Asymptotically Tight Bounds for Composing ORAM with PIR

Kartik Nayak[1], Ling Ren[2], Christopher W Fletcher[3], Ittai Abraham[4], and Benny Pinkas[5]

[1] University of Maryland, College Park, MD – kartik@cs.umd.edu,
[2] MIT, Cambridge, MA – renling@mit.edu,
[3] MIT, Cambridge, MA – cwfletch@mit.edu,
[4] VMware Research, Israel – iabraham@vmware.com,
[5] Bar IIan University, Israel – benny@pinkas.net

Abstract. Oblivious RAM (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. The last three decades of research on ORAMs have reduced the bandwidth blowup of ORAM schemes from $O(\sqrt{N})$ to $O(1)$. However, all schemes that achieve a bandwidth blowup smaller than $O(\log N)$ use expensive computations such as homomorphic encryptions. In this paper, we achieve a sub-logarithmic bandwidth blowup of $O(\log_d N)$ (where d is a free parameter) without using expensive computation. We do so by using a d-ary tree and a two server private information retrieval (PIR) protocol based on inexpensive XOR operations at the servers. We also show a $\Omega(\log_{cD} N)$ lower bound on bandwidth blowup in the modified model involving PIR operations. Here, c is the number of blocks stored by the client and D is the number blocks on which PIR operations are performed. Our construction matches this lower bound implying that the lower bound is tight for certain parameter ranges. Finally, we show that C-ORAM (CCS'15) and CHf-ORAM violate the lower bound. Combined with concrete attacks on C-ORAM/CHf-ORAM, we claim that there exist security flaws in these constructions.

1 Introduction

Oblivious RAM is a cryptographic primitive that allows a client to privately outsource storage to an untrusted server without revealing any information about its data accesses, i.e., the server learns nothing about the data or the sequence of addresses accessed. It was first proposed by Goldreich and Ostrovsky [22, 23]. Since the initial theoretical work three decades ago, there has been a lot of effort to improve ORAMs either as a stand-alone primitive [2, 9, 12, 19, 24, 25, 27, 37, 39, 40, 42, 44, 48, 51, 53, 58] or for applications including secure outsourced storage [3, 33, 41, 49, 50, 59], secure processors [15–17, 36, 43, 45, 46] and secure multi-party computation [20, 34, 35, 54, 55, 60].

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 [5], 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 assumes the existence of two non-colluding servers [34, 41, 49] with inexpensive server computation (e.g., XOR) or no server computation. But these constructions have been unable to surpass the $\Omega(\log N)$ bandwidth blowup barrier.

Another line of work allows the server to perform some computation. The most recent works involving server computation achieved $O(1)$ bandwidth blowup [2, 12, 39, 40]. But this improvement in bandwidth comes with a huge cost in the amount of server computation. In both Apon et al. [2] and Devadas et al. [12], the server runs 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.

Thus, the state of the art leaves the following natural question:

Can we construct a sub-logarithmic ORAM without expensive computation?

A recent construction called CHf-ORAM [39] claims to have solved the above challenge by combining ORAM with private information retrieval (PIR). Using four non-colluding servers, CHf-ORAM claims to achieve $O(1)$ bandwidth blowup using simple XOR-based PIR protocols. However, we realized that there exist security flaws in CHf-ORAM and its predecessor C-ORAM [40]. We give two concrete attacks on a slight variant of C-ORAM, highlighting some subtleties that the current C-ORAM proof does not capture.

Private information retrieval (PIR) and Oblivious RAM (ORAM) are two closely related concepts, and they both hide access patterns. In fact, PIR is frequently applied to ORAM constructions to improve bandwidth blowup [37, 39–41, 61]. This led us to ask the following question:

What is the asymptotically optimal bandwidth blowup one can achieve by using PIR in an ORAM construction?

In order to answer this question, we build on the seminal work of Goldreich and Ostrovsky [23] and derive a $\Omega(\log_{cD} N)$ bandwidth lower bound for ORAMs that leverage only PIR and PIR-write on top of the traditional model. Here, c is the number of blocks stored by the client and D is the number of blocks on which PIR/PIR-write operations are performed. C-ORAM and CHf-ORAM violate this lower bound, and thus cannot be secure.

Given the insecurity of C-ORAM and CHf-ORAM, the former question remains open. We then positively answer the former question with a concrete and provably secure construction. Our construction relies on a d-ary ORAM tree and a private information retrieval (PIR) protocol involving two non-colluding servers, where the servers perform simple XOR computations. Our construction achieves $O(\log_d N)$ bandwidth blowup with $c = O(1)$ blocks of client storage and PIR operations on $D = d \cdot \mathsf{polylog}(N)$ blocks. Therefore, it matches the $\Omega(\log_{cD} N)$ lower bound when $d = \Omega(\log N)$, implying that under certain parameter ranges our construction is asymptotically optimal and the lower bound is asymptotically tight.

We remark that there is a concurrent and independent work, MSKT-ORAM, that achieves comparable bandwidth blowup using similar techniques [62].[6] Our construction has several advantages over the concurrent work and we make a more detailed comparison in Section 2.

1.1 Our Contributions

Our contributions in this paper can be summarized as follows:

1. **ORAM with sub-logarithmic bandwidth blowup.** We show a provably secure ORAM construction that achieves a bandwidth blowup of $O(\log_d N)$ (where d is a parameter) using $O(1)$ blocks of client storage. Our construction uses a d-ary tree and a PIR protocol (Section 4).
2. **Extending the Goldreich-Ostrovsky lower bound to allow PIR operation.** For a client storing c blocks of data and performing a PIR on D blocks at a time, we show that the ORAM bandwidth blowup is lower bounded by $\Omega(\log_{cD} N)$ (Section 5). Our construction matches this lower bound implying that the lower bound is tight and that our construction is asymptotically optimal for certain parameter ranges.
3. **Security flaws in prior works.** Using our lower bound and other concrete attacks, we show that the bandwidth blowup claimed by C-ORAM and CHf-ORAM [39, 40] is not achievable (Section 6).

1.2 Overview of Our Construction

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. Many recent ORAM constructions [12, 44, 53, 54] are based on binary trees, in which the bandwidth overhead on retrieval and eviction are both $\Theta(\log N)$ due to the tree height.

[6] The title of that paper claims "constant bandwidth", which would have been immediately ruled out by our lower bound. On a closer look, the bandwidth blowup is actually $O(\log_d N)$. This calls for our lower bound to clear the confusion in this direction.

Table 1: **Comparison with existing Oblivious RAM schemes.** N denotes the number of logical blocks stored by the ORAM. In [44,53], a stash of $\Omega(\lambda)$ blocks ensures a failure probability of $e^{\Omega(-\lambda)}$. For a negligible (in N) failure probability, these works set $\lambda = \omega(\log N)$.

Construction	Bandwidth Blowup	Client Storage	Server Computation	#Servers
Path ORAM [53]	$8 \log N$	$O(\lambda)$	-	1
Ring ORAM [44]	$2.5 \log N$	$O(\lambda)$	XOR	1
Onion ORAM [12]	$O(1)$	$O(1)$	Homomorphic encryption	1
This work (with $d = \log N$)	$4 \log_d N$ $4 \log N / \log \log N$	$O(1)$	XOR	2

Our construction uses a tree with larger fan-out $d = \omega(1)$, which decreases the tree height to $O(\log_d N) = O(\frac{\log N}{\log d})$. Based on a d-ary tree, we design a new eviction algorithm whose bandwidth overhead is $O(\log_d N)$. However, it increases the bandwidth overhead by more than a factor of d on retrieval in the standard model. We then use two-server private information retrieval (Section 3.3) to reduce the retrieval bandwidth to $O(1)$ (assuming moderately large block size). Our basic eviction algorithm also requires $\Omega(d \log N)$ blocks of client storage. We again rely on two-server PIR to reduce the client storage to $O(1)$. Overall, we obtain a two-server ORAM with $O(1)$ client storage and $O(\log_d N)$, i.e., sub-logarithmic bandwidth overhead.

Although our bandwidth blowup decreases with the tree fan-out d, we cannot keep increasing d for free due to block metadata. We discuss the trade-off regarding d in Section 4.4.

2 Related Work

Oblivious RAM was first introduced by Goldreich and Ostrovsky around three decades ago [22, 23]. They proposed two constructions. The latter of the two used a hierarchy of buffers of exponentially increasing size, which was later known as the hierarchical ORAM framework.

They achieved $O(\log^3 N)$ amortized bandwidth blowup under constant client storage, $\Omega(\log N)$ block size and computational security. Their model assumes the server to be a simple storage device that is capable of only "read" and "write" operations. In this model, they show an $\Omega(\log_c N)$ lower bound on the bandwidth blowup, where c is the number of blocks stored by the client.

Follow-up works [24,25,27,57,58] in the hierarchical ORAM framework reduced the bandwidth blowup from $O(\log^3 N)$ to $O(\log^2 N / \log \log N)$. Most of these works also used constant client storage and computational security, and bandwidth blowups are amortized and holds for $\Omega(\log N)$ block size.

Ajtai [1] and Damgård et al. [10] showed ORAM constructions that are statistically secure. This was followed by the statistically secure ORAM construction by Shi et al. [48], who introduced the tree-based paradigm. ORAM constructions in the tree-based paradigm have improved the bandwidth blowup from $O(\log^3 N)$ to $O(\log N)$ [9,19,44,48,53,54]. Circuit ORAM [54] gets very close to the Goldreich-Ostrovsky lower bound, achieving $O(\log N)\omega(1)$ bandwidth blowup with constant client storage for moderately large blocks of size $\Omega(\log^2 N)$. Most tree-based ORAMs achieved statistical access pattern security, and obtained the desired bandwidth blowup in the worst-case instead of an amortized blowup. But the reported bandwidth results only hold for moderately large blocks of size $\Omega(\log^2 N)$ due to the use of the ORAM recursion technique [48].

Many works deviated from the above traditional ORAM model defined by Goldreich and Ostrovsky by introducing multiple non-colluding servers and/or server-side computation. Some of these papers refer to their work as oblivious outsourced storage, but we still refer to them as ORAMs. We review these works below.

ORAMs using multiple non-colluding servers. Constructions in this category so far have not been able to surpass the $\Omega(\log N)$ bandwidth barrier (except CHf-ORAM [39] which we discuss later in this section) [34, 41, 49]. Lu and Ostrovsky [34] achieved a bandwidth blowup of $O(\log N)$. In their scheme, each non-colluding server performs permutations that are hidden to the other server due to which the

Goldreich-Ostrovsky lower bound does not apply. Stefanov and Shi [49] implemented a practical system using two servers and $O(\sqrt{N})$ client storage. Their client storage can be reduced to $O(1)$ using the standard recursion technique [48]. Their construction required $O(1)$ client-to-server bandwidth blowup and $O(\log N)$ server-to-server bandwidth blowup.

ORAMs with server computation. There exist many ORAM schemes that allow the server to do computation on data blocks [2, 11, 12, 20, 37, 40, 44, 50, 51, 57, 59, 61]. Most of these works still require $\Omega(\log N)$ bandwidth blowup, except the following ones. Apon et al. [2] use fully homomorphic encryption to achieve an $O(1)$ bandwidth blowup. However, the large overhead of FHE makes the scheme impractical. Onion ORAM [12] improves upon Apon et al. to achieve an $O(1)$ bandwidth blowup by using only additively homomorphic encryption or somewhat homomorphic encryption. The amount of server computation is significantly reduced (compared to FHE) but is still quite large. In addition, the $O(1)$ bandwidth blowup of Onion ORAM can only be achieved for very large block sizes ($B = \Omega(\log^5 N)$). Both these schemes circumvent the Goldreich-Ostrovsky lower bound by using homomorphic operations on the server side that require little client intervention.

Independent and concurrent work. MSKT-ORAM [62] is an independent and concurrent work that achieves comparable bandwidth blowup using similar techniques, i.e., a d-ary tree and two-server PIR applied to a poly-logarithmic number of blocks. Our construction has several advantages stemming from the following major differences: While we extended the most recent tree-based ORAM, Onion ORAM [12], to a d-ary tree, MSKT-ORAM builds on top of the very first tree-based ORAM by Shi et al. [48] and extends it to a d-ary tree. Thus, MSKT-ORAM does not take advantage of the new techniques invented afterwards, such as small block recursion [52], reverse lexicographical order [19], higher bucket load [44], reduced eviction frequency [44], and an empty bucket invariant [12]. As a result, MSKT-ORAM requires a block size as large as $\Omega(N^\epsilon)$ for some constant ϵ, while we only require blocks of size $\mathsf{polylog}(N)$ bits; MSKT-ORAM has a $\omega(\log N)$ server storage blowup, while our construction has a constant size server storage blowup (Section 4.1); MSKT-ORAM needs a PIR, a physical read and a physical write operation to evict each block, while we can eliminate the need for the physical read due to the empty bucket/slice invariant (cf. Lemma 2 and Section 4.3); MSKT-ORAM also spends at least $2\times$ more bandwidth for both blocks and metadata during eviction, since Shi et al. [48] requires two evictions after every access.

Oblivious RAM lower bound. As mentioned earlier, Goldreich and Ostrovsky presented a lower bound of $\Omega(\log_c N)$ where c is the amount of client storage in blocks. Their lower bound modeled the server as a simple storage device capable of reading and writing blocks. Boyle and Naor revisit the ORAM lower bound to relate it to the size of circuits for sorting [5]. In our work, we extend the lower bound suggested by Goldreich and Ostrovsky to encompass private information retrieval (PIR) as a possible operation performed by the client and obtain a lower bound of $\Omega(\log_{cD}(N))$ in Section 5. Here, c is the number of blocks stored by the client and D is the number of blocks that a PIR is performed on. C-ORAM [40] and CHf-ORAM [39] violate the lower bound and must have security flaws. Boyle and Naor showed that an ORAM lower bound is difficult to obtain in a general model, i.e., if the client is not restricted to a small set of operations.

Other related work. There has also been work to optimize ORAM for the number of rounds of communication [14, 18, 57], response time [11], parallelism [4, 7] and various other parameters [3, 47]. Liu et al. developed compiler techniques to achieve obliviousness with fewer ORAM accesses [30–32]. Some data structures can be made oblivious without using a full ORAM [26, 38, 56].

Private information retrieval. A Private information retrieval (PIR) protocol allows a user to retrieve some data block from a server without revealing the block that was retrieved. It was first introduced by Chor et al. [8]. In our work, we use a simple two server $O(N)$ scheme from [8] to reduce the bandwidth cost of accessing a block.

```
1: function Access(a, op, data)
2:     l ← PosMap[a]
3:     data ← ReadBlock(l, a)
4:     l′ ← UniformRandom(0, d^L − 1)
5:     PosMap[a] ← l′
6:     if op = read then
7:         return data to client
8:     else
9:         data ← data′
10:    Write data to the root bucket
11:    evict()
```

Fig. 1: **Tree-based ORAM data access algorithm.** Here, PosMap is a map from an address a to a leaf l of the tree. ReadBlock(l, a) retrieves a block of data with address a from a path of buckets along leaf l.

3 Preliminaries

3.1 Problem Definition

Consider a scenario where a trusted client wishes to store data to a remote untrusted server while preserving privacy. First, the client can protect confidentiality of the data using standard encryption schemes. However, the access pattern of the client, i.e., the order in which the client accesses the data, can also reveal information. Oblivious RAM algorithms address this problem by *hiding* the data access pattern, i.e., hiding which blocks were read/written from the server. Intuitively, a server observing the resulting *physical* access pattern should not be able to learn anything about the *logical* access pattern of the client.

The ORAM model traditionally treats the server as a simple storage device [22, 23]. But recent works have extended the ORAM model to allow for server computation [2, 12]. Informally, an ORAM that allows server computation can be defined as follows:

Definition 1 (Informal). *Let* $\boldsymbol{y} = ((a_1, \mathsf{op}_1, \mathsf{data}_1), \ldots, (a_t, \mathsf{op}_t, \mathsf{data}_t))$ *be the client's logical data request sequence of length* t*. For each tuple* $\boldsymbol{y}_i = (a_i, \mathsf{op}_i, \mathsf{data}_i)$*,* a_i *represents the logical address of the data block accessed by the client,* $\mathsf{op}_i \in \{\mathsf{Read}, \mathsf{Write}\}$ *and* data_i *is the data being written (*$\mathsf{data}_i = \perp$ *if* $\mathsf{op}_i = \mathsf{Read}$*).*

Let $ORAM(\boldsymbol{y})$ *represent the ORAM client's sequence of interactions with the server. We say an ORAM algorithm is correct if for each access* $i \in [t]$*,* $ORAM(\boldsymbol{y}_i)$ *returns data that is consistent with* $\boldsymbol{y}_i$ *except with* $negl(|\boldsymbol{y}|)$ *probability. We say an ORAM algorithm is secure if for two access patterns* $\boldsymbol{y}$ *and* $\boldsymbol{z}$ *with* $|\boldsymbol{y}| = |\boldsymbol{z}|$*, their access patterns* $ORAM(\boldsymbol{y})$ *and* $ORAM(\boldsymbol{z})$ *are computationally or statistically indistinguishable. Respectively, the ORAM algorithms are called computationally or statistically secure.*

The sequence of interaction $ORAM(\boldsymbol{y})$ may include simple physical read/write requests, PIR requests, or any other complex protocols between the client and the server. For a more formal definition of ORAM with server computation, please refer to Appendix A.

Bandwidth blowup. In order to hide data access patterns, $ORAM(\boldsymbol{y})$ involves more communication between the server and the client than $\boldsymbol{y}$. We define *bandwidth blowup* as the ratio between the amount of communication (measured in bits) in $ORAM(\boldsymbol{y})$ to the amount of communication in $\boldsymbol{y}$. Each unit of logic data accessed by a client is referred to as a *block*. We denote N to be the total number of logic data blocks in the ORAM.

3.2 Tree-based ORAMs

In a tree-based ORAM, server storage is organized as a binary tree [48]. As mentioned in the introduction, instead of a binary tree, in this work we use a d-ary tree. Hence this brief introduction presents the general case and considers d as an independent parameter.

Server storage. We consider d-ary tree with $L+1$ levels, from level 0 to level L. Thus, level i has d^i nodes. Recall that N is the total number of logical blocks stored by the client. 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 stores B bits of information. Dummy blocks and real blocks are both encrypted using randomized symmetric encryption.

Metadata. Aside from the B bits of block data, tree-based ORAMs also store some metadata for each block. The metadata stores the block identifier and whether the block is real or dummy. The client also maintains a position map PosMap that maps each real block to a random leaf in the tree.

In this work, we first assume that the client stores all the metadata locally. We then describe how this metadata can be offloaded to the server (Section 4.3) to achieve $O(1)$ client storage.

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

Access. The pseudo-code for an access algorithm in a tree-based ORAM is described in Figure 1. 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 (line 2).
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. This is done by ReadBlock(l, a) in Figure 1 line 3.
3. Remap block a to a new random path l' (i.e., update PosMap), i.e. logically remove block a from its old position (lines 4 and 5).
4. Re-encrypt block a and append it to the root bucket (line 10, encryption is not shown in the figure).
5. Invoke an eviction procedure to percolate blocks down the tree towards leaves (line 11).

The first four steps correspond to the retrieval phase, and are similar for many tree-based ORAMs [12,48,53]. 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 4.

3.3 Private Information Retrieval

Private information retrieval (PIR) allows a user to download one item from an unprocessed database known to a server, without revealing to the server which item is downloaded [8]. More formally, the setting has a server which is holding a list of records $Y = (y_1, y_2, \cdots, y_m)$, and a user who wants to download record y_i without revealing i to the server. A PIR scheme must enable this operation while requiring communication that is strictly smaller than the size of the database (otherwise, a trivial solution could have the user hide i by simply downloading the entire database.) The database records are usually public data records, or records which are owned by the server, and therefore the user cannot encrypt or otherwise preprocess them.

Two categories of PIR techniques exist – one operates in a setting with a single server and the other requires the existence of two or more non-colluding servers. Single-server PIR protocols, such as [6,21,28], have been adopted by Path-PIR [37] and Onion ORAM [12] to improve bandwidth. A downside, however, is that they require the server to perform operations on homomorphically encrypted ciphertexts [29], making server computation the new bottleneck. PIR in the presence of two or more non-colluding servers is conceptually simpler and involves much less computation — typically only simple XOR operations. It can also guarantee information-theoretic security (whereas it is known that single-server PIR cannot be unconditionally secure).

The original investigation of two-server PIR assumed that each database record is a single bit. The initial PIR paper described a two-server PIR protocol with $O(m^{1/3})$ communication [8] (and more efficient protocols with more than two servers). This result was only recently improved to obtain a communication of $m^{O(\sqrt{\log\log m/\log m})}$ [13].

In the setting of ORAM, we are interested in a PIR of long records, where the number of bits in each record $|y_j|$ is in the same order as the total number of records m. In this case there is a simple PIR protocol that was adopted in [41]: The database of records is replicated across the two servers, $\mathcal{S}_1$ and $\mathcal{S}_2$. Suppose that the user is interested in retrieving record i. For the request, the user generates a random bit string of length m, $X = (x_1, x_2, \cdots, x_m)$. He 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 a bit-wise XOR, and $\cdot$ represents a bit-wise 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 $\mathsf{TwoServerPIR}(\mathcal{S}_1, \mathcal{S}_2, Y, i)$. The communication overhead is $O(|y_j| + m) = O(|y_j|)$.

PIR-writes. Analogous to PIR, we can define PIR-write operations. Our construction in this paper does not use PIR-writes, but we briefly mention it below since our lower bound in Section 5 allows PIR-writes.

A PIR-write operation lets a user update one record among a list of records on a server without revealing to the server which record is updated. Notice that now the records can no longer be public data; they have to be encrypted. Otherwise, the server can trivially figure out which record is updated by comparing their values before and after the update.

4 The Construction

Our construction follows the tree-based ORAM paradigm in the previous section (Section 3.2). 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 3 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 3 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 that can each store one block. Slots from the non-root buckets are equally divided into d *slices*, each of size Z/d. Each leaf bucket has an auxiliary bucket aux that can store Z blocks.

Metadata. Our construction requires metadata similar to the description in Section 3.2, i.e., the position map 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 blowup (Section 4.3).

Initialization. Initially, the ORAM tree at both servers contain all dummy blocks. The position map is initialized to contain independent and uniformly random numbers for each block. The client initializes each block using a logical write operation. If the client issues a logical read operation to a block that has never been initialized, the behavior of the ORAM is undefined.

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 (refer Figure 2):

1. The client looks up position map $\mathsf{PosMap}[a]$ to obtain the leaf l associated with block a (line 6).
2. Let $\mathcal{P}(l)$ represent the path from root to leaf l, and $\mathcal{P}(l, k)$ represent the k-th bucket on $\mathcal{P}(l)$. 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 $\mathsf{id}_i = a$ (lines 21 and 22).
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 (line 23).
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 (lines 8-14).

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 $\mathcal{P}(l)$ be the path from root to leaf l, and $\mathcal{P}(l, k)$ be the k-th bucket on $\mathcal{P}(l)$.

function Access(a, op, data$'$)
  $l \leftarrow \mathsf{PosMap}[a]$
  $\mathsf{data} \leftarrow \mathsf{ReadBlock}(l, a)$
  if $\mathsf{op} = \mathsf{read}$ **then**
    return data to client
  else
    $\mathsf{data} \leftarrow \mathsf{data}'$
  $l' \leftarrow \mathsf{UniformRandom}(0, d^L - 1)$
  $\mathsf{PosMap}[a] \leftarrow l'$
  Write data to the cnt-th slot of the root bucket
  $\mathsf{cnt} := \mathsf{cnt} + 1 \bmod Z/2$
  if $\mathsf{cnt} = 0$ **then**
    $l_e \leftarrow \mathsf{reverse}(G)$
    $\mathsf{EvictAlongPath}(l_e)$
    $G \leftarrow G + 1 \mod d^L$

function ReadBlock(l, a)
  $(\mathsf{id}_1, \mathsf{id}_2, \ldots, \mathsf{id}_{ZL}) \leftarrow$ Retrieve block identifiers on $\mathcal{P}(l)$
  Suppose $\mathsf{id}_i = a$
  return $\mathsf{TwoServerPIR}(\mathcal{S}_1, \mathcal{S}_2, \mathcal{P}(l), i)$

function EvictAlongPath(l_e)
  for $k \leftarrow 0$ to $L - 1$ **do**
    Let s be the $(k+1)$-th digit of G *// For each bucket, $(k+1)$-th digit accesses slices in a round-robin manner.*
    $\mathsf{EvictToSlices}(l_e, k, s)$
  // Additional processing for the leaf bucket $\mathcal{P}(l_e, L)$ to make it empty
  Read all blocks in $\mathcal{P}(l_e, L)$ and its auxiliary bucket $\mathcal{P}(l_e, \mathsf{aux})$
  Move all real blocks from $\mathcal{P}(l_e, L)$ to $\mathcal{P}(l_e, \mathsf{aux})$

function EvictToSlices(l_e, k, s)
  // Evict from bucket $\mathcal{P}(l_e, k)$ to the s-th slice of each of its d children
  Download all blocks in $\mathcal{P}(l_e, k)$
  for $t \leftarrow 1$ to d
    Let S be the s-th slice of the t-th child of $\mathcal{P}(l_e, k)$
    Let T be the set of real blocks in $\mathcal{P}(l_e, k)$ that can be evicted to S
    Upload T to S and pad remaining slots in S with dummy blocks

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

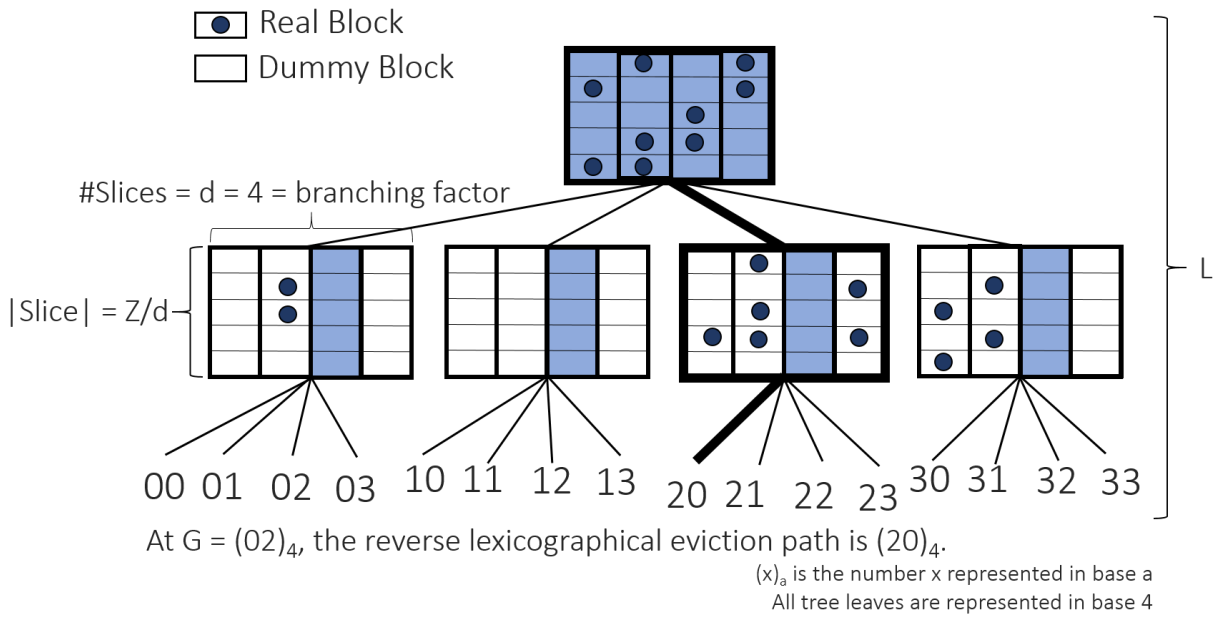

Fig. 3: **Example eviction path for a three-level 4-ary tree at $G = 2$ i.e. $G = (02)_4$.** For evicting the root bucket into its children buckets, the client downloads blue colored root bucket and writes to the blue colored slices of its children. The figure shows load of the buckets just before eviction from the root bucket.

5. The client increments cnt. If $\mathsf{cnt} = Z/2$, the client resets cnt and performs the eviction procedure described below (lines 15-19).

Eviction. The eviction procedure of our construction is a generalization of the eviction procedure of Onion ORAM [12]. It differs from Onion ORAM in the following two ways. First, we apply the eviction scheme on a reverse lexicographical ordering [19] 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). This is essential for our construction to achieve sub-logarithmic bandwidth blowup.

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 each of its children where s is the $(k+1)$-th digit of G. (We show in Section 4.2 that there will be sufficient room in these slices.) 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 (refer Figure 4 for an example), during each of which the slice being written to will be empty. For the last level (level L), the client downloads all blocks in the leaf bucket $\mathcal{P}(l_e, L)$ and its 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.

Example. An example showing $2d$ consecutive evictions is in Figure 4 for $d = 4$. In the example, we start with eviction number $G = (02)_4$. Observe that the third child of the root bucket is emptied at $G = (02)_4$ as the reverse lexicographic eviction path $(20)_4$ passes through it. In the next $d-1$ evictions, one slice of the bucket is written to in a round-robin manner. Finally, at eviction number $G = (12)_4$, when the path $(21)_4$ passes through it again, the last slice is written into after which the entire bucket is emptied again. Similarly,

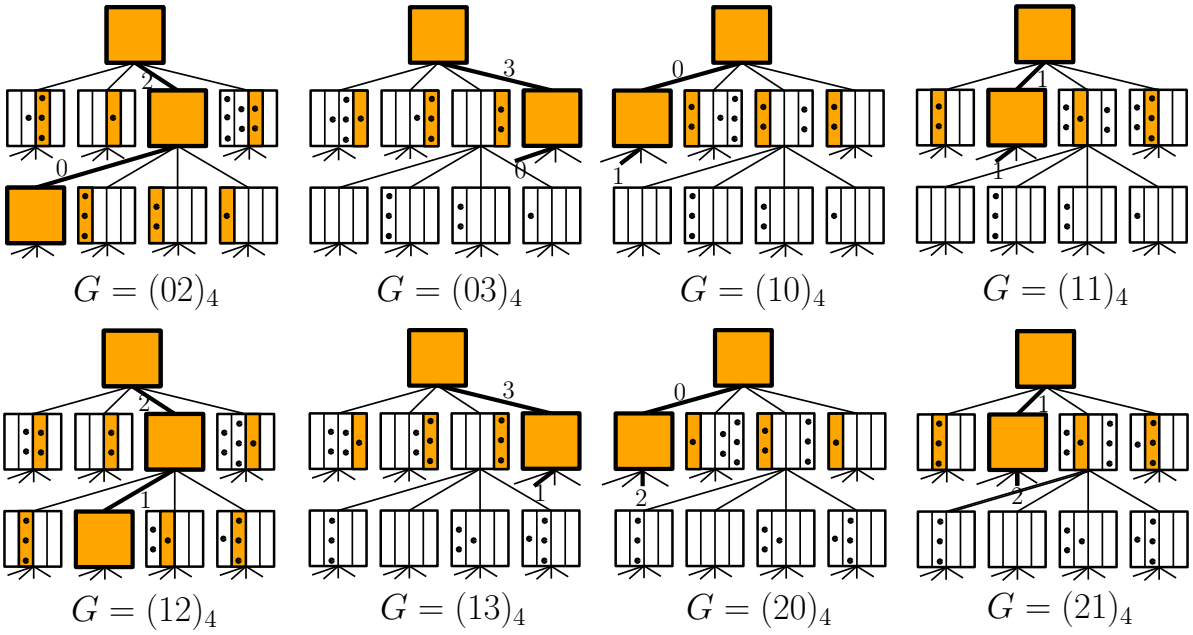

Fig. 4: **Buckets and slices accessed for $2d$ consecutive evictions.** Here, $d = 4$ and $G = \#$ evictions $\mathsf{mod}\ d^L$. $(x)_a$ denotes the number x represented in base a. The dots in the slices represent real blocks at the end of the eviction operation. Note that for each bucket, slices are accessed (written into) in a round-robin manner. If an eviction path passes through a bucket at level i at t-th eviction then it passes through it again at $t + d^i$ evictions.

it can be easily seen that for each bucket at level i, a slice is written into every d^{i-1} evictions and the bucket is emptied every d^i evictions.

4.1 Parameterization and Overflow Analysis

We show that the buckets (and slices) in the tree overflow with negligible probability. In our construction, the root bucket and the auxiliary buckets are not partitioned into slices. Eviction is performed every $Z/2$ accesses, so the root bucket never overflows. Below, Lemma 1 analyzes auxiliary buckets while Lemma 2 analyzes slices in non-root non-auxiliary buckets.

Lemma 1. *If the size of auxiliary buckets Z_{aux} satisfies $N \leq d^L \cdot Z_{\mathsf{aux}}/2$, the probability that an auxiliary bucket overflows is bounded by $e^{-\frac{Z_{\mathsf{aux}}}{6}}$.*

Proof. For an auxiliary bucket b, define $Y(b)$ to be the number of real blocks in b. Each of the N blocks in the ORAM has a probability of d^{-L} to be mapped to b independently. Thus, $E[Y(b)] \leq N \cdot d^{-L} \leq Z_{\mathsf{aux}}/2$, and a simple Chernoff bound completes the proof. □

The following lemma generalizes Onion ORAM [12] Lemma 1 to the scenario of a d-ary tree.

Lemma 2. *The probability that a slice of a non-root and non-auxiliary bucket overflows after an eviction operation is bounded by $e^{-\frac{Z}{6d}}$.*

Proof. Consider a bucket b, and its i-th slice b_i. Define $Y(b)$ to be the number of real blocks in b, and $Y(b_i)$ to be the number of blocks in b_i after an eviction operation.

We will first assume that all slices have infinite capacity and show that $E[Y(b_i)] \leq Z/2d$, i.e., the expected number of blocks in a non-root slice after an eviction operation is no more than $Z/2d$ at any time. Then, we bound the overflow probability given a finite capacity.

For a non-root and non-auxiliary bucket b, we define variables $\overline{m}$ and $m_i, 1 \le i \le d$: the last EvictAlongPath operation where b is on the eviction path is the $\overline{m}$-th EvictAlongPath operation, and the EvictAlongPath operation where b is a sibling bucket with eviction happening to slice i is the m_i-th EvictAlongPath operation. Clearly, during eviction to one of the d slices, the bucket b is on the eviction path. Thus, one of m_i is equal to $\overline{m}$. We also time-stamp the blocks as follows. When a block is accessed and remapped, it gets a time stamp m^*, if the next EvictAlongPath would be the m^*-th EvictAlongPath operation.

Now consider b_i and $Y(b_i)$. There exist the following cases:

1. If $\overline{m} \ge m_i$, then $Y(b_i) = 0$, because the entire bucket b becomes empty when it is a parent bucket during the $\overline{m}$-th EvictAlongPath operation, and the next eviction that evicts blocks to slice b_i has not occurred.
2. If $\overline{m} < m_i$, we must have $m_{i-1} < m_i$. Otherwise, m_i is the smallest among $m_1, \ldots, m_d$ and it must be that $\overline{m} \ge m_i$. We consider blocks with what time stamp range can end up in b_i.
   - Blocks with time stamp $m^* \le \overline{m}$ will not be in b_i as these blocks would have been evicted out of b in the $\overline{m}$-th EvictAlongPath operation.
   - Blocks with time stamp $\overline{m} < m^* \le m_{i-1}$ or $m^* > m_i$ will not be in b_i as these blocks are evicted to either slices $\le i-1$ or slices $> i$ respectively.
   - Blocks with time stamp $m_{i-1} < m^* \le m_i$ can be evicted to b_i.

   There are at most $(m_i - m_{i-1})Z/2$ blocks with time stamp $m_{i-1} < m^* \le m_i$. Each of these blocks go to bucket b independently with probability d^{-j}, where j is the level of b. Due to the deterministic reverse lexicographic ordering of eviction paths, it is easy to see that $m_i - m_{i-1} = d^{j-1}$. Therefore, $E[Y(b_i)] \le d^{j-1} \cdot Z/2 \cdot d^{-j} = Z/2d$.

In either case, we have $\mu = E[Y(b_i)] \le Z/2d$. Now that we have independence and the expected number of blocks in a bucket, using a Chernoff bound with $\delta = 1$, a slice b_i overflows with probability

$$\Pr[Y(b_i) > (1+\delta)u] \le e^{-\frac{\delta^2 \mu}{3}} = e^{-\frac{Z}{6d}}.$$

□

Combining the two lemmas, we can set $Z = \Omega(d\lambda)$ and $Z_{\mathsf{aux}} = \Omega(\lambda)$. The probability that any slice or any bucket overflows is $e^{-\Omega(\lambda)}$. Following prior work [12, 44, 53], it suffices to set $\lambda = \omega(\log N)$ for $N^{-\omega(1)}$ failure probability, i.e., negligible in N.

Server Storage The amount of server storage in our construction is

$$Z_{\mathsf{aux}} \cdot d^L + Z \cdot \Sigma_{i=0}^{L} d^i = \Theta(N).$$

4.2 Security Analysis

Similar to all tree based ORAMs, for each access, the 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.

4.3 Reducing Client Storage

In the construction described so far, the client stores the $\Theta(N \log N)$-bit position map, $\Theta(N \log N)$-bit metadata for all block and uses $\Theta(d\lambda)$ blocks of temporary storage during the eviction operation. In this section, we optimize our scheme to reduce the client storage to $O(1)$ blocks.

A. Position map. The position map for the main ORAM has a $\Theta(\log N)$-bit entry for each of the N blocks, amounting to $\Theta(N \log N)$ bits of storage.

```
1: function EvictToSlices(l_e, k, s)
2:   // Evict from bucket P(l_e, k) to the s-th slice of each of its d children
3:   Download metadata for bucket P(l_e, k) from S_1
4:   for t ← 1 to d
5:     Let S be the s-th slice of the t-th child of P(l_e, k) and S_i be its i-th slot // S is empty
6:     for each S_i ∈ S
7:       if ∃j such that the j-th block in P(l_e, k) can be evicted to S then
8:         block = TwoServerPIR(S_1, S_2, P(l_e, k), j)
9:         Locally update the metadata for the j-th block in P(l_e, k) to be dummy
10:        Upload block along with its metadata to S_i on both servers
11:      else // no such j exists, do a dummy PIR and a dummy upload
12:        Run TwoServerPIR(S_1, S_2, P(l_e, k), 1) and discard its output
13:        Upload a dummy block with a dummy identifier to S_i on both servers
14:  Upload the updated metadata of P(l_e, k) to S_1
```

Fig. 5: **Evicting to children slices using $O(1)$ blocks of client storage.**

Position map can be stored recursively in smaller ORAMs as discussed by Shi et al. [48]. As discussed in [52], when the data block size is $\Omega(\log^2 N)$ (which is the case for our scheme), using a small block size for recursive position map ORAMs, the asymptotic cost of recursion would be insignificant compared to the main ORAM tree. Hence, recursion does not increase to the bandwidth blowup asymptotically.

B. Metadata for each block in the tree. For each block of the tree, we store whether the block is real or dummy. If it is real, the identifying address is stored. This amounts to another $\Theta(N \log N)$ bits of storage.

We can store the metadata of each block along with the block data on the server. However, this would require downloading metadata from the server during retrieval before performing each PIR operation. For $Z = O(d\lambda)$, $L < \log_d N$ and a size of $O(\log N)$ bits for storing the identifier and whether the block is dummy, the total amount of metadata downloaded for an access is $O(d\lambda \log N \log_d N)$. Thus, for a block size of $\Omega(d\lambda \log N \log_d N)$ bits, the asymptotic bandwidth for downloading this metadata is absorbed.

C. Temporary storage for an eviction operation. During an eviction operation, the client downloads a bucket and a slice from each of its d children. This is equivalent to downloading two buckets. Thus, for each step of the eviction operation the client needs to store $Z = O(d\lambda)$ blocks.

We now show how this client storage can be reduced to $O(1)$. At a high level, the client needs to perform the eviction from a bucket to its children buckets without downloading the entire buckets. If the client can only store one block, it needs to download one block at a time from the parent bucket and upload it to one of its children buckets. And the client needs to do so obliviously. We achieve this by hiding which block from the parent bucket is downloaded, again using PIR, and letting the client upload to the children buckets in a deterministic order. The new EvictToSlices algorithm for evicting a parent bucket to its children slices is shown in Figure 5.

To perform the eviction from a bucket $\mathcal{P}(l_e, k)$ to a slice S of its t-th child, the client first downloads the metadata corresponding to $\mathcal{P}(l_e, k)$ (line 3). The client uploads to each slot i in S (denoted S_i) sequentially, one slot at a time (line 6). Before this eviction, each slot S_i will be empty due to Lemma 2. There are two cases:

1. If there exists a real block in $\mathcal{P}(l_e, k)$ that can be evicted to S, the client downloads that block from $\mathcal{P}(l_e, k)$ using PIR (thus hiding its location in $\mathcal{P}(l_e, k)$), and uploads it (re-encrypted) to S_i (lines 7-10).
2. If no real block in $\mathcal{P}(l_e, k)$ can be evicted to S, the client performs a dummy PIR to download an arbitrary block from $\mathcal{P}(l_e, k)$, discards the PIR output, and uploads an encrypted dummy block to S_i (lines 11-13).

Thus, for each $S_i \in S$ in order, the client downloads a block from the parent bucket using PIR (without revealing its position or whether its a dummy PIR) and uploads a block to S_i. This eviction process requires $O(1)$ blocks of storage.

4.4 Bandwidth Analysis

Bandwidth blowup. We analyze the bandwidth blowup of our construction while temporarily ignoring metadata for simplicity. The bandwidth blowup for retrieving a block using PIR is $O(1)$. On evictions, for each bucket on the path, the client downloads the parent bucket and uploads to one slice from each of the d child buckets, which is equivalent to two buckets of bandwidth. Thus, an eviction costs $2ZL$ blocks of bandwidth and it is performed every $Z/2$ accesses, giving an amortized bandwidth blowup of $4L < 4\log_d N$. Overall, the bandwidth blowup of our scheme is $O(\log_d N)$.

Trade-off regarding d. Although our bandwidth blowup decreases with d, we cannot keep increasing d for free. The reason is that the client needs to download a $\Theta(\log N)$-bit metadata for all $d\lambda \log_d N$ blocks on a path, on each access and eviction. Recursion contributes another $O(\log^3 N)$ bits, but that is no greater than the metadata overhead. So the raw bandwidth (in bits) per access is $O(B \log_d N + d\lambda \log_d N \log N)$. While we usually focus on the multiplicative blowup term, when d becomes too large, the additive term will dominate. Thus, the aforementioned bandwidth blowup only holds if the block size is $B = \Omega(d\lambda \log N)$. (If the client has large local storage and stores metadata locally, the block size B can be a $\log N$ factor smaller.)

In other words, the optimal d should be determined as a function of the block size B and the number of blocks N. For instance, for an application using moderately large block size $B = \Omega(\lambda \log^2 N)$, we can set $d = \Theta(\log N)$ and the bandwidth blowup is $O(\log N / \log\log N)$. If some application uses very large blocks such as $B = \Omega(\sqrt{N}\lambda \log N)$, then we can set $d = \Theta(\sqrt{N})$ and achieve a bandwidth blowup of $O(1)$.

5 Extending the Goldreich-Ostrovsky Lower Bound

Goldreich and Ostrovsky [23] gave an $\Omega(\log_c N)$ lower bound on the bandwidth overhead assuming perfect correctness, perfect security and assuming the client to be restricted to the following operations: reading from a memory location and writing to a memory location. Here, N represents the number of logical blocks stored on the server and c is the number of logical blocks stored by the client.

In this section, we first review (a slight variant of) the original Goldreich-Ostrovsky lower bound and its proof from [23]. We then extend the model to include PIR and PIR-write as possible client operations, and analyze how this changes the bound.

Following Goldreich-Ostrovsky and Boyle-Naor [5, 23] (both papers considered the client as a CPU) we will use client and CPU interchangeably, and refer to client storage as c registers.

5.1 Original Lower Bound

We present a slight variant of the CPU used in the original lower bound work [23]:

Definition 2. *A CPU is modeled as a probabilistic random-access-machine (*RAM*) which has c registers and is restricted to the following operations:*

- READ_i *reads a block from memory into a CPU register: For an* $\mathsf{addr} \in [N]$*, set* $\mathsf{Reg}[i] \leftarrow \mathsf{val}$*, where* val *is the content of the memory at location* addr*.*
- WRITE_i *moves a block from a CPU register to memory: For some* $i \in [c]$ *and* $\mathsf{addr} \in [N]$*, write the data in* $\mathsf{Reg}[i]$ *to location* addr *in the memory.*

Our RAM model has slightly different operations than the original analysis in [23] to simplify the exposition. This will cause only constant factor differences in the bound. Given this model, the original lower bound can be stated as:

Theorem 1. *Every statistically oblivious simulation of a probabilistic* RAM *(Definition 2), which can access N blocks of data and hold c blocks in the CPU, on input y makes at least $max\{|y|, \Omega(t \cdot \log_c N)\}$ accesses to simulate $t = \mathsf{poly}(N)$ steps.*

Proof. As described by Goldreich and Ostrovsky [23], the task of simulating an ORAM can be modeled as a game in between three players

1. The CPU, which can hold at most c blocks and has access to a random function f
2. the program request sequence of length t
3. the observer

The CPU knows the program request sequence ahead of time. Suppose the request access sequence is represented as $r_1, \ldots, r_t$. The game takes place on a memory consisting of N cells, each capable of storing a single block.

The game proceeds for t rounds. In each round i, the CPU loads r_i in one of its registers in such a way that the observer does not learn r_i. The CPU does so by making a sequence of operations as described in Definition 2. At any time during the game, the CPU can store at most c blocks in memory. Round i ends when the CPU holds block r_i.

For a t sized request sequence, the CPU's response can be represented by two q sized sequences. The first sequence consists of an externally visible access pattern. Each entry in this sequence is a single address (each of which is sufficient to implement either READ_i or WRITE_i). The second sequence consists of hidden operations performed by the CPU. Let us consider the possible number of *hidden actions* that can be taken by the CPU, *for a fixed visible access pattern.* For each visible read address, the block that is read can be stored in one of the c registers in the CPU. Likewise, for each visible write address, the block written back could have been removed from any one of the c registers in the CPU. Thus, hidden from the observer, the system can evolve in c ways on a READ and c ways on a WRITE.

Given that the CPU stores c blocks, a q length sequence can satisfy at most c^q program access sequences. Therefore, each visible access sequence can satisfy at most $(2c+1)^q c^q$ program request sequences. For perfect security, the visible access sequence should be able to satisfy all N^t possible request sequences, i.e.,

$$N^t \leq (2c)^q c^q$$

or $q \geq \frac{t \log N}{\log c + \log(2c)} = \Omega\left(t \log_c N\right)$. □

The above is a bound on the number of operations. Since each operation incurs at least 1 block of bandwidth, we also obtain an amortized bandwidth blowup lower bound of $\Omega(\log_c N)$.

5.2 Augmented Lower Bound (after adding PIR)

We now extend the above result to allow the CPU to perform PIR and PIR-write.

Definition 3. *A PIR-augmented CPU is modeled as a probabilistic random-access-machine* PIR-RAM *which has c registers and is restricted to the following operations:*

- READ_i *as described in Definition 2.*
- WRITE_i *as described in Definition 2.*
- $\mathsf{PIR\text{-}READ}_i$ *reads a block from memory into a CPU register using PIR: For a set of at most D addresses, set* $\mathsf{Reg}[i] \leftarrow \mathsf{val}$*, where* val *can be the content of the memory at any of the locations in the set.*
- $\mathsf{PIR\text{-}WRITE}_i$ *moves a block from a CPU register into memory privately using a PIR-WRITE operation: For a set of at most D addresses, write the data in* $\mathsf{Reg}[i]$ *to a location among one of the D addresses.*

Theorem 2. *Every statistically oblivious simulation of a probabilistic* PIR-RAM *(Definition 3), which can access N blocks of data and hold c blocks in the CPU and perform PIR on a maximum of D blocks, on input y makes at least* $max\{|y|, \Omega(t \cdot \log_{cD} N)\}$ *accesses to simulate* $t = \mathsf{poly}(N)$ *steps.*

Proof. The proof follows the same framework as the original lower bound. The number of operations in the visible and hidden sequences due to READ_i or WRITE_i operations is unchanged. Now, the visible sequence additionally reveals the set of D addresses accessed on a PIR request for $\mathsf{PIR\text{-}READ}_i/\mathsf{PIR\text{-}WRITE}_i$. In each

of these operations, the client can select one out of D possible memory blocks to read/write in the visible memory. Furthermore, for each of the above D outcomes, the client can add the read block to (or remove the written block from) any one of the c local registers. Thus, the system can evolve in cD possible ways for each of the PIR-READ and PIR-WRITE operations.

Extending the original argument, each visible access sequence can satisfy $(2c + 2cD)^q c^q$ program request sequences. For perfect security, the visible access sequence should be able to satisfy all N^t possible request sequences, i.e.,

$$N^t \leq (2c + 2cD)^q c^q$$

or $q \geq \frac{t \log N}{\log c + \log(2c+2cD)} = \Omega\left(\frac{t \log N}{\log(cD)}\right)$. □

Again, the bound is on the number of operations. Since each of the four operations incurs at least 1 block of bandwidth, a bound on the number of operations translates to a bound on amortized bandwidth blowup.

5.3 Discussion

Accounting for failure probability. The above lower bound assumes perfect security, i.e., each visible physical access sequence should be able to satisfy all possible program request sequences. However, using an argument similar to Wang et al. [54], the same lower bound can be extended to work for up to $O(1)$ failure probability (and hence, negligible failure probability).

PIR as a black box. Our lower bound is independent of the implementation details of the PIR and PIR-write operations. The bound is applicable to any statistically secure PIR construction that meets the interface in Definition 3, regardless of the number of servers it uses. We also note that although the lower bound considers PIR-WRITE as a possible operation, our construction does not use this primitive.

Our construction and the lower bound. Our construction matches this lower bound for certain parameter ranges. We use $c = 1$ register and perform a PIR operation on $D = O(d \cdot poly(\log N))$ blocks. Thus, our lower bound is asymptotically tight for $d = \Omega(\log N)$ when the data block size $B = \Omega(d \log^2 N)$.

C-ORAM, CHf-ORAM and the lower bound. We discuss how the lower bound is applicable to C-ORAM [40] and CHf-ORAM [39] in Section 6.2.

Circumventing the lower bound. The lower bound on bandwidth only applies to black-box usage of PIR. Onion ORAM [12] circumvents the lower bound and achieves $O(1)$ bandwidth blowup. The reason is that the homomorphic select operation in Onion ORAM (a non-black-box usage of PIR) does not consume one unit of bandwidth. Therefore, while the number of operations in Onion ORAM is still subject to the bound, the bound does not translate to bandwidth blowup. It is possible to circumvent the lower bound by adding other operations (e.g., FHE [2]).

6 Security Analysis of C-ORAM

C-ORAM [40] is a CCS'15 paper that achieves constant bandwidth blowup over smaller block sizes and performs less server computation (compared to Onion ORAM [12]). C-ORAM introduces an eviction procedure that publicly and homomorphically merges bucket contents. CHf-ORAM [39] extends C-ORAM with four non-colluding servers to avoid homomorphic encryption. In this section, we first give a short review of C-ORAM and CHf-ORAM. We then use the lower bound described in the previous section to show that the results obtained by C-ORAM and CHf-ORAM are impossible. Lastly, we give two concrete attacks that apply to both C-ORAM and CHf-ORAM.

6.1 A Review of C-ORAM

C-ORAM follows the tree-based ORAM framework in Section 3.2. It has a large bucket size $Z = \omega(\log N)$ and performs one eviction every $\chi = O(Z)$ accesses. On accesses, it relies on single-server PIR (or 2-server

PIR in the case of CHf-ORAM) to achieve constant bandwidth. Each eviction goes down a path in the reverse lexicographical order. For each bucket on the path, C-ORAM moves all blocks in it into the two child buckets. To perform this eviction procedure using constant bandwidth, C-ORAM proposes the following "oblivious merge" operation.

Each bucket may contain three types of blocks: real, noisy and zero. Essentially, C-ORAM has two types of dummy blocks. A zero block is a dummy block whose plaintext value is 0; a noisy block is a dummy block whose plaintext value is arbitrary. Metadata in each bucket or maintained by the client tracks the type of each block. C-ORAM then encrypts each block using an additive homomorphic encryption. Notice that if the server homomorphically merges an encrypted real block with an encrypted zero block, the result would be an encryption of the real block, i.e., $E(r) + E(0) = E(r)$ for a plaintext real block r. However, if a real block is merged with a noisy block or another real block, then the content cannot be recovered. If a zero block is merged with a noisy block, it is "contaminated" and becomes a noisy block. Therefore, in order to merge two buckets, C-ORAM needs to permute and align the two buckets in a very specific way, i.e., a real block in one bucket must always be aligned with a zero block in the other bucket. Crucially, C-ORAM also prioritizes aligning two noisy blocks such that it contaminates as few zero blocks as possible.

To make the presentation clear, we distinguish "permute" and "shuffle" operations. Whenever we say a set of blocks are "shuffled", we mean the client downloads all the blocks, shuffles them secretly and uploads them back to the server; the server has no idea how the blocks are shuffled. Whenever we say a set of blocks are "permuted", we mean the client instructs the server to permute them, and the server sees the permutation. Therefore, permuting a set of blocks does not provide any obfuscation effect. Its only purpose is to enforce the merging rules in C-ORAM, i.e., a real block should be merged with a zero block, and a noisy block should be merged with another noisy block if possible.

Each eviction goes down a path, and merges each bucket on the path into its two children. Note that shuffling all buckets involved an eviction would take more than constant bandwidth. Therefore, when two buckets need to be merged in C-ORAM, they are permuted and not shuffled, and the server sees the permutations. It is unnecessary to permute both buckets. It is equivalent to permuting only the parent bucket and merging it into the child bucket. Now we try to analyze whether these permutations leak information about the access pattern. C-ORAM argues that if the client secretly and randomly shuffles the root bucket before each eviction, then all permutations look random and leak no information to the server. Unfortunately, this belief is incorrect.

6.2 C-ORAM, CHf-ORAM and the Lower Bound

C-ORAM and CHf-ORAM introduced three new operations on top of the standard ORAM model: download a block from a path of poly-logarithmic blocks using PIR-READ, upload a block to one hidden location in a bucket using PIR-WRITE, and an oblivious merge operation. In an oblivious merge operation, the server applies plaintext permutations (chosen by the client) to buckets before merging them. This operation creates only one possible outcome to the system state, since no action is hidden from the server. Thus oblivious merge does not affect the lower bound in Section 5.

CHf-ORAM achieves statistical security with negligible failure probability and is thus subject to the lower bound in Theorem 2. The number of operations required for t logical accesses is $\Omega(\frac{t \log N}{\log(cD)})$ where $c = O(1)$ and $D = \mathsf{polylog}(N)$. Thus, its bandwidth blowup is lower bounded by $\Omega(\frac{\log N}{\log \log N})$. Instead, CHf-ORAM claims to have achieved $O(1)$ bandwidth, implying a flaw in its construction.

C-ORAM achieves computational security due to the use of single-server PIR-READ/PIR-WRITE, and thus does not directly violate the lower bound. However, unless carefully shown otherwise, it is extremely unlikely that any security flaw of CHf-ORAM can be fixed by merely replacing information theoretically secure PIR with computationally secure PIR.

6.3 An Attack on the Optimized Construction of C-ORAM

This subsection and the next one give two concrete attacks to C-ORAM to give some insights on why it is insecure. Before we start, the following analogy may aid understanding. Imagine a trivially broken ORAM as

follows. The client randomly shuffles all N blocks only *once* initially and keeps track of the mapping locally. Then for each request, the client simply retrieves the requested block. Each access is clearly to a random location due to the initial shuffle. But if the same block is requested multiple times, these accesses will go to the same location and this correlation reveals information.

C-ORAM essentially used a flawed argument like the above. While each permutation looks random in isolation, there is correlation among permutations, and the correlation leaks information. Both of our attacks exploit this fact.

Our first attack is on the optimized construction of C-ORAM, i.e., the "Second Construction" in Section 3.3. The goal of the second construction is to decrease D, i.e., the number of blocks to perform PIR on. The idea is to, on every access, "clone" the requested path to temporary memory, and perform a C-ORAM eviction operation (which we call a "shadow eviction") along the cloned path. By the ORAM invariant, the block of interest now lives in the leaf bucket and PIR to only the leaf bucket (not the entire path) is sufficient to retrieve the block. The cloned path is thrown out after the PIR operation.

This scheme suffers from correlations among permutation operations when a pair of buckets (a parent and its child) are part of multiple shadow evictions in between being involved in two regular evictions. Note that due to randomness (even if one eviction happens after every access, as suggested in C-ORAM), there is non-negligible probability that a pair of buckets deep in the tree are involved in more than 1 shadow evictions between two regular evictions. In each shadow eviction, the normal C-ORAM eviction rules apply: a real block can only be merged with a zero block, and a noisy block is prioritized to be merged with another noisy block. Since the contents of the two buckets remain the same across these shadow evictions, it is easy to see that certain slots in one bucket (e.g., the real blocks) will repeatedly "prefer" certain slots in the other bucket (e.g., zero blocks). This bias can reveal the number of real blocks in the bucket. It is well known that revealing bucket load in tree ORAMs is sufficient to leak the access pattern [48]: more recently accessed blocks will be in buckets higher in the tree than less recently used blocks.

6.4 An Attack on the Basic Construction of C-ORAM

Our second attack applies to both the basic version and the optimized version of C-ORAM. For this attack, we need the first three evictions. Recall that the basic C-ORAM performs one eviction every χ accesses, so we need client access sequences of length 3χ. Concretely, consider the following two client access patterns:

1. Access the same block 3χ times, i.e., $X = \{a_1, \ldots, a_{3\chi}\}$ where $a_i = a, \forall i$
2. Access 3χ distinct blocks, i.e., $X' = \{a_1, \ldots, a_{3\chi}\}$ where $a_i \neq a_j$, $\forall i \neq j$.

In this attack, we assume that initially all blocks in the C-ORAM tree are zero blocks. Our attack also work if initially the server stores all real blocks in leaf buckets, and all non-leaf buckets only contain zero blocks. We believe these are the two most natural initial states for tree-based ORAMs.[7] With access pattern X, the root will contain 1 real block, $\chi - 1$ noisy blocks and $Z - \chi$ zero blocks before each eviction. With access pattern X', the root will contain χ real blocks, 0 noisy block and $Z - \chi$ zero blocks before each eviction.

Figure 6 walks through the first three evictions in C-ORAM, and highlights a pair of correlated permutations during the 2nd and 3rd eviction. The figure shows the first three levels of the tree. O represents a bucket full of zero blocks. Initially, all blocks are zero. A, B and C are the three buckets of blocks injected into the root before the 1st, 2nd and 3rd eviction, respectively. A, B and C are all randomly shuffled by the client.

On the first eviction (first row), A is injected to the root. The bottom of the figure depicts an example of A assuming a small bucket size $Z = 8$ and $\chi = 4$. It contains $\chi = 4$ real blocks, denoted r, and 4 zero blocks, denoted o. This would be the case when the original access pattern is X' and four distinct blocks are accessed. A is then evicted to the next level, producing A_1 and A_2. They are then merged with the two children (both are all-zero buckets O), leaving the root empty. A_1 contains the set of blocks mapped to the left half of the tree, and A_2 contains the set of blocks mapped to the right half of the tree. In the example at the bottom, we assume 3 blocks are mapped to the left half and 1 block is mapped to the right half. Notice that A_1 and

[7] Through personal communication with the C-ORAM authors, we learnt that C-ORAM does not start in these two initial states. However, the C-ORAM paper did not specify what the initial state is.

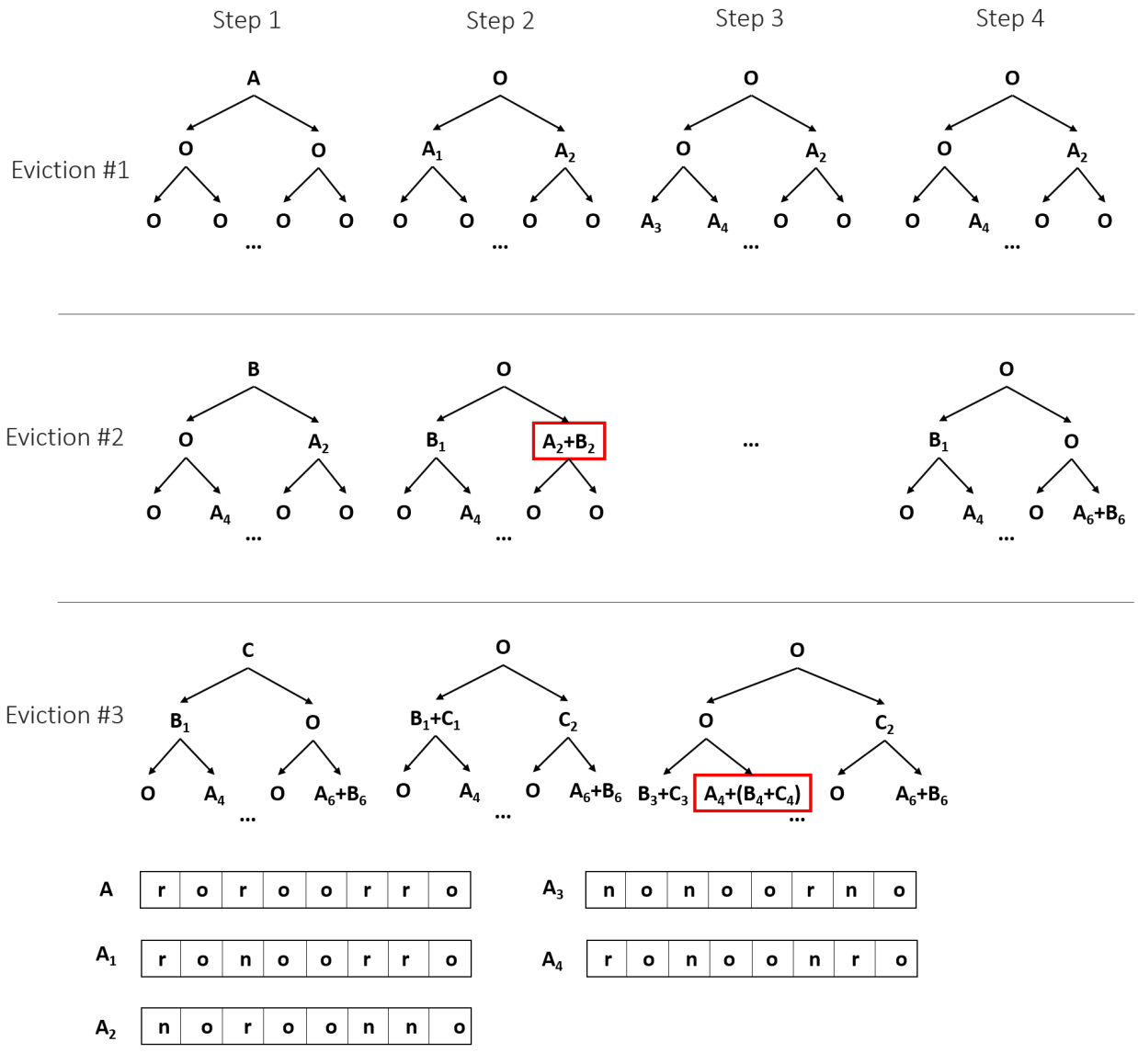

Fig. 6: **Public permutations in C-ORAM are correlated.**

A_2 are correlated. All the real blocks in A_1 are noisy blocks (denoted n) in A_2 and vice versa. On the other hand, all the zero blocks in A_1 are zero blocks in A_2 as well. We remark that A_1 and A_2 will be independently permuted (not shown in the figure). The two permutations will be truly random because merging with empty buckets imposes no restrictions on how the two buckets are aligned. But as we noted earlier the permutations do not provide any security benefits since the attacker sees the permutation in clear. The attacker can easily apply the inverse permutation to get the same view as our example in which the correlation exists.

The first eviction continues down the leftmost path and evicts A_1 into A_3 and A_4, and further evicts A_3 down the tree (not shown). Again, a public random permutation is applied for every merge, and similar correlations exist among all the derivative of A once the attacker applies the inverse permutation.

The second eviction (second row) injects another shuffled bucket B. B produces B_1 and B_2. B_1 is randomly permuted and merged with a zero bucket. B_2, however, needs to be permuted according to the C-ORAM rules (described in Section 6.1) to align with A_2. After that, the eviction goes on to evict the merged bucket $\mathsf{A}_2 + \mathsf{B}_2$ and its children (not shown).

On the third eviction (third row), we focus on the left half of the tree. C similarly produces C_1 and C_2. C_1 is permuted and merged with B_1. The merged bucket is then permuted again to be merged with A_4. This latter permutation (to align $\mathsf{B}_4 + \mathsf{C}_4$ with A_4) will have a strong correlation with the one that aligns A_2 and B_2 in the second row. More crucially, the type of correlation is very different depending on whether the client access pattern is X or X', thereby revealing the access pattern.

First consider access pattern X. In this case, A_2, B_2, A_4 and B_4 mostly contain noisy and zero blocks (there are at most 3 real blocks in the system). Furthermore, the noisy blocks occupy the same set of locations in A_2 and A_4, and also in B_2 and B_4. If two noisy blocks are aligned during $\mathsf{A}_2 + \mathsf{B}_2$, those two slots are also likely to be aligned in $\mathsf{A}_4 + \mathsf{B}_4$ because C-ORAM prioritizes noisy-noisy merge. If we simply count the number of repeated entries between these two permutations, it will be significantly higher than 1, which is the expected value for two random permutations.

Now consider access pattern X'. In this case, A_2, B_2, A_4 and B_4 will all contain a moderate number of real blocks. Recall that all real blocks in A_2 (B_2) are noisy blocks in A_4 (B_4) and vice versa, while all zero blocks in A_2 (B_2) are zero blocks in A_4 (B_4). Now once a real block in A_2 is aligned with a zero block in B_2, that same slot in A_4—a noisy block—tends to avoid that previous slot in B_4—a zero block—again because C-ORAM prioritizes noisy-noisy merge. If we again count the number of repeated entries between these two permutations, it will be much lower than the expected value 1 for two random permutations.

Utilizing these two different types of correlation, the attacker can easily distinguish X and X' by counting the repeated entries between the two highlighted permutations above. We implement the above attack and run the experiment 10000 times with $Z = 60$ and $\chi = 20$. For access pattern (i), the average number of repetition we get is 1.96, much higher than the expected value 1. For access pattern (ii), the average number of repetition is merely 0.81. We repeat the same experiment with $Z = 120$ and $\chi = 40$, and reproduce the results: 1.94 and 0.86. This shows our attack easily distinguishes the above two access patterns.

7 Conclusion and Open Problems

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. We show a lower bound of $\Omega(\log_{cD} N)$ for bandwidth blowup for a client storing c blocks of data and performing a PIR on D blocks of data at a time. Our construction matches our lower bound under certain parameter ranges. C-ORAM [40] and CHf-ORAM [39] violate the lower bound and have security flaws.

While we do achieve a sub-logarithmic bandwidth blowup, we do so by using a two server PIR and server computation. It is still an open question whether a sub-logarithmic bandwidth blowup can be obtained in the original model defined by Goldreich and Ostrovsky (the GO bound does not rule it out if the client uses $c = \omega(1)$ storage). Also, all known ORAM schemes that achieve $O(\log N)$ bandwidth blowup require a block size of $\Omega(\log^2 N)$. Whether this bound (or a sub-logarithmic bound) can be obtained for smaller block sizes remains open.

8 Acknowledgements

We would like to thank authors of C-ORAM (Tarik Moataz, Travis Mayberry and Erik-Oliver Blass) for discussions and inputs on algorithmic details of C-ORAM. We would like to thank Dahlia Malkhi, Jonathan Katz, Elaine Shi, Hubert Chan and Xiao Wang for helpful discussions on this work. This work is funded in part by NSF awards #1111599, #1563722 and a Google Ph.D. Fellowship award.

References

1. M. Ajtai. Oblivious rams without cryptogrpahic assumptions. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 181–190. ACM, 2010.
2. D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In *International Workshop on Public Key Cryptography*, pages 131–148. Springer, 2014.
3. V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 837–849. ACM, 2015.
4. E. Boyle, K.-M. Chung, and R. Pass. Oblivious parallel ram and applications. In *Theory of Cryptography Conference*, pages 175–204. Springer, 2016.
5. 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.
6. C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In J. Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 402–414. Springer, 1999.
7. B. Chen, H. Lin, and S. Tessaro. Oblivious parallel ram: Improved efficiency and generic constructions. In *Theory of Cryptography Conference*, pages 205–234. Springer, 2016.
8. B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.
9. K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure oram with\ tilde {O}(\ logˆ 2 n) overhead. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 62–81. Springer, 2014.
10. I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In *Theory of Cryptography Conference*, pages 144–163. Springer, 2011.
11. J. Dautrich, E. Stefanov, and E. Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, 2014.
12. 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.
13. Z. Dvir and S. Gopi. 2-server PIR with sub-polynomial communication. In R. A. Servedio and R. Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 577–584. ACM, 2015.
14. C. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov. Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM. Technical report, 2015.
15. C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.
16. C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas. Freecursive oram:[nearly] free recursion and integrity verification for position-based oblivious ram. In *ACM SIGPLAN Notices*, volume 50, pages 103–116. ACM, 2015.
17. C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, E. Stefanov, D. Serpanos, and S. Devadas. A low-latency, low-area hardware oblivious ram controller. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 215–222. IEEE, 2015.
18. S. Garg, P. Mohassel, and C. Papamanthou. Tworam: Round-optimal oblivious ram with applications to searchable encryption. Cryptology ePrint Archive, Report 2015/1010, 2015.
19. 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.

20. C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with he-over-oram architecture. In *International Conference on Applied Cryptography and Network Security*, pages 172–191. Springer, 2015.
21. C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 803–815. Springer, 2005.
22. O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM, 1987.
23. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.
24. M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages, and Programming*, pages 576–587. Springer, 2011.
25. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 157–167. SIAM, 2012.
26. M. Keller and P. Scholl. Efficient, oblivious data structures for mpc. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 506–525. Springer, 2014.
27. E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.
28. E. Kushilevitz and R. Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 364–373. IEEE Computer Society, 1997.
29. H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *International Conference on Information Security*, pages 314–328. Springer, 2005.
30. C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 87–101. ACM, 2015.
31. C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient ram-model secure computation. In *2014 IEEE Symposium on Security and Privacy*, pages 623–638. IEEE, 2014.
32. C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE, 2015.
33. J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 199–213, 2013.
34. S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography*, pages 377–396. Springer, 2013.
35. S. Lu and R. Ostrovsky. How to garble ram programs? In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–734. Springer, 2013.
36. M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324. ACM, 2013.
37. T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS*. Citeseer, 2014.
38. J. C. Mitchell and J. Zimmerman. Data-Oblivious Data Structures. In *Theoretical Aspects of Computer Science (STACS)*, 2014.
39. T. Moataz, E.-O. Blass, and T. Mayberry. Chf-oram: a constant communication oram without homomorphic encryption. Cryptology ePrint Archive, Report 2015/1116, 2015.
40. 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.
41. 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.
42. B. Pinkas and T. Reinman. Oblivious ram revisited. In *Annual Cryptology Conference*, pages 502–519. Springer, 2010.
43. A. Rane, C. Lin, and M. Tiwari. Raccoon: closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, 2015.

44. 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.
45. L. Ren, C. W. Fletcher, X. Yu, M. Van Dijk, and S. Devadas. Integrity verification for path oblivious-ram. In *High Performance Extreme Computing Conference (HPEC)*. Institute of Electrical and Electronics Engineers (IEEE), 2013.
46. L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 571–582. ACM, 2013.
47. C. Sahin, V. Zakhary, A. El Abbadi, H. R. Lin, and S. Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *Security and Privacy (SP), IEEE Symposium on*, 2016.
48. 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.
49. 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.
50. E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267. IEEE, 2013.
51. E. Stefanov, E. Shi, and D. X. Song. Towards practical oblivious ram. In *NDSS*. The Internet Society, 2012.
52. E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. Cryptology ePrint Archive, Report 2013/280 v3, 2013. `http://eprint.iacr.org/2013/280`.
53. 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.
54. 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.
55. X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 191–202, New York, NY, USA, 2014. ACM.
56. X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226. ACM, 2014.
57. P. Williams and R. Sion. Sr-oram: Single round-trip oblivious ram. *ACNS, industrial track*, pages 19–33, 2012.
58. 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*, pages 139–148. ACM, 2008.
59. P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 977–988. ACM, 2012.
60. S. Zahur, X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 218–234, 2016.
61. J. Zhang, Q. Ma, W. Zhang, and D. Qiao. Kt-oram: A bandwidth-efficient oram built on k-ary tree of pir nodes. 2014.
62. J. Zhang, Q. Ma, W. Zhang, and D. Qiao. Mskt-oram: A constant bandwidth oram without homomorphic encryption. IACR Cryptology ePrint Archive, Report 2016/882, 2016.

A Server Computation in ORAM

We define ORAMs with server computation similar to Apon et al. [2]. They define ORAM with server computation as a reactive two party protocol between client and server in the universal composability framework. The notation

$$((c_{out}, c_{state}), (s_{out}, s_{state})) \leftarrow \mathsf{protocol}((c_{in}, c_{state}), (s_{in}, s_{state}))$$

represents the inputs and outputs of a client-server protocol $\mathsf{protocol}$ where c_{in} and c_{out} represents the client's input and outputs, s_{in} and s_{out} represent the server's input and output and c_{state} and s_{state} represent the client and server's state before and after the protocol.

An Oblivious RAM with server computation can be defined as follows:

Definition 4. *An ORAM with server computation consists of the following interactive protocols between client and server:*

- $((\perp, z), (\perp, Z)) \leftarrow \mathsf{Initialize}(1^\lambda, (D, \perp), (\perp, \perp))$: An interactive protocol where the client's input is a memory array $D[1, \ldots, n]$ where each block has bit-length B; and the server's input is $\perp$. At the end of the protocol, the client holds secret state z and the server holds Z (which typically encodes the memory array D).

- $((\mathsf{data}, z'), (\perp, Z') \leftarrow \mathsf{Access}(((a, \mathsf{op}, \mathsf{data}'), z), (\perp, Z)))$: Let the initial memory array used by the client in the initialization protocol be denoted by D. Access is an interactive protocol where the client has an initial state z and an input tuple $(a, \mathsf{op}, \mathsf{data}')$. Here, a refers to an index in memory array D, $\mathsf{op} \in \{\mathsf{Read}, \mathsf{Write}\}$ and data' is the data to be written ($\mathsf{data}' = \perp$ if $\mathsf{op} = \mathsf{Read}$). The server has no input and starts with state Z'. At the end of a correct execution of the protocol, the client has output data where $\mathsf{data} = D[a]$ (for writes, the output is the old value of $D[a]$ before the write takes place). The client and server have updated output states z' and Z' respectively. The client outputs $\mathsf{data} = \perp$ if the protocol execution is aborted.

We say that a server-computation ORAM scheme is correct, if for any initial memory $D \in \{0,1\}^{Bn}$, for any input sequence $((a_1, \mathsf{op}_1, \mathsf{data}_1), \ldots, (a_m, \mathsf{op}_m, \mathsf{data}_,))$ where $m = \mathrm{poly}(\lambda)$, a client input (a, Read) would always return the last value written to the logical location a except with negligible probability.

Security Definition. We adopt a standard simulation-based definition of secure computation requiring that a real-world execution simulate an ideal-world (reactive) functionality $\mathcal{F}$.

Ideal world. We define an ideal functionality $\mathcal{F}$ that maintains an up-to-date version of the data D on behalf of the client and answers client's access queries.

- *Initialize.* An environment $\mathcal{Z}$ gives an initial database D to the client. The client sends D to the ideal functionality $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ about the execution of the initialization protocol and $|D|$. The ideal-world adversary $\mathcal{S}$ responds with ok or abort. $\mathcal{F}$ then says ok or $\perp$ to the client accordingly.
- *Access.* In each time step, the environment $\mathcal{Z}$ specifies a tuple $(a, \mathsf{op}, \mathsf{data})$ where $\mathsf{op} = \{\mathsf{Read}, \mathsf{Write}\}$ and $\mathsf{data} = \perp$ if $\mathsf{op} = \mathsf{Read}$ as the client's input. The client sends op to $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$(without revealing client's inputs). If $\mathcal{S}$says ok to $\mathcal{F}$, $\mathcal{F}$ sends $D[a]$ to the client and updates $D[a] = \mathsf{data}$ if $\mathsf{op} = \mathsf{Read}$. The client then forwards $D[a]$ to the environment $\mathcal{Z}$. If $\mathcal{S}$says abort to $\mathcal{F}$, $\mathcal{F}$ sends $\perp$ to the client.

Real world. In the real world, an environment $\mathcal{Z}$ gives an honest client a database D. The honest client runs the $\mathsf{Initialize}$ protocol with the server $\mathcal{A}$. Then at each time step, $\mathcal{Z}$ specifies an input tuple $(a, \mathsf{op}, \mathsf{data})$ (where $\mathsf{op} = \{\mathsf{Read}, \mathsf{Write}\}$ and $\mathsf{data} = \perp$ if $\mathsf{op} = \mathsf{Read}$) to the client. The client then runs the Access protocol with the server. The environment $\mathcal{Z}$ gets the view of the adversary $\mathcal{A}$ after every operation. The client outputs to the environment the data fetched or $\perp$ in case of an abort.

Definition 5. *We say that a protocol $\Pi_\mathcal{F}$ securely computes the ideal functionality $\mathcal{F}$ if for any probabilistic polynomial-time real-world adversary (i.e, server) $\mathcal{A}$, there exists an ideal-world adversary $\mathcal{S}$, such that for all non-uniform, polynomial-time environment $\mathcal{Z}$, there exists a negligible function* negl *such that*

$$|Pr[Real_{\Pi_\mathcal{F}, \mathcal{A}, \mathcal{Z}}(\lambda) = 1] - Pr[Ideal_{\mathcal{F}, \mathcal{S}, \mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$