Recursive Matrix Oblivious RAM: An ORAM construction for constrained storage devices

Steven Gordon, Xinyi Huang, Atsuko Miyaji, Chunhua Su, Karin Sumongkayothin, and Komwut Wipusitwarakun

***Abstract*—Oblivious Random Access Machine (ORAM) constructions can be used to hide a client's access pattern from a trusted but curious storage server. The privacy provided comes at the cost of increasing communication overhead, storage overhead, and computation overhead of the system. Recursive Matrix-based ORAM (RM-ORAM) is a new ORAM construction which is designed for constrained storage space devices. 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 ORAM schemes, e.g. recursive Path ORAM. In this paper, we present the construction and its theoretical analysis. In addition, we present how to select the appropriate number of data blocks which are being downloaded per level of recursion and the appropriate size of reserved space on the client. We provide theoretical security and performance analysis, as well as experimental results to illustrate how RM-ORAM satisfies security requirements and provides improved performance compared to other ORAM schemes.**

***Index Terms*—ORAM, secure communication, secure access pattern**

I. Introduction

DESPITE the many benefits and widespread use of cloud computing, security still remains a major drawback. Encryption by the client is to ensure data confidentiality at the cloud service provider, however additional techniques are needed to allow the full benefits of secure cloud computing. For example, searchable encryption can be used to search an encrypted database without revealing the contents. Although many searchable encryption algorithms have been presented (e.g. [1]–[4]), it is still possible for an adversary to observe and analyse the access patterns between client and server [5], [6]. Private information retrieval [7]–[9] can hide access patterns, but is designed for the client reading (not writing) data on the server. One method that allows reading and writing, and keeps both the data and access patterns confidential is Oblivious Random Access Machine (ORAM). In this paper, we present a new ORAM construction called Recursive Matrix ORAM (RM-ORAM) which uses M-ORAM [10] as its basis. RM-ORAM can reduce client storage space requirement without significantly increasing computational overheads.

K. Sumongkayothin and K. Wipusitwarakun are with the School of Information, Communication and Computer Technologies, Sirindhorn International Institute of Technology, Thammasat University.

K. Sumongkayothin and A. Miyaji are with the School of Information Science, Japan Advance Institute of Science and Technology.

K. Sumongkayothin is the corresponding author.
Email: s1420209@jaist.ac.jp

A. Miyaji is with the Graduate School of Engineering, Osaka University.

C. Su is with the Division of Computer Science, University of Aizu

S. Gordon is with the School of Engineering and Technology, Central Queensland University.

X. Huang is with the School of Mathematics and Computer Science, Fujian Normal University.

Manuscript received February 18, 2017; revised June 4, 2017; accepted June 30, 2017.

A. Oblivious RAM

Oblivious RAM [11] was first introduced as a technique for hiding communication patterns between CPU and memory. More recently [10], [12], [12]–[27] ORAM has been applied for networked computing, in particular for a client to access a server without any disclosures of access pattern or data being accessed. The key components of an ORAM system are: storage on the server, storage on the client, and an algorithm for the client to access the server. We will explain these components via an example of the client uploading data to the server.

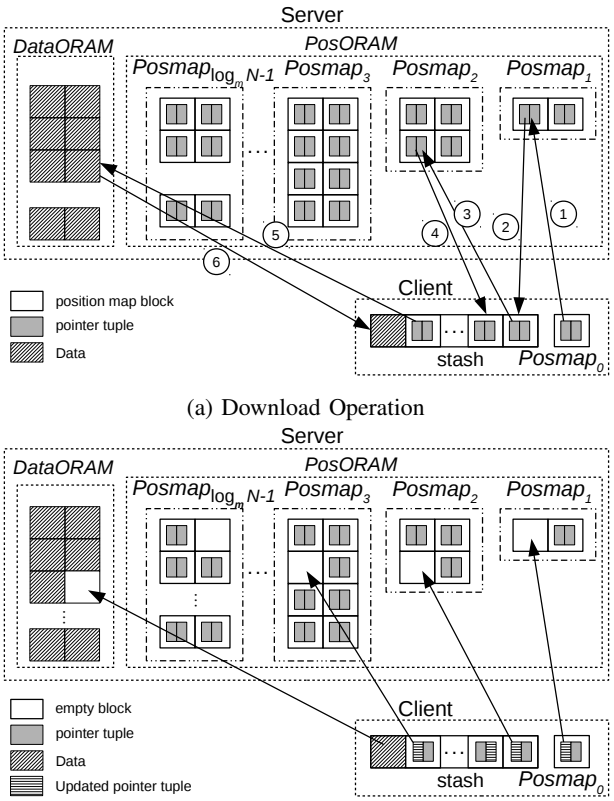
Suppose the client has N data blocks kept on the server. The server's storage, which we refer to as *DataORAM*, is logically organised as N fixed sized *blocks*; and the ORAM client accesses the blocks using address b_j, where $j \in \{1, 2, \ldots, N\}$. At the client side, we distinguish between a user and the ORAM client software acting on behalf of the user. The user has *data of interest* d_i with unique identifier ID_i to upload to the server, where $i \in \{1, 2, \ldots, N\}$. Both download or upload operations by the user triggers the ORAM client to perform an *access operation* which consists of a series of downloads and uploads from/to the server. For example, the ORAM client uploads data with ID_i to block b_j on the server. The ORAM client stores a mapping of ID_i to block b_j in a *position map*. When the ORAM client downloads data, it is temporarily stored locally in a *stash*.

The detailed design of storage organisation, use of position map and stash, and the access algorithm distinguishes the many different ORAM systems [10]–[12], [12]–[25]. They all aim to achieve the security requirements, while minimising performance overheads in three areas: bandwidth incurred by the extra downloads/uploads; storage necessary for client and server; and computations performed on client and server. A broad classification of ORAM systems is those that require significant client/server computations [11], [12], [12]–[21], [23]–[25], and those that don't [10], [22]. We focus on the latter, which are appropriate for devices with constrained resources. However such systems require moderate to large client storage space for the position map and stash. A general approach to further reduce the client storage space requirements is to store the position map on the server (in ORAM) and have a second,

smaller position map on the client. This leads to *recursive ORAM*.

B. Recursive Oblivious RAM

Recursive ORAM first appeared in [20]. It reduces the position map size on a client to $O(1)$ while incurring logarithmic growth of bandwidth overhead according to the size of ORAM. Contrary to a non-recursive ORAM construction which has to retain the position map on the client, a recursive ORAM construction stores most of the position map on the server. Therefore, server storage may be viewed in two parts as *DataORAM* that stores N data blocks, and *PosORAM* that stores N' position map blocks. PosORAM can be divided into i groups of position map blocks, each group referred to as $Posmap_i$. The number of groups is one less than the number of levels of recursion. The recursive operation is explained with the aid of Figure 1, where the circled numbers in Figure 1 indicate the order of steps in downloading. Let every position map block contains m tuples, called *pointer tuple*. Each pointer tuple contains a pointer, and the pointers of the current level are an injective mapping to position map blocks of the next level of recursion. To download data of interest from DataORAM, a client looks for the associated pointer in $Posmap_i$ which points to a position map block located somewhere in $Posmap_{i+1}$. Then the client downloads that block and stores it in the storage named *stash*. The operation will start from $Posmap_0$ and repeatedly goes on until the data of interest is downloaded from DataORAM. After finishing the download, the client uploads blocks back to the server. The procedure to upload the information depends on which ORAM algorithm is applied.



(a) Download Operation

(b) Upload operation

Fig. 1: General recursive operation for ORAM

C. Related Work

Proposed ORAM algorithms can be classified by their storage structure—hierarchical [12]–[20], tree [21], [22], [24], [25], and matrix [10]—which have different trade-offs of bandwidth, client's space requirement and computational complexity. Some of these algorithms also define a recursive construction to reduce the client storage space requirement.

The first ORAM algorithm [11] was introduced in a format of hierarchical data structure to hide the access pattern between CPU and memory. A top level of ORAM has smaller capacity compared to its lower level, and the topmost level is most frequently accessed by the CPU. The recently accessed data will be put on the topmost level and it is continuously pushed down to deeper levels if it is not accessed by the CPU in a specific period of time. To push the information down, the CPU has to perform obfuscation sorting and merging of the data which are stored in consecutive levels, and then the merged data will be written to the next lower level. During an access operation, to hide the data of interest from the adversary, the CPU performs dummy scanning of every subsequent level even though the data of interest has been found. The CPU keeps only the data of interest, therefore the first ORAM algorithm requires $O(1)$ for stash buffer but incurs $O(\log^3 N)$ bandwidth cost. ORAM revisited by Pinkas *et. al* [12] and introduced as an algorithm for hiding the access pattern between client and server. However, some researchers [28] have pointed out a security flaw in Pinkas's construction. Partition ORAM or SSS-ORAM is an altered hierarchical structure ORAM that was introduced by Stefanov *et. al* [20]. SSS-ORAM separates single ORAM of N blocks to $\sqrt{N}$ partitions which can be concurrently accessed via its corresponding stash. Although SSS-ORAM incurs $O(\log^2 N)$ bandwidth overhead, which is less than [11], it requires $O(\sqrt{N})$ blocks in the client.

Binary tree ORAM was introduced by Shi *et. al* [21]. Similar to hierarchical ORAM, the client uploads the recently accessed data to the top level (root) of binary tree ORAM and it will be continually pushed down close to lowest level (leaf) on the binary tree if it has not been accessed for a period of time. Unlike hierarchical construction, once the data from the top level is pushed down to the next lower level, it is randomly pushed to one of the two nodes (either left or right child) and dummy data will be written to another node. To define the location of data of interest, Shi's ORAM keeps the information called *leaf ID* within a client. Leaf ID is unique for each leaf of binary tree data structure, and a data item is associated with one of them. The leaf ID tells a client the path to root on which the associated data is located. Shi's ORAM incurs $O(N)$ client storage and $O(\log^2 N)$ bandwidth cost. Path ORAM [22] use the leaf ID to identify the path of interest in the same way as [21]. However, it reduces the computational complexity by downloading every block within the path to the client instead of scanning on the server. The leaf ID of the data of interest is randomly changed after download, and then

the client will try to upload the data within its stash back to the server. By using simpler operations and incurring $O(\log N)$ bandwidth cost, Path ORAM has a significant advantage over hierarchical ORAMs and Shi's scheme. However, Path ORAM still requires $O(N)$ blocks on a client to contain a position map. Therefore the recursion technique introduced in [20] is applied to Path ORAM which can reduce the client storage space requirement from $O(N)$ to $O(\log N) \cdot \omega(1)$ while the bandwidth cost is increased from $O(\log N)$ to $O(\log^2 N)$.

Hierarchical and binary tree ORAM have common disadvantages. System bandwidth overhead depends upon the height of the ORAM construction which is increasing, according to the growing size of ORAM. Unlike any existing schemes, the ORAM of [10] builds upon the matrix data structure which is called M-ORAM. When the size of ORAM is growing, the width of the construction is increasing instead of height. In the same way as the hierarchical and binary tree ORAM, the bandwidth overhead of M-ORAM is dependent upon the height of the construction. Therefore, the bandwidth is constant even though the size of ORAM is varied. The disadvantage of M-ORAM is it requires N blocks of position map on the client. [29] outlines an idea for a recursive construction of M-ORAM, and gives the theoretical analysis over computation complexity, security, and bandwidth cost. However, the previous design has some limitations and lacks experimental results. Hence, in this paper, we present an improved design, new performance analysis, and results from the experiments.

D. Our Contribution

This research proposes a new ORAM construction which dramatically reduces the storage space requirement of a client but does not significantly increase the overhead of bandwidth and computation complexity of the system. Our major contributions are:

- **Design of RM-ORAM construction.** We present the detailed design of recursive M-ORAM (RM-ORAM) and show how it can operate in a constrained storage space environment.
- **Optimise bandwidth cost, calculation overhead, and client's storage usage.** We introduce the ORAM construction which can achieve $O(\log N)$ costs for all three important factors.
- **Efficient storage usage.** In our construction, every block in reserved space can be real information rather than dummy as used by other ORAM constructions.
- **Theoretical security analysis.** We give the theoretical analysis of RM-ORAM security which includes the randomness of access pattern, probability of secret key re-use, and the minimum number of blocks downloaded per level of recursion, demonstrating RM-ORAM can achieve the same security level as Path ORAM.
- **Theoretical performance analysis.** We give a proof for RM-ORAM compared with Path ORAM under the same conditions to define the appropriate number of blocks that should be downloaded from DataORAM per access request. In addition, we give the theoretical performance models of RM-ORAM in three aspects: the storage usage efficiency, bandwidth cost and computational overhead.
- **Experimental analysis.** To provide further insights into RM-ORAM performance and security which are not given by the theoretical analysis, we have implemented RM-ORAM and provide experimental analysis of the random movement of information and size of storage used for the stash.

TABLE I: Notation

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.
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 stash_data after finish download operation.
c	Size of counter.

E. Paper Organization

The remainder of this paper is structured as follows. Details of construction and operation of Recursive M-ORAM are introduced in Section II. Theoretical analysis of RM-ORAM security is presented in Section III, while the theoretical analysis of performance is given in Section IV. Experimental results from our implementation compared with Path ORAM are given in Section V, and we conclude in Section VI. Notation used in this paper is summarised in Table I.

II. Recursive M-ORAM Construction and Operation

RM-ORAM inherits components of matrix-based ORAM, of which the key concepts are presented in Section II-A. Then a top-level view of RM-ORAM design is given in Section II-B, followed by details of the data structures in Section II-C and access operations in Section II-D.

A. Basic Principle of M-ORAM

A matrix based ORAM (M-ORAM) was introduced by Gordon *et al.* [10], where the server storage of N blocks was

TABLE II: Differentiation of M-ORAM and RM-ORAM

Structure/Operation	M-ORAM	RM-ORAM
ORAM	DataORAM	DataORAM, PosORAM
stash	h stashes with fixed size	h stashes for data block with fixed size, h stashes for position map block with $O(\log N)$ size
position map	N	$O(\log N)$
Download	Download h data blocks	Download h blocks per download of $\lceil \log_m(N) \rceil$ performed downloads.
Upload	Upload random h data blocks	Upload random $h * \lceil \log_m(N) \rceil$ blocks

organised as a x_a-by-$y_{N/a}$ matrix as shown by the *DataORAM* structure in Figure 2. The motivation of the structure was to optimize bandwidth usage during information retrieval. The client locally stores the address (i.e. row and column) of all N blocks in a position map. To access (either download or upload) a block of interest, x_a blocks are accessed (i.e. a block from each row), where a small number of columns are from the previous access, and the remainder are randomly chosen columns. The x_a blocks are downloaded and stored in a stash on the client. Then x_a random blocks are chosen from the stash for upload to DataORAM (they may or may not include the block of interest). The key strength of M-ORAM is the bandwidth overhead during access operation is kept small as the size of the DataORAM grows. However M-ORAM requires large client storage of N blocks, unsuitable for clients with limited storage space.

B. Overview of RM-ORAM

The aim of RM-ORAM is to reduce the client storage space compared to M-ORAM. This is achieved by recursively storing the position map blocks on the server. To support this recursive operation, changes to M-ORAM are needed as summarised in Table II. Specifically, RM-ORAM requires additional storage on the server for blocks related to the position map (i.e. PosORAM). The structure of the storage is described in Section II-C. Also, to ensure the security properties are met while keeping the bandwidth low, a different algorithm for download/upload is necessary. This is described in Section II-D.

As a light-weight ORAM, M-ORAM and Path ORAM have similar design concepts. Therefore, the performance aspects of M-ORAM were compared with Path ORAM in [10]. Since both RM-ORAM and recursive Path ORAM are inherited from their original constructions, recursive Path ORAM is used as a reference to measure the performance of RM-ORAM. The performance comparison of both constructions is illustrated in Section IV and Section V.

C. RM-ORAM Storage Structure

RM-ORAM storage (Figure 2) consists of two types of ORAM on a server: *DataORAM* and *PosORAM*; and four types of storage on a client: *StashData*, *StashPath*, *Posmap*$_0$ and *history list*. DataORAM and PosORAM may differ by block size and number of blocks, the values of which are

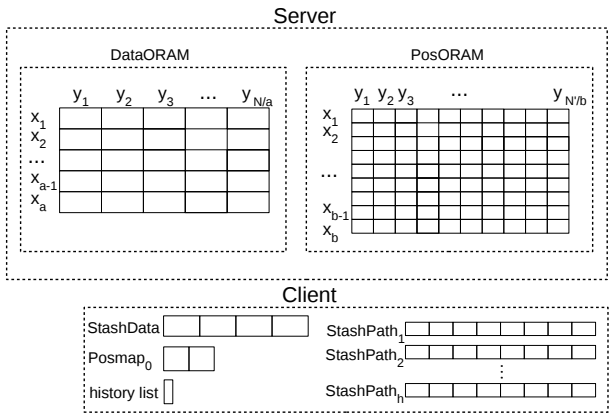


Fig. 2: RM-ORAM storage

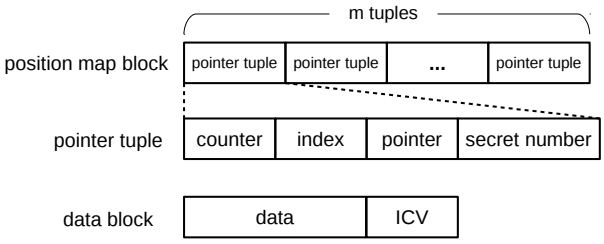


Fig. 3: Details of data block and position map block

known by the server. Similar to other recursive ORAM constructions, DataORAM and PosORAM of RM-ORAM are used to store data blocks and position map blocks, respectively. While PosORAM contains Posmap_1 through to $\text{Posmap}_{\log_m N-1}$ (see Figure 1), the position map blocks are actually randomly distributed within PosORAM. Each position map block consists of m tuples called *pointer tuple*, and each tuple consists of four parts: counter, index, pointer, and common share seed which are used for generating an encryption key. A data block consists of only data content (see Figure 3). StashPath and StashData are used to store downloaded position map blocks and data blocks, respectively. Posmap_0 contains pointer tuples which have a pointer pointing to the blocks in PosORAM. The size of PosORAM is $\sum_{i=1}^{r-1} m^i$ and must be greater than Posmap_0. The size of StashData is greater than the number of data blocks downloaded during an access operation. History list is a constant size buffer which contains the list of addresses having been accessed by a client. Block addresses in the history list are sorted from oldest access to newest access.

D. RM-ORAM Access Operation

Algorithms 1, 2, and 3 illustrate the details of operation when the client wants to access a data block of interest. Once a client wants to access (either download or upload) a data block, it starts by checking the data blocks remaining in StashData as shown in line 1-7 of Algorithm 1. If data of interest is in StashData, the client performs a local access. If it is not, the client randomly chooses $h-1$ other data and the data of interest to be downloaded. Among the $h-1$ other data, there are

TABLE III: Function and Parameter Description for Algorithm 1, 2 and 3

Function / Parameter	Description
$\{x\}$	Set of parameter x.
ID	ID of data of interest.
ID_{all}	ID of data in DataORAM.
ID_{prev}	ID of data in 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 $\{ID_{prev}\}$.
d_{ID}	Content of data of interest that 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^*	Content to update.
d_{oth}	Content of other 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.
ReadStashData(ID_i)	Read data which belongs to ID_i from stash. Return: d_i.
UpdateStashData($ID, data^*$)	Update content of data ID in stash.
RndSelect($\{ID\}, x$)	Random select x *IDs* from set of *IDs*. Return: $\{ID\}$ size x.
ReadPos(*pointer*, ID)	Read data of ID from PosORAM which is pointed by *pointer*. Return: $\{d\}$ of $\{ID\}$, $\{b\}$ to block of next level of recursion.
ReplaceHist($\{ID\}$)	Randomly replace l elements in $\{ID_{hist}\}$ with $\{ID\}$.
AppendStashPath(x)	Append parameter x to StashPath.
AppendStashData(x)	Append parameter x to StashData.
ReadData($\{b\}$)	Read data from DataORAM from $\{b\}$. Return: $\{data\}$.
RndRetrievStashData(x)	Retrieve x data blocks from StashData. Return: $\{ID_{prev}\}$,$\{d_{rnd}\}$.
RndAssignNewPointer(x)	Random assign new pointer and address for parameter x. Return: $\{b_{rnd}\}$.
WriteData()	Write data blocks to DataORAM.
WritePos()	Write position map blocks to PosORAM.
UploadOperation()	Upload selected blocks to DataORAM and PostORAM.
ClearStashPath()	Clear content from StashPath.

o data blocks selected from data blocks of previous uploads, and l data blocks (not the data blocks of previous uploads) are selected from the history list. Once h blocks have been chosen, the client is looking for all corresponding pointers of the data blocks being downloaded in Posmap_0 (line 9). Then the list of the oldest blocks in the history list is replaced by the list of new selected blocks (line 10). The pointers retrieved from Posmap_0 tell the location of position map blocks to be downloaded within Posmap_1, then the client downloads those blocks to StashPath. The downloaded position map blocks are distributed to h StashPaths (line 14-16). The same procedures are repeated until blocks within Posmap_2, which correspond to the pointers retrieved from Posmap_1, are downloaded. After downloading, 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).

Algorithm 1 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,$\{ID_{prev}\}$) #See. Algorithm 3
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: UploadOperation()
26: **return** d_{ID}

Algorithm 2 RM-ORAM's upload operation (UploadOperation())

1: $\{ID_{prev}\} \leftarrow$ RndRetrievStashData(h)
2: RndAssignNewPointer($\{$StashPath$\}$, Posmap_0)
3: WriteData()
4: WritePos()
5: ClearStashPath()

In the next phase of the access operation, the same number of downloaded blocks is uploaded to the server (Algorithm 2). The 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 previous download. The addresses that have been accessed in DataORAM and PosORAM from the download operation are randomly assigned to selected data blocks and position map blocks, respectively. The pointer in selected position map blocks is set to point to the new corresponding address (line 2). Finally, all of the selected data blocks and position map blocks are uploaded to the DataORAM and PosORAM, respectively, according to their new location (line 3-4).

Algorithm 3 SelectBlocks()

1: **Input:** ID, $\{ID_{prev}\}$
2: **if** $ID \in \{ID_{prev}\}$ **then**
3: $\{ID_{old}\} \leftarrow$ 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: **return** $\{ID_{old}\}, \{ID_{hist}\}, \{ID_{new}\}$

III. Security Analysis

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

A. ORAM Security 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 the different access operations.
4) Randomness of sequence of an access request is secure under polynomial-time adversary.

To achieve first and second requirements, the content of data before download and after upload must look different. One of the solutions is re-encrypting the content with a different secret key. These two requirements are covered by the security proof in Section III-B 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 III-E, we start by introducing the appropriate number of blocks selected from the previous access operation in Section III-C. RM-ORAM requires downloading some blocks which were accessed in the previous access operation. If this wasn't the case, if the client consecutively accesses a different set of blocks, then the set of blocks accessed would be distinct. Then the history list is introduced in Section III-D to cover the issue of distinguishable access pattern in specific circumstances.

B. Random Re-encryption

The 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-updated content will reveal the identity of information to the server. Therefore, $k_{(i,t)}$ must not equal $k_{(i,t+1)}$.

To generate the secret key, the *index*, *counter* and shared secret *seed* 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 3. Index tells what data block is relevant to its pointer which is always updated during an 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 to by the pointer. Seed is a parameter which the client keeps as a secret, and used to make the key generator to be more flexible. As the PRF is deterministic, and the same values of index and counter may be used as input, the seed should change periodically. To determine the period of changing the seed, we measure it as the number of upload operations.

Let c be the size of the counter, counter $\in \{i \mid 0 < i < c, i \in \mathbb{Z}\}$, s be the size of StashData after download operation, and h be the number of data blocks downloaded during the access operation. Assume the client repeatedly requests one specific data block without changing the content. The period of changing the secret seed is given in Theorem 1.

Theorem 1. *Let T denotes the time period (number of upload operations) to change a secret seed. Suppose a counter has size c bits and StashData has size equal to s blocks. $P(X = t)$ is the probability that the client spends t trials until at least one data block is assigned to the same block as it was during the download operation with probability of success p. Therefore, the expected period of time to change a seed is:*

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

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

Proof. Since *PRF* is deterministic function, the same input value must be used to generate the same output. The c bits of counter must take c uploads before the same value of the counter will return. Hence, the data block must be successfully chosen c times before the same secret key is reused. Therefore, we use the geometric distribution to accurately calculate T. From the expected value of the geometric distribution, the equation of expected value of x failure attempts to get 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 encryption function is deterministic, the same encrypted data will be generated when the same data content is encrypted by the same secret key. Furthermore, the same secret key will be generated by PRF if every input has the same value. As an index at the last level of recursion relates to one of data blocks in DataORAM, there is an position map block containing the same value of index if at least one of downloaded data block is selected to be uploaded to its previous location. For now let the counter does not change over the time. The number of solutions which at least one data block is assigned to the same block where it used to be 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 where it used to be during the download operation. Therefore, the probability of *S1DW* is:

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

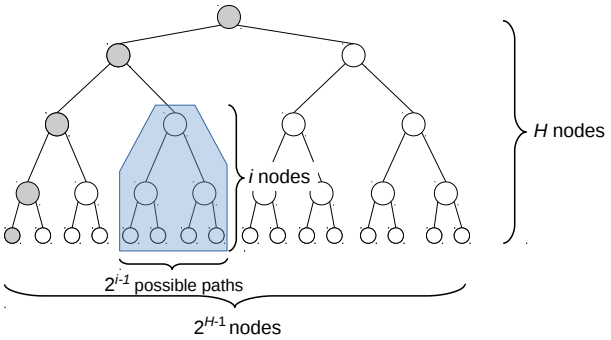


Fig. 4: Possible path of $H - i$ overlapped nodes

Hence, the expected value of number of uploads is:

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

where $p = \Pr(S1DW)$

However, the counter has changed over time when a data block is uploaded to the server. Suppose the counter is size c bits, a system needs to have exactly c successes to retrieve data of interest from stash when the same secret key is reused. Therefore, 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 secret seed has to be changed is as in Theorem 1. □

C. Choosing the Previous Accessed Block

According to the requirement in Section III-A, RM-ORAM is required to download some blocks which were accessed in the previous access operation. To accurately determine the proper number of the blocks having to be chosen, 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 2. *The average number of overlapped blocks across two consecutive access operations of Path ORAM is 2 when the height of binary trees tends to infinity.*

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

Proof. Figure 4, shows the binary-tree structure of Path ORAM with height H. Since the binary-tree has 2^{H-1} leaf nodes, there are 2^{H-1} possible paths in total from a leaf node to a root node. Suppose the path chosen during the previous access operation is represented as the gray circles of Figure 4, and the path of next access operation is the path that is associated with a node of interest. As the node of interest is assigned to a path uniformly at random (according to [22]), 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:

$$\begin{aligned} \bar{o}_{path\_oram} &= \frac{1}{2^{H-1}} H + \sum_{i=1}^{H-1} \frac{2^{i-1}}{2^{H-1}} (H-i) \\ &= \frac{\sum_{i=0}^{H} 2^i}{2^{H-1}} \end{aligned} \tag{3}$$

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

Therefore, to achieve the same or better security level of Path ORAM, the number of o having to be accessed by RM-ORAM must be equal to or greater than 2.

D. Distinguishable Series of Accesses in Specific Circumstance

In some specific circumstances the different access sequences over RM-ORAM can be distinguishable. The vulnerability occurs when the blocks having been accessed are requested to be accessed again in a short period of time. For example, let two sequences of accesses be A' and A''. Suppose A' and A'' consist of sequence 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 finishing the first access operation, and it is not 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 reading d_3 is $(h-o)/N$. On the other hand, as d_1 is requested to be read by 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 predict with high probability which block is interested. However, the client is the person who controls the pattern of access requests. Therefore, client can design how often the same information will be repeatably requested. It leads to two solutions that can be used to solve this problem.

1) **Do not request same block address twice more often than one time of $N/(h-o)$ accesses:** To make the block selection looks like a uniform distribution, a client has to limit the number of accesses to the block that has been accessed. Therefore, the number of accesses to those blocks should be approximately 1 time of $N/(h-o)$ accesses according to the example given in the beginning of this section.
2) **Random selecting from the history list:** Another solution to keep A' indistinguishable from A'' is to create the *history list*. History list contains list of blocks having been accessed in the past, and a client randomly chooses l blocks from this list as additional blocks for each access operation. To create history list, a client starts with the *'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 predefined 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 the two proposed solutions, the first solution is limited by a 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 impractical. On the other hand, the second solution is more flexible and practical. The trade-off is client has to perform warm-up accesses to construct the history list. However, the warm-up method will be performed once on new communication which is negligible for long-term communication. Regarding the appropriate number of l chosen from history list; it is defined in Theorem 3.

Theorem 3. *The appropriate number of blocks chosen from the history list, l when $N \gg h$ is:*

$$1 \leq l \leq \frac{(h-o)}{2} \tag{4}$$

Proof. Suppose h blocks are chosen to be accessed in each access request. o blocks come from the 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 are $h-l-o$ blocks, which 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 block addresses in the history list is:

$$\text{number of block list} \leq \left(\frac{N}{(h-o)} \times (h-l-o)\right) \tag{5}$$

According to the security issue that has been discussed at the beginning of this section, 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 block addresses contained in the history list is taken into consideration. At the maximum number of block addresses in the history list, the average frequency of accessing the history list is equal to the new list. In other words, the probability of data of interest coming from the chosen block from the history list is equal to the probability of data of interest coming from the chosen block from the new list. Suppose that $N \gg h > o$, from Equation 5 we can derive the possible solution of l as:

$$\begin{aligned} \frac{N}{2} &= \left(\frac{N}{(h-o)} \times (h-l-o)\right) \\ l &= \frac{(h-o)}{2} \end{aligned} \tag{6}$$

The probability of data of interest coming from the history list depends on a ratio of l and the number of block addresses contained in the history list. Therefore, the number of l holds by Equation 4. □

E. Randomness Over Access Pattern

Recall from Section II-C, the block size of PosORAM and DataORAM might differ and are known by a server. Therefore, once an access operation has been executed, a server knows which type of ORAM is being accessed by client. The randomness of access pattern is therefore considered into two parts which are the randomness of access pattern over DataORAM and PosORAM.

Suppose A is an access sequence length i where each access (either upload or download) reveals a block address, $adr_j[data_j]$ to a server. Let $adr_j[data_j]$ be the block address revealed during j^{th} access where $j \in \{1 \leq x \leq i \mid x \in \mathbb{Z}\}$. Hence, A can be illustrated as Equation 7.

$$A = (adr_i[data_i], adr_{i-1}[data_{i-1}], \ldots, adr_1[data_1]) \tag{7}$$

Since the block size of PosORAM and DataORAM might differ and are known by the server, the set of block addresses having been accessed and seen by the server during j^{th} access can be defined as:

$$adr_j[data_j] = AddrData_j \uplus AddrPos_j \tag{8}$$

where $AddrData_j$ and $AddrPos_j$ are the set of addresses that have been accessed during j^{th} access on DataORAM and PosORAM, respectively. Therefore, the series of access requests A from Equation 7 can be separated to A_{data} and A_{pos} which are an access sequence length i on DataORAM and PosORAM, respectively. We can define the A_{data} and A_{pos} as:

$$A_{data} = (AddrData_i, AddrData_{i-1}, \ldots, AddrData_1) \tag{9}$$

$$A_{pos} = (AddrPos_i, AddrPos_{i-1}, \ldots, AddrPos_1) \tag{10}$$

There are three possible patterns of two series of accesses A and A' generated by a client. Those patterns can be described by Lemma 1, Lemma 2, and Lemma 3. To show the random access pattern of RM-ORAM is secure, the access patterns are proven case-by-case.

Lemma 1. *Different access sequences 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 III-B, the content of a series of accessed blocks is indistinguishable from a random string by randomised encryption whether or not the content has been changed. In addition, since the content of data blocks of each access is randomly changed according to RM-ORAM access operation, a server cannot identify the relationship between the content and the data block containing the content. Although A_{data} and A'_{data} access the same set of data blocks, the uploaded content of both access sequences are not necessary the same. Therefore A_{data} and A'_{data} are indistinguishable by the polynomial time adversary. □

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

Proof. Suppose two access sequences length i: A and A' access on different data blocks. According to Section II-D, o blocks are selected from the previous operation to be accessed in current operation to make sure that two consecutive operations do not access on a completely different set of data blocks. Furthermore, the 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 a client tries to access two different groups of data. Therefore, Lemma 2 holds by the same proof of Lemma 1. □

Lemma 3. *Different access sequences on the some (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 discussed in Section III-D, 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 the history list, a client can control the probability of block that will be accessed in the ORAM. High probability means the block is possible to be accessed more often, although it does not contain a data of interest. Suppose the probability of data of interest is among l chosen data blocks from history list size L is l/L. It means the data of interest is possibly selected every L/l accesses. By maintaining this probability to be greater than or equal to l/L, two different sequences of accesses will look similar from the server's perspective although they have different frequencies of accessing data of interest. Hence, Lemma 3 holds when the content of data blocks is indistinguishable by randomised encryption. □

Lemma 4. *Different access sequences to 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 to 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. In the same manner of proving in Lemma 1, 2 and 3, 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 a client. B_A and $B_{A'}$ is the set of blocks accessed by A and A', respectively, which $B_A, B_{A'} \in B_N$ 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 is addressed by Lemma 1 and 4; case $B_A \cap B_{A'} = \emptyset$ is addressed by Lemma 2 and 4; and for the last case $(B_A \cap B_{A'} \neq \emptyset) \wedge (B_A \neq B_{A'})$, it is addressed by Lemma 3 and 4. Therefore, any sequences of access pattern are secure under the advisory whose computational power is bounded by polynomial time. □

TABLE IV: Performance Comparison of Different ORAM Schemes

Scheme	Client Storage (#Block)	Bandwidth Cost (#Block)	Computational Overhead
Hierarchical Structure			
GO-ORAM [11]	$O(1)$	$O(\log^3 N)$	$O(\log^3(N))$
SSS-ORAM [20]	$O(N)$	$O(\log N)$	$O(\log^2(N))$
Recurisve SSS-ORAM [20]	$O(\sqrt{N})$	$O(\log^2 N)$	$O(\log^3(N))$
Tree Structure			
Tree-ORAM [21]	$O(N)$	$O(\log^2 N)$	$O(\log^3(N))$
Recursive Tree-ORAM [21]	$O(1)$	$O(\log^3 N)$	$O(\log^4(N))$
Path ORAM [22]	$O(N)$	$O(\log N)$	$O(\log^2(N))$
Recursive Path ORAM [22]	$O(\log N)\cdot\omega(1)$	$O(\log^2 N)$	$O(\log^3(N))$
Matrix Structure			
M-ORAM	$O(N)$	$O(1)$	$O(1)$
Recursive M-ORAM	$O(\log N)$	$O(\log N)$	$O(\log N)$

IV. Performance Analysis

In this section, the performance of RM-ORAM is analysed from 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. The performance metric of RM-ORAM compared against other ORAM schemes yields the results are per Table IV. Analysis of efficient storage usage is discussed in Section IV-A. Analysis of bandwidth cost of RM-ORAM system is provided in Section IV-B while analysis of computational overhead is given in Section IV-C.

A. Storage Usage Efficiency

Efficiency of storage usage in ORAM is one of the important performance aspect. Decreasing of storage space reservation means increasing the number of operations to ensure the designed data structure can achieve all of ORAM's security requirements. Generally, the storage of ORAM system can be categorised into storage on a server and storage on a client as it has been described in Section II-C. Therefore, the efficiency analysis in this section is focused on the use of storage on the client and server of RM-ORAM.

1) Client Storage Usage Efficiency: In M-ORAM, a client stores N logical addresses of the data block in position map and downloaded data's content in stash. Stash has a constant size while the size of position map varies according to the size of ORAM. Therefore, there are $O(N)$ blocks reserved on the 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. History list has a constant size, which its size depends on the security thresholds as defined in Section III-D. 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

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 on a client.

2) Server Storage Usage Efficiency: Some ORAM constructions store both data and dummy information on a server (e.g. [11] and [21]). The dummy information is necessary to preserve the security requirement of upload and download, 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 a server which contains the 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 data structure formats. For example, RM-ORAM uses matrix data structure for Posmap while Path ORAM uses binary tree data structure. 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 not, there are some unused pointer tuples. For Path ORAM, as the number of nodes in a 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 the binary tree Posmap of i^{th} level of recursion and $h_{i+1} > h_i$. Since RM-ORAM and recursive Path ORAM have same r when they have the same number of m and N (Page 9 of [22] and Lemma 7), we show in Theorem 5 that the number of non-used spaces in PosORAM of RM-ORAM is equal to or less than the number of non-used spaces in PosORAM of Path ORAM when both constructions share the same r.

Lemma 5. *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:*

$$m \cdot x_i \leq x_{i+1} < m \cdot (x_i + 1) \tag{11}$$

Lemma 6. *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:*

$$m \cdot (2^{y_i} - 1) \leq 2^{y_{i+1}} - 1 < m \cdot (2^{y_i+1} - 1) \tag{12}$$

Theorem 5. *Let the number of blocks in binary tree Posmap and the number of blocks in RM-ORAM's Posmap at the i^{th} level of recursion be $|BPos_i|$ and $|RPos_i|$, respectively. The number of non-used pointer tuples in PosORAM of binary tree ORAM are equal to or greater than the number of non-used pointer tuples in PosORAM of RM-ORAM.*

$$\sum_{i=1}^{r-1}(m \cdot |BPos_i| - |BPos_{i+1}|) \geq \sum_{i=1}^{r-1}(m \cdot |RPos_i| - |RPos_{i+1}|) \tag{13}$$

Proof. Suppose there are x_a blocks (nodes of binary tree) in Posmap_a, and each position map block has m pointer tuples. Lemma 5 and Lemma 6 represent the range of increasing the number of position map blocks in Posmap_i to cover every position map block in Posmap_{i+1}. Let $x_{i+1} = z = 2^{y_{i+1}} - 1$, we can derive Equation 11 and Equation 12 to:

$$m \geq m \cdot (x_i + 1) - z > 0 \tag{14}$$

$$m \cdot 2^{y_i} \geq m \cdot (2^{y_i+1} - 1) - z > 0 \tag{15}$$

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 5 holds when Equation 14 and Equation 15 are true. □

B. Bandwidth Cost

In RM-ORAM the new client has to construct the history list from the very beginning of communication to server. It does so by generating the 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 the dummy accesses can be ignored for long term communication. In RM-ORAM the bandwidth spent in 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 7. Let the number of blocks accessed per upload/download operation be h, therefore the asymptotic bandwidth cost of RM-ORAM is $O(\log N)$ as described in Theorem 6.

Lemma 7. *The number of levels of recursion of RM-ORAM construction is:*

$$r = \lceil \log_m N \rceil \tag{16}$$

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 16. □

Theorem 6. *Let BW refer to bandwidth cost of RM-ORAM:*

$$BW = O(\log N) \tag{17}$$

Proof. As h and m are constant and independent of the size of ORAM they do not impact on the bandwidth cost. The number of levels of recursion, r, and the position map block's index header are dependent on N. However the index header is very small compared to the position map size, therefore the bandwidth cost depends primarily on r. Therefore using Lemma 7, the asymptotic bandwidth of RM-ORAM is bound by $O(\log N)$. □

C. Computation Overhead

We measure the computation overhead in terms of time complexity of the operations. From Section II-D, 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:** 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 old blocks during the download operation:** To achieve ORAM security requirements, random o from h blocks of previous operation are chosen. Since o and h are constant, the cost of randomly choosing a number of elements from a finite set is $O(1)$.
- **Randomly choose some blocks from history list:** As the cost of randomly choosing a 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 the 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:

$$TC = O(\log N) \tag{18}$$

D. Suggested Value of h

Recall from Section III and Section IV, h is significant to the security and performance of RM-ORAM system. As a lower limit to ensure the server cannot identify the data of interest from other data items, $h-o$ must be greater than or equal to 2. Furthermore, Theorem 2 shows that to provide the equivalent average number of accessed old data blocks as Path ORAM, o should be 2 or more. In addition, according to Theorem 3, the number of chosen new blocks and chosen blocks from history list should be equal, and each should be greater than or equal to 1. Therefore h must be greater than or equal to 4.

As our aim is to provide equal or better performance than Path ORAM, we consider h that makes RM-ORAM consume less bandwidth than Path ORAM when both constructions share the same number of N and m. Section II-D shows 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 7. *RM-ORAM requires fewer downloaded position map blocks compared with Path ORAM when:*

$$h \cdot (r-1) \leq \left\lceil \log\left(\frac{N}{m}+1\right)\right\rceil + \sum_{i=1}^{r-2}(H_i) \tag{19}$$

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 [22]). 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, client accesses a different binary tree Posmap for each level of recursion. At each level, the number of position map blocks that client has to download is equal to the height of binary tree of this level. Recursive Path ORAM has a binary tree at level i of height $\left\lceil \log\left(\frac{2^{(H_{i+1})}-1}{m}+1\right)\right\rceil$ and the height of the 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:

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

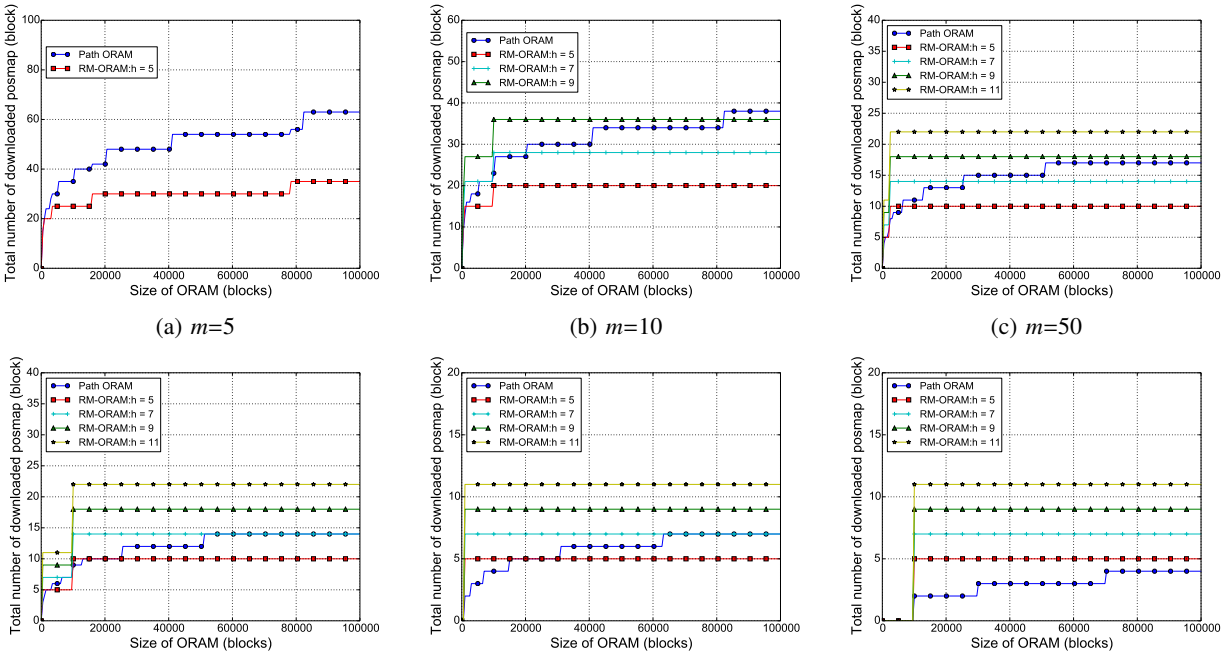
$$h \cdot (r-1) \leq \left\lceil \log\left(\frac{N}{m}+1\right)\right\rceil + \sum_{i=1}^{r-2}(H_i) \tag{20}$$

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

By applying the different values of N, m and h to Equation 19, the comparison of the number of downloaded position map blocks per access operation between RM-ORAM and recursive Path ORAM is as illustrated in Figure 5. Even though Path ORAM has fewer 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. 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. Considering the implementation of RM-ORAM from the experimental results, $4 \leq h \leq 7$ is a recommended value.

V. Experimental Results

Section III presented the security analysis, followed by the theoretical performance analysis of RM-ORAM in Section IV.



(a) m=5 (b) m=10 (c) m=50

(d) m=100 (e) m=1000 (f) m=10000

Fig. 5: Number of position map blocks downloaded per access request of different m

Since Path ORAM has the lowest computational overhead over other existing recursive ORAM constructions, we use Path ORAM as a standard for our experiment. In this section additional experimental results using an implementation of RM-ORAM and recursive Path ORAM are provided. These results provide insights about the stash usage in RM-ORAM. We implement recursive Path ORAM according to the algorithm in [22]. Both recursive Path ORAM and RM-ORAM are implemented in Python. We use AES-CBC with 256 bit key from the library Crypto to encrypt both the data blocks and position map blocks. Experiments are performed on a PC running Windows 7 64 bit on Intel i3 3.3 GHz CPU with 8GB of RAM. Despite RM-ORAM being designed for constrained storage devices, as the experiments focus on bandwidth cost and storage space which are measured in block sizes, the results are independent of hardware architecture. Therefore a PC is used due to ease of implementation and testing.

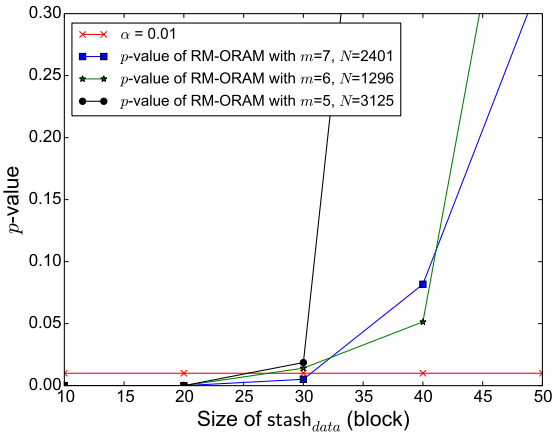


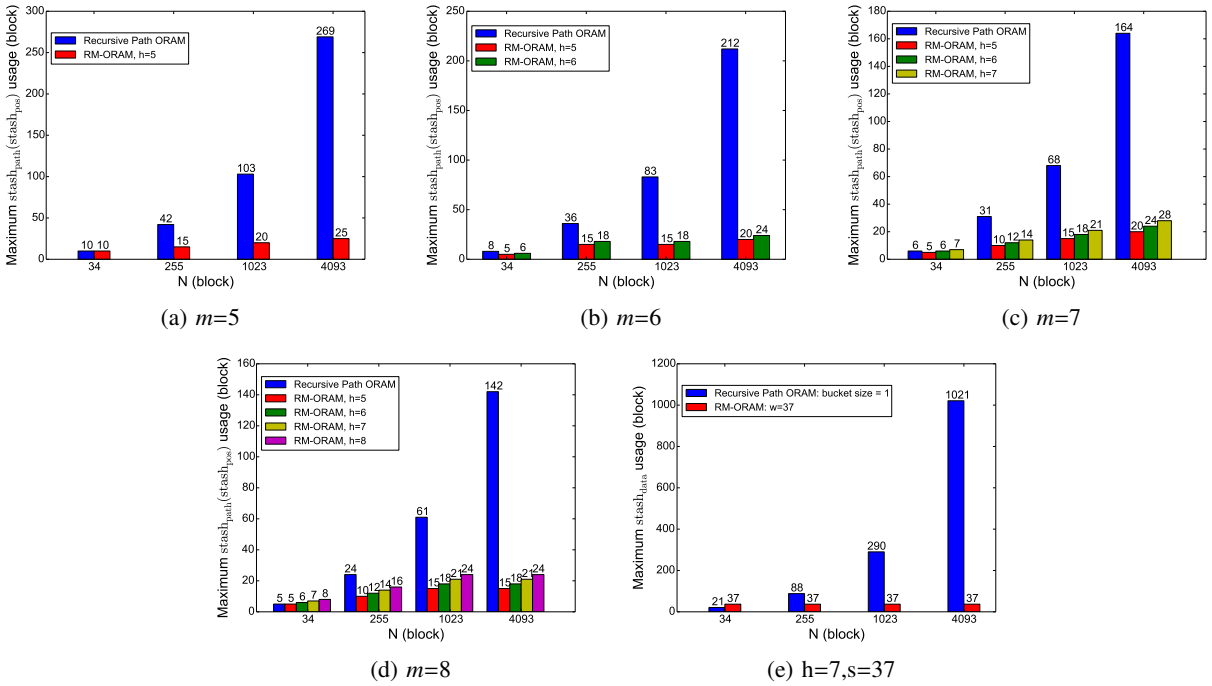
Fig. 6: p-value of χ^2 test over varied size of StashData

A. Suggested Size of StashData

In addition to storing the downloaded data block, StashData data structure is beneficial for random selection of data to be uploaded. In other words, the size of StashData impacts the data relocation and it should be large enough to allow the movement of data block to achieve random relocation. As an experiment, we move a specific block (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 m is equal to 5, 6, and 7 with $h = 6$. 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)* [30] 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 illustrates the p-value from χ^2 test of the different StashData sizes. After $s = 30 + h$ the movement of data block can be considered as uniform random. Therefore, the suggested size of StashData after finishing an upload operation is greater than 30 blocks.

B. Comparison of Stash Usage

This section compares the stash space requirements of RM-ORAM and recursive Path ORAM. Figures 7a to 7d show the maximum number of position map blocks stored in stash during a download operation. We refer to the size of the stash



(a) m=5 (b) m=6 (c) m=7

(d) m=8 (e) h=7,s=37

Fig. 7: Maximum stash usage of Recursive Path ORAM and RM-ORAM with different m

to store the position map blocks and data blocks as N_p and N_d, respectively.

For the thesis, you need to present the analysis better. You should first explain what the figures show, and then explain the reasons for the results

As Path ORAM uses leaf-ID of data to identify a path which contains data of interest, there are some data (whether it is position map block or data block) that cannot be uploaded back to the path when their leaf ID has been changed. Those data are left in the stash. The number of data items remaining in the stash is increasing when the size of DataORAM of Path ORAM is growing, therefore the space requirement for stash is also increasing. In RM-ORAM, although N_p is also growing when the size of DataORAM is increasing, it grows by a factor of $\log_m N$ which is significantly slower than Path ORAM as shown in Figures 7a to 7d. Figure 7e shows the comparison of the maximum number of N_d which is constant for any N of RM-ORAM while it is increasing when N is growing in Path ORAM. RM-ORAM requires constant space for storing a data block. Since h data blocks are downloaded, the random h data blocks of StashData are being uploaded back to a server whether or not it is the same set of data blocks. On the other hand, N_d of Path ORAM is growing according to the size of ORAM which is increasing.

VI. Conclusion

RM-ORAM is a novel ORAM structure which is designed for constrained storage devices. It uses recursion to reduce the space that is required for storing the location of information on the server. By designing the construction based on M-ORAM, RM-ORAM consumes $O(\log N)$ blocks on the client instead of $O(N)$ as required in M-ORAM. However, the recursion introduces additional bandwidth and computational overhead: in M-ORAM both are $O(1)$, while in RM-ORAM they increase to $O(\log N)$.

Future work includes evaluating the computational complexity of RM-ORAM by optimising the implementation for constrained devices, i.e. phones, embedded computers. One avenue of optimisation is using a parallel architecture to improve the performance of operations on the matrix. Further study is also needed in how RM-ORAM can be used with multiple servers and/or multiple clients.

Acknowledgment

This work is partially supported by JSPS KAKENHI Grant (C)(JP15K00183), (JP15K00189), and (JP15K16005) and Japan Science and Technology Agency, CREST (JPMJCR1404) and Infrastructure Development for Promoting International S&T Cooperation and Project for Establishing a Nationwide Practical Education Network for IT Human Resources Development, Education Network for Practical Information Technologies. This work also partially supported by Thailand's "National Electronics and Computer Technology Center" (NECTEC).

References

[1] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi, "Searchable encryption revisited: Consistency properties, relation to anonymous ibe, and extensions," *Journal of Cryptology*, vol. 21, pp. 350–391, Mar 2008.

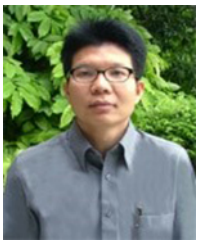
[2] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. 2012 ACM Conference on Computer and Communications Security (CCS)*, (Raleigh, North Carolina, USA), pp. 965–976, ACM, Oct 2012.

[3] K. Liang, X. Huang, F. Guo, and J. K. Liu, "Privacy-preserving and regular language search over encrypted cloud data," *IEEE Transactions on Information Forensics and Security*, vol. 11, pp. 2365–2376, Oct 2016.
[4] C. Zuo, J. Macindoe, S. Yang, R. Steinfeld, and J. K. Liu, "Trusted boolean search on cloud using searchable symmetric encryption," in *Proc. IEEE Trustcom*, (Tianjin, China), pp. 113–120, IEEE, Aug 2016.
[5] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.," in *Proc. 19th Annual Network and Distributed System Security Symposium, NDSS*, (San Diego, California, USA), The Internet Society, Feb 2012.
[6] C. Liu, L. Zhu, M. Wang, and Y.-A. Tan, "Search pattern leakage in searchable encryption: Attacks and new construction," *Inf. Sci.*, vol. 265, pp. 176–188, May 2014.
[7] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *ACM*, vol. 45, no. 6, pp. 965–981, 1998.
[8] X. Ding, Y. Yang, and R. H. Deng, "Database access pattern protection without full-shuffles," *IEEE Transactions on Information Forensics and Security*, vol. 6, pp. 189–201, Mar 2011.
[9] X. Yi, R. Paulet, E. Bertino, and G. Xu, "Private cell retrieval from data warehouses," *IEEE Transactions on Information Forensics and Security*, vol. 11, pp. 1346–1361, June 2016.
[10] S. Gordon, A. Miyaji, C. Su, and K. Sumongkayothin, "A matrix based ORAM: design, implementation and experimental analysis," *IEICE Transactions on Information and System*, vol. 99-D, no. 8, pp. 2044–2055, 2016.
[11] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, pp. 431–473, May 1996.
[12] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *Proc. 30th Annual Cryptology Conference*, (Santa Barbara, CA, USA), pp. 502–519, Springer, Aug 2010.
[13] D. Boneh, D. Mazieres, and R. A. Popa, "Remote oblivious storage: Making oblivious RAM practical," Tech. Rep. MIT-CSAIL-TR-2011-018, Massachusetts Institute of Technology, Mar 2011.
[14] J. Dautrich, E. Stefanov, and E. Shi, "Burst ORAM: Minimizing ORAM response times for bursty access patterns," in *Proc. 23rd USENIX Security Symposium*, (San Diego, CA, USA), pp. 749–764, USENIX Association, Aug 2014.
[15] C. Gentry, K. A. Goldman, S. Halevi, C. Jutla, M. Raykova, and D. Wichs, "Optimizing ORAM and using it efficiently for secure computation," in *Proc. 13th International Symposium PETS*, (Bloomington, IN, USA), pp. 1–18, Springer, Jul 2013.
[16] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," in *Proc. 23rd ACM-SIAM Symposium on Discrete Algorithms*, (Kyoto, Japan), pp. 157–167, SIAM, Jan 2012.
[17] N. P. Karvelas, A. Peter, S. Katzenbeisser, and S. Biedermann, "Efficient privacy-preserving big data processing through proxy-assisted ORAM," *Proc. IACR Cryptology ePrint Archive*, vol. 2014, p. 72, 2014.
[18] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: a parallel oblivious file system," in *Proc. ACM Conference on Computer and communications security*, (New York, NY, USA), pp. 977–988, ACM, Oct 2012.
[19] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," in *Proc. IEEE Symposium on Security and Privacy*, (Berkeley, CA, USA), pp. 253–267, IEEE Computer Society, May 2013.
[20] E. Stefanov, E. Shi, and D. X. Song, "Towards practical oblivious RAM," in *Proc. 19th Annual Network & Distributed System Security Symposium*, (San Diego, CA, USA), The Internet Society, Feb 2012.
[21] E. Shi, T. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O(log^3N)$ worst-case cost," in *Proc. 17th International Conference on the Theory and Application of Cryptology and Information Security*, (Seol, South Korea), pp. 197–214, Springer, Dec 2011.
[22] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proc. ACM SIGSAC Conference on Computer and Communications Security*, (Berlin, Germany), pp. 299–310, ACM, Nov 2013.
[23] T. Moataz, T. Mayberry, E. Blass, and A. H. Chan, "Resizable tree-based oblivious RAM," in *Proc. 19th International Conference, Financial Cryptography and Data Security*, (San Juan, Puerto Rico), pp. 147–167, Springer, Jan 2015.
[24] L. Ren, C. W. Fletcher, X. Yu, A. Kwon, M. van Dijk, and S. Devadas, "Unified oblivious-RAM: Improving recursive ORAM with locality and pseudorandomness," *Proc. IACR Cryptology ePrint Archive*, vol. 2014, p. 205, 2014.
[25] J. Zhang, Q. Ma, W. Zhang, and D. Qiao, "KT-ORAM: A Bandwidth-efficient ORAM Built on K-ary Tree of PIR Nodes," *Proc. IACR Cryptology ePrint Archive*, vol. 2014, p. 624, 2014.
[26] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion oram: A constant bandwidth blowup oblivious ram," in *Proc. Theory of Cryptography: 13th International Conference (TCC 2016-A), Part II*, (Tel Aviv, Israel), pp. 145–174, Springer, Jan 2016.
[27] B. Carbunar and R. Sion, "Write-once read-many oblivious ram," *IEEE Transactions on Information Forensics and Security*, vol. 6, pp. 1394–1403, Dec 2011.
[28] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via Oblivious RAM simulation," in *Proc. Automata, Languages and Programming: 38th International Colloquium*, (Zurich, Switzerland), pp. 576–587, Springer, Jul 2011.
[29] K. Sumongkayothin, S. Gordon, A. Miyaji, C. Su, and K. Wipusitwarakun, "Recursive M-ORAM: A matrix ORAM for clients with constrained storage space," in *Proc. Applications and Techniques in Information Security - 6th International Conference, ATIS*, (Cairns, QLD, Australia), pp. 130–141, Springer, Oct 2016.
[30] A. Rukhin, J. Soto, J. Nechvatal, E. Barker, StefanLeigh, M. Levenson, D. Banks, A. Heckert, J. Dray, S. Vo, A. Rukhin, J. Soto, M. Smid, S. Leigh, M. Vangel, A. Heckert, J. Dray, and L. E. Iii, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," 2010.

Atsuko Miyaji received the B.Sc., the M.Sc., and the Dr. Sci. degrees in mathematics from Osaka University, Osaka, Japan in 1988, 1990, and 1997, respectively. She joined Panasonic Co., LTD from 1990 to 1998 and engaged in research and development for secure communication. She was an associate professor at the Japan Advanced Institute of Science and Technology (JAIST) in 1998. She joined the computer science department of the University of California, Davis from 2002 to 2003. She has been a professor at Japan Advanced Institute of Science and Technology (JAIST) since 2007 and the director of Library of JAIST from 2008 to 2012. She has been a professor at Graduate School of Engineering, Osaka University since 2015. Her research interests include the application of number theory into cryptography and information security. She received Young Paper Award of SCIS93 in 1993, Notable Invention Award of the Science and Technology Agency in 1997, the IPSJ Sakai Special Researcher Award in 2002, the Standardization Contribution Award in 2003, Engineering Sciences Society: Certificate of Appreciation in 2005, the AWARD for the contribution to CULTURE of SECURITY in 2007, IPSJ/ITSCJ Project Editor Award in 2007, 2008, 2009, 2010, and 2012, the Director-General of Industrial Science and Technology Policy and Environment Bureau Award in 2007, Editorial Committee of Engineering Sciences Society: Certificate of Appreciation in 2007, DoCoMo Mobile Science Awards in 2008, Advanced Data Mining and Applications (ADMA 2010) Best Paper Award, The chief of air staff: Letter of Appreciation Award, Engineering Sciences Society: Contribution Award in 2012, and Prizes for Science and Technology, The Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology, and International Conference on Applications and Technologies in Information Security (ATIS 2016) Best Paper Award. She is a member of the International Association for Cryptologic Research, the Institute of Electronics, Information and Communication Engineers, the Information Processing Society of Japan, and the Mathematical Society of Japan.

Steven Gordon obtained a Ph.D. in Telecommunications from the University of South Australia in 2001. He has held positions as Senior Lecturer at University of South Australia, Associate Professor at Thammasat University, Thailand, and is currently Senior Lecturer at CQUniveristy, Australia. His research interests include: formal analysis of protocols; design of IP-based wireless ad-hoc, sensor and vehicular networks; and Internet security and privacy protocols. He serves on the editorial board and TPC of various international journals and conferences, and is a member of IEEE, ACM and IEICE.

Chunhua Su is an Associate Professor of University of Aizu, Japan. He received the B.S. degree from Beijing Electronic and Science Institute in 2003 and received his M.S. and PhD of computer science from Faculty of Engineering, Kyushu University in 2006 and 2009, respectively. He joined the Division of Computer Science, University of Aizu in April 2017. He has worked as a research scientist in Cryptography & Security Department of the Institute for Infocomm Research, Singapore from 2011-2013. From 2013-2016, he has worked as an Assistant Professor in School of Information Science, Japan Advanced Institute of Science and Technology. He also has worked as Assistant Professor in the Graduate School of Engineering, Osaka University from 2016-2017. His research interests include cryptanalysis, cryptographic protocols, privacy-preserving technologies in data mining and IoT security & privacy.



Komwut Wipusitwarakun is currently an Associate Professor at Sirindhorn International Institute of Technology(SIIT), Thammasat University, Thailand. He received the B.Eng. in Electrical Engineering from Chulalongkorn University,Thailand in 1993 and received his M.Eng. and Ph.D. in communication Engineer from Faculty of Engineering, Osaka University,Japan in 1996 and 1999, respectively. He joined the SIIT in April 1999. He works as a system manager of computer center and a professor of Information Computer&Communication Technology (ICT) School at SIIT from 1999-2017. His research interests include network reliability analysis and design, network virtualization, big scale sensor networking and community contributed networks.

Xinyi Huang received his Ph.D. degree from the School of Computer Science and Software Engineering, University of Wollongong, Australia. He is currently a Professor at the School of Mathematics and Computer Science, Fujian Normal University, China, and the Co-Director of Fujian Provincial Key Laboratory of Network Security and Cryptology. His research interests include applied cryptography and network security. He has published over 100 research papers in refereed international conferences and journals. His work has been cited more than 1600 times at Google Scholar (H-Index: 23). He is an associate editor of IEEE Transactions on Dependable and Secure Computing, in the Editorial Board of International Journal of Information Security (IJIS, Springer) and has served as the program/general chair or program committee member in over 60 international conferences.

Karin Sumongkayothin received the B.Eng degree from Kasetsart University in 2001, and received his M.Eng of Microelectronic from Asian Institute of Technology in 2003. He has worked as instructor in computer science department of Suan Dusit University from 2004-2013. He is currently a dual degree PhD. Candidate of Japan Advance Institute Science and Technology, Japan and Sirindhorn International Institute of Technology, Thailand. His research interests includes: code obfuscation; reverse code engineering; network and protocol security; and cryptography. He received Best Paper Award from International Conference on Applications and Technologies in Information Security (ATIS) in 2016.