Hindawi
Security and Communication Networks
Volume 2018, Article ID 2138147, 15 pages
https://doi.org/10.1155/2018/2138147

WILEY | Hindawi

Research Article

Practical Suitability and Experimental Assessment of Tree ORAMs

Kholoud Al-Saleh and Abdelfettah Belghith

Department of Computer Science, College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia

Correspondence should be addressed to Abdelfettah Belghith; abelghith@ksu.edu.sa

Received 21 April 2018; Revised 30 October 2018; Accepted 6 November 2018; Published 19 November 2018

Academic Editor: Bela Genge

Oblivious Random-Access Memory (ORAM) is becoming a fundamental component for modern outsourced storages as a cryptographic primitive to prevent information leakage from a user access pattern. The major obstacle to its proliferation has been its significant bandwidth overhead. Recently, several works proposed acceptable low-overhead constructions, but unfortunately they are only evaluated using algorithmic complexities which hide valuable constants that severely impact their practicality. Four of the most promising constructions are Path ORAM, Ring ORAM, XOR Ring ORAM, and Onion ORAM. However, they have never been thoroughly compared against each other and tested on the same experimental platform. To address this issue, we provide a thorough study and assessment of these recent ORAM constructions and implement them under the same testbed. We perform extensive experiments to provide insights into their performance characteristics, simplicity, and practicality in terms of processing time, server storage, client storage, and communication cost. Our extensive experiments show that despite the claimed algorithmic efficiency of Ring and Onion ORAMs and their judicious limited bandwidth requirements, Path ORAM stands out to be the simplest and most efficient ORAM construction.

1. Introduction

Outsourcing data to the cloud is becoming the norm nowadays, whether it is for business, research, leisure, or everyday life. The amount of data stored in the cloud is growing every day. However, outsourcing data to the cloud brings great security and privacy challenges [1, 2].

Encryption on the client side has been used to provide privacy to the stored data. Unfortunately, encryption is not enough since a lot of information can be inferred from the user's access pattern [3]. One cryptographic primitive that can be used to hide the access pattern is Oblivious Random-Access Memory (ORAM). ORAM was initially developed by Goldreich [4] and Ostrovsky [5] to provide software protection and later was used to provide protection for the access pattern of stored data. Various ORAM constructions, extensions, and improvements have been developed since the initial development of ORAM: Hierarchal ORAM [6], Partition ORAM [7], Tree ORAM [8], Path ORAM [9], SCORAM [10], Ring and XOR Ring ORAM [11, 12], and Onion ORAM [13].

However, in most of previous papers and studies, the focus has been on algorithmic complexity order of growth using the Big-O notation. As such, these studies often overlooked the values of the complexity constants in the Big-O notation that can be very large and therefore greatly affects the overall real performance of an ORAM. Moreover, many of the ORAMs have never been tested with large amounts of data [14–16]. Only few studies conducted experimental investigations, even though being limited to some ORAMs, like the work by Chang et al. [15].

In this paper, we investigate the practical performance of the latest and main tree based ORAMs. Using a unified platform, we implemented and compared Path [9], Ring [12], XOR Ring [11, 12], and Onion ORAMs [13], while Path ORAM was compared in one experimental study by Chang et al. [15] to some earlier ORAMs, to the best of our knowledge, there exists no experimental investigation to later and more

advanced ORAMs such as Ring, XOR Ring, and Onion ORAMs, yet there is still a lack of open source implementations of ORAMs [16, 17].

Our contributions can be summarized as follows:

(1) Critical presentation of Path ORAM, Ring ORAM, XOR Ring ORAM and Onion ORAM

(2) Qualitative performance comparison between the four considered ORAM schemes

(3) Implementation of these four considered ORAMs on a unified experimental platform

(4) Experimental assessment of the practical performance of the four considered ORAM schemes

(5) Analysis of the practical suitability of each of the considered ORAMs

The rest of the paper is organized as follows. In Section 2, we present a general description and the evolution of the ORAM techniques that are relevant to the completeness of the paper and show how these constructions are related to each other. In Section 3, we present a detailed description of the four Tree ORAMs discussed and compared in this paper. In Section 5, we present a qualitative analysis and comparison of the four ORAMs and discuss their similarities and differences. In Section 4, we present the most relevant work related to experimental evaluations of ORAMs. In Section 6, we first present the details of our unified experimental platform. Then we provide the experimental results and their analysis. We conclude the paper in Section 7.

2. General Description and Evolution of ORAMs

Probabilistic encryption must be used with ORAMs to insure obliviousness [19]. A block must be encrypted by the user using a probabilistic encryption before being stored in any ORAM. To read or write a block received from the server, the user has to decrypt it using the same probabilistic encryption that was used to encrypt it. Before returning the block to the server, the data block must be reencrypted again.

The trivial ORAM solution is to read and then write every memory location when a single read or write is requested. For storage of N data blocks, a single read or write amounts then to read and rewrite each of the N blocks not to mention the need to decrypt and then reencrypt every block. This tacitly amounts to a prohibitively large and unacceptable overhead.

The first developed ORAM was the Basic Square Root ORAM (Basic-SR) [4, 5]. SR-ORAM replaces the scanning of the entire memory by a relatively small number of blocks, in addition to a regular reshuffling of the blocks in memory [15]. The Basic-SR-ORAM needs $N + 2\sqrt{N}$ blocks of data storage on the server to store the N data blocks. The extra $2\sqrt{N}$ blocks of data storage are divided into two parts. The first $\sqrt{N}$ are used to store dummy data and the second $\sqrt{N}$ to cache the read/written blocks and are called the shelter locations. The Basic-SR-ORAM has a client storage complexity of O(1) and a server storage complexity of O(N). The amortized communication complexity is $O(\sqrt{N} \log N)$ [4].

The second developed ORAM was the hierarchal ORAM [6]. Hierarchal ORAM uses the basic idea of the shelter locations used in SR-ORAM, but rather than having just one level of shelter locations, it uses a hierarchy of shelters called levels. Moreover, the levels in the Hierarchical ORAM are hash tables. To store N blocks, the hierarchal ORAM requires $(\log N) + 1$ levels where Level i is of size 2^i buckets, $1 \leq i \leq (\log N) + 1$. As such, each level of the hierarchy has a different epoch equal to its size. Each bucket is capable of storing $\Theta(\log N)$ blocks. Furthermore, a random hash function is used for each level except the first. The hash function enables the direct lookup of a logical address in the level instead of having to scan the complete level [6]. The Hierarchical ORAM has a client storage complexity of O(1) and a server storage complexity of $O(N \log N)$, while the amortized communication cost is $O(\log^3 N)$. However, the worst-case communication cost is bad and is $O(N \log^2 N)$ [19].

An improvement to the hierarchal ORAM was suggested by using Cuckoo hashing [20]. Using Cuckoo hashing in the hierarchal ORAM allows for the removal of the overhead associated with having buckets of size $\Theta(\log N)$ [21]. This results in an improvement of all complexities by a factor of $\log N$ [19]. Further improvements to the Hierarchical ORAM were suggested, such as partial sorting at the client [22], and the use of bloom filters [23, 24] which was later improved in [25, 26].

The third developed ORAM was TP-ORAM [7] which is an improved version of the hierarchal ORAM [7, 19]. Hierarchal ORAM has a bad worst-case communication complexity since the merging and shuffling must be done over many levels. TP-ORAM tries to solve this issue by dividing the server storage into partitions. The number of partitions is $p = \sqrt{N}$, where N is the number of data blocks in the database. Each data block is randomly assigned to a partition. Every partition is viewed as a black box ORAM. When a block needs to be read or written, the corresponding partition is looked up and control is transferred to the black box ORAM of that partition. The black box ORAM has a smaller size than the big ORAM and thus the merging and shuffling happen on levels having smaller sizes. This has a direct impact on the worst-case complexity.

The TP-ORAM has a client storage complexity of O(N) because of the use of the partition map. To reduce the client storage complexity, recursion can be used to store the partition map on the server. This process can reduce the client storage complexity to $O(\sqrt{N})$. The server storage complexity is O(N), while the amortized communication cost is $O(\log N)$ for the nonrecursive version and is $O(\log^2 N)$ for the recursive version. On the other hand, the worst-case complexity for communication is the same for the recursive and the nonrecursive versions and is $O(\sqrt{N})$ [19].

The fourth developed ORAM was Tree ORAM [8]. Tree ORAM builds upon the ideas of the hierarchal ORAM and TP-ORAM [8, 19]. The server storage is viewed as a binary tree rather than a hierarchy of levels; the rationale behind this is to eliminate the need to merge and reshuffle complete levels as in hierarchal ORAM. The merging and shuffling of

Table 1: Notations used for Path ORAM.

N	The total number of buckets and data blocks
L	The height of the tree
Z	The size of each bucket in blocks
Path(x)	The path from leaf node x to the root
Path(x, i)	The bucket at level i on path x
St	The client's local stash
PMap	The position map stored on the client
x:=PMap[a]	Block a is assigned to leaf node x
RBuck(i)	Read a complete bucket i
WBuck(i, y)	Write bucket i with the contents from y
URand	Uniform Random distribution

complete levels have a negative impact on the performance of the ORAM. Nodes in the Tree ORAM are called buckets.

The client in Tree ORAM stores a leaf map. The leaf map stores, for every block, the corresponding leaf it is assigned to. When a block is assigned to a leaf, it means that the block must reside in one of the buckets on the path from the root to that leaf node. To read/write a block the leaf map has to be inspected to find out the leaf the block is assigned to. Then the complete path has to be read, and the desired block is removed from its bucket. All the buckets are written back but with the desired block written in the root bucket.

As in hierarchal ORAM, a bucket as well as a level can overflow. To solve this issue, two steps are taken. First, an eviction process is done periodically. The eviction process involves only few randomly selected buckets of every level and not all the buckets of a level, which reduces the overhead. Second, the size of each bucket is $\Theta(\log N)$ even though a constant number of blocks is expected to be stored in the bucket. This results in O(N log N) server storage complexity. The client storage complexity is O(N) and can be reduced to $O(\log^2 N)$ by using recursion to store the leaf map in a Tree ORAM on the server. The worst-case communication cost is $O(\log^2 N)$ for the nonrecursive version and $O(\log^3 N)$ for the recursive version [8, 19].

3. Detailed Description of the Considered Tree ORAMs

We are concerned here with the most promising Tree ORAMs which are Path ORAM [9], Ring ORAM [12], XOR Ring ORAM [11, 12], and Onion ORAM [13].

All the four ORAMs use the basic tree structure used in Tree ORAM. However, there is a difference in writing back the data to the tree and the amount of data stored in the tree. Moreover, the first three introduce a small storage at the client side called a stash, while the fourth uses an Additive Homomorphic Encryption (AHE). All Tree ORAM variants need to store a leaf map hereafter called a position map. To store the position map the client storage size has to be $\theta(N)$, where N denotes the total number of nodes (buckets) in the tree. This storage can be reduced by using recursion to store the position map itself as an ORAM on the server. However, we will restrict our discussion to the nonrecursive versions of their algorithms.

3.1. Path ORAM. Path ORAM adopts a binary tree structure to store the data at the server. The nodes in the binary tree are called buckets just like in Tree ORAM. However, Path ORAM tries to solve the logarithmic storage complexity of the Tree ORAM by fixing the number of blocks a bucket can hold to a constant number, say z [9]. In Path ORAM [9, 16, 27], each block is mapped to a leaf of the binary tree. This mapping is stored in a leaf map at the client called the position map. When a block is mapped to a leaf it means that it can reside in one of the buckets on the path from the root of the tree to that leaf. The client in Path ORAM also has a local buffer called a stash to hold the blocks read from the server. As such the stash is of capacity at least $z \log(N)$, where N is the number of nodes and the number of data blocks that can be stored in the binary tree. The main invariant in Path ORAM is that, at any time, a block is either in the tree on the path it has been mapped to or in the stash waiting to be mapped to one of the buckets of its designated path. To read/write a block, all the buckets on the path from the root to the mapped leaf are read into the stash. The requested block is then remapped to another uniformly randomly chosen leaf and the position map is updated accordingly. Now, using a bottom up greedy filling strategy (i.e., from the leaf to the root) the blocks read in the stash are rerewriting into the same path. This may also include the requested block as well as any additional blocks already in the stash in a way that the Path ORAM invariant is maintained [9]. The notation used for Path ORAM is shown in Table 1 and the Path data access algorithm is shown in Algorithm 1.

3.2. Ring ORAM. Path ORAM might be considered slow as it must read a whole path from the root to the designated leaf into the stash and then overwrite that same path with blocks from the stash. Ring ORAM tries to combine the benefits of TP-ORAM and Path ORAM and can be parameterized for either small or large client storage [12]. The benefit of TP-ORAM is that it only reads the desired block and returns it to the client unlike Path ORAM that must read a complete path. On the other hand, Path ORAM benefit is that it only requires small client storage unlike TP-ORAM.

Input: operation,a,data*
1: $x \longleftarrow PMap[a]$
2: $PMap[a] \longleftarrow URand(0 \ldots 2^L - 1)$
3: **for** $i = 0$ to L **do**
4: $St \longleftarrow (St \cup RBuck(Path(x, i))$
5: **end for**
6: **if** $(operation = write)$ **then**
7: $St \longleftarrow (St - (a, data)) \cup (a, data^*)$
8: **end if**
9: **for** $i = L$ downto 0 **do**
10: $St \longleftarrow St \cup RBuck(Path(x, i))$
11: $St' \longleftarrow (a', data) \in St : Path(x, i) = Path(PMap[a'], i)$
12: $St' \longleftarrow Min(|St'|, Z)$ blocks from St‘
13: $St \longleftarrow St - St'$
14: $WBuck(Path(x, i), St')$
15: **end for**
16: **return** data

Algorithm 1: Algorithm for data access of Path ORAM.

Ring ORAM uses the same tree structure as Path ORAM [11, 18] and has a position map and a stash at the client. However, there are several differences with Path ORAM. The buckets in Ring ORAM store z real blocks, s dummy blocks, and some meta-data. The meta-data in each bucket indicate the addresses of the blocks stored within the bucket as well as a validity flag for each block. When a block is read, its validity flag is set to false. Furthermore, each bucket has an epoch equal to the number s of dummy blocks. To keep track of the epoch, each bucket has a counter also stored within the meta-data to count the number of blocks read so far. When this counter reaches s, meaning the epoch of the bucket has ended, the bucket needs to be reshuffled. The shuffling is done using a different random permutation for each epoch. The permutation is also stored within the meta-data to ease reaching the blocks. The meta-data are encrypted except for the count and validity fields that do not need to be encrypted.

To read/write a block the position map is checked first to find out the leaf the block is mapped to (the designated path). The client then successively reads the meta-data of all the buckets along the designated path. From the meta-data of a bucket, the client knows whether the requested block is within this bucket. If it is and it has not been read before, the client asks for it. Otherwise, the client asks instead for a dummy block from this bucket. As a result, in Ring ORAM, we do not read complete buckets but only the meta-data and a single block from each bucket on the path. After reading a path, the path is not overwritten with contents from the stash, only the meta-data of the path are written back after performing the appropriate changes. The requested block is assigned to a new random leaf and the position map is accordingly updated.

There must then be a separate eviction process in Ring ORAM to empty the stash and fill up the tree. The eviction process is scheduled statically and is controlled by a public parameter named τ. Moreover, the paths selected to be evicted are chosen using a reverse lexicographic order. When a path is selected for eviction, z unread blocks are read from each bucket on the path and written to the stash. Then the buckets on the path are filled from the stash using a bottom up greedy filling strategy just like Path ORAM. There are two invariants for Ring ORAM. The first invariant is the same as in Path ORAM; that is, each block is mapped to a random leaf and the block resides either in the path of the leaf or within the stash. The second invariant concerns the physical positions of the z real blocks and the s dummy blocks in every bucket in the tree; they are randomly permuted [18]. This second invariant was not present in Path ORAM and is necessary in Ring ORAM for obliviousness since after a read/write operation the path is not overwritten. To maintain this second invariant, a reshuffling function is introduced in Ring ORAM algorithm. The reshuffle algorithm is called whenever a bucket is read s times. When a bucket is read s times, it must be shuffled and then written back to the tree in the server. After every read/write operation all the buckets on the path need to be checked if they need a reshuffle. After shuffling a bucket all the valid flags are set to true and the counter field in the meta-data is set to zero.

We shall show that this reshuffling consumes a lot of bandwidths and is called many times which degrades the overall performance of Ring ORAM. Recall that Ring ORAM is meant to be an improvement over Path ORAM [11, 12]. We experimentally show that this is not the case. The data access algorithm of Ring ORAM is more involved than that of Path ORAM and is displayed in Algorithms 2, 3, 4, and 5. The notations used for Ring ORAM are displayed in Table 2.

3.3. XOR Ring ORAM. XOR Ring ORAM takes advantage of the fact that the blocks returned to the client are all dummy blocks except the desired block. All dummy blocks, in XOR Ring ORAM, are assumed to have the same plaintext, which is known by the client. When the server reads a path, it

Table 2: Notations used for Ring ORAM.

N	The total number of leaves
L	The height of the tree
Z	The number of real blocks in each bucket
S	The number of dummy blocks in each bucket
Path(x)	The path from leaf node x to the root
Path(x, i)	The bucket at level i on path x
GetPath(x,a)	Read the path x looking for block with address a
St	The client's local stash
PMap	The position map stored on the client
x:=PMap[a]	Block a is assigned to leaf node x
EvPath()	Evict path
ShuffPath(x)	Shuffle the buckets on path x
offs	The offset of a block stored in a bucket
block[offs]	The block stored in the bucket at offset offs
RBuck(i)	Read a complete bucket i
WBuck(i, y)	Write bucket i with the contents from y
URand	Uniform Random distribution
GetBOffs(Path(x,i),a)	Gets the block offset of the block with address a on path x on level i.

Input: operation,a,data*
Public variable: counter
$x \longleftarrow PMap[a]$
$PMap[a] \longleftarrow URand(0 \dots 2^L - 1)$
$data \longleftarrow GetPath(x, a)$
if (data is empty) **then**
  $data \longleftarrow Get$ data a from st
  Remove data a from st
end if
if $(operation = read)$ **then**
  give client the data
end if
if $(operation = write)$ **then**
  $data \longleftarrow data'$
end if
$St \longleftarrow St \cup (a, data)$
$counter \longleftarrow counter + 1 \bmod A$
if $(counter = 0)$ **then**
  EvPath()
end if
ShuffPath(x)

Algorithm 2: Algorithm for data access of Ring ORAM.

$data \longleftarrow null$
for $i = 0$ to L **do**
  offs ⟵GetBOffs(Path(x,i),a)
  $data' \longleftarrow block$[offs]
  set the valids flag of the read block to false
  if ($data'$ is not empty) **then**
    $data \longleftarrow data'$
  end if
  $Path(x, i).count \longleftarrow Path(x, i).count + 1$
end for
return data

Algorithm 3: Algorithm for GetPath(x,a).

Public variable: K initialized to zero
$l \longleftarrow K \bmod 2^L$
$K \longleftarrow K + 1$
for $i = 0$ to L **do**
  $St \longleftarrow (St \cup RBuck(Path(l, i))$
end for
for $i = L$ downto 0 **do**
  $WBuck(Path(l, i), St)$
  $Path(l, i).count \longleftarrow zero$
end for

Algorithm 4: Algorithm for EvPath.

first performs an XOR operation on all the blocks and then only sends the resulting block. When the client receives the Xored cipher block, it first performs an XOR operation on the ciphertext of the known dummy block and the received block from the server which results in the desired block [12]. The algorithm of XOR Ring ORAM is displayed in Algorithms 6 and 7. The algorithm is the same as the algorithm for Ring ORAM except for the GetPath function which we renamed GetXORPath. Moreover, the notation used for XOR Ring ORAM is the same notation used for Ring ORAM with the addition of three notations. The first concerns an array of offsets (Aoffs) filled by the client and sent to the server, to indicate the blocks that should be fetched by the server and then Xored. The second notation concerns the function that takes the array of offsets (Aoffs) and returns to the client the unique block resulting from the XOR performed on the

```
1: for i = 0 to L do
2:     if (Path(x, i).count = zero) then
3:         St ⟵ St ∪ (a, data)
4:         WBuck(Path(x, i), St)
5:         Path(x, i).count ⟵ zero
6:     end if
7: end for
```

Algorithm 5: Algorithm for ShuffPath.

```
Input: operation,a,data*
       Public variable: counter
1: x ⟵ PMap[a]
2: PMap[a] ⟵ URand(0 ... 2^L − 1)
3: data ⟵ GetXORPath(x, a)
4: if (data is empty) then
5:     data ⟵ Get data a from st
6:     Remove data a from st
7: end if
8: if (operation = read) then
9:     give client the data
10: end if
11: if (operation = write) then
12:     data ⟵ data'
13: end if
14: St ⟵ St ∪ (a, data)
15: counter ⟵ counter + 1 mod A
16: if (counter = 0) then
17:     EvPath()
18: end if
19: ShuffPath(x)
```

Algorithm 6: Algorithm for data access of XOR Ring ORAM.

```
1: data ⟵ null
2: for i = 0 to L do
3:     Aoffs[i] ⟵GetBOffs(Path(x,i),a)
4:     set the valids flag of the read block to false
5:     increment the counter field of the meta-data
6: end for
7: data ⟵GetXOR(Aoffs)
8: for i = 0 to L − 1 do
9:     data⟵ data XOR EncDummy
10: end for
11: return data
```

Algorithm 7: Algorithm for GetXORPath(x,a).

blocks with offsets in (Aoffs). The third notation concerns (EncDummy) representing an encrypted dummy block.

3.4. Onion ORAM. The Onion ORAM uses the standard tree structure of Tree ORAM. The tree is composed of buckets and each bucket contains z blocks. Meta-data are also stored on the server for each bucket. These meta-data are encrypted with a semantically secure encryption algorithm. The meta-data store the addresses of the blocks in the corresponding bucket and the path that each block is assigned to. The client in Onion ORAM has a leaf map called the position map to store the path each block is assigned to. However, unlike previous ORAMs, Onion ORAM does not have a stash. As a result, the main invariant in Onion ORAM is that a block must be in one of the buckets along the path it is assigned to. To read/write a block the position map is looked up to find the path the block is mapped to; then the meta-data of the corresponding path are fetched from the server. The client then searches the meta-data for the desired block and constructs a selection vector for each bucket along the path. Each selection vector is composed of z slots set to zero except the one corresponding to the desired block which is set to one. The selection vectors are then encrypted using AHE and sent to the server. The server performs an AFH selection operation using the received selection vectors which amounts to the desired block encrypted with AHE. The client upon the reception of the result performs the necessary AHE decryptions to obtain the plaintext of the block. The client randomly assigns to this block a new path, updates accordingly the position map, updates the address of this block to that of a dummy block, and inserts this address in the meta-data of the root bucket. Finally, the client reencrypts the block using AHE and inserts it in the root bucket [13, 18].

We note that AHE can be performed more than once; for each application an extra layer of encryption is added, hence the name onion (i.e., like the layers of an onion). The first application of the AFH is on the plaintext belonging to the plaintext space donated by L_1, which transforms the plaintext into a ciphertext that belongs to ciphertext space donated by L_2. The ciphertext belonging to L_2 may be further encrypted using AHE and consequently transformed into ciphertext space L_3. This process can go on as long as needed. However, to decrypt a ciphertext and obtain the plaintext an equal number of decryptions is needed; that is, a ciphertext belonging to ciphertext space L_3 requires two decryptions.

We also note that, for the proper operation of the selection operation, the selection vector should have one additional encryption layer than the blocks to which it is going to be applied. Moreover, all the blocks on which the selection operation is applied should have the same number of layers of encryption. As a result, the server must first bring all the designated blocks to the same number of layers of AHE encryption as well as bring the selection vector to one higher number of encryption. The resulting encrypted block from the selection operation will have the same number of encryption layers as that of the select vector. That is, the selection operation tacitly adds one layer of encryption to the selected block.

To prevent the number of layers of encryption from being incremented indefinitely, the block written back to the root should be encrypted with only one layer of encryption. In addition, the blocks at the leaf level of the tree need to be periodically stripped off their layers of encryption and reencrypted under only one layer of encryption. This process is an integral part of the eviction algorithm used in Onion ORAM. Onion ORAM also needs an eviction

algorithm; otherwise the buckets would overflow. Eviction in Onion ORAM is then performed periodically for every τ read/write operations. τ denotes hereafter the rate of eviction. The path to be evicted is selected according to the reverse lexicographical order of the paths in the tree. Onion ORAM introduces a new eviction strategy called the triplet eviction. Like other ORAMs, the eviction procedure starts with the root of the selected path down to the leaf. For each bucket on the designated path, the triplet eviction considers its two children nodes (buckets).

One of the children is then on the eviction path and called the child bucket and the other child not on the eviction path is called the sibling bucket. The bucket to be evicted needs to be emptied completely into its two children. The eviction process proceeds as follows: the blocks of the parent node are scanned sequentially where each block is either (randomly) written to the child bucket or written to the sibling bucket. To preserve obliviousness, a dummy block is also written to the other child/sibling. This triplet eviction is performed by the server as instructed by the client. For the client to be able to instruct the server to perform the triplet eviction for a certain path, the client needs to download the meta-data of the evicted path and all the meta-data of all the children of the buckets in the path. Then the client scans the meta-data starting with the root and its two children and constructs a selection vector for each slot to be evicted to the two child buckets based on the triplet eviction procedure. The client then sends these selection vectors to the server which performs a homomorphic select operation using the selection vectors received from the client, the data in the parent bucket, and the data in the two children. The result of each homomorphic select will be a block written in one of the slots in the children [13, 18]. At the end of the eviction, all the blocks in the leaf bucket are sent to the client. The client strips off their layers of encryption and encrypts them under one layer of encryption and sends them back to be saved on the server. The Onion ORAM algorithm is sketched in Algorithms 8, 9, 10, and 11, and the notation used is the same notation used for the previous algorithms.

A further observation about Onion ORAM concerns the space expansion associated with using AHE and its resulting multilayering. The ciphertext resulting from AHE is larger than the plaintext. In the Paillier AHE [28], the ciphertext size is double that of the plaintext. As such, using such an AHE will cause the server storage to grow exponentially. Fortunately, the AHE Damgard-Jurik cryptosystem [29] has an expansion equal to $(1 + 1/s_0)$. The larger is s_0, the closer is the expansion factor to 1. We may deliberately put s_0 to a large value. However, the induced cost to this saving in ciphertext expansion is a higher computing time to perform the encryption and decryption [30]. The Paillier AHE is actually a special case of the Damgard-Jurik cryptosystem, where s_0 is set to 1.

Input: operation,a,data*
1: $x \longleftarrow PMap[a]$
2: $PMap[a] \longleftarrow URand(0 \dots 2^L - 1)$
3: $data \longleftarrow GetPath(x, a)$
4: **if** ($operation = read$) **then**
5: give client the data
6: **end if**
7: **if** ($operation = write$) **then**
8: $data \longleftarrow data'$
9: **end if**
10: write data to the root
11: EvPath()

Algorithm 8: Algorithm for data access of Onion ORAM.

1: Read all the meta-data on path x
2: Construct the select vector and send to server
3: Update the meta-data and send back to server

Algorithm 9: Algorithm for GetPath(x,a).

1: Public variable: counter and K initialized to Zero
2: $counter \longleftarrow counter + 1 \pmod A$
3: **if** ($counter = 0$) **then**
4: $x \longleftarrow ReverseBits(K)$
5: EvAlPath(x)
6: $K \longleftarrow K + 1 \pmod 2^l$
7: **end if**

Algorithm 10: Algorithm for EvPath().

1: **for** $i = 0$ to $L - 1$ **do**
2: Read whole Path(x,i) and its right and left children
3: Empty blocks in Path(x,i) to appropriate child
4: **end for**

Algorithm 11: Algorithm for EvAlPath(x).

4. Related Work

There have been various extensions and refinements of the above ORAMs. However, these ORAMs have not been thoroughly compared against each other on an experimental basis using the same testbed. Williams et al. [24] were the first to provide an actual implementation of an ORAM scheme. They provided a Java implementation for the hierarchal ORAM with bloom filters. However, their implementation is not open source, yet they did not perform any comparison with other ORAMs.

Chapman [15] provided a survey and compared SR-ORAM and hierarchal ORAM but only on a theoretical basis.

Kushilevitz et al. [31] provided an analysis of the theoretical complexity of the basic and recursive SR-ORAM and many of the Hierarchical ORAM variants. Furthermore, they pointed out a certain weakness in the security of previous schemes that used hashing and proposed a solution to fix it.

Ren et al. [12] introduced Ring and XOR Ring ORAMs and compared their complexities with Path ORAM. They also conducted a case study on secure processors to compare between Path ORAM and Ring ORAM. They showed that Ring and XOR Ring ORAMs achieve better online and overall bandwidth than Path ORAM. The same authors also provided a detailed theoretical analysis of Ring ORAM and compared it to Path ORAM and TP-ORAM [11]. They also performed a second case study to compare Ring ORAM to TP-ORAM and showed that Ring ORAM performs better than TP-ORAM.

Devadas et al. [13] introduced Onion ORAM and provided a theoretical algorithmic analysis and comparison with Path ORAM, Private Information Retrieval (PIR) [32], and Circuit ORAM [33].

Paul Teeuwen [19] provided an excellent comparison between many ORAM constructions. The basic ideas of the different constructions are discussed, and a chronological view is given for the different constructions and how they are related to each other. They also summarized the theoretical asymptotic complexities of each.

Chang et al. [17] also conducted interesting research on SR-ORAM, Hierarchical ORAM, TP-ORAM, Tree ORAM, and Path ORAM. They implemented these ORAMs on a unified platform and provided a deep comparison between them. However, this comparison is based on their theoretical algorithmic complexities instead of their experimental results. Moreover, they did not include the later ORAMS: Ring, XOR Ring, and Onion ORAMs.

5. Qualitative Comparison and Analysis of Tree ORAMs

We now detail our considered four ORAM schemes, discuss some of the most relevant similarities and differences, and present a qualitative comparison taking into account their particularities. We show the differences/similarities regarding server storage, number of blocks in a bucket, use of meta-data, client storage, communication bandwidth, the need for bucket reshuffles, and the need for a separate eviction process.

Ring, XOR Ring, and then Onion ORAMs have been proposed as enhancements of Path and TP ORAMs for much better performances. Ring ORAM tries to combine the benefits of TP-ORAM and Path ORAM and can be parameterized for either a small or large client storage [12]. TP-ORAM [7] only reads the desired block and returns it to the client unlike Path ORAM that must read a complete path of buckets. But Path ORAM requires smaller client storage compared to TP-ORAM. Ring and XOR Ring ORAMs build on Path ORAM and were intended to further improve its required bandwidth by sending a path of blocks instead of a path of buckets (in Path ORAM a bucket contains few, in general 4, blocks). Indeed by adding some dummy blocks in each bucket (i.e., node of the tree) and some meta-data (storing the addresses of the blocks in the corresponding bucket), Ring ORAM constructs in an oblivious manner a path from the root of the tree to the designated leaf containing the real block and all the others are dummy. Moreover, assuming that dummy blocks have the same plain content (but different encrypted contents for obliviousness) and assuming that the client knows such content, XOR Ring is meant to further improve the performance by letting the server apply successive XOR operations on the blocks of the selected path and therefore sends just one block to the client. Authors in [11, 12] showed that the XOR Ring ORAM improves the theoretical performance. We will show that this is not the case as Ring and XOR Ring ORAMs use more server space, need the exchange of meta-data between the client and the server, and require frequent oblivious shuffling. These issues not counted for in their theoretical Big-O complexity, hinder their practical use as compared to the simpler Path ORAM.

The Onion ORAM builds rather on the standard tree structure of the Tree ORAM and not that of Path ORAM like Ring and XOR Ring ORAMs. In Onion ORAM [13], the client based on the meta-data contained along the designated path and received from the server, constructs a selection vector pinpointing the block to select from each bucket along this designated path. The selection vector is then encrypted using Additive Homomorphic Encryption (AHE) and sent to the server. The server uses this selection vector to obliviously select the desired block AHE encrypted and sends it to the client. While it is true that only the desired block is sent to the client and this can be done obliviously using an AHE, several issues regarding essentially the practical performance are not taking care of when we restrict ourselves to the Big-O complexity measure. We will show that Onion ORAM does not outperform the simpler Path ORAM even when we assume a zero processing time AHE encryption (which is far away to be the case as of today's AHE implementation). Using the best AHE known so far, Onion ORAM is still far from being practical.

Asymptotically all four ORAMs require O(N) server storage. However, the number of blocks in a bucket in Ring and XOR Ring ORAMs is bigger than that of Path and Onion ORAMs. Furthermore, Path ORAM is the only one among the four ORAMs that does not require meta-data; all the other three ORAMs require storing meta-data on the server. Regarding client storage all four ORAMs require a position map to be stored on the client side. The position map is of equal size in all four ORAMs. However, the Onion ORAM has an advantage of not using a stash on the client. The communication bandwidth is the same asymptotically for Path, Ring, and XOR Ring and is equal to $O(\log N)$ [11, 12, 18, 19]. The Onion ORAM has a better communication bandwidth than the other three and is O(1) [19]. As for the need for bucket reshuffles and a separate eviction process, Path ORAM does not need both, making the algorithm very simple. On the other hand, Ring and XOR Ring need both, making these algorithms much more complicated. Onion ORAM only needs a separate eviction process and does not need any bucket reshuffling. However, the AHE used in Onion ORAM makes it not practical at all. We shall show that Onion ORAM yields no improvement even if the used AHE is infinitely fast (requiring null computation time); otherwise it is just not practical.

The similarities and differences between Path ORAM, Ring ORAM, XOR ORAM, and Onion ORAM are displayed in Table 3.

Table 3: Similarities and differences between Path ORAM, Ring ORAM, XOR ORAM, and Onion ORAM.

	Path ORAM	Ring ORAM	XOR Ring ORAM	Onion ORAM
Server storage size	O(N) [9]	O(N) [18]	O(N) [18]	O(N) [13]
Number of blocks in a bucket	Z	Z + S	Z + S	Z
Use of meta-data	No	Yes	Yes	Yes
Client storage size	Stash size = $O(\log N).\omega(1)$ blocks [9] Position map size = O(N)	Stash size = $O(\log N).\omega(1)$ blocks [11, 12] Position map size =O(N)	Stash size = $O(\log N).\omega(1)$ blocks [11, 12] Position map size = O(N)	No stash [13] Position map size = O(N)
Communication Bandwidth	$O(\log N)$ [9]	$O(\log N)$ [19]	$O(\log N)$ [19]	O(1) [19]
Read all buckets along the path	Yes	No, only read all the meta-data on a path, then read one block from each bucket along the path	No, only read all the meta-data on a path, then read one block from each bucket along the path	No, only read all the meta-data
Number of blocks returned to client after a read or a write operation	$(\log(N) + 1) * Z$	$(\log(N) + 1)$ blocks of meta-data and $\log(N) + 1$ blocks of size B all of which are dummy but one	$\log(N) + 1$ blocks of meta-data and one block of data	$\log(N) + 1$ blocks of meta-data and one block of data
After a read/write operation all buckets along the path are rewritten back	Yes, and may contain new blocks from stash if they satisfy the invariant.	Only the meta-data of all the buckets on the path are written back.	Only the meta-data of all the buckets on the path are written back.	The meta-data of all the buckets along the path are written back and a AHE select operation on the root.
Bucket reshuffling	No	Yes, when the bucket epoch ends	Yes, when the bucket epoch ends	No
Separate eviction process	No	Yes	Yes	Yes

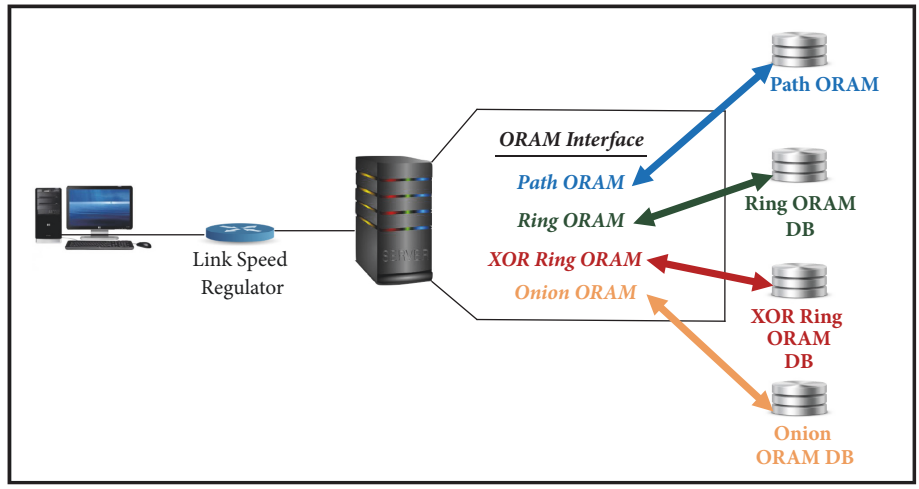

Figure 1: Experimental platform.

6. Experimental Analysis of Tree ORAMs

6.1. Experimental Set Up. The experimental platform is composed of three main components: a server, a client, and an adaptable communication link. The server comprises 8 cores, 1 TB hard disk, and 128 GB of memory. The client is a laptop with Intel core i7-7 500u, up to 3.5 GHz with 1 TB hard disk and 8 GB of memory. The switch used to connect the client and the server has a base speed of 1.09 Gbps. Both the server and the client are running Ubuntu 14.04. The cloud server hosts a MongoDB instance as the outsourced cloud database and storage. Furthermore, the switch can throttle the available bandwidth so that we can control the communication speed between the server and the client. The platform is sketched on Figure 1.

We implemented the nonrecursive versions of Path, Ring, XOR Ring, and Onion ORAM using the C++ programming language to evaluate, compare, and position these ORAMs. In all our experiments, we assumed that blocks are requested according to a Poisson distribution with an interarrival time of one minute. However, for each request, the real block is chosen randomly among all real blocks in the ORAM. We used the AES/CFB for encryption with a key length of 128 bits. The block size is set to 4096 Bytes. Moreover, we set the number Z of real blocks in a bucket to 4 and the number s of dummy blocks in a bucket to 2. In addition, we used an eviction rate $\tau = 3$. The ORAMs have each N = 2047 nodes or buckets in the binary tree, and hence the number of leaves is 1024. We used 4096 real data blocks. We ran the experiments until 600 requests were issued by the user. Seventy-five percent of the requests were reads, and twenty-five percent were writes. We measured and timed many operations during the execution of the four ORAMs.

6.2. Experimental Results, Assessment and Analysis. We run several experiments to collect the request average response time, the system latency, the average number of rounds (the number of read/write requests) a block waits in the stash before being returned to the server, the average time a block waits in the stash, and the average number of blocks waiting in the stash after each read/write. The request average response time is defined as the average time from the instant the user issues a read/write request until the requested data is returned. The system latency is defined as the average time from the instant the user issues a read/write request until the data is returned to the binary tree on the server (i.e., the data has been successfully removed from the stash). The system latency is very important since it measures the time needed before the user can turn off his/her device; otherwise the data is not written on the server and would be lost. We stress here the fact that for Onion ORAM all the AHE operations are assumed to be infinitely fast and take zero processing time. This is adopted to show that even with infinitely fast AHE, Onion ORAM is still outperformed by the much simpler version of Path ORAM. Furthermore, in the next paragraph we shall measure the experimental processing time to perform the AHE using the Damgard-Jurik cryptosystem [29]. For our first experiment, the link speed between the user and the server is set to 1 Gbps. The results are displayed on Table 4.

We observe from Table 4 that Path ORAM has the best response time and the best system latency. In Path ORAM, the stash is empty almost all the time and a read block is returned to the tree on the server either immediately or by the next write path operation. On the other hand, in Ring and XOR Ring ORAMs a read block must wait for the path eviction process to be written back and that if there is space and if the first invariant is satisfied. This waiting results in a significant increase in the system latency of Ring and XOR Ring ORAMs. It is here notable to recall that Ring ORAM was designed [11, 12] to reduce the communication overhead in the first place. That was the rationale behind introducing meta-data in Ring ORAM so that instead of reading a complete path of data buckets, we only read the meta-data of the complete path, and then only one block is read from each bucket along that path. Furthermore, in Ring ORAM, the read path may only be written back on the server when the eviction process is called which is controlled by the constant rate τ. Surprisingly,

Table 4: Comparison between Path, Ring, XOR Ring, and Onion ORAM in terms of average response time, system latency, and the number of rounds and time a block waits in the stash, using 1 Gbps link speed.

ORAM	Path ORAM	Ring ORAM	XOR Ring ORAM	Onion ORAM
Average response time	12.930	103.650	103.260	148.400
System latency	1642.930	563283.450	317973.243	148.400
Average number of rounds a block must wait for in the stash	0.027	9.386	5.298	No stash
Average time a block waits in the stash	1630.000	563179.800	317869.980	No stash
Average number of blocks remaining in stash after a read/write	0.027	75.516	73.384	No stash

Table 5: Processing time to perform the different AHE operations for Onion ORAM using the Damgard-Jurk cryptosystem, in ms.

Operation	Time in ms
Construct select vector for a complete path	46200
Time to perform select operation on a complete path	22660
Time to decrypt the received block under two levels of encryption	478.4
Time to encrypt a data block before writing it back in the root	3464
Time to encrypt triplet eviction vectors for a complete path	720000
Time to perform select operation on a complete path for eviction	329600
Time to strip off encryption layers of leaf bucket and encrypt with a single layer	24384.4
Total AHE time for a single evict operation	1073984.4
Number of evictions for 600 requests	200
Amortized eviction cost over the 600 requests per read/write	357994.8
Total AHE cost for per read/write	430797.2

the claim that Ring ORAM lowers the required bandwidth is not true in practice. Our conducted experiments clearly show that the overall number of exchanged blocks is larger in Ring ORAM than in Path ORAM. This is the opposite of what the authors in [11, 12] claimed and were trying to achieve, not to mention the extra overhead of the meta-data. This increase in the number of blocks exchanged between the server and the user is essentially caused by the reshuffling process needed in Ring ORAM that was not present in path ORAM. Recall that the reshuffling process is needed in Ring ORAM to maintain its second invariant which states that every bucket is permuted randomly and is different from the past and the future writes of a bucket. This invariant requires that whenever s blocks are read from a bucket (i.e., the same as the number of dummy blocks in the bucket), this bucket must be completely transferred to the client then reshuffled and written back. This operation requires a higher exchange of data blocks as well as much more processing time at the client.

The average response time and the system latency for Onion ORAM are the same since it does not use a stash. Moreover, what comes as a surprise is that Path ORAM outperforms Onion ORAM even accounting for all the processing time required for all the needed AHE operations in Onion ORAM.

In our experiments, we have also investigated the time to perform the AHE operations using the Damgard-Jurik cryptosystem with a key size of 1024 bits [29]. We measured the time to construct a selection vector for a complete path, the time to decrypt a received data block with a double layer of encryption since this is the minimum number of layers for any received block by the client, the time to encrypt a data block before writing it back in the root, the time to construct triplet eviction vectors for a complete path, the time to strip off the layers of encryption in the leaf level blocks of an evicted path, the cumulative number of evictions for 600 requests, and the amortized eviction cost for a read/write operation. The total time for performing all the AHE operations for a single read/write request amounts to 430797.2 ms which is huge and has a tremendous impact on the average response time of Onion ORAM. The average response time and the system latency are then 430945.6 ms. The results are displayed in Table 5.

The read bucket, shuffle, and write bucket operations of Ring and XOR Ring ORAMs take a considerable amount of time. Furthermore, the meta-data adopted in Ring and XOR Ring ORAMs introduce an extra overhead to their reading and writing. In addition, the offset of every block that needs to be read must be first calculated using the meta-data, due to the continuous shuffling of the blocks in the buckets. The processing overhead in Ring ORAM and XOR Ring is presented in Table 6.

Table 7 shows, for each of the considered ORAMs, the total number of data blocks sent from the client to the server, those sent from the server to the client, the total number of exchanged data blocks, and the total number of exchanged meta-blocks. As can be noted from this table, the number of blocks sent from the client to the server is the same as the number of blocks sent from the server to the client in Path ORAM. Indeed, in Path ORAM every read/write operation performs a read on the complete path that is reading all the blocks in every bucket on the selected path. Then a complete

Table 6: Processing overhead in Ring and XOR Ring ORAM per read/write in ms.

ORAM	Ring ORAM	XOR Ring ORAM
Average read bucket time per read/write	43.275	41.400
Average reshuffle bucket time per read/write	73.335	69.415
Average write bucket time per read/write	79.005	74.295
Average get offset time per read/write	26.105	26.580
Average read meta time per read/write	43.280	43.765
Average write meta time per read/write	60.210	58.855
Cumulative overhead per read/write	325.210	314.310

Table 7: Total number of data blocks sent from client to server, server to client, and their sum in Path, Ring, and XOR Ring ORAM.

ORAM	Path ORAM	Ring ORAM	XOR Ring ORAM	Onion ORAM
Total number of data blocks sent from client to server	26400	30794	30830	1400
Total number of data blocks sent from server to client	26400	27134	21158	1400
Total number of data blocks transmitted	52800	57928	51988	2800
Total number of meta blocks transmitted	0	23465	23476	25800

write of the same path is performed trying to fill the path from the leaf bucket to the root in a greedy manner. Recall, in our experiments, the number of leaves is set to 1024 which amounts to a tree height equal to 11 (the total number of nodes in the tree is N = 2047). To read/write a complete path (i.e., to read/write 11 buckets where each bucket contains 4 blocks) amounts to read/write $11*4 = 44$ blocks and since we performed 600 request operations, it yields $600*44 = 26400$ blocks. However, in Ring and XOR Ring ORAMs, the number of blocks sent from the client to the server is different from the number of blocks sent from the server to the client. This is due to several reasons. First, when a client reads a path from the server it does not rewrite it back to the server. The read and write path operation necessary to perform the eviction operation happens according to a rate τ. Second when a complete bucket is read from the server, only the $Z = 4$ blocks are read. But, when a bucket is written to the server the client must send $Z + s$ blocks, that is, the complete bucket with its dummy blocks. In our experiment $s = 2$; hence, a complete bucket holds 6 blocks and the client must send 6 blocks to write a bucket. Compared to Path ORAM, Ring ORAMs transmits a larger number of data blocks. This again contradicts the aim of proposing Ring ORAM to lower the bandwidth; the only bandwidth lowered is the online bandwidth, which does not justify all the overhead in the processing time and the overhead of storage on the server needed for the dummy blocks. Ring and XOR Ring ORAMs also exchange a considerable amount of meta-data while Path ORAM does not.

As for Onion ORAM, the number of exchanged data blocks is rather high equal to 2800. This contradicts the sole aim of Onion ORAM not to move other blocks between the server and the client by having the server perform additional AHE processing to just send a unique block per read/write request. 1200 blocks out of the 2800 blocks are necessary for reading and writing the data blocks of the 600 requests, but the question arises as to where did the extra 1600 data blocks come from? It turns out that these extra 1600 exchanged data blocks are the data blocks from the leaf level of evicted paths that had to be stripped off from their encryption layers. Since 200 evictions happened in our experiment (recall that $\tau = 3$) and for each eviction the Z blocks at the leaf bucket had to be sent to the client to be firstly stripped off and then encrypted with only a single layer of encryption and consequently sent back to the server. Furthermore, compared to Path ORAM, Onion ORAM amounts to a total of 25800 exchanged meta-data blocks while Path does not use any meta-data.

Then, we performed experiment on the four ORAMs with the same parameters but varying the link speed. We used the practical following link speeds 512 Kbps, 768 Kbps, 1 Mbps, 1.5 Mbps, 2 Mbps, 3 Mbps, 4 Mbps, and 100 Mbps. Table 8 shows the average response time and the system latency obtained for each speed in addition to the original speed we started with of 1 Gbps. All times are in milliseconds. Figure 2 portrays the average response time as a function of the client-server link speed up to 4 Mbps.

While varying the link speed, Path ORAM clearly outperforms Ring and Onion ORAMs in terms of system latency. Path ORAM also outperforms Ring ORAM and Onion ORAM in terms of average response time. However, for link speeds lower than 1.5 Mpps, the XOR Ring ORAM attains lower values of the average response time which is shown in Figure 2. We notice that the faster the link speed, the more advantageous the Path ORAM against XOR Ring ORAM as its main overhead lies in sending and receiving of a complete path.

We also computed the server storage used up by Path ORAM, Ring ORAM, XOR Ring, and Onion ORAM while varying the number of nodes N as 1024, 4096, 16384, 65536, 262144, 1048576, 4194304, and 16777216. In Onion ORAM, we used a very small ciphertext space expansion of 1.02 as suggested by [29]. We showed the size needed by both of Onion ORAM when AHE is used without a space expansion (i.e., the expansion parameter is equal to 1) and when using AHE with an expansion factor of 1.02. The results are displayed in Table 9 and also portrayed on Figure 3. We observe

Table 8: Average response time and system latency for Path, Ring, XOR Ring, and Onion ORAMs as a function of different link speeds.

ORAM	Link Speed	PATH	RING	XOR RING	ONION w/o AHE	ONION with AHE
Average response time	**512 Kbps**	5907.530	6741.290	4802.480	386.310	431183.510
	768 Kbps	3946.340	4079.590	3544.750	282.450	431079.650
	1 Mbps	3029.430	3173.750	2740.870	240.600	431037.800
	1.5 Mbps	1972.860	2300.560	1857.400	198.010	430995.210
	2 Mbps	1516.150	1781.600	1537.450	190.990	430988.190
	3 Mbps	1011.380	1266.350	1104.024	161.940	430959.140
	4 Mbps	760.077	986.550	861.776	155.100	430952.300
	100 Mbps	39.520	230.930	221.590	150.540	430947.740
	1 Gbps	12.930	103.650	103.260	148.400	430945.600
System latency	**512 Kbps**	7527.530	569921.090	567982.270	386.310	431183.510
	768 Kbps	5566.340	567259.390	566724.550	282.450	431079.650
	1 Mbps	4649.430	566353.550	565920.670	240.600	431037.800
	1.5 Mbps	2372.860	56503.194	5850.930	198.010	430995.210
	2 Mbps	2616.148	564961.400	564797.120	190.990	430988.190
	3 Mbps	1131.380	578646.400	580704	161.940	430959.140
	4 Mbps	960.075	564166.400	564041.600	155.100	430952.300
	100 Mbps	940.800	563410.730	700024.140	150.540	430947.740
	1 Gbps	1642.930	563283.450	317973.243	148.400	430945.600

Table 9: Server storage size for Path, Ring and XOR Ring ORAMs the numbers are in M bytes.

N	Path	Ring	XOR Ring	Onion w/o AHE	Onion with AHE
1024	33.54	50.80	50.80	33.77	40.90
4096	134.20	203.27	203.27	135.12	170.27
16384	536.85	813.15	813.15	540.52	708.65
65536	2147.47	3252.66	3252.66	2162.15	2963.83
262144	8589.92	13010.71	13010.71	8648.64	12331.96
1048576	34359.72	52042.90	52042.90	34594.60	51311.27
4194304	137438.93	208171.67	208171.67	138378.46	213499.06
16777216	549755.80	832686.76	832686.76	553513.90	888345.90

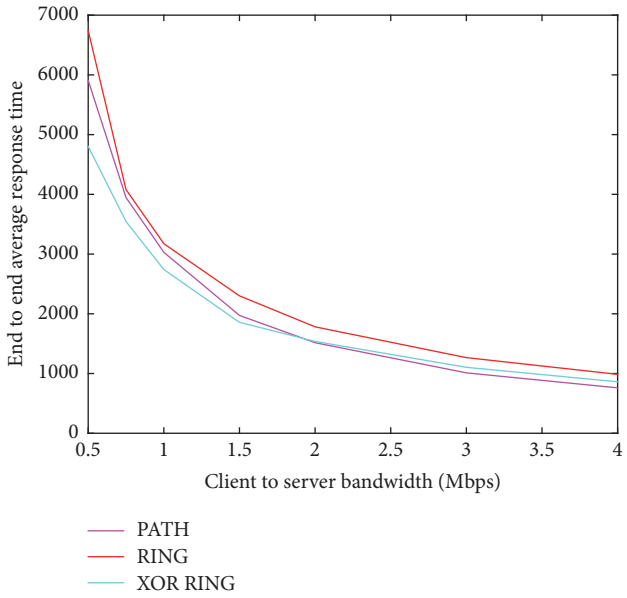

Figure 2: Average response time as a function of the link speed.

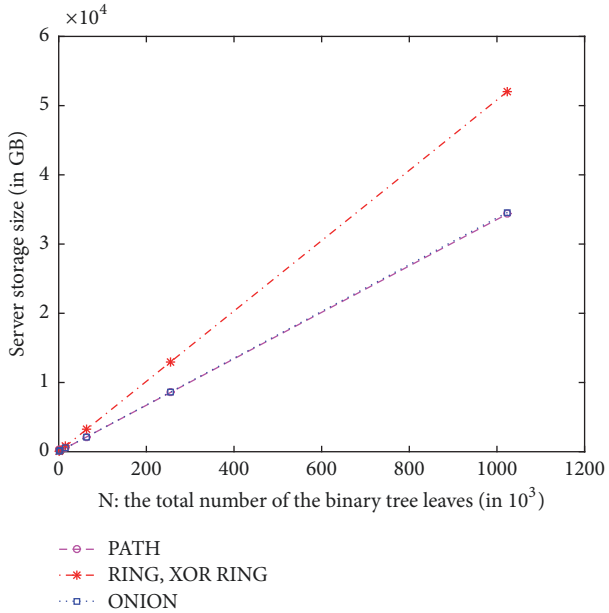

Figure 3: Required server storage size as a function of N.

from this table that Path ORAM is much more efficient in terms of server storage. The server storage consumed by Path ORAM is around 40% less than that consumed by Ring and XOR Ring ORAMs. The difference in the required server storage between Path and Onion ORAM for small values of N is limited given that the ciphertext size expansion is properly selected. However when N gets larger, the required storage of Path ORAM becomes notably lower than that of Onion ORAM. This is essentially due to the meta-data used in Onion ORAM. Figure 3 shows that the storage usage of Onion ORAM is just underneath that of Path but very close.

7. Conclusion

ORAM has become an important component of modern secure outsourced storage. However, this tacitly comes at the expense of much more required bandwidth and storage. We explored and assessed the relevance, efficiency, and practicality of four recent ORAM constructions Path, Ring, XOR Ring, and Onion ORAMs using the same experimental platform. Although Ring and Onion ORAMs are meant as improvements of Path ORAM as they limit the number of exchanged data blocks between the server and the user, their additional requirements in terms of eviction and reshuffling hinder their practical deployment. We experimentally showed that Path ORAM outperforms the three others in terms of many efficiency and performance metrics.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

We would like to thank King Abdulaziz City for Science and Technology (KACST) for funding this research with research (Grant no. 1-17-00-001-0004).

References

[1] T. T. W. Group, *The treacherous 12: Cloud computing top threats*, Cloud Security Alliance, 2016.

[2] B. Li, Y. Huang, Z. Liu, J. Li, Z. Tian, and S. Yiu, "HybridORAM: Practical oblivious cloud storage with constant bandwidth," *Information Sciences*, 2018.

[3] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," *Ndss*, vol. 20, p. 12, 2012.

[4] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *Proceedings of the the nineteenth annual ACM conference*, pp. 182–194, NY, USA, 1987.

[5] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *Proceedings of the the twenty-second annual ACM symposium*, pp. 514–523, Baltimore, Maryland, United States, May 1990.

[6] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[7] E. Stefanov, E. Shi, and D. Song, *Towards practical oblivious RAM*, 2011, http://arxiv.org/abs/1106.3652.

[8] E. Shi, T. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o((logn)3) worst-case cost," in *Advances in cryptology – ASIACRYPT*, vol. 7073 of *Lecture Notes in Comput. Sci.*, pp. 197–214, Springer, Heidelberg, 2011.

[9] E. Stefanov, M. van Dijk, E. Shi et al., "Path oram: An extremely simple oblivious ram protocol," in *Proceedings of the in Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security, ser. CCS 13*, pp. 299–310, New York, NY, USA, 2013, http://doi.acm.org/10.1145/2508859.2516660.

[10] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi, "SCORAM," in *Proceedings of the the 2014 ACM SIGSAC Conference*, pp. 191–202, Scottsdale, Arizona, USA, November 2014.

[11] L. Ren, C. W. Fletcher, A. Kwon et al., "Ring oram: Closing the gap between small and large client storage oblivious ram," *Cryptology ePrint Archive*, vol. 2014, p. 997, 2014.

[12] L. Ren, C. Fletcher, A. Kwon et al., "Constants count: Practical improvements to oblivious ram," in *IACR Cryptology ePrint Archive 997, Tech. Rep*, 2014.

[13] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion oram: A constant bandwidth blowup oblivious ram," in *Theory of Cryptography Conference*, vol. 9563 of *Lecture Notes in Comput. Sci.*, pp. 145–174, Springer, Berlin, 2016.

[14] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing Oblivious Access on Cloud Storage," in *Proceedings of the the 22nd ACM SIGSAC Conference*, pp. 837–849, Denver, Colorado, USA, October 2015.

[15] E. Chapman, "A Survey and Analysis of Solutions to the Oblivious Memory Access Problem," Tech. Rep., Portland State University, 2000.

[16] K. Al-Saleh and A. Belghith, "Locality Aware Path ORAM: Implementation, Experimentation and Analytical Modeling," *Computers*, vol. 7, no. 4, 2018. https://doi.org/10.3390/computers7040056.

[17] Z. Chang, D. Xie, and F. Li, "Oblivious RAM: A dissection and experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1113–1124, 2016.

[18] C. W. Fletcher, *Oblivious ram: from theory to practice [Ph.D. thesis]*, Massachusetts Institute of Technology, 2016.

[19] P. Teeuwen, "Evolution of oblivious ram schemes," Tech. Rep., Technische Universiteit Eindhoven, 2015.

[20] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms in Cognition, Informatics and Logic*, vol. 51, no. 2, pp. 122–144, 2004.

[21] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *Advances in cryptology—CRYPTO*, vol. 6223 of *Lecture Notes in Comput. Sci.*, pp. 502–519, Springer, Berlin, 2010.

[22] P. Williams and R. Sion, "Usable pir," *in NDSS*, pp. 139–152, 2008.

[23] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[24] P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: practical access pattern privacy and correctness on untrusted storage," in *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS'08*, pp. 139–148, October 2008.

[25] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *Proceedings of the the 2012 ACM conference*, p. 293, Raleigh, North Carolina, USA, October 2012.

[26] P. Williams and R. Sion, "Access privacy and correctness on untrusted storage," *ACM Transactions on Information and System Security*, vol. 16, no. 3, pp. 1–29, 2013.

[27] E. Stefanov, M. Van Dijk, E. Shi et al., "Path ORAM: an extremely simple oblivious RAM protocol," *Journal of the ACM*, vol. 65, no. 4, pp. 1–18, 2018 (Croatian).

[28] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology—EUROCRYPT*, J. Stern, Ed., vol. 1592, pp. 223–238, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[29] I. Damgård and M. Jurik, "A Generalisation, a Simpli.cation and Some Applications of Paillier's Probabilistic Public-Key System," in *Public Key Cryptography*, vol. 1992 of *Lecture Notes in Computer Science*, pp. 119–136, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[30] I. Damgård, M. Jurik, and J. B. Nielsen, "A generalization of Paillier's public-key system with applications to electronic voting," *International Journal of Information Security*, vol. 9, no. 6, pp. 371–385, 2010.

[31] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the security of hash-based oblivious ram and a new balancing scheme," in *Proceedings of the the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms, ser. SODA '12*, pp. 143–156, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2012.

[32] T. Mayberry, E. Blass, and A. H. Chan, "Efficient Private File Retrieval by Combining ORAM and PIR," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA.

[33] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 850–861, Denver, Colorado, USA, October 2015.