An Oblivious RAM with Sub-logarithmic Bandwidth Blowup

Kartik Nayak
kartik@cs.umd.edu

Ling Ren
renling@mit.edu

Ittai Abraham
iabraham@vmware.com

Benny Pinkas
benny@pinkas.net

1 Introduction

Oblivious RAM (also known as ORAM) is a cryptographic primitive that allows a trusted client to outsource storage to an untrusted server while hiding the client's memory access patterns to the server. It was first proposed by Goldreich and Ostrovsky [6, 7]. Since the initial theoretical work three decades ago, there has been a lot of effort to improve ORAMs [14, 16, 13, 9, 1, 4, 11, 10].

The standard ORAM model assumes the server to be a simple storage device that only supports "read" and "write" operations. In this model, numerous works have improved the bandwidth blowup (or bandwidth overhead) — the amount of communication between the client and the server relative to an insecure scenario that does not protect access patterns — from $O(\log^3 N)$ to $O(\log N)$ where N is the number of logical data blocks. But none could achieve sub-logarithmic bandwidth blowup so far. In this sense, though not provably insurmountable [2], the $\Omega(\log N)$ bandwidth blowup barrier does seem hard to surpass.

To this end, a line of work deviates from the standard model and allows the server to do computation. The most recent works in this line achieved $O(1)$ bandwidth blowup [1, 4]. But this improvement in bandwidth comes with a huge cost in the amount of server computation. In both Apon et al. [1] and Devadas et al. [4], the idea is to have the server run the ORAM algorithm using homomorphic encryption (fully homomorphic and additively homomorphic, respectively) with little client intervention. In practice, in both schemes, the time for server computation will far exceed the time for server-client communication and become the new bottleneck.

Another line of work assumes the existence of two non-colluding servers [12, 8, 15] with very simple server computation (e.g., XOR) or no server computation. But these constructions have been unable to surpass the $\Omega(\log N)$ bandwidth blowup barrier.

The state of the art leaves the following natural question:

Can we construct a sub-logarithmic Oblivious RAM without expensive computation?

In this paper, we positively answer the above question with a concrete and provably secure construction. Our construction relies on two non-colluding servers but very simple computation (XOR) by the servers.

There exist works which claim to achieve $O(1)$ bandwidth blowup without using expensive computation such as homomorphic encryption [11, 10]. However, based on the best of our understanding of these works, we are concerned about the security of these schemes. We are also aware of the existence of independent and concurrent work [18] that seemingly achieve results similar to ours.

1.1 Overview of Our Approach

On a high level, an ORAM access has two phases. The first phase, called *retrieval*, fetches and possibly updates the data block requested by the client. The second phase, called *eviction*, reshuffles some data blocks on the server. The most efficient ORAM constructions [16, 17, 13, 4] to date are all based on binary trees. The bandwidth overhead on retrieval and eviction are both $\Theta(\log N)$ due to the tree height.

Table 1: **Comparison with existing Oblivious RAM schemes.**

Construction	B/W blowup	Client Storage	Server Computation	#Servers
Path ORAM [16]	$8 \log N$	$O(\lambda)$	-	1
Ring ORAM [13]	$2.5 \log N$	$O(\lambda)$	XOR	1
Onion ORAM [4]	$O(1)$	$O(1)$	Homomorphic encryption	1
This work	$4 \log_d N$	$O(d\lambda)$	XOR	2
This work ($d = \log N$)	$4 \log N / \log \log N$	$O(\lambda \log N)$		

Our construction uses a tree with larger fan-out $d = \omega(1)$, which decreases the tree height to $O(\log_d N)$. We design a new eviction algorithm whose bandwidth overhead is $O(\log_d N)$. However, it increases the bandwidth overhead on retrieval to $O(d \log_d N)$ in the standard model. We then use two-server private information retrieval (Section 2.1) to reduce the retrieval bandwidth back to $O(1)$ (assuming reasonably large block size). Overall, we obtain a two-server ORAM with $O(\log_d N)$, i.e., sub-logarithmic bandwidth overhead.

2 Preliminaries

2.1 Two-server Private Information Retrieval

Private information retrieval (PIR) allows a user to download one item from an unprocessed database on the server without revealing which item to the server [3]. More formally, a server is holding a list of records $Y = (y_1, y_2, \cdots, y_m)$. A user wants to download y_i without revealing i. These are usually public data records, so the user cannot preprocess them. Two categories of PIR techniques exist – one requires a single server and the other requires the existence on two non-colluding servers.

Single-server PIR protocols have been adopted by Path-PIR [9] and Onion ORAM [4] to improve bandwidth. A downside, however, is that they require the server to perform operations on homomorphic encryption ciphertexts, making server computation the new bottleneck.

PIR in the presence of two non-colluding server is conceptually simple and involves much less computation. Two-server PIR achieves low bandwidth assuming the number of bits in each record $|y_j|$ is much larger than the number of records m, which will be the case for ORAMs. Below we present a simple solution adopted in [12]. The list of records are replicated across the two servers, $\mathcal{S}_1$ and $\mathcal{S}_2$. Suppose, the client is interested in retrieving record i. For the request, the user generates a random bit string of length m, $X = (x_1, x_2, \cdots, x_m)$. It then generates $X' = (x'_1, x'_2, \cdots, x'_m)$ by flipping the i-th bit in X, i.e., $x'_i = \bar{x_i}$ and $x'_j = x_j$ for $j \neq i$. The user then sends X to $\mathcal{S}_1$, and X' to $\mathcal{S}_2$. $\mathcal{S}_1$ computes and responds with $\sum_j x_j \cdot y_j$ while $\mathcal{S}_2$ computes and responds with $\sum_j x'_j \cdot y_j$. Here, the sums represent bit XOR, and $\cdot$ represents bit AND. The user then sums up (XORs) the two responses to obtain $\sum_j (x_j + x'_j) \cdot y_j = y_i$. The above protocol is denoted as $\mathsf{TwoServerPIR}(\mathcal{S}_1, \mathcal{S}_2, Y, i)$.

2.2 Tree-based ORAMs

In a tree-based ORAM, server storage is organized as a binary tree [14]. As mentioned, instead of a binary tree, in this paper we will use a d-ary tree. Hence this brief introduction presents the general case and leaves d as an independent parameter.

Server storage. The tree has $L + 1$ levels, from level 0 to level L. Level i hence has d^i nodes. Let N be the total number of logical blocks in the system. Then L is roughly $\log_d N$. Each node in the tree is called a *bucket* and each bucket contains Z slots for logical blocks. A slot can also be empty — in this case, we say it contains a *dummy* block; otherwise, we say it contains a *real* block. Each block contains B bits. Dummy blocks and real blocks are both encrypted using randomized symmetric encryption.

Metadata. Typically in tree-based ORAMs, each slot should also store some metadata about the block including the block identifier. However, since we focus on the cloud storage scenario where the client has moderate storage, we let the client store all metadata locally. The client tracks locally for each slot whether it contains a real block, and if it does, the block identifier. The client also maintains a position map (PosMap) that maps each real block to a random leaf in the tree. An L-depth d-ary tree has d^L leaves. (The client can easily achieve small client storage by storing per-bucket metadata on the server and applying the standard recursion technique [14].)

Invariant. Tree-based ORAM maintains the invariant that if a block is mapped to a leaf, the block must be somewhere on the path from the root to that leaf. Since a leaf uniquely determines a path and vice versa, we use the two terms interchangeably.

Access. To access a block with logical address a, the client performs the following operations:

1. Look up the local PosMap to figure out the path l it is mapped to.
2. Download and decrypt every block on path p, discarding every block that does not have address a. Due to the invariant, the client is guaranteed to find block a on path l.
3. Remap block a to a new random path l' (i.e., update PosMap). Logically remove block a from its old slot (mark that slot to contain a dummy block in local metadata).
4. Re-encrypt block a and append it to the root bucket.
5. Invoke an eviction procedure to percolate blocks down the tree towards leaves.

The first 4 steps correspond to the retrieval phase, and are the same for many tree-based ORAMs [16, 14, 4]. Tree-based ORAMs differ in their eviction procedures (which also affect the bucket size Z). No existing tree-based ORAM can be efficiently extended to a d-ary tree, so a main contribution of this paper is to construct an eviction scheme that works well with a d-ary tree in Section 3.

3 Scheme

Our construction follows the tree-based ORAM paradigm in Section 2.2. In this section, we present the changes in server storage and the retrieval and eviction strategies to obtain a sub-logarithmic bandwidth overhead. Figure 2 shows the pseudocode of our construction. Figure 1 shows how servers store blocks and an example eviction for our construction.

Server storage. Our construction uses two servers $\mathcal{S}_1$ and $\mathcal{S}_2$, both storing identical information (hence, Figure 1 shows only one tree). Our d-ary tree has $L+1$ levels, numbered from 0 (the root) to L (the leaves). Each node in the tree is called a *bucket*. Each bucket consists of Z slots to store blocks. These slots are equally divided into d *slices*, each of size Z/d. Each leaf bucket has a corresponding auxillary bucket aux that also contain Z slots.

Metadata. Our construction requires metadata similar to the description in Section 2.2, i.e., the PosMap and a block identifier for each slot. As mentioned, we assume the client stores all metadata locally for the cloud storage application, but can easily outsource them to the server (without asymptotically increasing bandwidth overhead).

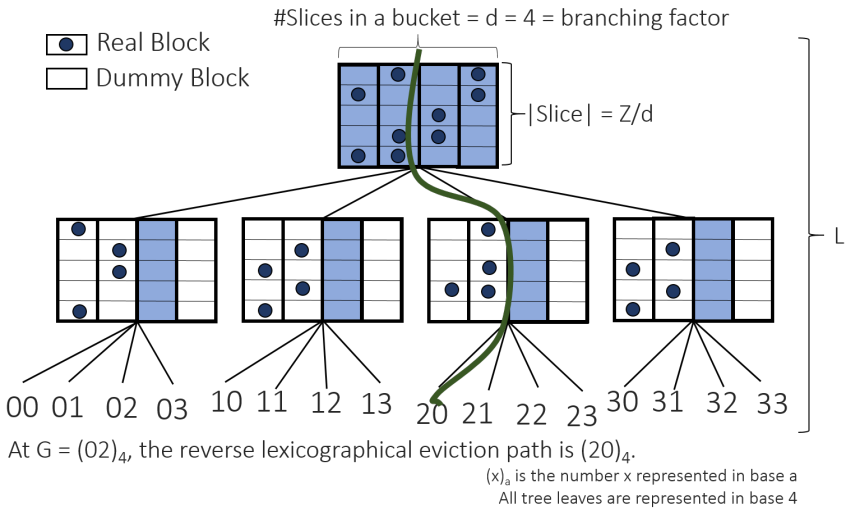

Figure 1: **Example eviction path for a three-level 4-ary tree at $G = 2$ i.e. $G = (02)_4$.** For evicting the root bucket into its children buckets, the client downloads blue colored root bucket and writes to the blue colored slices of its children.

Access. Each client request is represented as a tuple $(a, \mathsf{op}, \mathsf{data}')$ where a is the address of the block, $\mathsf{op} \in \{\mathsf{Read}, \mathsf{Write}\}$ and data' is the data to be written ($\mathsf{data}' = \perp$ if $\mathsf{op} = \mathsf{Read}$). The client maintains a counter cnt for the total number of accesses made so far.

For each access $(a, \mathsf{op}, \mathsf{data}')$, the client does the following:

1. The client looks up position map $\mathsf{PosMap}[a]$ to obtain the leaf l where the block is stored.
2. Let $\mathcal{P}(l)$ represent the path from root to a leaf l. The client retrieves the block identifiers on the path $(\mathsf{id}_1, \mathsf{id}_2 \ldots, \mathsf{id}_{ZL})$ from its local storage. Due to the tree-based ORAM invariant, one of the identifiers on the path will be a. Without loss of generality, assume $\mathsf{id}_i = a$.
3. The client invokes a two-server PIR protocol $\mathsf{TwoServerPIR}(\mathcal{S}_1, \mathcal{S}_2, \mathcal{P}(l), i)$ to retrieve the block with address a.
4. The client updates the data field of the block a to data' if $\mathsf{op} = \mathsf{Write}$. It sets a new leaf l' for the block and updates PosMap. It updates the metadata to remove the block from the tree. It appends the block a to the cnt-th slot of the root bucket.
5. The client increments cnt. If $\mathsf{cnt} = Z/2$, the client resets cnt and performs the eviction procedure described below.

Eviction. The eviction procedure of our construction is a generalization of the eviction procedure of Onion ORAM [4]. It differs from Onion ORAM in the following two ways. First, we apply the eviction scheme on a reverse lexicographical ordering [5] over a d-ary tree instead of a binary tree. Second, when evicting from each bucket along a path, we write to only one slice of each child bucket (instead of writing to the entire child buckets).

As shown in Figure 2, we evict every $Z/2$ accesses along reverse lexicographical ordering of paths. Given that we have a d-ary tree instead of a binary tree, we represent the paths as numbers with base d. We use a counter G to maintain the next path l_e that should be evicted. Eviction is performed for each non-leaf bucket on path $\mathcal{P}(l_e)$. For the k-th bucket from the root, denoted $\mathcal{P}(l_e, k)$, the client first downloads the bucket $\mathcal{P}(l_e, k)$. It then uploads all real blocks to the s-th slice (which will be empty before this operation) of

```
Persistent variables cnt, G initialized to 0
cnt is the number of accesses performed so far since the previous eviction
G is the number of evictions performed so far, represented in base d
Let P(l) represent the path from root to a leaf l.

function Access(a, op, data')
  l' ← UniformRandom(0, d^L − 1)
  l ← PosMap[a]
  PosMap[a] ← l'
  data ← ReadBlock(l, a)
  if op = read then
    return data to client
  else
    data ← data'
  Write data to the cnt-th slot of the root bucket
  cnt := cnt + 1 mod Z/2
  Evict()

function ReadBlock(l, a)
  (id_1, id_2, ..., id_ZL) ← Retrieve block identifiers on P(l)
  Suppose id_i = a
  return TwoServerPIR(S_1, S_2, P(l), i)

function Evict()
  if cnt = 0 then
    l_e ← reverse(G)
    EvictAlongPath(l_e)
    G ← G + 1  mod d^L

function EvictAlongPath(l_e)
  for k ← 0 to L − 1 do
    Let s be the (k + 1)-th digit of G
    Download all blocks in P(l_e, k) and all blocks in the s-th slice of its children
    Move all real blocks in P(l_e, k) to s-th slices of its children
  Read all blocks in P(l_e, L) and its auxiliary bucket P(l_e, aux)
  Move all real blocks from P(l_e, L) to P(l_e, aux)
```

Figure 2: **Access and eviction algorithm for our oblivious RAM construction.**

each of its children where $s = (k+1)$-th digit of G. After this operation, the bucket $\mathcal{P}(l_e, k)$ will be empty. Due to the reverse lexicographical order of eviction paths, $\mathcal{P}(l_e, k)$ will be a child bucket for the next $d-1$ evictions involving it, during each of which the slice being written to will be empty. An example of the data downloaded or uploaded by the client for eviction of the root bucket is shown in Figure 1. For the last level (level L), the client downloads all blocks in the leaf bucket $\mathcal{P}(l_e, L)$ and its the auxiliary bucket $\mathcal{P}(l_e, \mathsf{aux})$. It moves all real blocks to the auxiliary bucket $\mathcal{P}(l_e, \mathsf{aux})$ and uploads both buckets to the server.

3.1 Security Analysis

Parameterization and overflow probability. During an eviction, every real block has an independent choice on which child it is mapped to. Each slice in a child bucket receives $Z/2d$ blocks in expectation. The probability that any bucket overflows (receives more than Z/d blocks) is less than $e^{-\frac{Z}{6d}}$ by a simple Chernoff Bound. To achieve $2^{-\lambda}$ overflow probability, we set $Z = O(d\lambda)$. (Most prior works only guarantee $N^{\omega(1)}$ failure probability, and we can set $Z = \omega(d \log N)$ in that case.) A similar analysis gives the required tree

height L. Each bucket at level L has N/d^L blocks mapped to it in expectation. We want this expectation to be no more than $Z/2$, such that the same Chernoff bound applies. This gives $N/d^L \leq Z/2$, and thus $L = \lceil \log_d(2N/Z) \rceil < \log_d N$.

Obliviousness. Similar to all tree based ORAMs, for each access, client performs the retrieval phase on a random path. The use of PIR hides the location of the requested block on that random path. Eviction is performed on a publicly known reverse lexicographical ordering of paths. Along the eviction path, each bucket and a predetermined slice in each child buckets are downloaded/uploaded. Thus, all client operations observed by the servers are independent of the logical client access patterns.

3.2 Cost Analysis

Access bandwidth. With two-server PIR, the client uploads ZL bits (one bit for each block on the path) to each server, and retrieves one block (B bits) from each server. When $B = \Omega(ZL) = \Omega(d\lambda \log_d N)$, the overhead of an access is $O(1)$. (If the client has small local storage and retrieves all metadata from the servers, the block size needs to be a $\log N$ factor larger to absorb the metadata cost.)

Eviction bandwidth. For each bucket on the path, the client downloads the parent bucket and one slice from each of the d child buckets, which is equivalent to two buckets. Thus, an eviction costs an overhead of $2ZL$ (blocks of bandwidth) and it is performed every $Z/2$ accesses. Thus, the amortized eviction overhead is $4L < 4 \log_d N$.

Client storage. For each bucket on the path, the client moves the blocks in the parent bucket to d slices in the d child buckets. The client storage is thus $O(Z) = O(d\lambda)$ blocks.

4 Conclusion

In this work, we design an Oblivious RAM with sub-logarithmic overhead where the servers only perform XOR operations. We achieve this by using a novel eviction scheme over a d-ary tree to obtain an eviction overhead of $O(\log_d N)$ and using two-server PIR to reduce the cost to retrieve a block to $O(1)$.

References

[1] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In *International Workshop on Public Key Cryptography*, pages 131–148. Springer, 2014.

[2] E. Boyle and M. Naor. Is there an oblivious RAM lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 357–368. ACM, 2016.

[3] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[4] 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*, pages 145–174. Springer, 2016.

[5] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 1–18. Springer, 2013.

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

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

[8] S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography*, pages 377–396. Springer, 2013.

[9] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS*. Citeseer, 2014.

[10] T. Moataz, E.-O. Blass, and T. Mayberry. Chf-oram: a constant communication oram without homomorphic encryption. Cryptology ePrint Archive, Report 2015/1116, 2015.

[11] T. Moataz, T. Mayberry, and E.-O. Blass. Constant communication ORAM with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 862–873. ACM, 2015.

[12] R. Ostrovsky and V. Shoup. Private information storage. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 294–303. ACM, 1997.

[13] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, 2015.

[14] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *International Conference on The Theory and Application of Cryptology and Information Security*, pages 197–214. Springer, 2011.

[15] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 247–258. ACM, 2013.

[16] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[17] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM, 2015.

[18] J. Zhang, Q. Ma, W. Zhang, and D. Qiao. TSKT-ORAM: A two-server k-ary tree ORAM for access pattern protection in cloud storage, 2016.