IEICE TRANS. INF. & SYST., VOL.E99–D, NO.8 AUGUST 2016

PAPER *Special Section on Security, Privacy and Anonymity of Internet of Things*

A Matrix Based ORAM: Design, Implementation and Experimental Analysis*

Steven GORDON[†a)], **Atsuko MIYAJI**[††,†††b)], **Chunhua SU**[††c)], ***Members***,
and **Karin SUMONGKAYOTHIN**[†,††d)], ***Nonmember***

SUMMARY Oblivious RAM is a technique for hiding the access patterns between a client and an untrusted server. However, current ORAM algorithms incur large communication or storage overhead. We propose a novel ORAM construction using a matrix logical structure for server storage where a client downloads blocks from each row, choosing the column randomly to hide the access pattern. Both a normal construction and recursive construction, where a position map normally stored on the client is also stored on the server, are presented. We show our matrix ORAM achieves constant bandwidth cost for the normal construction, uses similar storage to the existing Path ORAM, and improves open the bandwidth cost compared to Path ORAM under certain conditions in the recursive construction.

key words: *ORAM, secure communication, secure access pattern, secure protocol*

1. Introduction

Oblivious RAM (ORAM) [1] is seen as beneficial for cloud computing, specifically for hiding access patterns from client to server. Encrypting the data before uploading to the server is not sufficient for privacy as it has been shown that the sequence of server storage locations read/written by the client may reveal valuable information to the server [2], [3]. Therefore, ORAM can be used to hide the access pattern by making the reads/writes indistinguishable from random accesses to the server. When a client wants to access (read or write) a block of data on the server, the client reads multiple blocks (including the block of interest) and writes back multiple blocks. The multiple blocks accessed are selected to make it impossible for the server to identify which block is of interest to the client and what operation is being performed. Despite many ORAM schemes being designed [4]–[12], achieving a satisfactory performance tradeoff remains a challenge. Key performance objectives for ORAM are:

- Minimize client storage, the persistent and temporary storage needed on the client.
- Minimize bandwidth cost, the total blocks transferred between client and server in order to access one block of interested data.
- Maximize server storage usage efficiency, that is, given the server can store N blocks of data, ensure as much as possible is available for storing real data.

In this paper we introduce a new ORAM that uses a matrix data structure on the server, called M-ORAM. The design of the matrix data structure allows us to keep the bandwidth cost independent of the number of blocks stored on the server, thereby reducing bandwidth cost compared to other ORAM schemes. In addition, M-ORAM's read and write operations are performed using a simple pseudo-random function without any complex operations such as shuffling, sorting and merging.

1.1 Related Work

ORAM algorithms can generally be divided by the server storage structure: either a hierarchical structure [4]–[12] where each layer is independent of each other; or a tree–structure [13]–[16] where nodes in neighboring layers have a relation of child and parent.

1.1.1 Hierarchical Based ORAM

The original ORAM was introduced by Goldreich *et. al* [1] with a hierarchical structure (pyramid structure) which each level is larger than its previous level. The accessing is done by reading from the most top level to the lowest bottom level. For each level of not the target information, a dummy information will be retrieved. The target information will be read, updated, and then uploaded to the most top level of ORAM. To protect the each level from buffer overflow, the eviction scheduling must be done at a particular level when it is accessed by given number of operation. The element from n^{th} level will be evicted then merged and shuffled with element of the next below level $(n+1)^{th}$, and stored at this level without leaking any critical information to the server.

Manuscript received November 30, 2015.
Manuscript revised April 13, 2016.
Manuscript publicized May 31, 2016.

[†]The authors are with Sirindhorn International Institute of Technology, Khlong Nung, Khlong Luang District, Pathum Thani 12121, Thailand.
[††]The authors are with Japan Advanced Institute of Science and Technology, Nomi-shi, 923–1211 Japan.
[†††]The author is with the Graduate School of Engineering, Osaka University, Suita-shi, 565–0871 Japan.

[*]This work is supported by the project "The Security infrastructure Technology for Integrated Utilization of Big Data" of Japan Science and Technology Agency CREST. This study is also partly supported by Grant-in-Aid for Scientific Research (C) (5K00183), (15K16005) and (15K00189) and Japan Science and Technology Agency (JST), Infrastructure Development for Promoting International S&T Cooperation.

a) E-mail: steve@siit.tu.ac.th
b) E-mail: miyaji@comm.eng.osaka-u.ac.jp
c) E-mail: chsu@jaist.ac.jp
d) E-mail: s1420209@jaist.ac.jp (Corresponding author)

DOI: 10.1587/transinf.2015INP0012

It incurs $O(\log^3 N)$ bandwidth cost with $O(1)$ client storage.

Partition ORAM or SSS-ORAM is an altered hierarchical structure ORAM that was introduced by Stefanov *et. al* [12] for cloud storage. Instead of using single ORAM, SSS-ORAM provides multiple ORAM partitions which can concurrently access by the client via the local reshuffling management module. By using this module, SSS-ORAM can manage to reduce the number of request to the server and achieves $O(\log N)$ bandwidth cost with $O(N)$ client storage. In addition, they proposed the *recursive ORAM* which reduces client storage to $O(\sqrt{N})$ by storing logical address of data in other SSS-ORAM on the server. Although the new method can reduce local storage requirement, it incurs increasing the bandwidth cost when compared to the normal SSS-ORAM construction.

1.1.2 Binary Tree Based ORAM

Binary tree based ORAM was first proposed by Shi *et al.* [14] with $O(\log^2 N)$ bandwidth cost, and $O(N)$ client storage under normal construction and $O(\log^3 N)$ bandwidth cost and $O(1)$ client storage under recursive construction. Unlike the hierarchical structure, the block (node) in the adjacent layer is related as parent and child. Every node at the bottom layer has a unique ID, called *leafID*, which identifies the path to access; the target information of interest is on that path. Even once the target information is found, the client continues to access other nodes in the path so that the server cannot identify the target data. Similar to hierarchical-based ORAM, the recently accessed information will be put in the top layer of the binary tree. To protect the top node from overflowing, an evict operation will be invoked every constant period of time which is called *background eviction*. During background eviction, two nodes from each layer will be randomly chosen. The element from the parent node will be mixed with its children nodes and randomly put to one of them. The remaining children nodes will be filled with dummy information.

Stefanov *et al.* enhance Shi's scheme by removing the background eviction operation from their design which is called Path ORAM [15]. Same as Shi's scheme, each data item in Path ORAM is associated with the leafID which identifies the path in the binary tree. Instead of accessing one by one on the path, every element in the path is downloaded and temporarily stored in the client. The leafID that is associated with the target information is randomly changed, then the client will try to upload the elements that are stored in its buffer to the previous downloaded path, under the condition that each element must be located on a path starting at its leafID. In the case that there are no elements that can be stored in the node, dummy information will be stored instead. Path ORAM has $O(\log N)$ bandwidth cost with $O(N)$ client storage using a normal construction, and $O(\log^2 N)$ bandwidth cost with $O(\log N) \cdot \omega(1)$ client storage when using a recursive construction (where $\omega(1)$ is some constant number).

1.1.3 Contribution

One aspect of ORAM's inefficiency is the bandwidth cost. To hide access patterns, ORAM requires the client to download/upload multiple blocks, even though it is interested in either reading or writing just one of those blocks. The number of blocks to access usually depends on the total number of blocks the server may store. This paper provides an alternative ORAM structure, called M-ORAM. It makes the bandwidth cost independent of the total number of blocks, instead dependent on the matrix height which is controllable by the designer.

Unlike any existing schemes, M-ORAM is built upon a matrix data structure for server storage. With the normal construction, it achieves $O(1)$ bandwidth cost, the constant size of stash buffer, and N blocks of position-map. For the recursive construction, M-ORAM can achieve the best bandwidth cost at $O(\log N)$ under $O(\log N)$ client storage. Therefore, our major contributions are:

- **The design of normal and recursive constructions for M-ORAM.** We present the detailed design of the M-ORAM normal construction and show how it can be modified to recursively store the position map in ORAM. Both constructions have a slightly different operation for retrieving the information of interest from the server.

- **Bandwidth cost independent from the ORAM size.** We introduce the first matrix based ORAM structure where the number of downloaded blocks per access request is independent of the size of ORAM storage. Therefore, it can achieve the improved bandwidth cost with any size of ORAM compared with the existing schemes, and especially Path ORAM.

- **Maximize and efficient stash usage.** In our construction the reserved space for stash buffer is being used about 90% of the time and never overflows (whereas there is a very small probability that an overflow occurs in Path ORAM). Also with an appropriate stash width, M-ORAM can utilize a smaller stash than Path ORAM.

- **Theoretical performance analysis.** We give the theoretical performance models of both normal and recursive constructions and compare to Path ORAM under same conditions. In addition, we give a novel proof of the appropriate height of M-ORAM in order to achieve the same security when transmitting information as Path ORAM.

- **Experimental analysis.** To provide further insights into M-ORAM performance and security not given by the theoretical analysis, we have implemented M-ORAM and provide experimental analysis of the

Table 1 Notation

Parameter	Description
N	Size of ORAM [blocks]
H	Height of ORAM logical structure [blocks]
W	Width (length) of each stash buffer [blocks]
SecretKey	Common secret key for encryption/decryption key generator
K	Encryption/Decryption key
dataID	Data identification number
stash	Temporary buffer for downloaded information
(x, y)	Position of rows and columns in matrix structure
m	Number of addresses in single block of ORAM position-map
counter	Individual counter of each information
Pos$[i]$	[*counter*, (x, y)], Position map of *dataID* i
loclist	Temporary address list of downloaded data
oldlist	Address list of previous access operation
PRF()	Pseudo-Random Function

bandwidth cost, stash usage and probability of reading/writing information to the same location.

The design of the M-ORAM normal construction has been reported in [17]. In addition to providing updated design details in this paper, we also introduce the recursive construction, its analysis and comparison with Path ORAM, new security analysis and results from the experimental analysis.

1.1.4 Paper Organization

The remainder of this paper is structured as follows. We provide an overview of M-ORAM storage structure in Sect. 2, and then its constructions and operations are presented in Sect. 3. In Sect. 4 the theoretical performance analysis of M-ORAM is provided and used to compare M-ORAM to other schemes. Section 5 gives a discussion of M-ORAM's security analysis. The experimental results from our implementation compared with Path ORAM are given in Sect. 6, and we conclude in Sect. 7. The notation used in this paper is summarized in Table 1.

2. M-ORAM Storage Structure

Two key design decisions for ORAM are the logical address structure and the operation for accessing information. In this section we describe the server and client storage structures used by M-ORAM (Fig. 1), while its operations will be described in Sect. 3.

2.1 Server Storage Structure

In other ORAM schemes, the bandwidth cost depends on the total number of the blocks stored on the server. For example, Path-ORAM [15] has bandwidth cost $O(\log N)$ for N blocks of ORAM. Our aim is to improve on this bandwidth cost. We do so by introducing a matrix structure for server storage which is illustrated in Fig. 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 where $i \in \{0, 1, \ldots, H\}$ and $j \in \{0, 1, \ldots, \frac{N}{H}\}$. Those logical

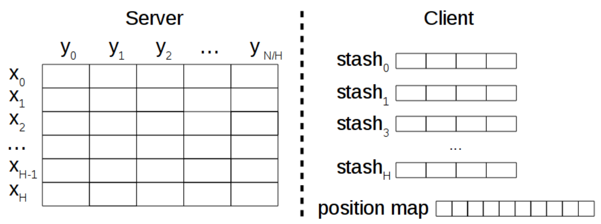

Server
Client
y_0 y_1 y_2 … $y_{N/H}$
x_0 x_1 x_2 … x_{H-1} x_H
stash$_0$
stash$_1$
stash$_3$
…
stash$_H$
position map

Fig. 1 M-ORAM structure

addresses are stored in the client and they will be accessed whenever the client needs to retrieve content from the server. 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.

2.2 Client Storage Structure

Similar to other ORAM schemes, the client uses a buffer called *stash* for temporarily storing the downloaded blocks, and another buffer called *position map* for mapping the logical addresses to *dataIDs*. However unlike other schemes, M-ORAM uses a separate stash for each row as illustrated in Fig. 1. In the normal M-ORAM construction, every address of the data is in the client's position map. To reduce the client storage requirements, the recursive construction is used, where most of the position map information is stored on the server, while leaving a small amount of position map information on the client. We denote each stash buffer as *stash*$_i$ where i is the matrix rows number.

3. M-ORAM Construction and Operation

We introduce two new matrix ORAM constructions: the *normal construction*, and the special construction for the client with constrained storage space called *recursive construction*. For both constructions, the operations can be categorized as Read, Write, Add, Delete, and Secret Key Management. Both constructions share the same Key Management, Add, and Delete operations; the other operations differ in details. Table 2 lists the functions used by some of the operations.

3.1 M-ORAM Normal Construction

As mentioned in the previous section, the bandwidth cost of M-ORAM is independent of the size but dependent on the height of ORAM. It gives us the ability to control the system bandwidth by keeping the height at some 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 maximize the efficiency of storage usage to 100% for containing the client's information. Another advantage of the matrix based structure is the independence of the block in the different levels of the hierarchy.

Table 2 Description of M-ORAM functions

Function Name	Description
ReadBl(x, y)	Read information from server at position x and y
RndStash(*data*)	Randomly put data to stashes without duplication
RndData($stash_i$)	Randomly pick up data from $stash_i$ without duplication
RndOld(*oldlist*, n)	Randomly pick up n addresses from *oldlist*
UpdatePos$(i, (x, y))$	Update position-map of *dataID* i
WriteBl(*data*, (x, y))	Write information to the server at position x and y

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 (Sect. 5.3). In this section, Read/Write, Add/Delete, and Secret Key Management operation will be introduced respectively.

3.1.1 Read/Write Operation

In M-ORAM whenever the client wishes to read or write data, it must actually read multiple blocks and then write multiple blocks back and forth to the server. One of the downloaded blocks must be the information that is required by the client, whereas the set of uploaded blocks is not necessarily the same set of previously downloaded blocks. The operation will begin whenever the client requests information from the server. The address of requested information is derived from the position map by using its *dataID*, then the read operation is invoked. The intersection of a distinct x_i and y_j is represented as the block position of information. As described in Algorithm 1, y_j will be uniformly randomly chosen if x_i does not belong to the requested information. The information in that location will be retrieved whether it is real information or not. The same process will be repeatedly applied to the next row and so on. The dummy contents are thrown away, while the real information is randomly spread to $stash_i$ without duplication (see Fig. 2 (a)). With this method, the location in X–axis of every content is changed and is independent from its original position.

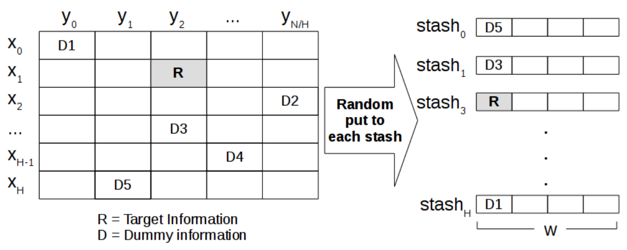
The randomization in the Y–axis will take place during the write operation as illustrated in Fig. 2 (b). The information will be uploaded by selecting from each stash uniformly at random, starting from $stash_0$ through to $stash_H$. This is to ensure the write access pattern is deterministic over indistinguishable information. As a block is selected from the entire $stash_i$, an emply block may be selected. In this case dummy information is uploaded. The chosen content from $stash_i$ will be stored at the previously accessed position (x_i, y_j). However to avoid the server being able to identify whether one access is of the same or different data as the previous access, at least one block location of the previous access must be randomly selected for the current access. In Sect. 4.1 we analyse how many elements from a previous access must be re-used in the current access, proving that choosing randomly between 1 and 3 blocks will given similar functionality as Path ORAM.

Algorithm 1 Read operation

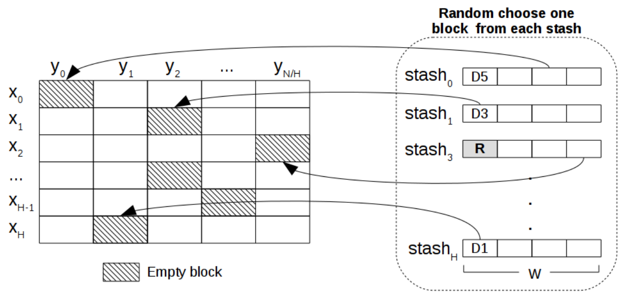
```
Input: $dataID, UpdateData$
$(x_d, y_d) \leftarrow Pos[dataID]$
$n \overset{\$}{\leftarrow} \{1, 2, 3\}$
$o \leftarrow \mathrm{RndOld}(oldlist, n)$
for $i \in \{0, 1, 2, 3, \ldots, H\}$ do
    if $x_i = x_d$ then
        $data \leftarrow \mathrm{ReadBl}(x_d, y_d)$
        $loclist \leftarrow loclist \cup \{(x_d, y_d)\}$
        if update operation then
            $data \leftarrow UpdateData$
        end if
        RndStash($data$)
    else
        if $x_i \in o$ then
            $loclist \leftarrow loclist \cup \{(x_i, y_*)\}$
            RndStash(ReadBl($x_i, y_*$))
        else
            $y_j \overset{\$}{\leftarrow} \{0, 1, 2, 3, \ldots, \frac{N}{H}\}$
            $loclist \leftarrow loclist \cup \{(x_i, y_j)\}$
            RndStash(ReadBl($x_i, y_j$))
        end if
    end if
end for
return $data$
```

Algorithm 2 Write operation

```
Input: $loclist, stash$
for $i \in \{0, 1, 2, 3, \ldots, H\}$ do
    $(x_i, y_j) \leftarrow loclist[i]$
    $loclist \leftarrow loclist - \{(x_i, y_j)\}$
    $dataID, data \leftarrow \mathrm{RndData}(stash_i)$
    UpdatePos($dataID, (x_i, y_j)$)
    WriteBl($data, (x_i, y_j)$)
end for
```



(a) Read operation



(b) Write operation

Fig. 2 M-ORAM read and write operations

3.1.2 Add/Delete Operation

In M-ORAM an Add operation involves the client putting new data in the a stash buffer (with the position chosen uniformly at random). There are no communications with the

Algorithm 3 Secret key management

```
Input: $dataID, SecretKey$
$counter \leftarrow Pos[dataID]$
$K \leftarrow \text{PRF}(dataID, counter, SecretKey)$
$text \leftarrow \text{Decrypt}_K(data)$
$counter \leftarrow counter + 1$
$Pos[dataID] \leftarrow counter$
$K \leftarrow \text{PRF}(dataID, counter, SecretKey)$
$data \leftarrow \text{Encrypt}_K(text)$
return $data$
```

server. A Delete operation is identical to a Read operation, hwoever after data to be deleted is downloaded, it is not stored in the stash buffer.

3.1.3 Secret Key Management

As with other ORAM schemes, data is re–encrypted using symmetric key encryption. A new fresh key is applied to an information before it is uploaded. Therefore, the server cannot identify even if a subsequent upload is the same data as a previous upload. The details of the encryption algorithm and management of encryption keys differ amongst ORAM schemes. In our construction we use AES as our encryption method; use the *counter, dataID,* and a *common secret key* as the initial elements for generating the block's encryption key. Every counter is kept within the position map along with its associated block location, and it is increased whenever the information has been downloaded. To generate the encryption key, the client will use those three elements as input into a pseudo-random function (PRF) as described in Algorithm 3.

3.2 M-ORAM Recursive Construction

The position map on the client can introduce a significant overhead for a client with limited storage size. Different ORAM schemes [12], [14], [15] have therefore proposed a *recursive construction* where the majority of the position map is stored on the server with ORAM, and the remaining part of the position map is on the client. In this section the discussion of the difference between M-ORAM normal and recursive constructions is given. Then, as they differ from the normal construction, Read/Write operations for the recursive construction are introduced. Finally, the bandwidth cost for the recursive construction is discussed.

3.2.1 Difference between Normal Construction and Recursive Construction

In M-ORAM normal construction, all of the position map is stored on the client. Suppose the ORAM has the capacity to contain N data blocks, the space requirement for the position map on the client is bounded by $O(N)$ blocks. On the other hand in the recursive construction, client storage space reduction can be achieved by storing the position map of the data also in ORAM, and a position map of that position map being stored on the client. This can be applied recursively to reduce the client storage requirements. Suppose that there

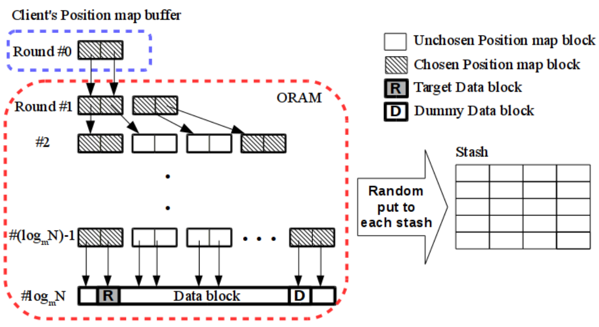


Fig. 3 Read operation for recursive construction

are m addresses be stored within a block, which is called a *position map block*, and these addresses point to either the data block or another position map block in next recursion. With N data blocks, $\log_m N$ recursions for retrieving the position map block are needed to cover every address of the N data blocks and allow the client to store only $O(1)$ blocks for the position map. Therefore, this recursive construction can keep the client storage usage at particular constant value but consumes more bandwidth cost as a trade-off as each access to a position map requires communication overhead.

3.2.2 Read/Write Operation of Recursive Construction

Let the position map block contains m block's addresses of next recursion, and H blocks are downloaded in each recursion from N blocks ORAM, where $H \leq m < N$. The Read/Write in recursive construction, the $H \log_m N$ blocks will be downloaded and randomly put into any empty blocks within arbitrary stashes. Then the same amount of information will be randomly chosen and uploaded back to the server. However, the blocks that have been chosen must relate to each other as a link list, because the information which is kept in the position map block will be used to determine the address of either data block or position map block for next recursion. The new address is randomly assigned for each chosen block. By the reason that every downloaded block is logically related, its new address in the position map block can be updated without failure.

To be more precise, the toy example is given as the Fig. 3. Suppose recursive M-ORAM has a height equal to 2 blocks, and the position map block contains $m = 2$ addresses of the next round operation. Therefore, 2 blocks will be randomly downloaded for each round. The first round (Round 0) position map that is stored on the client will point to two blocks of the next round position map (Round 1). At this point, another two blocks that are pointed from the position map of current round will be randomly picked up. The same procedure will be applied for each round until reaching the data block in the final round (Round $\log_m N$). During the write operation, the same number as previous downloading of "related block" will be randomly chosen from the stash. The new location of each related position map block will be randomly chosen by the client then uploaded back to the server.

3.2.3 Bandwidth Cost of Recursive Construction

The bandwidth cost of the M-ORAM normal construction with height H is $2H$. With the recursive construction with $\log_m N$ recursions the bandwidth cost is therefore $2H \log_m N$ blocks. As described in the previous section, the read/write operations differ slightly from what was used in the normal construction. Rather than downloading and then uploading for an access to the position map in ORAM, and then downloading and uploading for an access to the data in ORAM, in the M-ORAM recursive construction the client downloads all necessary blocks from both the position map and data ORAM, stores necessary blocks in the stash, and then uploads back to the server. This increases the stash size necessary for the client, which will be bounded by $O(\log_m N)$ (in the normal construction it was $O(1)$). Despite the extra stash size needed for the recursive construction, the overall client storage requirements (stash plus position map) for recursive construction ($O(\log_m N)$) is less than that needed for the normal construction ($O(N)$).

4. Performance Analysis

The key aim of M-ORAM is to decrease the bandwidth cost when the client accesses data on the server. To optimize the bandwidth cost, we first give a proof in Sect. 4.1 that the height in our construction is bottom bounded by 5 blocks. In Sects. 4.2 and 4.3, we analyze the bandwidth cost in both normal and recursive constructions of M-ORAM. Then we show in Sect. 4.4 that to achieve the better bandwidth cost comparing with Path ORAM, it depends on two factors: the number of addresses in position map block and the number of data block in ORAM. We also show in Sect. 4.5 that it is impossible to overflow the M-ORAM stash if the recommended size is used.

4.1 Lower Bound of the Height of M-ORAM Construction

The design of M-ORAM means we can freely choose any height for constructing the ORAM regardless of the number of blocks to be stored. However, varying the height may change the bandwidth cost and the security level of the ORAM. Therefore, we should consider the appropriate minimum/maximum height for M-ORAM. To do so, we compare our construction with Path ORAM.

As outlined in Sect. 3, M-ORAM requires one access operation to download some blocks which were accessed in the previous operation. If this wasn't the case, if the client accesses the same data consecutively, then there will be one block accessed through both operations. If the client, however, accesses different data, then the set of blocks accessed would be distinct. Hence, the server could distinguish if an operation is on the same data or different data. Therefore, M-ORAM requires o_m blocks (data blocks, in the case of recursive construction) from the previous operation to be accessed in the next operation. To determine an appropriate value of o_m we aim to provide equivalent functionality as Path ORAM. In Path ORAM the average number of previous buckets chosen to be downloaded at height H, $\bar{o}_p$ is:

$$\bar{o}_p = \frac{\sum_{i=1}^{H} 2^{i-1}}{2^{H-1}} \tag{1}$$

where $i \in \{1, 2, \dots H\}$. A proof is given in Appendix. Considering as a geometric series, as H tends to infinity, $\bar{o}_p$ tends to 2. Therefore, M-ORAM should download on average 2 blocks from the previous operation. To achieve this the client can choose 1, 2 or 3 blocks from the previous operation, uniformly at random, to download.

As the M-ORAM client may select 3 blocks from the previous operation to download, and one new block (the data of interest), the matrix height, H, must be at least 5 (i.e. 2 new blocks, 3 old blocks) to ensure the server cannot identify the data of interest.

4.2 Bandwidth Cost of M-ORAM: Normal

Using the normal construction, the M-ORAM client must download H blocks and upload H blocks for every access operation. Hence, the bandwidth cost is $2H$. The matrix height, H, can be set independently from the ORAM size, N. The height can be kept constant as the ORAM grows. However, there are limits on the height: those limits and the impact on security are discussed in Sect. 5. Assuming these limits are considered, we claim that the bandwidth cost of the M-ORAM normal construction is independent of ORAM size and bounded by $O(1)$.

4.3 Bandwidth Cost of M-ORAM: Recursive

Using the recursive construction and assuming r recursions, the M-ORAM client must download H blocks and upload H blocks r times. Hence the bandwidth cost is $2rH$. Suppose each position map block contains m addresses of blocks for the next recursion, then $\log_m N$ rounds are needed to keep the number of blocks of the first recursion to be 1. As with the normal construction, the matrix height is independent of N, and therefore the M-ORAM recursive construction can achieve $O(\log_m N)$ bandwidth cost.

4.4 Bandwidth Cost Comparison: M-ORAM and Path ORAM

With the normal construction, the bandwidth cost of our proposed M-ORAM is better than existing schemes such as Path ORAM [15]. M-ORAM has an $O(1)$ bandwidth cost and Path ORAM is $O(\log N)$.

With the recursive construction, we have shown that the M-ORAM bandwidth cost is bounded by $O(\log N)$, which is better than Path ORAM (bounded by $O(\log^2 N)$). In Sect. 4.1 we give a discussion about the lower bound of bandwidth requirement. In this section the upper bound of bandwidth is discussed. In the normal construction, as long as the height of M-ORAM is less than Path ORAM, we can achieve better bandwidth cost with any size of ORAM. However in the recursive construction, the bandwidth cost is depends on the the number of recursions, and the number of recursions depends on the number of addresses within a position map block. Therefore, the number of addresses within a position map block is another parameter (as well as height) that affects the bandwidth cost of recursive M-ORAM.

In Path ORAM the number of access requests depends on the height of binary tree structure: $\log N$, and the number of addresses that is stored within a position map block: m. Furthermore, after running $\log_m N$ recursions, we can construct the general equation of the total number of downloaded blocks ($total_p$) as:

$$\begin{aligned} total_p &= \sum_{i=0}^{\log_m N} \log \frac{N}{m^i} \\ &= \sum_{i=0}^{\log_m N} \log N - \sum_{i=0}^{\log_m N} \log m^i \\ &= (\log N)(\log_m N + 1) \\ &\quad - \frac{(\log m)(\log_m N)(\log_m N + 1)}{2} \\ &= \frac{\log^2 N}{2 \log m} + \frac{\log N}{2} \end{aligned} \tag{2}$$

Suppose M-ORAM has size N blocks with height equal to H, and it uses $\log_m N$ rounds for recursion. Therefore, the total bandwidth for downloading during an access operation ($total_m$) is:

$$total_m = H \log_m N \tag{3}$$

Comparing Eq. (2) with Eq. (3), we see M-ORAM can achieve better bandwidth cost than Path ORAM if:

$$H < \left\lfloor \frac{\log N + \log m}{2} \right\rfloor \tag{4}$$

4.5 Stash Usage

In M-ORAM, the stash usage is bounded by the controllable parameters the height of ORAM and the width of stash. In the case of the recursive construction a third parameter, the number of recurions, also impacts on stash size. However in this paper we focus on the stash usage in the normal construction, leaving the formalization for the recursive construction as future work.

In the normal construction, ORAM's height and stash's width are used to control the obfuscation property of the system. We consider the obfuscation of the system as the probability that the block will be retrieved, then written back to its original location in one access operation (called *probability of duplication*). The lower probability means more obfuscated and vice versa. We can create the equation of the probability when a particular block has been put into one of H stashes, and then written back to its previous location as follows:

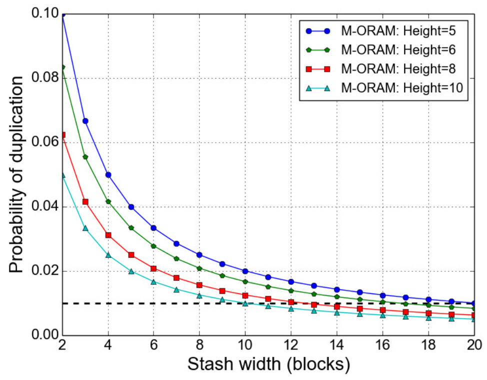


Fig. 4 Probability of duplication in M-ORAM

$$\text{probability of duplication} = \frac{1}{H \times W} \tag{5}$$

where H is number of stash buffer (ORAM's height) and W is the width of each stash.

Figure 4 shows the plots of probability of duplication with the different ORAM's height and stash's width. With fixed height of the ORAM matrix, increasing the stash's width decreases the probability of duplication. For our construction, we consider the probability of duplication must not be over 0.01 (1%), giving a guideline for the recommend stash width.

An 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 that is uploaded back to the server during an access operation, the stash always has empty spaces for the upcoming downloaded elements from next access operation. In addition, as we will show in Sect. 6.1.1, the reserved space for stash is efficiently used, i.e. almost 90% of its space is used all the time.

5. Security Analysis

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

5.1 ORAM Security Requirements

We can generally summarize the security requirements of the ORAM problem as follows:

1. The server cannot observe the relationship between the data and its address.
2. The server cannot distinguish between updated and non-updated information when it is written back to the server.
3. Two sequences of accesses to the server of the same length are computationally indistinguishable.

To achieve these properties, random re-encryption (Sect. 5.2) and the randomization over read/write operations (Sect. 5.3) are needed.

We define a series of access requests from client that the server will see as:

$$A = (pos_i[dataID_i]), pos_{i-1}[dataID_{i-1}], \dots, pos_1[dataID_1])$$

where $pos_j[dataID_j]$ is the set of addresses that have been accessed during retrieving information $dataID_j$ where $j \in \{1, 2, \dots, i\}$. Each block of information is given in the format $(counter_j, (x_m, y_n))$, where $counter_j$ is a counter for the re–encryption operation; x_m and y_n are the row m and column n of the matrix, respectively.

5.2 Random Re-Encryption

Every time the client has data to upload, the client first encrypts the data using a different key for each upload. We use the set of $(dataID_j, counter_j)$ together with a common *SecretKey* as the input to a pseudo-random function (PRF) to generate a key for encrypting the data. The reasons for using these three inputs are as follows. First, the *dataID* is unique per data block. Second, the *counter* is introduced so that each time the same data block is uploaded, a different value input to the PRF is used to ensure that the server cannot identify multiple uploads of the same content. Third, the *SecretKey* is secret, only known by the client and is necessary as the server may be able to learn the *dataID* and *counter*. Combining these three values as input to the PRF ensures that a "unique secret encryption key" will be used before the uploading. Hence, the *SecretKey* is secured and the server cannot distinguish encrypted information uploaded by the client.

5.3 Randomization over Access Pattern

In M-ORAM, the client downloads H blocks where, as described in Sect. 3, the dummy blocks are selected from each row with the column chosen uniformly at random for every row that does not belong to the requested block. Then every downloaded block is randomly pushed to each stash without duplication. At this point the number of ways to arrange the downloaded data in the stashes without duplication is $H!$. Suppose each stash has W blocks. Therefore, the number of ways to choose the data from each stash for writing back is W. Hence, the probability that downloaded contents will be written back to their previous location is:

$$\Pr(pos_j(dataID_j))_{\text{norm}} = \frac{1}{H! \cdot W^H} \tag{6}$$

where $j \in \{1, 2, \dots, i\}$

Suppose we have access request sequence A of size i and $j < k \in i$. When the $pos_j(dataID_j)$ is revealed to the server, it will be randomly remapped to the new position by probability $1 - \Pr(pos_j(dataID_j))$. Therefore the $pos_j(dataID_j)$ is statistically independent of $pos_k(dataID_k)$, with $dataID_j = dataID_k$. In the case of $dataID_j \neq dataID_k$, the address of different information does not have any relation, then those addresses are statistically independent of each other. Using Bayes rule, we can describe the probability of remapping addresses over a series of access requests A as:

$$\begin{aligned}\Pr(A)_{\text{norm}} &= \prod_{j=1}^{i} \Pr(pos_j(dataIDs_j))_{\text{norm}} \\ &= \left(H! \cdot W^H\right)^{-i}\end{aligned} \tag{7}$$

For now, lets assume M-ORAM has H and $W = 2$. Therefore, $\Pr(A)_{\text{norm}} = \frac{1}{2^{3i}}$ which is computationally indistinguishable from a random sequence of bit strings As H and W can be greater than 2, therefore, $\frac{1}{(H!\cdot W^H)^i} \leq \frac{1}{2^{3i}} < \epsilon$. Hence, $\Pr(A)_{\text{norm}}$ is computationally indistinguishable from a random sequence of bit strings for every $H, W \geq 2$

In the recursive construction, a single access request starts from reading the set of information and then writing the set of information. We replace $dataID_j$ with the set of information, $dataID_j[z], z \in \{0, 1, \dots, r\}$, where $dataID_j[z]$ is a set of data d blocks during z^{th} recursion, and r is the total number of recursions. Therefore, the probability of remapping addresses for a single request under the recursive construction is:

$$\begin{aligned}\Pr(pos_j(dataID_i[z]))_{\text{rec}} = &\left(\left(\prod_{k=1}^{d-1}(HW - (\varphi - k)\right)\right. \\ &\left.\times\left(\prod_{k=1}^{d-1}(HW-(\omega-k))\right)\right)^{-1}\end{aligned} \tag{8}$$

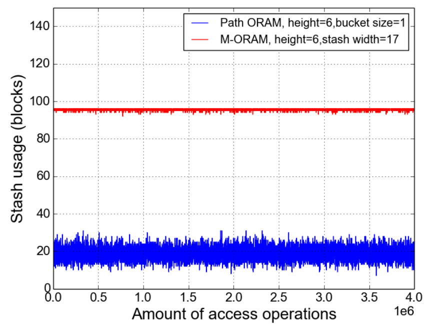
with $HW - \omega \leq d \leq HW - \varphi$ and where φ is the number of remaining elements in the stash, ω is the number of blocks that cannot be updated to the position map, and d is the number of data blocks among downloaded elements. Hence, the probability of a sequence request i is:

$$\begin{aligned}\Pr(A)_{\text{rec}} &= \prod_{j=1}^{i} \Pr(pos_j(dataID_j[z]))_{\text{rec}} \\ &= \left(\left(\prod_{k=1}^{d-1}(HW-(\varphi-k))\right)\times\left(\prod_{k=1}^{d-1}(HW-(\omega-k))\right)\right)^{-i}\end{aligned} \tag{9}$$

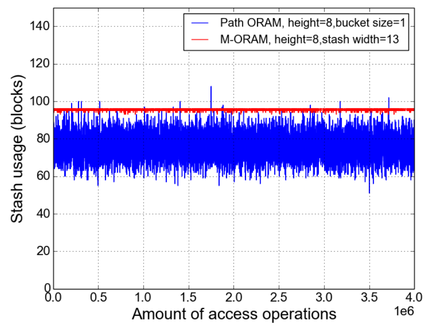
In the same manner as the normal construction, we can claim that a series of access requests using the recursive construction is also computationally indistinguishable from a random sequence of bit strings.

6. Experimental Results

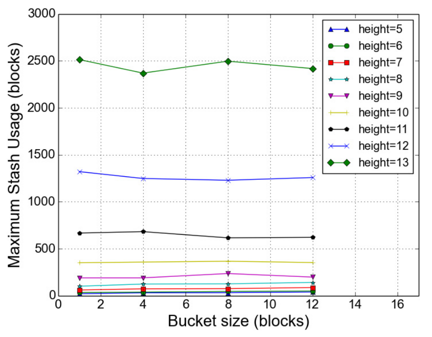
Section 4 presented theoretical performance analysis of M-ORAM, followed by security analysis in Sect. 5. However, the impact of some parameters, especially stash size, is not fully captured by this analysis. In this section, we present experimental results using implementations of Path ORAM and M-ORAM to compare their performance and security properties. Performance comparison is limited to the normal constructions and the experimental analysis of the recursive construction is left for future work.



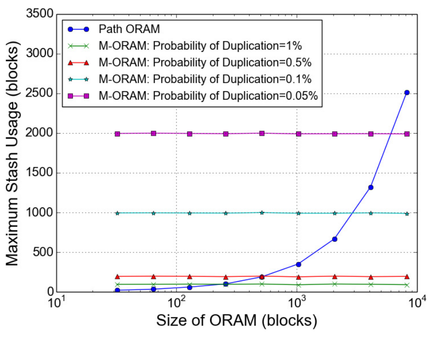
(a) ORAM height = 6



(b) ORAM height = 8



(c) Maximum with Path ORAM



(d) Maximum with M-ORAM

Fig. 5 M-ORAM and Path ORAM Stash Usage

6.1 Performance Comparison

We implemented both Path ORAM and M-ORAM in Python. Using a dataset of 20,000 strings from the UCI Machine Learning Repository [18] as input, we measure the performance of both ORAMs for random reads/writes across the data set. In each experiment the data is randomly stored in the ORAM, and then 4,000,000 accesses are made (that is, adding the new data into ORAM is not included in the experimental results).

The key performance metrics for ORAM are the client storage size and bandwidth cost per access request. To compare client storage size we consider only the stash size (assuming the position map of each ORAM is approximately the same size). Bandwidth cost is defined as the number of blocks uploaded and downloaded for each single block accessed.

6.1.1 Comparison of Stash Usage

In both Path ORAM and M-ORAM, the stash size is the main difference with respect to client storage requirements. Path ORAM has a single stash while M-ORAM has H stashes, all of the same width. Figures 5 (a) and 5 (b) show the total stash usage for two selected experiments, one with ORAM height of 6 and the other 8, for all access operations. The maximum stash usage across multiple experiments is shown in Fig. 5 (c) and 5 (d) for different parameters. As mentioned in Sect. 4.1, the lower bound of M-ORAM's height is 5 blocks. However, we want the probability distribution of choosing the old block to be similar to choosing the new block. Therefore, we first consider height, H, equal to 6 blocks (3 from the old block and 3 from the new block for worst case scenario). Figure 5 (a) 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 (stash width of 17); on the other hand, Path ORAM uses only 34 blocks, significantly better than M-ORAM. However with a larger height (Fig. 5 (b)), although the average stash usage of Path ORAM is still lower than M-ORAM, the maximum stash usage exceeds M-ORAM. The maximum stash usage should be the allocated storage to avoid overflow.

Analysis of Path ORAM maximum stash usage (Fig. 5 (c)) shows it increases as the height of the construction increases. In contrast, the maximum stash usage of M-ORAM is determined by two parameters: height and stash width. Figure 5 (d) shows the maximum stash usage is constant for every size of M-ORAM under the same probability of duplication. However if we consider the probability of duplication must be lower than 0.01, Path ORAM has lower stash size requirements than M-ORAM when ORAM's height is lower than 8.

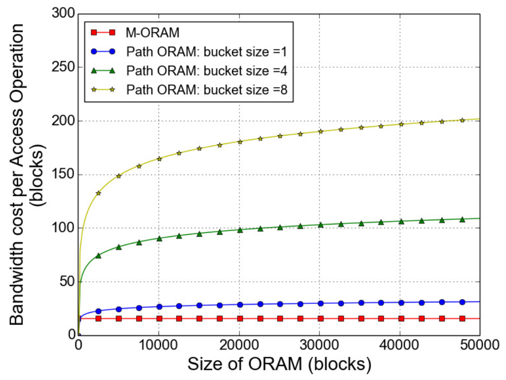


Fig. 6 Bandwidth cost with height = 8

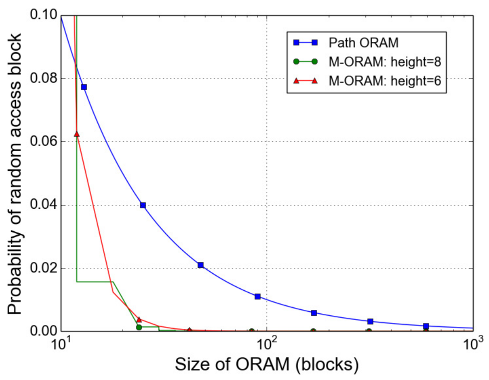


Fig. 7 Probability client reads same set of blocks

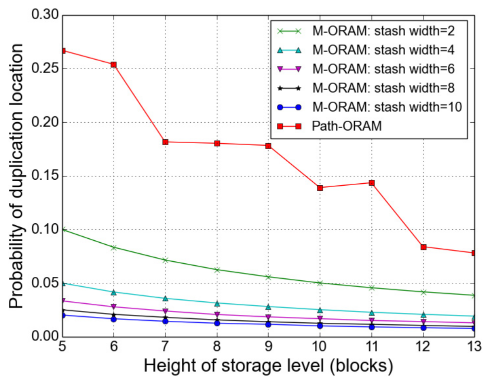


Fig. 8 Probability client writes to same address

6.1.2 Recommended Parameters for M-ORAM

Observing the experimental results together with the theoretical analysis given in Sect. 4, we give recommended values for key M-ORAM parameters. To achieve a probability of duplication of 0.01, we require $\lceil \frac{1}{H \times W} \rceil = 0.01$. In this case the matrix height, H, should be $8 \leq H \leq 50$, while the stash width, W, should be $2 \leq W \leq 13$.

Note the stash width is more significant than height with respect to randomization of a series of accesses (Eq. (7)). We also should minimize the height to achieve a low bandwidth cost. Therefore, $H = 8$ and $W = 13$ are recommended values for a normal M-ORAM construction.

6.1.3 Comparison of Bandwidth Cost

Figure 6 shows the bandwidth cost (per access operation) of Path ORAM and M-ORAM for different sized ORAMs (i.e. server storage space). M-ORAM uses a height of 8. In Path ORAM, there are two ways to increase the size of ORAM: increasing the number of blocks within a bucket, or increasing the height of the ORAM. However, both cause an increase in bandwidth cost. With Path ORAM as the ORAM size increases, the path length increases, leading to increasing bandwidth cost. On the other hand, M-ORAM has a fixed number of accessed block 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.

6.2 ORAM's Randomization Characteristic Comparison

In Sect. 5 we show that a series of accesses using M-ORAM is indistinguishable from random accesses. Here we look closer at the accesses with different parameters, specifically the probability that a client accesses the same set of blocks in two read operations, and the probability that a client writes data back to the same location. We compare with Path ORAM, with the aim of showing the probabilities with M-ORAM are no worse than Path ORAM.

6.2.1 Probability of Reading Same Blocks

During the reading period, the randomization of read information depends on the matrix's width. We show in Fig. 7 that the probability that the client will access the same set of information is decreases exponentially as the matrix width increases (i.e. the ORAM size increases). For both M-ORAM heights (6 and 8) and with a constant 2 blocks from a previous operation downloaded, the probability of reading the same location is less than Path-ORAM for all but very small ORAMs measured.

6.2.2 Probability of Writing to Same Address

During the write operation, one block from each stash will be randomly selected then uploaded back to the server. The percentage of times information is written to its previous location can be used to determine the randomization behavior of ORAM's writing operation.

In Fig. 8, we give a comparison between M-ORAM and Path ORAM with different height (from 5 to 13 blocks) and stash widths (2 to 10). M-ORAM with any height and any stash's width has less probability of writing information back to its old location, compared to Path ORAM.

7. Conclusion

M-ORAM uses a matrix storage structure to provide the same security as other ORAM schemes while reducing the bandwidth cost. In the normal construction, it gives $O(1)$ bandwidth cost with $O(N)$ client storage, while the recursive construction gives $O(\log N)$ bandwidth cost with $O(\log N)$ client storage. To prevent the server from distinguishing between two consecutive access operations on different data, we prove that the height of our construction should be at least 5 blocks, where 1 to 3 blocks are randomly selected from the previous operation. The results from the experimental analysis show the appropriate range of height is $8 \leq H \leq 50$ with stash width $2 \leq W \leq 13$. With these parameter values, M-ORAM can achieve better bandwidth cost than Path ORAM at the same probability of duplication.

References

[1] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," J. ACM, vol.43, no.3, pp.431–473, May 1996.

[2] M.S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," Proc. 19th Annual Network & Distributed System Security Symposium, San Diego, California, USA, Feb. 2012.

[3] C. Liu, L. Zhu, M. Wang, and Y.-A. Tan, "Search pattern leakage in searchable encryption: Attacks and new construction," Information Sciences: an International Journal, vol.265, pp.176–188, May 2014.

[4] 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, March 2011.

[5] J. Dautrich, E. Stefanov, and E. Shi, "Burst ORAM: Minimizing ORAM response times for bursty access patterns," Proc. 23rd USENIX Security Symposium, San Diego, CA, USA, pp.749–764, Aug. 2014.

[6] C. Gentry, K.A. Goldman, S. Halevi, C. Jutla, M. Raykova, and D. Wichs, "Optimizing ORAM and using it efficiently for secure computation," Proc. 13th International Symposium PETS, Bloomington, IN, USA, vol.7981, pp.1–18, Springer, July 2013.

[7] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," Proc. 23rd ACM-SIAM Symposium on Discrete Algorithms, pp.157–167, Kyoto, Japan, Jan. 2012.

[8] 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.

[9] B. Pinkas and T. Reinman, "Oblivious RAM revisited," Proc. 30th Annual Cryptology Conference, Santa Barbara, CA, USA, vol.6223, pp.502–519, Springer, Aug. 2010.

[10] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: a parallel oblivious file system," Proc. ACM Conference on Computer and communications security, New York, NY, USA, pp.977–988, ACM, Oct. 2012.

[11] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," Proc. IEEE Symposium on Security and Privacy, Berkeley, CA, USA, pp.253–267, IEEE Computer Society, May 2013.

[12] E. Stefanov, E. Shi, and D.X. Song, "Towards practical oblivious RAM," Proc. 19th Annual Network & Distributed System Security Symposium, San Diego, California, USA, Feb. 2012.

[13] 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.

[14] E. Shi, T.H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O(log^3 N)$ worst-case cost," Proc. 17th International Conference on the Theory and Application of Cryptology and Information Security, Seol, South Korea, vol.7073, pp.197–214, Springer, Dec. 2011.

[15] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," Proc. ACM SIGSAC Conference on Computer and Communications Security, Berlin, Germany, pp.299–310, ACM, Nov. 2013.

[16] 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.

[17] S. Gordon, A. Miyaji, C. Su, and K. Sumongkayothin, "M-ORAM: A matrix ORAM with $\log N$ bandwidth cost," Information Security Applications, Lecture Notes in Computer Science, vol.9503, pp.3–15, Springer International Publishing, Cham, 2016.

[18] M. Lichman, "UCI machine learning repository."

Appendix: Proof of Average Number of the Same Block Will be Downloaded between the Difference Operation in Path-ORAM

In this section we show how to calculate the average number of blocks that are accessed in one Path ORAM operation, which were also accessed in the previous operation. These blocks are referred to as duplicate blocks.

Theorem 1: The average number of duplicate blocks downloaded across two Path ORAM operations $\bar{o}_p$, is:

$$\bar{o}_p = \frac{\sum_{i=1}^{H} 2^{i-1}}{2^{H-1}} \tag{A· 1}$$

where $i \in \{1, 2, \ldots H\}$

Proof : Figure A· 1, which shows the binary-tree structure of Path ORAM with height H, is used in explaining the proof. Suppose that the set of blocks chosen during the previous access operation is represented as the gray circles in the binary tree. There are 2^{H-1} possible paths from a leaf node to a root of the tree. The next access operation downloads from one of the possible paths. As data blocks are assigned random leaf IDs in Path ORAM (they are changed for each accessing), 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

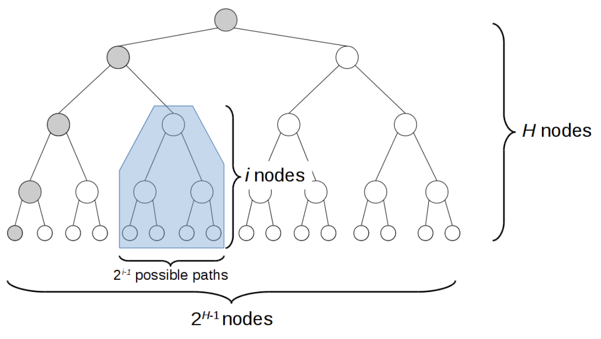


Fig. A· 1 Possible path of $H - i$ overlapped nodes exact same path as the previous operation). If the exact same path is not chosen, then there are 2^{i-1} possible paths that result in $H - i$ duplicate nodes, where $i \in \{1, 2, \ldots, H - 1\}$. Therefore, the average number of duplicate nodes/blocks is:

$$\begin{aligned}\bar{o}_p &= \frac{1}{2^{H-1}} H + \sum_{i=1}^{H-1} \frac{2^{i-1}}{2^{H-1}} (H - i) \\ &= \frac{\sum_{i=0}^{H} 2^i}{2^{H-1}} \qquad (A\cdot 2)\end{aligned}$$

Steven Gordon obtained a Ph.D. in Telecommunications from the University of South Australia in 2001, where he then worked as a senior researcher up until 2006. He has since been with Thammasat University, Thailand, currently as an Associate Professor within Sirindhorn International Institute of Technology. His research interests includes: formal analysis of protocols; integration of IP-based mobile networks with ad hoc networks; and Internet security protocols. He serves on the editorial board and TPC of various international journals and conferences. He is a member of IEEE, ACM and IEICE.

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 in 1998. She has joined the computer science department of the University of California, Davis since 2002. She has been a professor at the Japan Advanced Institute of Science and Technology since 2007 and the director of Library of JAIST since 2008. Her research interests include the application of number theory into cryptography and information security. She received Young Paper Award of SCIS'93 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, and 2010, 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 (2010) Best Paper Award, and The chief of air staff: Letter of Appreciation Award. She is a member of the International Association for Cryptologic Research, IEICE, IPSJ, and MSJ.

Chunhua Su received the B.S. degree for Beijing Electronic and Science Institute in 2003 and recieved his M.S. and PhD of computer science from Faculty of Engineering, Kyushu University in 2006 and 2009, respectively. He is currently working as an Assistant Professor in School of Information Science, Japan Advanced Institute of Science and Technology. He has worked as a Scientist in Cryptography & Security Department of the Institute for Infocomm Research, Singapore from 2011–2013. His research areas include algorithm, cryptography, data mining and RFID security & privacy.

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.