Security and Experimental Performance Analysis of a Matrix ORAM

Steven Gordon*, Atsuko Miyaji† Chunhua Su‡ and Karin Sumongkayyothin§
*§ Sirindhorn International Institute of Technology (SIIT), Thammasat University
† Graduate school of engineering, Osaka University
†‡§ Japan Advanced Institute of Science and Technology (JAIST)
† Japan Science and Technology Agency (JST) CREST
Corresponding Author: §s1420209@jaist.ac.jp,

***Abstract*—Oblivious RAM can hide a client's access pattern from an untrusted storage server. However current ORAM schemes incur a large communication overhead and/or client storage overhead, especially as the server storage size grows. We have proposed a matrix-based ORAM, M-ORAM, that makes the communication overhead independent of the server size. This requires selecting a height of the matrix; we present how to select the height to match the functionality of the well-known Path ORAM. We then given both theoretical models and experimental results that show M-ORAM can achieve a lower communication overhead than Path ORAM, without a significant increase in maximum client storage overhead.**

I. Introduction

Cloud computing has many benefits; however, raises significant privacy issues. For example, with cloud-based storage, a client stores data remotely on a server, accessing that data when necessary. The client may encrypt the data before uploading to the server to ensure the server cannot read the data. However by observing the pattern in which the client accesses the data (e.g. the address locations on the server, the order of reading and writing data), it is possible for the server to learn valuable information about the client [1], [2]. Oblivious RAM (ORAM) [3] is an approach for the client to hide these access patterns. The aim is to allow the client to access the data on the server without the server knowing: whether the client is performing a read (download) or write (upload); which data the client intends to access; and if a sequence of accesses is the same or different an observed previous sequence.

The general approach for ORAM schemes to achieve these aims is for every time the client wants to access one block of data, the client actually reads multiple blocks of data (with one being the block of interest) and then writes the same number of blocks back to the server. The blocks written back may not be the same as those read, as the aim is to move the data to random addresses on the server. With this approach it is possible to ensure the server cannot distinguish a sequence of accesses by the client from random accesses, thereby providing privacy of the access pattern. The main limitation of ORAM is however performance. Ideally, an ORAM scheme should minimize the: bandwidth cost (number of reads/writes); computation on the client; storage on the client; and wasted storage on the server. Various ORAM schemes have been proposed that consider different tradeoffs of these performance requirements [4]–[12].

Our previously proposed Matrix ORAM [13] is one such scheme. In this paper, we provide new design details of M-ORAM, analysis of the security parameters, and experimental evaluation of the bandwidth cost and client storage usage.

The remainder of the paper is organized as follows. Section II reviews related work and summarizes our contribution. Section III presents the design of our M-ORAM. Section IV gives theoretical performance analysis of M-ORAM. Section V discusses M-ORAM's security properties. Section VI presents experimental results comparing M-ORAM to Path ORAM. Section VII concludes the paper.

II. Related Work and Our Contribution

A. 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 [14]–[17] where nodes in neighboring layers have a relation of child and parent. The size of storage in both constructions is counted as a bucket, with each bucket containing either a single or multiple blocks of encrypted data.

The first ORAM was introduced by Goldreich *et al.* [3] with a hierarchical structure. Each hierarchy level i has 2^i buckets where $i \in \{1, 2, \ldots, \log N\}$ and N is the number of blocks in server's storage. To get the information of interest, buckets are scanned from level 1 to level $\log N$. By using a permutation function and unique secret key for each level, the client can determine the specific bucket on each level to be scanned for retrieving the target information. For each level that does not contain target information, a dummy block will be retrieved. Ultimately, the target information is moved to level 1. To protect each level from overflowing, information is evicted from level i to the level $i+1$; this is done every 2^i access operations and a new secret key for level $i+1$ is applied. Hierarchical-based ORAM incurs $O(\log^3 N)$ bandwidth cost with $O(1)$ client storage and involves complex processing by the client.

Binary tree based ORAM was first proposed by Shi *et al.* [15] with $O(\log^2 N)$ bandwidth cost and $O(N)$ client storage when using the normal construction and $O(\log^3 N)$

bandwidth cost and $O(1)$ client storage with the recursive construction (where data stored on the client in the normal construction is instead stored on the server in a second ORAM). This construction was improved by Stefanov *et al.* in the scheme called Path ORAM [16]. Path ORAM focused in simplifying the client operations, and can improve the performance to $O(\log N)$ bandwidth cost with $O(N)$ client storage for normal construction and $O(\log^2 N)$ bandwidth cost with $O(\log N) \cdot \omega(1)$ client storage for recursive construction where $\omega(1)$ is a constant number. Each block of real data in Path ORAM is associated with the a single leaf node, with a path from leaf to root. Instead of searching through the path, every element that is associated with that path will be downloaded and temporarily stored in the client (in a data structure call a *stash*). A new leaf node is chosen uniformly at random for the target information. Then the client will try to upload the blocks that are stored in the stash to the path that was previously downloaded from. However, not all blocks can fit in the path, meaning the stash will not be empty.

B. Our Contribution

Many ORAM schemes make a performance tradeoff of bandwidth cost versus client storage/processing. In particular, Path ORAM offers low bandwidth cost of $O(\log(N))$ with only a minor increase of client processing/storage.

In [13] we proposed a theoretical construction called Matrix ORAM (M-ORAM), which uses similar concepts as Path ORAM but the server storage structure is based on a matrix. The design allows the bandwidth cost to depend on a system parameter, the matrix height, rather than the size of the ORAM, thereby allowing a reduction in bandwidth cost for a fixed server storage size. A theoretical analysis of the bandwidth cost and client storage was given in [13]. In this paper, we offer the following new contributions:

1) Updated design of M-ORAM, including the method at which the matrix height is determined.
2) Analysis of the minimum recommended height for M-ORAM, so it provides similar behavior as Path ORAM.
3) The python-based prototype of M-ORAM.
4) Experimental results obtained to confirm the theoretical performance model, in particular that M-ORAM can reduce the bandwidth cost compared to Path ORAM while requiring only small increases in the client storage.

III. M-ORAM Storage and Operation

In this section we present the design of M-ORAM. We focus only on the normal (non-recursive) construction. As with other ORAM schemes, a recursive construction is possible that allows reducing the size of the client storage at the expense of increased bandwidth cost. The feasibility of the recursive construction is given in [13]; however it is not within the scope of this paper. Key notations of the M-ORAM design are summarized in Table I.

TABLE I: Notation

Parameter	Description
N	Total number of data blocks stored in server [blocks]
H	Height of ORAM logical structure in server-side
S	Size of each stash buffer [blocks]
SecretKey	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 column and row in matrix structure
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

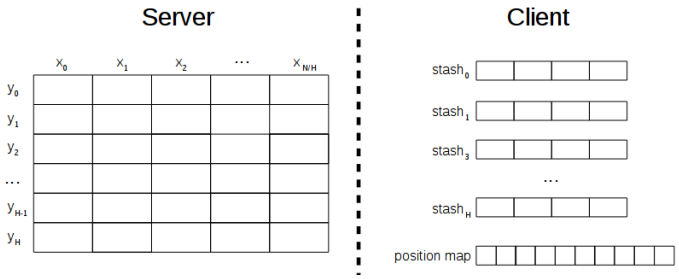


Fig. 1: M-ORAM structure

A. M-ORAM Storage Structure

In M-ORAM, N blocks of data are stored on the server in a matrix data structure of height H and width N/H as illustrated in Figure 1. An element in column i and row j is referred to as (x_i, y_j).

The client uses a position map to store the (x, y) values of each block of data stored on the server. Note that blocks are identified by a unique *dataID*. The client also uses a stash to temporarily store downloaded blocks. Unlike other ORAM schemes which have a single stash, M-ORAM uses a separate stash for each row in the matrix as illustrated in Figure 1. We denote each stash as $stash_j$ where j is the matrix row number. Each stash has a fixed length of S blocks, however not all blocks are necessarily used at any one time.

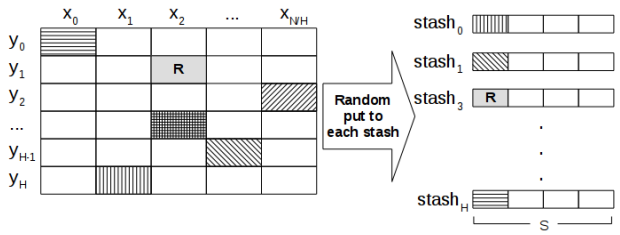
B. M-ORAM Operations

The operations that a client can perform with M-ORAM are: read data from the server, write/update data to the server, add new data to the server, and delete data from the server. The main operations are read and write, involving interactions with the server. The add/delete operations involve only client-side operations of inserting/deleting data into/from the stash, respectively. In this section, we present the read/write operations as well as the secret key management procedure.

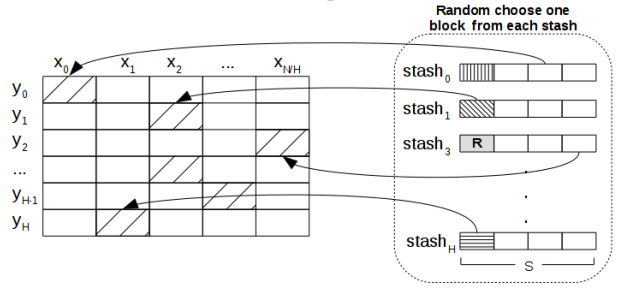
1) Read/Write Operation: In M-ORAM, whenever the client wishes to read or write data, it must actually read multiple blocks and then write multiple blocks back to the server. One among the many downloaded blocks must be the information that is required by the client, whereas the uploaded blocks are not necessarily the same set as previously downloaded. Whenever the client wants to access (read or

TABLE II: Description of 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 server at position x and y



(a) Read Operation



(b) Write Operation

Fig. 2: M-ORAM Operation

write) data *dataID* from the server, the address (x_i, y_j) is obtained from the position map. The client then reads H blocks from the server, one block from each row in the matrix. The columns of the matrix are chosen as follows: If the current row is y_j, then the column is x_i. Otherwise, o block locations are chosen uniformly at random from the set of blocks accessed by the previous operation, and the columns are chosen uniformly at random for the remaining $(H-o-1)$ rows. The purpose of selecting columns randomly (in addition to the block with the data of interest) is so the server cannot identify which data is of interest. However, some block locations must be the same as the previous operation so that the server cannot distinguish if this access is different from the previous operation. That is if we did not select some addresses from the previous operation, then accessing two different blocks would result in two distinct sets of (x, y) being accessed (with high probability), allowing the server to know the accesses are different. We discuss the appropriate value of o in Section IV-A.

Algorithm 1 and 2 show the read and write operations, respectively. Supporting functions are in Table II.

2) Secret Key Management: As with other ORAM schemes, each time a data block is accessed, it is re–encrypted using symmetric key encryption. After a block is downloaded it is decrypted. Before it is uploaded again it is encrypted using a new key. Therefore, the server cannot identify that the uploaded data is the same as previously accessed. In M-ORAM, we use AES for encryption, where the data block and its ID are encrypted using a key generated from a pseudo-random function (PRF) as described in Algorithm 3. Importantly, the PRF takes as input the *dataID* (unique to each block), a secret key (common across all blocks) and a counter (specific to each block). The counter is incremented after each access and stored in the position map on the client.

Algorithm 1 Read Operation

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

Algorithm 2 Write Operation

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

Algorithm 3 Secret Key Management

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

IV. Performance Analysis

The key aim of M-ORAM is to decrease the bandwidth cost usage when the client accesses data on the server. The bandwidth cost will depend on the matrix height. In Sec-

tion IV-A, we analyze the appropriate height, comparing it to the equivalent measure in Path ORAM. In Section IV-B, we analyze the bandwidth cost of M-ORAM and then in Section IV-C show that with M-ORAM, the stash buffer is impossible to be overflow even though its size is fixed.

A. Height of M-ORAM Storage

In M-ORAM, the height (H) of the matrix storage used in constructing the ORAM can be chosen independently of the number of blocks N stored on the server. However, varying the height leads to the changing bandwidth cost and the security level of the ORAM. To determine the appropriate height from a security perspective, we aim to provide equivalent functionality as Path ORAM.

The M-ORAM read operation downloads a block from each row in the matrix structure. One row contains the information of interest while the columns in the other rows are chosen randomly. However, to ensure two different accesses cannot be distinguished by the server, the columns in some of those other rows must be the same as in the previous read operation. Path ORAM also uses some blocks from the previous read operation. We therefore aim to set the number of blocks from the previous operation to be the same or greater than for Path ORAM.

First we find the average number of blocks that are re-used in Path ORAM, denoted as $\overline{o}_{poram}$. Consider the example Path ORAM binary tree in Figure 3. If the tree height is H nodes, then there are 2^{H-1} leaf nodes, and therefore 2^{H-1} possible paths. Suppose that a set of blocks chosen in the previous access operation are the gray nodes in Figure 3. Of the 2^{H-1} possible paths, only one of them will result in all H nodes being the same as the previous operation. Alternatively, there are 2^i possible paths for which $H-i$ nodes being the same as the previous operation (for $i \in \{1, 2, \ldots, H-1\}$). Hence, the average number of re-used blocks in Path ORAM is:

$$\begin{aligned}\overline{o}_{poram} &= \frac{H + \sum_{i=1}^{H-1}(H-i)\cdot 2^{i-1}}{2^{H-1}} \\ &= \frac{\sum_{i=1}^{H} 2^{i-1}}{2^{H-1}}, \text{where } i \in \{1, 2, \ldots H\}\end{aligned}$$

As H tends to infinity, using the geometric series of $\frac{1}{2^i}$, the average number of blocks re-used between two operations in Path ORAM tends to 2.

Therefore, if M-ORAM chooses 1 to 3 blocks from the previous operation uniformly at random, then on average it selects the same number of re-used blocks as Path ORAM. Hence, the minimum height requirements for M-ORAM is 5 blocks: 2 new blocks (one being the data of interest) and 3 re-used blocks. However, a larger height can be used to allow for more new blocks.

B. M-ORAM Bandwidth Cost

In M-ORAM, the bandwidth cost depends on the number of blocks read and written for each data access. With a matrix height of H, there are $2H$ blocks accessed (read then write for each accessed block). Similarly, Path ORAM (in the non-recursive construction) requires $2P$ blocks to be accessed where P is the path length. Hence, M-ORAM achieves the same bandwidth cost as Path ORAM if the matrix height is the same as the path length. However a key difference is that the path length in Path ORAM depends on the server storage size N, i.e. $P \approx \log(N)$. In M-ORAM H can be set independent of N (although there are constraints, as discussed in the previous section), allowing for a lower bandwidth cost than Path ORAM. Experimental results for bandwidth cost are given in Section VI-B.

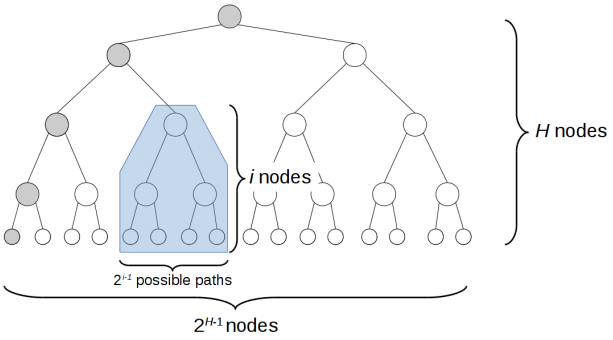


Fig. 3: Possible path of $H-i$ overlapped nodes

C. M-ORAM Stash Size and Usage

In M-ORAM, the stash usage is bounded by two controllable parameters which are height (H) of ORAM and width of stash buffer (S). As mentioned in Section III, the number of downloaded blocks is equal to the number of blocks that are uploaded back to the server during an access operation. Therefore the stash buffer always have empty spaces and impossible to be overflow for the upcoming downloaded elements from next access operation. In addition, the reserved space for stash buffer is efficiently used which is almost 90% of its space all the time (see Section VI-A).

V. Security Analysis

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

A. ORAM's Security Requirements

The security requirements of ORAM are:

1) The server cannot observe the relationship between data and its address.
2) The server cannot distinguish between updated and non-updated information when they are written back to the server.
3) The sequence of requested information cannot be differentiated from a random bit string.

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

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

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

B. 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 shared *SecretKey* as the inputs to a strong pseudo-random function (PRF) to generate a secret key for encrypting the data. The reasons for using these three inputs are: 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, known by the only 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.

C. Randomization Over Access Pattern

In M-ORAM the client downloads H blocks from a matrix of height H. The column is chosen uniformly at random for every row that does not contain the information of interest. Then every downloaded block is randomly pushed to the stashes without duplication. Therefore, the possible ways to store the downloaded blocks to stashes is $H!$. Suppose each stash has S blocks and we need to randomly choose a block from each stash to write back to the server. Therefore, the possible ways to choose the data from each stash is S. Hence, the probability that the same set of blocks will be written back to their previous location is:

$$Pr(pos_j(dataID_j)) = \frac{1}{H! \cdot S^H}, \text{ where } j \in \{1, 2, \ldots, i\}$$

Suppose we have access request sequence A 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; thus, those addresses are statistically independent of each other. As Bayes rule, we can describe the statistically independent of A as:

$$\prod_{j=1}^{i} Pr(pos_j(dataIDs_j)) = \left(H! \cdot S^H\right)^{-i}$$

It proves that the series of access requests is indistinguishable from a random sequence of bit string.

VI. Experimental Results

We implemented M-ORAM and Path ORAM in Python as a means for comparing the performance of our proposed scheme against one of the best-known ORAMs. In this paper, we focus on bandwidth cost and stash usage. Using 20,000 string datasets from UCI Machine Learning Repository [18] as data input, 100,000 access operations were made on each ORAM. Experiments were run on an Intel Core i5 CPU with 2 GB of memory, with both client and server running on the same computer.

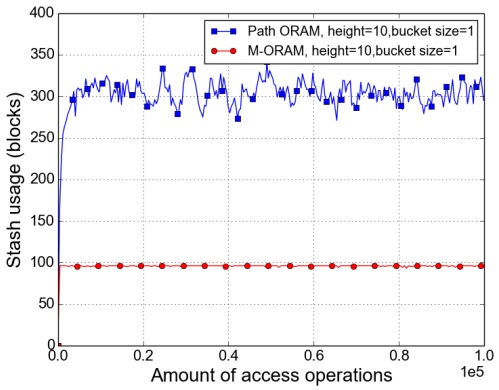


Fig. 4: Stash usage for all 1×10^5 access operations

A. Maximum Stash Size Requirement

An important performance metric for ORAM is the usage of storage at the client. It is desirable for it to be as small as possible. The position maps for M-ORAM and Path ORAM both require one entry for each data block. However, there are significant differences in the stash design: Path ORAM uses a single stash, M-ORAM has H stashes of length S. We measured the stash usage using the two different ORAMs.

Therefore, with the same number of downloaded elements per access request,the ORAM size was set to approx 1000 blocks. This was chosen to allow a M-ORAM can be competed with Path ORAM under same number of downloaded elements per access request while keeping the experiment run-time small. By giving around 1000 blocks of ORAM size, Path ORAM has height of 10 blocks which is larger than the minimum requirement of M-ORAM (Section IV-A).

Figure 4 shows the stash usage for both ORAMs which Path ORAM and M-ORAM bucket size of 1. For M-ORAM, we fix the height H to 10 and the stash size S to 10. The tree height in Path ORAM is the same as M-ORAM height. With a height of 10 blocks, M-ORAM requires 100 blocks for stash buffer but Path ORAM needs more than 300 blocks. To consider Path ORAM, the bucket size is increased, the stash size is also increased. Therefore, M-ORAM will always has a smaller stash than Path ORAM for height equal to 10. We expect this to hold for lager height as well.

B. Bandwidth Cost

In this paper, the bandwidth cost is approximated as the average number of blocks read/written per each access request. We measured the bandwidth cost for M-ORAM and Path ORAM with different ORAM sizes. For M-ORAM, the matrix

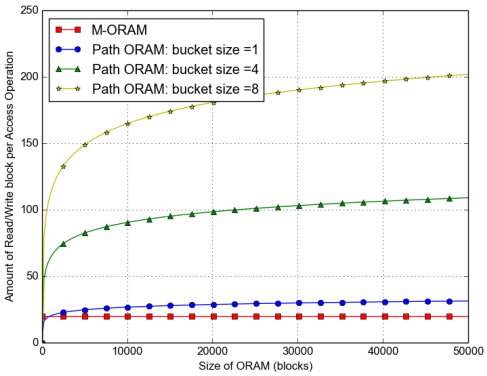


Fig. 5: Bandwidth cost for Path ORAM and M-ORAM

height is set to 10. For Path ORAM each node in the binary tree stores a bucket of blocks; we consider bucket sizes of 1, 4 and 8 blocks (4 is recommended by [16]). Figure 5 compares the bandwidth cost for the ORAMs. The results show the bandwidth cost of using M-ORAM is independent of the ORAM storage size, whereas for Path ORAM the it depends on the ORAM size. With Path ORAM, increasing the bucket size can decrease the stash usage; however, it leads to a significant increase in bandwidth cost.

VII. Conclusion

M-ORAM uses a matrix storage structure to provide the same security as other ORAM schemes while reducing the bandwidth cost and maintaining a manageable client storage size. We have presented the design of M-ORAM, based on an earlier version [13], specifically addressing the way in which M-ORAM selects blocks from a previous operation to ensure the server cannot identify two different access requests. We have also given analysis the number of blocks to be selected to give equivalent functionality to one of the best performing binary-tree ORAMs, Path ORAM. Our new experimental results backup the theoretical analysis that indicate M-ORAM can achieve a bandwidth cost independent of the server storage size, N (whereas Path ORAM bandwidth cost is $O(\log N)$) while keeping the practical stash size small. The client storage size can be further reduced using a recursive construction. That, and further analysis of different matrix and stash parameters, is left for future work.

Acknowledgement

This research is partly supported by Grant-in-Aid for Scientific Research (C) (5K00183) and (15K00189) and Japan Science and Technology Agency (JST), Infrastructure Development for Promoting International S&T Cooperation.

References

[1] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proc. 19th NDSS 2012*, San Diego, California, USA, Feb 2012.

[2] C. Liu, L. Zhu, M. Wang, and Y. Tan, "Search pattern leakage in searchable encryption: Attacks and new construction," *Proc. Inf. Sci.*, vol. 265, pp. 176–188, 2014.

[3] Goldreich, Oded, and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Proc. JACM*, vol. 43, no. 3, pp. 431–473, May 1996.

[4] D. Boneh, D. Mazieres, and R. A. Popa, "Remote oblivious storage: Making oblivious RAM practical," Massachusetts Institute of Technology, Tech. Rep. MIT-CSAIL-TR-2011-018, 2011. [Online]. Available: http://hdl.handle.net/1721.1/62006

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

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

[7] 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 2012*. Kyoto, Japan: SIAM, Jan 2012, pp. 157–167.

[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] Pinkas, Benny, and T. Reinman, "Oblivious RAM revisited," in *Proc. Advances in Cryptology - CRYPTO 2010*. Santa Barbara, CA, USA: Springer, Aug 2010, pp. 502–519.

[10] R. Sion and P. Williams, "Fast oblivious storage," *Proc. ACM Transactions on Information and System Security*, vol. 15, no. 4, Mar 2013.

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

[12] E. Stefanov, E. Shi, and D. X. Song, "Towards practical oblivious RAM," in *Proc. 19th NDSS 2012*. San Diego, California, USA: The Internet Society, Feb 2012.

[13] S. Gordon, A. Miyaji, C. Su, and K. Sumongkayothin, "M-ORAM: A matrix ORAM with $\log N$ bandwidth cost," in *Proc. 16th International Workshop on Information Security Applications 2015*. Jeju, South Korea: Springer, Aug 2015.

[14] 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.

[15] E. Shi, T. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O(log^3 N)$ worst-case cost," in *Proc. 17th Advances in Cryptology ASIACRYPT 2011*. Seol, South Korea: Springer, Dec 2011, pp. 197–214.

[16] 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 2013*. Berlin, Germany: ACM, Nov 2013, pp. 299–310.

[17] 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.

[18] M. Lichman. (2013) UCI machine learning repository. [Online]. Available: http://archive.ics.uci.edu/ml