information that is written back to same

Algorithm 9 Main Operation

```
1:  Input: operation, dataID, data*
2:  Output: data
3:  blockID, bucketID, leafID ← posmap[dataID]
4:  if blockID < 0 then
5:      if operation = update then
6:          data ← stash.Update(data*, dataID)
7:      else
8:          data ← stash.Read(dataID)
9:      end if
10: else
11:     path ← CreatePath(bucketID, leafID)
12:     for bucket ∈ path do
13:         content, IV, ICV ← Download(bucket)
14:         stash ← stash ∪ {Decrypt(content, IV, ICV)}
15:     end for
16:     leafID* ←$ {leafIDs} − leafID
17:     if operation = update then
18:         data ← stash.Update(data*, dataID)
19:     else
20:         data ← stash.Read(dataID)
21:     end if
22:     posmap[dataID] ← leafID*
23:     for bucket ∈ path do
24:         if There is data in stash can fit in bucket then
25:             content ← Encrypt(stash.Read(dataID))
26:             stash ← stash − {stash.Read(dataID)}
27:             Upload(bucket, content + IV + ICV)
28:         else
29:             Upload(bucket, dummy)
30:         end if
31:     end for
32: end if
33: return data
```

block location and the total amount of uploaded information. Although the amount of duplication is contrary to the height that is growing, the probability of duplication is still high at 0.8 with 13 levels binary-tree storage. Thus in conclusion, the AES-GCM is appropriate to be used compare with XTS-AES. However The relation of the suitable size of extra information or suitable methods than AES encryption is still an open question and it is on researching in our future work.

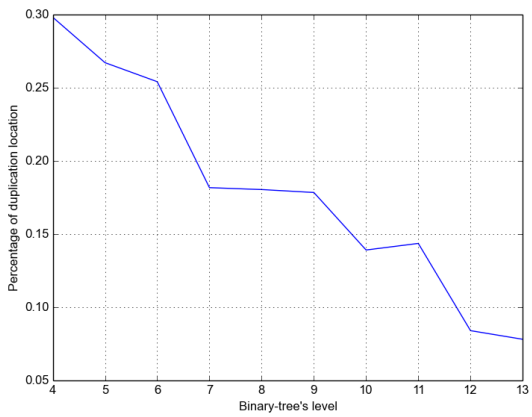


Figure A.4: Probability of duplicate location of Download/Upload same content

A.3.2 Stash Usage

The most important discussion topic of Path ORAM is the amount of stash memory usage. The amount of stash usage is based on probability of unable to upload the content within the stash buffer. By theoretical proof from Path ORAM paper, the stash usage is bounded by $O(\log N)\omega(1)$ blocks by secure parameter λ. However in real implementation, we cannot specify λ due to the Path ORAM's random characteristic. For our experiment, we run through read/write operation for 7 days with more than 10TB send/receive information each different height and bucket size of binary-tree. The graph in Figure A.5 and Figure A.6 were generated by the number of remain information within stash after upload operation. It shows the stash usage is slightly increasing over amount of request operation and be bounded by some value according to theoretical proof of Path ORAM. However, the upper bound of stash size based on our experiment is slightly larger than it was described in original paper. Ultimately, our implementation shows two factors that can reduce the bound of stash size which are number of block per bucket and height of binary-tree storage.

A.3.3 Number of Download/Upload per request

Slightly difference from illustrating by Stafanov, Path ORAM in our construction has offline process such local read and local update whenever requested information is found in stash buffer. By the experimental results from Section A.3.2, the number of remaining contents in stash are slightly increasing overtime. It means that the probability of content to be found in stash also proportionally increase. The balance point between client's storage consumption and communication complexity is still an open question which was not discussed in the original

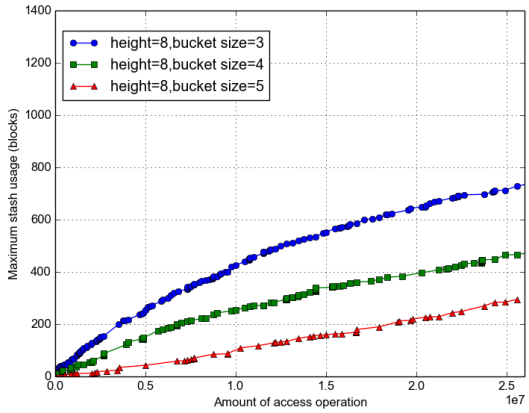


Figure A.5: Amount of stash usage(block) with different bucket size(block) at Binary-tree's hight = 8

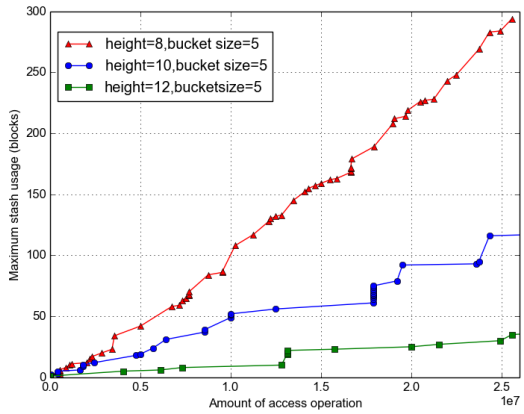


Figure A.6: Amount of stash usage(block) with different Binary-tree's hight at bucket size(block) = 5

paper. Now we are investigating on this area and it will be applied in our future scheme. By increasing either height or bucket size, the probability that the remaining elements in stash can be uploaded to server increases which means the size of stash for storing the remaining elements is reduced.

A.4 Conclusion

Path ORAM is one of the most efficient binary-tree based ORAM algorithms for hiding client access patterns to a server storage. However, there are several open areas with regards to the detailed design of Path ORAM. In this paper we addressed three issues, presenting detailed design and experimental analysis results using a Python implementation. First, we recommend using AES-GCM for encryption of blocks, as AES-XTS can lead to too many duplicated blocks on the server. The drawback of AES-GCM is an increase in storage space and communication overhead for the security parameters for each block. Second, we give results comparing communication complexity with the height of the binary tree. In the future we will design a scheme to reduce the path length to reduce communication complexity. Finally, we present a mechanism for using the client stash as a cache. This allows the client to access blocks directly (without contacting the server), reducing the communication overhead. This is only effective with a large stash size, hence the optimal tradeoff between client storage and communication overhead is an area for future research.

Bibliography

[1] Goldreich, O. & Ostrovsky, R. (1996), Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3), 431–473.

[2] Shi, E., Chan, T. H., Stefanov, E., & Li, M. (2011), Oblivious RAM with $O(log^3 N)$ worst-case cost. In D. H. Lee (Ed.), *International Conference on the Theory and Application of Cryptology and Information Security: Proceeding of an international conference held in Seol, South Korea, 4-8 December 2011* (pp. 197–214). Berlin, Heidelberg: Springer Berlin Heidelberg.

[3] Stefanov, E., Shi, E., & Song, D. X. (2012), Towards practical Oblivious RAM. *19th Annual Network and Distributed System Security Symposium: Proceeding of an international conference held in San Diego, California, USA, 5-8 February 2012*. Geneva, Switzerland : The Internet Society.

[4] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C. W., Ren, L., Yu, X., & Devadas, S. (2013), Path ORAM: An Extremely Simple Oblivious RAM Protocol. *ACM SIGSAC Conference on Computer and Communications Security: Proceeding of an international conference held in Berlin, Germany, 4-8 November 2013* (pp. 299–310). New York, NY, USA: ACM.

[5] Abdalla, M., Bellare, M., Catalano, D., Kiltz, E., Kohno, T., Lange, T., Malone-Lee, J., Neven, G., Paillier, P., & Shi, H. (2008), Searchable Encryption Revisited:Consistency Properties, Relation to Anonymous IBE, and Extensions. *Journal of Cryptology*, 21(3), 350–391.

[6] Kamara, S., Papamanthou, C., & Roeder, T. (2012), Dynamic Searchable Symmetric Encryption. *ACM Conference on Computer and Communications Security: Proceeding of an international conference held in Raleigh, North Carolina, USA, 16 - 18 October 2012* (pp. 965–976). New York, NY, USA: ACM.

[7] Liang, K., Huang, X., Guo, F., & Liu, J. K. (2016), Privacy-Preserving and Regular Language Search Over Encrypted Cloud Data. *IEEE Transactions on Information Forensics and Security*, 11(10), 2365–2376.

[8] Zuo, C., Macindoe, J., Yang, S., Steinfeld, R., & Liu, J. K. (2016), Trusted Boolean Search on Cloud Using Searchable Symmetric Encryption. *IEEE Trustcom/BigDataSE/ISPA: Proceeding of an international conference held in Tianjin, China, 23-26 Aug 2016* (pp. 113–120). Washington, DC, USA: IEEE Computer Society.

[9] Gordon, S., Miyaji, A., Su, C., Sumongkayothin. K. (2016), A Matrix Based ORAM: Design, Implementation and Experimental Analysis. *IEICE Transactions on Information and Systems*, 99-D(8), 2044–2055

[10] Sumongkayothin, K., Gordon, S., Miyaji, A., Su, C., Wipusitwarakun, K. (2016), Recursive M-ORAM: A Matrix ORAM for Clients with Constrained Storage Space. In L. Batten, G. Li (Eds.), *6th International Conference on Applications and Techniques in Information Security: Proceeding of an international conference held in airns, QLD, Australia, 26-28 October 2016* (pp. 130141). Singapore: Springer Singapore.

[11] Islam, M. S., Kuzu, M., & Kantarcioglu, M. (2012), Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. *19th Annual Network and Distributed System Security Symposium: Proceeding of an international conference held in San Diego, California, USA, 5-8 February 2012*. Geneva, Switzerland : The Internet Society.

[12] Liu, C., Zhu, L., Wang, M., & Tan, Y.-A. (2014), Search Pattern Leakage in Searchable Encryption: Attacks and New Construction. *Inf. Sci.*, 265, 176–188.

[13] Boneh, D., Mazieres, D., & Popa, R. A. (2011, Mar), Remote Oblivious Storage: Making Oblivious RAM Practical. (Report No.MIT-CSAIL-TR-2011-018). Retrieved September 11, 2016, from `http://hdl.handle.net/1721.1/62006`.

[14] Dautrich, J., Stefanov, E., & Shi, E. (2014), Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns. *23rd USENIX Security Symposium: Proceeding of an international conference held in San Diego, California, USA, 20-22 August 2014* (pp. 749–764). USENIX Association.

[15] Gentry, C., Goldman, K. A., Halevi, S., Jutla, C., Raykova, M., & Wichs, D. (2013), Optimizing ORAM and Using It Efficiently for Secure Computation. In E. De Cristofaro & M. Wright (Eds.), *Privacy Enhancing Technologies: 13th International Symposium: Proceeding of an international conference held in Bloomington, IN, USA, 10-12 July 2013* (pp. 1–18). Berlin, Heidelberg: Springer Berlin Heidelberg.

[16] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., & Tamassia, R. (2012), Privacy-preserving Group Data Access via Stateless Oblivious RAM Simulation. *23rd Annual ACM–SIAM Symposium on Discrete Algorithms: Proceeding of an international conference held in Kyoto, Japan, 17-19 January 2012* (pp. 157–167). Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[17] Karvelas, N. P., Peter, A., Katzenbeisser, S., & Biedermann, S. (2014), Efficient Privacy-Preserving Big Data Processing through Proxy-Assisted ORAM. *IACR Cryptology ePrint Archive.*, 2014, 72.

[18] Pinkas, B. & Reinman, T. (2010), Oblivious RAM Revisited. In T. Rabin (Ed.), *30th Annual Cryptology Conference: Proceeding of an international conference held in Santa Barbara, CA, 15-19 August 2010* (pp. 502–519). Berlin, Heidelberg: Springer Berlin Heidelberg.

[19] Williams, P., Sion, R., & Tomescu, A. (2012), PrivateFS: A Parallel Oblivious File System. *ACM Conference on Computer and Communications Security: Proceeding of an international conference held in Raleigh, North Carolina, USA, 16-18 October 2012* (pp. 977–988). New York, NY, USA: ACM.

[20] Stefanov, E. & Shi, E. (2013), ObliviStore: High Performance Oblivious Cloud Storage. *IEEE Symposium on Security and Privacy: Proceeding of an international conference held in San Francisco, California, USA, 19-22 May 2013* (pp. 253–267). Washington, DC, USA: IEEE Computer Society.

[21] Stefanov, E. & Shi, E. (2013 Dec), Multi-cloud Oblivious Storage. *ACM SIGSAC Conference on Computer and Communications Security: Proceeding of an international conference held in Berlin, Germany, 4-8 November 2013* (pp. 247–258). New York, NY, USA: ACM.

[22] Carbunar, B. & Sion, R. (2011), Write-Once Read-Many Oblivious RAM. *IEEE Transactions on Information Forensics and Security*, 6(4), 1394–1403.

[23] Ren, L., Yu, X., Fletcher, C. W., Dijk, M. V., & Devadas, S. (2013), Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. *40th Annual International Symposium on Computer Architecture: Proceeding of an international conference held in Tel-Aviv, Israel, 23-27 June 2013* (pp. 571–582). New York, NY, USA: ACM.

[24] Mayberry, T., oliver Blass, E., & Chan, A. H. (2014), Efficient Private File Retrieval by Combining ORAM and PIR. *21st Annual Network and Distributed System Security Symposium: Proceeding of an international conference held in San Diego, California, USA, 23-26 February 2014*. Geneva, Switzerland : The Internet Society.

[25] Ma, Q., Zhang, J., Peng, Y., Zhang, W., & Qiao, D. (2016), SE-ORAM: A Storage-Efficient Oblivious RAM for Privacy-Preserving Access to Cloud Storage. *3rd IEEE International Conference on Cyber Security and Cloud Computing: Proceeding of an international conference held in Haidian, Beijing, China, 25-27 June 2016* (pp 20–25). Washington, DC, USA: IEEE Computer Society.

[26] Zhang, J., Zhang, W., & Qiao, D. (2016, February), MU-ORAM: Dealing with Stealthy Privacy Attacks in Multi-User Data Outsourcing Services. (Report No.2016/073). Retrieved September 11, 2016, from `http://eprint.iacr.org/2016/073`.

[27] Jinsheng, Z., Wensheng, Z., & Qiao, D. (2014, April), A Multi-user Oblivious RAM for Outsourced Data. (Report No. 262). Retrieved September 11, 2016, from `http://lib.dr.iastate.edu/cs_techreports/262/`.

[28] Ren, L., Fletcher, C. W., Yu, X., Kwon, A., van Dijk, M., & Devadas, S. (2014, June), Unified Oblivious-RAM: Improving Recursive ORAM with. Locality and Pseudorandomness. (Report No. 2014/205). Retrieved September 11, 2016, from `https://eprint.iacr.org/2014/205`.

[29] Fletcher, C. W., Naveed, M., Ren, L., Shi, E., & Stefanov, E. (2015, November), Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM. (Report No. 2015/1065). Retrieved September 11, 2016, from `https://eprint.iacr.org/2015/1065`.

[30] Fletcher, C. W., Ren, L., Kwon, A., van Dijk, M., & Devadas, S. (2015), Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious

RAM. in *20th International Conference on Architectural Support for Programming Languages and Operating Systems: Proceeding of an international conference held in Istanbul, Turkey, 14-18 March 2015* (pp. 103–116). New York, NY, USA: ACM.

[31] Garg, S., Mohassel, P., & Papamanthou, C. (2016, February), TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption. (Report No. 2015/1010). Retrieved September 11, 2016, from `https://eprint.iacr.org/2015/1010`.

[32] Moataz, T., Mayberry, T., Blass, E., & Chan, A. H. (2015), Resizable Tree-Based Oblivious RAM. In R. Böhme & T. Okamoto (Eds.), *19th International Conference, Financial Cryptography and Data Security: Proceeding of an international conference held in San Juan, Puerto Rico, 26-30 January 2015* (pp. 147–167). Berlin, Heidelberg: Springer Berlin Heidelberg.

[33] Moataz, T., Mayberry, T., & Blass, E.-O. (2015), Constant Communication ORAM with Small Blocksize. *22nd ACM SIGSAC Conference on Computer and Communications Security: Proceeding of an international conference held in Denver, Colorado, USA, 12-16 October 2015* (pp. 862–873). New York, NY, USA: ACM.

[34] Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., & Devadas, S. (2015), Constants count: Practical improvements to Oblivious RAM. *24th USENIX Security Symposium: Proceeding of an international conference held in Washington, D.C., USA, 12-14 August 2015* (pp. 415–430). USENIX Association.

[35] Wolfe, N., Zou, E., Ren, L., & Yu, X., *Optimizing path ORAM for cloud storage applications*. Retrieved September 11, 2016, from `https://arxiv.org/abs/1501.01721`.

[36] Yu, X., Haider, S. K., Ren, L., Fletcher, C., Kwon, A., van Dijk, M., & Devadas, S. (2015), PrORAM: Dynamic Prefetcher for Oblivious RAM. *the 42nd Annual International Symposium on Computer Architecture: Proceeding of an international conference held in Portland, Oregon, USA, 13-17 June 2015* (pp. 616–628). New York, NY, USA: ACM.

[37] Wagh, S., Cuff, P., & Mittal, P., *Root ORAM: A Tunable Differentially Private Oblivious RAM*. Retrieved September 11, 2016, from `https://arxiv.org/abs/1601.03378`.

[38] Zhang, J., Ma, Q., Zhang, W., & Qiao, D. (2015, March), KT-ORAM: A Bandwidth-efficient ORAM Built on K-ary Tree of PIR Nodes. (Report No. 2014/624). Retrieved September 11, 2016, from `https://eprint.iacr.org/2014/624`.

[39] Jia, Y., Moataz, T., Tople, S., & Saxena, P. (2016), Oblivp2p: An oblivious peer-to-peer content sharing system. *25th USENIX Security Symposium: Proceeding of an international conference held in Austin, TX, USA, 10-12 August 2016* (pp. 945–962). USENIX Association.

[40] Devadas, S., van Dijk, M., Fletcher, C. W., Ren, L., Shi, E., & Wichs, D. (2016), Onion ORAM: A constant bandwidth blowup Oblivious RAM. In E. Kushilevitz & T. Malkin (Eds.), *13th Theory of Cryptography Conference: Proceeding of an international conference held in Tel Aviv, Israel, 10-13 January 2016* (pp. 145–174). Berlin, Heidelberg: Springer Berlin Heidelberg.

[41] Dachman-Soled, D., Liu, C., Papamanthou, C., Shi, E., & Vishkin, U. (2015), Oblivious Network RAM and Leveraging Parallelism to Achieve Obliviousness. In T. Iwata & J. H. Cheon (Eds.), *21st International Conference on the Theory and Application of Cryptology and Information Security: Proceeding of an international conference held in Auckland, New Zealand, 29 November-3 December 2015*, (pp. 337–359). Berlin, Heidelberg: Springer Berlin Heidelberg.

[42] Biryukov, A., Khovratovich, D., & Nikoli, I. (2009), Distinguisher and Related-Key Attack on the Full AES-256. *29th Annual International Cryptology Conference on Advances in Cryptology: Proceeding of an international conference held in Santa Barbara, CA, 16-20 August 2009* (pp. 231–249). Berlin, Heidelberg: Springer Berlin Heidelberg.

[43] Biryukov, A., Dunkelman, O., Keller, N., Khovratovich, D., & Shamir, A. (2010), Key recovery attacks of practical complexity on AES-256 variants with up to 10 rounds. In H. Gilbert (Ed.), *29th Annual International Conference on the Theory and Applications of Cryptographic Techniques: Proceeding of an international conference held in Riviera, French, 30 May 30-3 June 2010* (pp. 299–319). Berlin, Heidelberg: Springer Berlin Heidelberg.

[44] Agren, M. (2012), *On Some Symmetric Lightweight Cryptographic Designs*, Retrieved September 11, 2016, from `https://lup.lub.lu.se/search/publication/69a88ee7-0525-49dd-8327-fc6fe87933e3`.

[45] Gilbert, H.,& Peyrin, T. (2010), Super-Sbox Cryptanalysis: Improved Attacks for AES-Like Permutations. In S. Hong (Ed.), *Fast Software Encryption: 17th International Workshop: Proceeding of an international conference held in Seoul, South Korea, 7-10 February 2010* (pp. 365–383). Berlin, Heidelberg: Springer Berlin Heidelberg.

[46] Bogdanov. A., Khovratovich, D., & Rechberger, C. (2011), Biclique Cryptanalysis of the Full AES. In D. H.Lee & X. Wang (Eds.), *17th International Conference on The Theory and Application of Cryptology and Information Security: Proceeding of an international conference held in Seoul, South Korea, 4-8 December 2011.* (pp. 344–371). Berlin, Heidelberg: Springer Berlin Heidelberg.

[47] Tao, B. & Wu, H. (2015), Improving the Biclique Cryptanalysis of AES. In E. Foo & D. Stebila (Eds.), *Information Security and Privacy: 20th Australasian Conference: Proceeding of an international conference held in Brisbane, QLD, Australia, 29 June-1 July 2015* (pp. 39–56). Berlin, Heidelberg: Springer Berlin Heidelberg.

[48] Dworkin, M. (2001, December), NIST Special Publication 800-38A, 2001 Edition: Recommendation for Block Cipher Modes of Operation, Methods and Techniques. (Report No. 800-38A). Retrieved September 11, 2016, from `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a-add.pdf`.

[49] Rukhin, A., Soto, J., Nechvatal, J., Barker, E., Leigh, S., Levenson, M., Banks, D., Heckert, A., Dray, J., Vo, S., Rukhin, A., Soto, J., Smid, M., Leigh, S., Vangel, M., Heckert, A., Dray, J., & Iii, L. E. B. (2010, September), A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. (Report No. 800-22 Rev 1a). Retrieved September 11, 2016, from `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf`.

[50] Lucero, S. (2016, Mar), IoT platforms: enabling the Internet of Things. (Report No. IHS2016). Retrieved September 11, 2016, from https://cdn.ihs.com/www/pdf/enabling-IOT.pdf.

[51] Lichman, M., *UCI Machine Learning Repository*. Retrieved December 17, 2014, from http://archive.ics.uci.edu/ml/index.php

Publications

INTERNATIONAL JOURNAL

[1] Steven Gordon, Atsuko Miyaji, Chunhua Su, Karin Sumongkayothin," A Matrix Based ORAM: Design, Implementation and Experimental Analysis" IEICE Transactions, vol. 99–D(8), pp. 2044–2055, 2016. *(authors in alphabetical order)*

[2] Steven Gordon, Xinyi Huang, Atsuko Miyaji, Chunhua Su, Karin Sumongkayothin, Komwut Wipusitwarakun, " Recursive Matrix Oblivious RAM: An ORAM construction for constrained storage devices" IEEE Transactions on Information Forensics & Security, (To appear). *(authors in alphabetical order)*

INTERNATIONAL CONFERENCE

[1] Karin Sumongkayothin, Steven Gordon, Atsuko Miyaji, Chunhua Su, Komwut Wipusitwarakun, "Recursive M-ORAM: A Matrix ORAM for Clients with Constrained Storage Space", Applications and Techniques in Information Security - 6th International Conference (ATIS 2016), pp. 130–141, Cairns, QLD, Australia.

[2] Steven Gordon, Atsuko Miyaji, Chunhua Su, Karin Sumongkayothin, " Security and experimental performance analysis of a matrix ORAM", IEEE International Conference on Communications, IEEE-ICC 2016, pp. 1–6, Kuala Lumpur, Malaysia. *(authors in alphabetical order)*

[3] Steven Gordon, Atsuko Miyaji, Chunhua Su, Karin Sumongkayothin, "Analysis of Path ORAM toward Practical Utilization", In Proceeding of 18th International Conference on Network-Based Information Systems (NBis 2015), pp. 646–651, Taipei, Taiwan. *(authors in alphabetical order)*

[4] Steven Gordon, Atsuko Miyaji, Chunhua Su, and Karin Sumongkayothin, "M-ORAM: A Matrix ORAM with $\log N$ Bandwidth Cost", In Proceeding of Information Security Applications - 16th International Workshop (WISA 2015), pp. 3–15, Jeju Island, Korea. *(authors in alphabetical order)*