LatORAM: ORAMs from Lateral Stashes and Delayed Shuffling

Sarvar Patel* Giuseppe Persiano† Joon Young Seo‡ Kevin Yeo§

January 1, 2026

Abstract

We study the design of Oblivious RAMs (ORAMs) that allow a client to access memory outsourced to a remote, untrusted server without revealing the client's data access pattern. We are interested in concretely efficient constructions and prior works have yielded different ORAM frameworks with various trade-offs. Tree-based constructions such as RingORAM [Ren et al., USENIX'15] obtain low communication overhead, but require client storage of linear position maps and two roundtrip queries. Hierarchical schemes such as FutORAMa [Asharov et al., CCS'23] further reduce communication at the cost of more roundtrips during queries. Finally, SQRT-ORAM [Goldreich, STOC '87] enables fast queries of one roundtrip and one block of communication at the cost of larger amortized communication costs.

We present two new constructions, LatORAM and Lat^{2}ORAM, that simultaneously obtain the positive traits of all three types of ORAM constructions. Online queries are blazing fast with one roundtrip and a single block of communication like SQRT-ORAM. Fixing the client memory sizes for comparison, the online communication cost of our constructions are 5-8x smaller than RingORAM and 5-10x smaller than FutORAMa even though both RingORAM and FutORAMa require multiple roundtrips per online query. Furthermore, our total amortized communication is also up to 50% smaller. To obtain our constructions, we present a new lazy approach of lateral stash growth that delays large shuffles.

Of independent interest, we present improved oblivious merging schemes for specific settings important for our ORAMs. Our constructions solely rely on symmetric cryptography.

*Google, USA. sarvar@google.com
†Università di Salerno, Italy. giuper@gmail.com
‡Google, USA. jyseo@google.com
§Google, USA and Columbia University, USA. kwlyeo@google.com

Contents

1 Introduction **3**

2 Preliminaries and Basic Tools **5**
2.1 Oblivious RAMs . . . 5
2.2 Server Storage . . . 6
2.3 Client Storage . . . 6
2.4 The XOR Technique . . . 6

3 Shuffling and Merging **7**
3.1 Shuffling Algorithms . . . 7
3.2 Client-Stored Input Blocks . . . 8
3.3 Obliviously Merging Two Containers . . . 8

4 ORAM from Delayed Merging **10**
4.1 Idea: Delaying Large Merges . . . 12
4.2 First Construction: LatORAM . . . 13
4.3 Analysis of LatORAM . . . 15

5 Lat^{2}ORAM: Further Delayed Merging **18**
5.1 Second Construction: Lat^{2}ORAM . . . 19
5.2 Analysis of Lat^{2}ORAM . . . 21

6 Experimental Evaluation **23**

7 Related Works **26**

8 Conclusions **27**

A A General Approach **31**

B Oblivious Merging **32**

C Further discussions on CacheShuffle **33**
C.1 Description . . . 33
C.2 Analysis . . . 34

D TwoMerge Analysis **36**

E Additional Experimental Data **37**
E.1 The parameters used for LatORAM and Lat^{2}ORAM . . . 37
E.2 The parameters used for RingORAM . . . 37

1 Introduction

An Oblivious RAM (ORAM) [13, 14] allows a client to access a remote memory consisting of N blocks stored by an untrusted semi-honest server without leaking the patterns of access to the server. This primitive has found application in several fields including secure hardware [37, 22, 21, 29, 18], contact discovery for Signal [31], practical obfuscation [23] and cloud storage [32, 3, 5].

Given its wide applicability, there has been a long line of work studying ORAMs to reduce costs (see [13, 25, 15, 19, 26]) leading to OptORAMa [1] that uses constant client storage and obtains $O(\log N)$ bandwidth, that is known to be necessary [20]. Unfortunately, these optimal asymptotic constructions have extremely large constants (estimated by [8] to be at least 2^{228} for [1]) prohibiting them from practical deployment.

Prior works have aimed to design ORAMs that are concretely efficient. To circumvent the large constant barrier, these works consider constructions with either larger asymptotic communication costs and/or larger client storage. To date, these constructions can be roughly divided into three different types that we present below.

Tree-based ORAMs introduced by Shi *et al.* [30] build ORAMs using binary trees and a clever eviction algorithm that obliviously moves blocks from the root towards the leaves of the binary tree. To our knowledge, the most concretely efficient tree-based construction is RingORAM [28] with $O(\log N)$ bandwidth using a linear-sized client-stored position map. Online queries require two roundtrips for metadata reading and actual block downloading. We note it is possible to reduce the client memory by recursively storing the position map in smaller ORAMs; this approach increases both the roundtrips and the communication costs.

Hierarchical ORAMs, introduced by Ostrovsky [25], were mainly studied in the context of obtaining asymptotically small communication costs while keeping client memory constant. Indeed, this approach is used by the recent works obtaining logarithmic communication costs [1, 26]. Asharov *et al.* [2] showed that hierarchical ORAMs also yield practically efficient ORAMs by presenting FutORAMa. The authors showed that FutORAMa can obtain smaller communication costs than tree-based constructions without the need for large client-stored positions. However, FutORAMa requires more online roundtrips of 9-28 that may make queries slower in various network settings.

Finally, the last category is SQRT-ORAM, the original construction of Goldreich [13]. Zahur *et al.* [36] showed that SQRT-ORAM was surprisingly very concretely efficient despite being at the time about 30-year old. In particular, the online queries of SQRT-ORAM are extremely fast consisting of one roundtrip and only a single block of communication. Unfortunately, the amortized communication cost is $\sqrt{N}$ that is significantly larger than RingORAM and FutORAMa.

This leads to the following question:

> *Is it possible to obtain a concretely efficient construction that obtains the best of all three frameworks described above? In particular, can we obtain the fast online queries of* SQRT-ORAM *along with the superior communication costs of tree-based and hierarchical ORAMs without client-stored linear-sized position maps?*

Our Contribution. We answer the question in the affirmative by presenting two new constructions LatORAM and Lat^{2}ORAM that simultaneously obtain the best properties of all three types of ORAM frameworks. In particular, both constructions have blazing fast online queries that consist of a single roundtrip and one block of communication. Furthermore, the amortized communication costs of LatORAM and Lat^{2}ORAM are $O(N^{1/4})$ and $O(N^{1/6})$ with very small constants when the client has $O(\sqrt{N})$ client storage. Thus, our constructions have smaller amortized (concretely and asymptotic) cost than SQRT-ORAM. At first glance, these communication costs seem to be significantly larger than the $O(\log N)$ communication overhead of the tree-based RingORAM and hierarchical FutORAMa. It turns out that asymptotic analysis is not the best indicator for concrete efficiency. We show through empirical evaluation and analysis that both LatORAM and Lat^{2}ORAM have smaller or comparable communication costs while obtaining $1 + o(1)$ amortized round trips per query. In fact, this phenomenon was also observed previously in FutORAMa that consider shallow hierarchical ORAM constructions (using less recursive levels) with $O(N^{1/c})$ asymptotic cost for constant

$c > 1$, but obtain better concrete costs than asymptotically optimal ORAMs. We show our constructions have better concrete costs than these variants of FutORAMa as well.

A New Approach: Lateral Stash Growth. To obtain our new ORAM construction, we deviate from all three prior frameworks with a different approach of *lateral stash growth*. We use SQRT-ORAM as a conceptual starting point where the client stores $M = \Theta(\sqrt{N})$ blocks. At the start of SQRT-ORAM, the N real blocks are randomly shuffled along with M dummy blocks and stored in the server's container called Main. Each query downloads exactly one block (either the block sought or a dummy) from Main and thus, after M queries, the client's storage is full. SQRT-ORAM will then reshuffle the client's stash with the N blocks from Main that have not been touched by a query (i.e., they are *fresh*). This reshuffle costs $\Theta(N)$ communication yielding an amortized $\Theta(N/M) = \Theta(\sqrt{N})$ communication cost.

Our constructions will take a different approach. Instead, we leverage the key observation that shuffling small amounts of data that fit into client storage is much cheaper compared to shuffling larger amount of data. Following M queries, our constructions forego the large shuffle and, instead, upload a disjoint server container (i.e., a *lateral* stash) consisting of M blocks: the blocks in client storage as well as sufficiently many dummy blocks. Now, queries can continue as the client stash is empty. Each query will now access one block from each disjoint server container. By careful organization, we ensure that at most one real data block is queried and all others are pseudorandomly generated dummy blocks. Following the XOR technique [6], the server XORs all the accessed blocks and returns a single block that enables the client to retrieve the single real block by peeling off the pseudorandomly generated dummy blocks. Every time the client's storage is full, we can upload another server container (a *lateral stash*) up to P times, where P is a configurable parameter. As a result, we successfully postpone the large expensive shuffle involving Main and reduce the amortized communication cost without sacrificing the blazing efficient online queries of SQRT-ORAM. Furthermore, since stashes are not shuffled together, our constructions can select the $\leq P + 1$ blocks (one from each of the $\leq P$ containers plus one from Main) in a single roundtrip unlike hierarchical ORAMs that require one roundtrip per level to determine queries for the next level. The selected blocks can be xored together and sent as a single block to the client. In terms of position map, our schemes only require explicit positions for the at most PM blocks in lateral stashes (as the blocks in Main can be pseudorandomly permuted). For the settings of P and M that we use in this paper, we have $PM \in o(N)$, resulting in a much smaller position map than previous construction that had to explicitly store an $O(N)$ position map. We will describe the above construction, which we call LatORAM, in Section 4.

We can push our approach one step further. Specifically, after PM queries we do not shuffle back into Main but instead we create a *level-2* lateral shuffle that contains all the PM blocks and then we start again. Note that all subsequent queries will touch one block from each of the lateral stashes (of both levels) and from Main for a total of at most $P+2$ blocks and these can be selected in a single roundtrip, xored together, and sent to the client as one single block. We repeat this process for E times, where E is a configurable parameter, Specifically, following PM queries, we shuffle all the blocks into the level-2 lateral stash that will grow to have size EPM. After EPM queries, we shuffle all blocks into Main. The two-level construction described here, which we call Lat^{2}ORAM, will be formally presented and analyzed in Section 5. Lat^{2}ORAM will have a position map of size $O(EPM)$ and, by looking ahead, the parameters E and P that minimize the amortized communication for $M = \Theta(\sqrt{N})$ guarantee $EPM \in o(N)$ and thus Lat^{2}ORAM has a position map of size $o(N)$.

Clearly, the lateral-stash approach to ORAM design does not necessarily have to consider only two levels (and, for that matter, a second level with only one lateral stash) but it can be extended to any number of levels, with each level containing any number of lateral stashes of any size. As our main motivation is the design of practical ORAMs, we only present the two construction, LatORAM and Lat^{2}ORAM, that we consider to be the most practical and briefly discuss the general approach in Appendix A.

New Oblivious Shuffles. Of independent interest, our ORAM constructions rely on a variation of the notion of oblivious shuffling, that we call *oblivious merging*, in which the input consists of two disjoint server-stored containers C_1 and C_2 that are assumed to be randomly shuffled individually. We present a new algorithm that merges C_1 and C_2 into a single randomly shuffled container using only approximately $2(|C_1| + |C_2|)$ blocks of bandwidth improving by a multiplicative factor of two over prior shuffling algorithms [24, 27]

that use $4(|C_1| + |C_2|)$ bandwidth.

2 Preliminaries and Basic Tools

For natural numbers $m < n$, we set $[m, n] = \{m, m+1, \ldots, n\}$. We let $[n] = [1, n] = \{1, 2, \ldots, n\}$. For a list L, we denote $L[i]$ as the i-th entry in L where $i \in [|L|]$, and $L[m : n]$ to denote a sublist from indices m to n, inclusive. We denote $x \| y$ as the string concatenation of x and y.

We use symmetric IND-CPA encryption $\mathcal{E} = (\mathsf{Enc}, \mathsf{Dec})$ where each ciphertext is indistinguishable from a random string. We also use pseudorandom functions (PRFs) F such that the output $F(K, x)$ is indistinguishable from a random string for random private key K and any input string x. Both encryption and PRFs are used in all practically efficient ORAMs to date.

For algorithms A requiring joint execution between the client and the server, we will denote $A(X; Y)$ where X is the server's input and Y is the client's input.

2.1 Oblivious RAMs

In this section, we discuss the notion of an Oblivious RAM [13, 14] starting from the notion of a RAM simulation and then formalizing the security notion, obliviousness, we are interested in. We refer the reader to [26, 1] for formal definitions.

We consider a RAM machine $\mathcal{M}$ that performs operations on a *memory* $\mathtt{mem}$ consisting of N *blocks* each consisting of b bytes. Each operation is described by a triplet $O = (\mathtt{op}, \mathtt{addr}, \mathtt{data})$. If $\mathtt{op} = \mathtt{Read}$ then $\mathtt{data} = \perp$ and the operation returns $\mathtt{mem}[\mathtt{addr}]$. If $\mathtt{op} = \mathtt{Write}$ then $\mathtt{data} \in \{0,1\}^{8 \cdot b}$ and it has the effect of setting $\mathtt{mem}[\mathtt{addr}] = \mathtt{data}$.

A *simulator* for $\mathcal{M}$ is a two-party protocol $(\mathsf{Clt}, \mathsf{Srv})$ in which a client Clt with a local Stash that can hold up to M blocks wants to execute $\mathcal{M}$ and it is helped by a server Srv that has its own memory. We are interested in *online* simulation of $\mathcal{M}$ in which Clt receives the correct result of each operation (as defined above) before the next operation is disclosed. We consider the case in which $M \ll N$ (and thus Clt is unable to excute $\mathcal{M}$ by itself) but Srv is not trusted to see the sequence of operations executed by $\mathcal{M}$. We want to design, that is, *Oblivious* RAM simulators (*ORAMs*, in short).

More formally, we consider any honest-but-curious PPT adversarial server $\mathcal{S}$ that interacts with a client and adaptively selects the next operation to be performed by the client. Obliviousness is formalized by means of two games: the *real* game $\mathsf{Real}^{\mathsf{Clt},\mathcal{S}}(1^N, 1^M)$ and the the *ideal* game $\mathsf{Ideal}^{\mathsf{Sim},\mathcal{S}}(1^N, 1^M)$. In both games the two parties receive the size N of $\mathtt{mem}$ and the size M of the client memory, and $\mathcal{S}$ outputs one bit at the end of the game. In the real game, the client is impersonated by the honest client Clt and every time $\mathcal{S}$ outputs the next operation to be executed, it is passed as an input to Clt that then engages in an interaction with $\mathcal{S}$. In the ideal game, instead, $\mathcal{S}$ interacts with a simulator Sim that does not receive the operations output by $\mathcal{S}$ but only a special message $\mathtt{next}$ that signals that a new operation must be executed. Obliviousness requires that $\mathcal{S}$ cannot distinguish the two games.

Definition 1 (Oblivious Simulation)**.** *We say that* $(\mathsf{Clt}, \mathsf{Srv})$ *is an* Oblivious RAM (ORAM) *if*

1. **Correctness.** *For every sequence of operations executed by* $(\mathsf{Clt}, \mathsf{Srv})$*,* Clt *outputs the correct value for every operation of the sequence, except with negligible probability.*
2. **Obliviousness.** *For every honest-but-curious PPT adversarial server* $\mathcal{S}$*, there exists a simulator* $\mathsf{Sim}_{\mathcal{S}}$ *and a negligible function* negl *such that*

$$\left|\text{Prob}\left[\mathsf{Real}^{\mathsf{Clt},\mathcal{S}}(1^N, 1^M) = 1\right] - \text{Prob}\left[\mathsf{Ideal}^{\mathsf{Sim}_{\mathcal{S}},\mathcal{S}}(1^N, 1^M) = 1\right]\right| \leq \mathsf{negl}(N).$$

As a note, we stress that we only consider honest-but-curious (semi-honest) adversaries. This is identical to the prior ORAM works that we compare against, including $\mathsf{RingORAM}$ and $\mathsf{FutORAMa}$. Also, we stress that in our definition the adversarial server $\mathcal{S}$ takes part to the protocol and thus sees all messages exchanged

with Clt (or $\mathsf{Sim}_{\mathcal{S}}$, depending on the game) as well as all memory locations accessed by the honest server Srv. This is slightly different from other definitions in which it is the honest server Srv to interact with Clt (or Sim) and $\mathcal{S}$ sees the access pattern of Srv to its own memory.

There are five main efficiency measures for an ORAM. First, the amortized communication overhead, that is the amount of client-server communication required for each operation amortized over a sequence of operations. Secondly, the amortized computation required across both the client and server for each operation. Next, the round complexity concerning the number of rounds between the client and server per operation. In general, we will exclusively focus on minimizing online rounds. The last two measures are the storage sizes of the client and server respectively required to jointly store $\mathtt{mem}$. In our work, we consider the worst-case (largest) storage required at any point in time during the execution of an operational sequence.

2.2 Server Storage

In our constructions, server storage is partitioned into disjoint *containers* (similar to levels in hierarchical ORAMs, but generated differently in our work), and each container will be given a unique label ℓ. A container stores two types of blocks: the *real* blocks, that are encryptions of one of the input blocks $B_1, \ldots, B_N$ of $\mathtt{mem}$, and *dummy* blocks, whose content is pseudo-randomly generated. All blocks (both real and dummy) will be tagged with some public label $\mathtt{lab}$ that is also pseudorandomly generated and thus a block is pair $(\mathtt{lab}, \mathsf{blk})$. We say that a block is *fresh* if it was uploaded to the server and never accessed during any ORAM operation. After a block is accessed, it ceases to be fresh and will never be accessed again. Every non-fresh block may be immediately deleted by the server after the operation is completed. All our constructions, will access server stored blocks within a specific container by specifying its public label $\mathtt{lab}$.

The label $\mathtt{lab}$ of a real block carrying block $B_{\mathtt{idx}}$ is generated pseudorandomly as a function of ℓ, the label of the container where the block is found, and the identifier $\mathtt{idx}$ of the real block; the second component blk is an encryption of $B_{\mathtt{idx}}$ with an IND-CPA encryption scheme whose key is private to Clt. The dummy blocks $(\mathtt{lab}, \mathsf{blk})$ of container ℓ should be indistinguishable from real blocks. More specifically, like real blocks each dummy block has a label $\mathtt{lab}$ that is derived pseudorandomly from the container label ℓ and the unique identifier $\mathtt{idx}$ within the container prefixed with the string $\mathtt{dmy}$ (so to distinguish it from the real block with the same identifier $\mathtt{idx}$). The second component blk instead is pseudorandomly derived from $\mathtt{lab}$ using PRF F_D and randomly generated seed K_D. This allows to deterministically derive the blk component of a dummy block given only its label (and the seed); this property is crucial for the success of the XOR technique (see below). Moreover, it is possible to check if $(\mathtt{lab}, \mathsf{blk})$ is a dummy block by checking if $F_D(K_D, \mathtt{lab}) = \mathsf{blk}$.

2.3 Client Storage

In most practical ORAM constructions, including ours, the client secure storage may be split into three distinct sections. First, the *stash*, Stash, is used to securely store up to M blocks. M is typically a parameter of constructions and much smaller than the memory size N. The next component is the *position map*, Map, that allows the client to locate queried blocks in server storage. In our constructions, Map associates the index $\mathtt{idx} \in [N]$ of a block to the container ℓ_c holds the current copy of the block. To actually address the block in the container its label is pseudorandomly derived from $\mathtt{idx}$ and ℓ_c. Typically, the size of Map depends on the memory size N (such as $O(N)$ in [28] or $O(\sqrt{N})$ in [36]). Finally, the client also stores $O(1)$ cryptographic keys for encryption and PRFs that are much smaller than Stash and Map. The client storage size will be the sum of all three parts. In our work, we commonly parameterize client storage as M blocks of local memory to mean the stash size is M as the stash typically dominates the client storage size.

2.4 The XOR Technique

We use the XOR technique [6] to reduce communication costs. In many constructions (including ours), a query will access l blocks on the server with the restriction that at most one block is real. and at least $l-1$ blocks are dummy blocks. The client's goal is to download the single real block, but retrieves the remaining $l-1$ for security. One way to do this is for the client to simply download all l blocks. If the

dummy blocks are pseudorandomly generated by a PRF, it is possible to reduce communication, by having the server compute the XOR of all l blocks and return the resulting block to the client. If all blocks are dummy, then the block received can be ignored. If instead exactly one of the l blocks is real, the client can deterministically re-generate the content of the dummy blocks locally and it can XOR off the dummy blocks to obtain the single real block. In this way we have reduced communication from l blocks to a single block.

3 Shuffling and Merging

Our ORAM constructions rely on two special two-party protocols called oblivious Shuffling and oblivious Merging. In both problems, the common input consists of a set of N encrypted blocks of which some subset (or all) may be stored in trusted client storage. Furthermore, some of these blocks might be dummies. In both problems we suppose the client has secure storage of M blocks.

In oblivious shuffling the target is to obliviously shuffle the encrypted blocks to a permutation hidden from the server so that the server cannot link uploaded blocks to input blocks (i.e., obliviously). As a note, there already exist algorithms such as CacheShuffle [27] that perform this task using $4N$ communication when $M = O(\sqrt{N})$. We present more efficient algorithms for specific settings arising in our ORAM constructions.

In oblivious merging instead the blocks are partitioned into a certain number c of containers and the blocks within each container are already randomly shuffled. The goal here is to merge the blocks into one output container so that the server cannot link a block in the output container as originating from a specific input container. Note that any oblivious shuffling algorithm can also be used for merging: simply concatenate the c containers into one container and then perform oblivious shuffling. On the other hand, oblivious shuffling is the special case of oblivious merging in which each container has exactly one block.

Public Random Labels. In addition to obliviousness and random shuffling/merging, we require an additional property of public label tagging necessary for our ORAM constructions. At the end of the shuffle protocol, each block (real or dummy) stored on the server must be tagged with some public label that will be used to identify and access server-stored blocks later. As explained above, the labels will be PRF outputs with a client-stored private key and, thus, reveal nothing about the associated encrypted block. Furthermore, each time a block is shuffled, a new and independent public label is generated to ensure the server cannot link blocks across multiple shuffles, but that is the responsibility of the caller.

3.1 Shuffling Algorithms

We denote a shuffle algorithm A in the form of $Z \leftarrow A(X; Y)$ where X are server stored input blocks and Y are the client stored input blocks. Typically, the client's input also includes private keys for encryption and PRF. In our work, we focus on the problem of obliviously shuffling data to a (pseudo)randomly generated permutation that should remain unknown to the server. This is an easier task than prior works that obliviously shuffle to an input permutation, but it suffices for our work.

We will assume the following in all shuffle algorithms. Every time a block is downloaded, it will be decrypted by the client. Furthermore, every time a block is uploaded, it will be re-encrypted by the client. In this way, even if the same block is uploaded multiple times, the server will observe a uniformly random string each time. Finally, each uploaded block will contain encrypted metadata containing a bit of whether the block is dummy or real as well as the block's index (if real).

CacheShuffle. Let us now briefly review the oblivious shuffling protocol CacheShuffle [27] that obliviously shuffles N blocks using client storage $M = \Theta(\sqrt{N})$ and communication $\approx 4N$. CacheShuffle receives server-stored blocks X, encryption key K_E, label key K_L to associate all uploaded blocks with labels and dummy key for generating dummy block contents. CacheShuffle is used by our ORAM constructions and as a subroutine of our oblivious merging algorithm (see Section 3.3).

We next give an overview of a slightly adapted version of the original algorithm, which is better suited to our overall constructions, and we refer to the full description and analysis in Appendix C. At a high level, CacheShuffle models the configuration of the blocks as occupying entries within an $r \times c$ matrix where $r \cdot c = N$. We can view the overall goal as obliviously arranging the blocks into a uniform random permutation

Algorithm 1 SmallShuffle$(\perp; Y, I, K_E, K_L, K_D)$

Input: Y, I, K_E, K_L, K_D: client blocks, dummy indices, encryption key, and PRF seeds for labels and dummies.
Output: Z: server-stored output.

```
while |Y| + |I| > 0 do
    Generate random x from [|Y| + |I|].
    if x < |Y| then                                              ▷ Real block
        Extract random (idx, B_idx) from Y.
        Set lab ← F_L(K_L, idx).
        Set blk ← Enc(K_E, B_idx).
    else                                                         ▷ Dummy block
        Randomly select and remove idx from I.
        Set lab ← F_L(K_L, dmy ‖ idx).
        Set blk ← F_D(K_D, lab).
    Upload (lab, blk) to server.
```

across all matrix positions. CacheShuffle accomplishes this goal in two distinct phases: Spray and Recalibrate. The initial Spray phase aims to distribute the blocks randomly among the different rows, leaving the intra-row ordering undefined. Then, the Recalibrate phase performs an independent random shuffle of the blocks contained *within* each row. Together, these operations result in a final block arrangement corresponding to a uniformly random permutation. By setting $r \approx c \approx \sqrt{N}$, CacheShuffle obtains $\approx 4N$ communication overhead using $M = \Theta(\sqrt{N})$ client storage.

3.2 Client-Stored Input Blocks

In this section we look at the simplest, but arguably the most important, oblivious shuffling setting where all the input blocks are client-stored.

Description. SmallShuffle takes as input a list Y of M pairs of real blocks with their index $(\texttt{idx}, B_{\texttt{idx}})$. The pairs are stored on the client already and there will be no blocks originating on the server. The goal is to obliviously upload the M blocks received in input along with D extra dummy blocks and store them on the server. SmallShuffle takes as additional inputs encryption key K_E, that will be used to encrypt the real blocks, PRF seed K_L for F_L, that will be used to generate the labels of the real and the dummy blocks, and PRF seed K_D for F_D, that will be used to generate the dummy blocks. Moreover, SmallShuffle take as input a set I of D indices that will be used to generate the labels and the contents of the dummy blocks. See Algorithm 1 for the formal pseudocode.

Obliviousness. SmallShuffle is oblivious since the server simply sees exactly $M + D$ blocks being uploaded that is independent of the random generated permutation.

Efficiency. SmallShuffle is optimal in terms of bandwidth as it uploads exactly $N = M + D$ blocks. One cannot aim for any more efficient shuffling algorithm in terms of bandwidth. Note, the client should upload blocks in batches to avoid too many roundtrips between the client and server. In terms of client storage, we note SmallShuffle requires storing $O(M)$ blocks as well as $O(D)$ indices to keep track of used dummy blocks.

3.3 Obliviously Merging Two Containers

In this section, we consider the oblivious merging of $c = 2$ containers, C_1, C_2, for a total of $N = |C_1| + |C_2|$ blocks. The goal is to obliviously merge the two containers into a single container that is randomly shuffled. Our algorithm assumes no dummies need to be added. Instead, the input containers will contain dummies that need to be re-generated and shuffled in the output. We will use CacheShuffle as a sub-routine.

First Failed Attempt. We present an algorithm for this task with a shortcoming that we will fix later. The algorithm will proceed in $r = (|C_1| + |C_2|)/M$ rounds with each round consisting of downloading and uploading M blocks, where M is the number of blocks that can stored in client's stash. The M downloaded

blocks for each round are chosen according to the sizes of C_1 and C_2 where we download $M \cdot |C_1|/N$ blocks from C_1 and $M \cdot |C_2|/N$ blocks from C_2. In other words, we download $|C_1|/N$ fraction of the M blocks from C_1 and the remaining $|C_2|/N = 1 - (|C_1|/N)$ fraction from C_2. For each block that we need to upload, we flip a biased coin that also depends on the sizes of C_1 and C_2. We upload a block that was previously downloaded from C_1 with probability $|C_1|/N$. With the remaining $|C_2|/N$ probability, we upload a block from C_2. We repeat this M times in each round for uploading.

In the above, we ignore rounding for simplicity (but we handle this in our formal algorithm). We note that there is some additional care needed to ensure a random permutation (that we will show later). However, the bigger problem is that the algorithm fails if the client attempts to upload a block from either C_1 or C_2, but the client's storage contains no such block. In fact, this algorithm will almost certainly fail in the first round.

TwoMerge **Description.** We present our algorithm TwoMerge that fixes the problems above using the following observation. Even if the above algorithm fails, it is unlikely that the number of missing blocks at any point is very large. If we had a small cache of some additional blocks that could act as either blocks from C_1 or C_2, they could be used to avoid the algorithm from failing to upload blocks. To do this, we will pick two parameters $q_1 \geq 1$ and $q_2 \geq 1$ denoting extra blocks for each round. The server will pick and remove $q_1 \cdot r$ blocks from C_1 as well as $q_2 \cdot r$ blocks from C_2. Afterwards, the client and server execute a standard oblivious shuffle (such as CacheShuffle) to randomly permute all $(q_1 + q_2) \cdot r$ blocks into a single container denoted by SAux stored on the server. At this point, each of the block could either originate from C_1 or C_2. Due to the oblivious shuffle, the knowledge to link each block in SAux back to either C_1 or C_2 has been lost from the server's perspective.

Now, we modify each round as follows. Previously, we downloaded $M \cdot |C_1|/N$ blocks from C_1 and $M \cdot |C_2|/N$ blocks from C_2. We modify the downloads such that we replace q_1 downloads from C_1 and q_2 downloads from C_2 with blocks downloaded from SAux. In other words, we download $M \cdot |C_1|/N - q_1$ blocks from C_1, $M \cdot |C_2|/N - q_2$ blocks from C_2 and $q_1 + q_2$ blocks from SAux. We note that each of the $q_1 + q_2$ blocks downloaded from SAux could be from either C_1 or C_2.

Uploads proceed with the same idea as before, but we specify an algorithm to ensure a random permutation. The algorithm keeps two counters L_1 and L_2 that track blocks remaining in the two containers not yet uploaded to the server. Afterwards, we pick a uniformly random number r from $[L_1 + L_2]$. If $L_1 \leq r$, we pick a block from C_1 to upload. Otherwise, we pick a block from C_2 to upload. Finally, we decrement the counter (L_1 or L_2) depending on whether the uploaded block is from C_1 or C_2. To compute the largest index i, we utilize prefix sum data structures with binary search. This guarantees that the output is a uniformly random permutation. We denote this as the MultiPermDS data structure and the pseudocode is presented in Algorithm 2 generalized to an arbitrary number of containers $c \geq 2$. The MultiPermDS data structure uses prefix sum data structure PrefixSumDS that can be instantiated in many ways such as using Fenwick trees [10] as we do in our implementations.

We present the pseudocode of TwoMerge in Algorithm 3 that takes as parameter q_1 and q_2. The server receives two containers as input, C_1 and C_2, that are assumed to be randomly shuffled already. The client receives encryption key K_E, dummy PRF key K_D, new label PRF key $K_L^{\mathtt{new}}$ and the set I of indices to be used for the dummy blocks. Note that TwoMerge does not introduce new dummies but rather it produces an output that contains the same number of dummies of the c input containers. The dummies must be refreshed meaning it is assigned a random index from I and the label is computed by using the new PRF key $K_L^{\mathtt{new}}$. Recall one can use K_D to check if $(\mathtt{lab}, \mathsf{blk})$ is a dummy or a real block.

Obliviousness. We sketch the obliviousness of TwoMerge. To initialize underflows, the client and server execute CacheShuffle that is oblivious. The obliviousness of TwoMerge in the remaining part follows from the fact that, if the algorithm does not abort, the access patterns is deterministic: in each round, the client downloads a fixed number of blocks from each container and uploads exactly M blocks.

Abort Analysis. We prove that TwoMerge aborts with negligible probability as long as $q_1 = q_2 = O(\sqrt{M \ln N})$ where $N = |C_1| + |C_2|$. See the proof in Appendix D.

Theorem 1. *If* $q_i \geq \sqrt{2|C_i|M \ln(2N/(\delta \cdot M))/N}$ *for* $i \in [2]$*, then* TwoMerge *aborts with probability at most* δ*.*

Algorithm 2 MultiPermDS

Input: L: List of the form $L = (L_1, \ldots, L_c)$ indicating that the frequency of container i is L_i.
Output: Initialized data structure.

Setup(K, L) :
Generate random PRF key K.
Initialize DS $\leftarrow$ PrefixSumDS.Setup(L).
Initialize counter $i \leftarrow 1$.

Output: Next element in the permutation.

Next() :
Compute $r \leftarrow F(K, i)$.
Find largest $z \in [c]$ such that DS.PrefixSum$(z) < r$ using binary search over z.
Execute DS.Update$(z, -1)$.
Increment counter $i \leftarrow i + 1$.
Output z.

Therefore, setting $q_i = \Omega(\sqrt{M \cdot (\log(N/M) + \lambda)})$ suffices to obtain $2^{-\lambda}$ failure probability.

Efficiency. From the above analysis, the total number of underflow blocks is $U = N/M \cdot (q_1 + q_2)$. As $q_1 + q_2 = o(M)$ assuming standard sizes of stash such as $M = O(\sqrt{N})$, we know that the underflow blocks are $U = N/M \cdot o(M) = o(N)$. Underflow blocks are uploaded and downloaded twice costing approximately $4U$. All other $N - U$ blocks are uploaded and downloaded exactly once costing $2(N - U)$. Altogether, the cost of TwoMerge is $2N + 2U = 2N + o(N)$ communication. This is a factor of two improvement over prior algorithms [24, 27] including CacheShuffle that could have been applied directly.

Generalization to More Containers. We note that TwoMerge may be adapted to more than two containers that are each independently shuffled. The core idea would remain the same but the uploads and downloads must be generalized to more than two containers. We do not present this generalization as it is not useful to our ORAM constructions, but we note this possible modification as it could be useful for other future works.

4 ORAM from Delayed Merging

We quickly overview SQRT-ORAM [13] for N blocks and client storage of M blocks. The server memory consists of one container, Main, containing $N + M$ randomly permuted and labeled blocks: N real blocks and M dummy blocks. Each of the $N + M$ blocks has a public label generated using a PRF F_L with a client-stored key K_L. In particular, the N real blocks are labeled by $F_L(K_L, i)$ where $i \in [N]$. Similarly, the M dummy blocks are labeled by $F_L(K_L, j)$ where $j \in [N + 1, N + M]$. Real block contents are encryptions of the b-byte data blocks whereas dummy block contents are generated using another PRF $F_D(K_D, j)$ whose outputs are b-byte long. During initialization, the $N + M$ blocks are obliviously permuted and stored in Main on the server.

SQRT-ORAM consists of a single phase of M queries. The client stash with capacity M is initially empty. Suppose that the q-th query is for block B_{i_q}, for $q \in [M]$. The client checks if B_{i_q} is present in the stash. If not, the client requests the block with label $F_L(K_L, i_q)$ of Main from the server. The block returned by the server is decrypted and the resulting real block is stored in the stash. If B_{i_q} is found in the stash, the client asks for the q-th dummy block, that is the block of Main with label $F_L(K_L, N + q)$. The dummy block is downloaded and immediately discarded as the client already has the desired block.

After M queries when the client stash has potentially reached its capacity, SQRT-ORAM restores the initial state by performing a merging of the N fresh (unaccessed) blocks remaining in Main and the at most M real blocks stored in stash. Plugging in the best algorithm for this setting (that happens to be TwoMerge in Section 3), $2N + M$ bandwidth is required. As the total communication cost is $2N + 2M$ over the M queries, we obtain amortized cost of $2N/M + 2$ blocks per query.

Algorithm 3 $\mathsf{TwoMerge}_{q_1,q_2}(C_1, C_2; K_E, K_D, I, K_L^{\mathtt{new}})$

Input: Server-stored input containers, dummy indices, encryption key and PRF keys for labels and dummies.
Output: Z: server-stored output.

Initialize Underflow Store (executed by client and server):

Initialize additional server stores T_1, T_2.
Compute number of rounds $r \leftarrow \lfloor (|C_1| + |C_2|)/M \rfloor$.
Server moves $q_1 \cdot r$ blocks from C_1 into T_1.
Server moves $q_2 \cdot r$ blocks from C_2 into T_2.
Randomly select $K'_L \leftarrow \{0,1\}^\lambda$ and set $I \leftarrow [q_1 + q_2]$.
Execute $\mathsf{SAux} \leftarrow \mathsf{CacheShuffle}(T_1, T_2; I, K_E, K_D, K'_L)$.

Execute Rounds (executed by client and server):

Initialize $\mathsf{DS} \leftarrow \mathsf{MultiPermDS.Setup}(|C_1|, |C_2|)$.
Set $\ell_1 \leftarrow \lfloor |C_1| \cdot M/(|C_1| + |C_2|) \rfloor - q_1$.
Set $\ell_2 \leftarrow M - \ell_1 - q_1 - q_2$.
Initialize client stores S_1, S_2, CAux and upload list A.
for $i \in [r]$ **do**
  Download ℓ_1 next entries from C_1 into S_1.
  Download ℓ_2 next entries from C_2 into S_2.
  Download $q_1 + q_2$ entries from SAux into CAux.
  for $j \in [M]$ **do**
    Compute $z \leftarrow \mathsf{DS.Next}()$.
    if $|S_z| > 0$ **then**
      Remove labeled block $(\mathtt{lab}, \mathsf{blk})$ from S_z.
    else if $|\mathsf{CAux}| > 0$ **then**
      Remove block $(\mathtt{lab}, \mathsf{blk})$ from CAux
    else
      Abort and terminate algorithm.
    if blk is dummy **then** ▷ Dummy block
      Randomly select and remove $\mathtt{idx}$ from I.
      Set $\mathtt{lab} \leftarrow F_L(K_L^{\mathtt{new}}, \mathtt{dmy} \| \mathtt{idx})$.
      Set $\mathsf{blk} \leftarrow F_D(K_D, \mathtt{lab})$.
    else ▷ Real block
      Set $(i, B_i) \leftarrow \mathsf{Dec}(K_E, \mathsf{blk})$.
      Set $\mathtt{lab} \leftarrow F(K_L, i)$ and $\mathsf{blk} \leftarrow \mathsf{Enc}(K_E, i \| B_i)$.
    Append $(\mathtt{lab}, \mathsf{blk})$ to A.
  Upload A to the server and re-initialize empty A.
Download the remaining $\leq M$ blocks from C_1 and C_2.
Client locally shuffles and uploads remaining blocks.
Server concatenates all uploaded blocks to obtain Z.

For obliviousness, each query in SQRT-ORAM downloads exactly one new block from Main. In other words, the server sees a random subset of M blocks, each being accessed once. As this access pattern is independent of the ORAM operations, we obtain obliviousness.

Bottleneck. Observe that communication cost of SQRT-ORAM is dominated by the merge cost $2N + M$. If we could perform more queries between two shuffles, we would reduce the amortized cost. Unfortunately, SQRT-ORAM must stop after M queries because no space is available in the stash (if M distinct blocks are queried). The stash is the limiting factor of SQRT-ORAM and a larger stash would allow for more queries and thus the contribution of the merge to the amortized cost of a query would decrease. Client memory, however, is a severely limited resource and thus if we want to extend the stash size, we must find a different approach.

4.1 Idea: Delaying Large Merges

We present our main idea of *lateral stash growth* that extends the client storage capacity by critically leveraging server storage. With this approach, we will present our first construction LatORAM.

After performing M queries, LatORAM deviates from SQRT-ORAM using the following core observation. If we merge the stash of M blocks with the $O(N)$ blocks on the server like SQRT-ORAM does, the cost will be $\Omega(N)$. In most practical applications, the stash is actually much smaller, $M \ll N$ such as $M = O(\sqrt{N})$. Instead, we want to delay the large shuffle of mixing many blocks to avoid the $\Omega(N)$ cost. To do this, we instead only randomly shuffle the M blocks in the stash and outsource them to server storage as a new, disjoint lateral stash denoted by $\mathsf{LatStash}_1$. We will also add some number of dummy blocks D that we will specify later. As the M blocks are client-stored, we can obliviously shuffle using $M + D$ communication with SmallShuffle (see Section 3).

After this, the server's storage consists of two disjoint containers. We denote the original container as Main and the new container as $\mathsf{LatStash}_1$. For the next query, it is possible the block is stored in either container. To maintain obliviousness, each of the next M queries will access exactly one block from each of the two disjoint containers (see Figure 1). Furthermore, each query accesses a previously unaccessed (fresh) block. This is similar to SQRT-ORAM that only accesses previously unaccessed blocks.

Our construction LatORAM will generalize this approach beyond just two disjoint containers. LatORAM is parameterized by the number of *phases* P each consisting of M queries that will construct another lateral stash at the phase's completion. Parameter M is chosen as the size of the client stash size in blocks. At the beginning, the initial server storage Main contains N real blocks along with $P \cdot M$ dummy blocks. The first phase of M queries proceeds identically as SQRT-ORAM up to the merging algorithm. Instead of merging the M stash blocks with blocks in Main, LatORAM proceeds by shuffling the M stash blocks into a disjoint server container $\mathsf{LatStash}_1$ along with $(P-1) \cdot M$ dummy blocks. More generally, after the M queries of i-th phase are completed, the M stash blocks are shuffled into container $\mathsf{LatStash}_i$ with $(P-i) \cdot M$ dummy blocks for all $i \in [P]$. Finally, the client will also maintain a position map that will map previously queried real blocks to their container after the shuffle at the end of each phase. Note, the position map will only contain information about real blocks not stored in Main and that were queried in a previous phase. Therefore it has size $O(P \cdot M)$ that, for the values of P used in our ORAM constructions, will be $o(N)$.

Consider any query in the i-th phase where the server has i disjoint containers: Main, $\mathsf{LatStash}_1$, ..., $\mathsf{LatStash}_{i-1}$. First, the client can quickly determine the location of the queried block. It is either in the client stash, one of the $i-1$ lateral stashes (using information from the position map) or Main. By careful tracking and labeling of dummy and real blocks, the client accesses exactly one block from each disjoint container. Additionally, it can be guaranteed that the client will access at most one real block (the queried block) and the remaining accessed blocks will be dummies. We note the query algorithm requires a single roundtrip as the client may determine the location of the queried real block using information exclusively stored on the client. With this information, the client can determine whether to access a real or dummy block from each disjoint container. We refer readers to our formal description of LatORAM in Section 4.2 for more details on the query algorithm.

As the above query algorithm guarantees that at most one real block is downloaded from the server, we

also leverage the XOR technique (see Section 2). The server may return the XOR of all accessed blocks and return the result. If the XOR contains a real block, the client may pseudorandomly generate the dummy block's contents and XOR them off the server's response to obtain the encrypted real block. Now, we guarantee that each query requires only downloading a single block.

To see the benefit of our lateral stash approach, we note that it achieves our desired goals. First, online queries remain efficient with one roundtrip and one block of communication like SQRT-ORAM. On the other hand, we delayed the expensive full merge of Main and stash to reduce amortized communication costs. In SQRT-ORAM, the $\Omega(N)$ merge was performed every M queries where $O(M)$ is client storage size. Instead, our construction delays the merge with Main to be every $P \cdot M$ queries at the minimal cost of uploading the $O(M)$ client stash and $O(P \cdot M)$ dummies at the end of each of the P phases. By carefully choosing P, LatORAM requires smaller amortized communication cost than SQRT-ORAM and prior works while maintaining the very efficient online queries.

We note that in our approach, new server containers are grown in a "lateral" fashion and sit next to each other and never exchange blocks. Our LatStash's only receive blocks from the stash. This is markedly different from the hierarchical approach in which new containers are created "hierarchically" by moving blocks from "lower" containers to "higher" containers. Having the possibility of randomly shuffling blocks in the stash makes the oblivious creation of a new container extremely efficient. On the other hand, the hierarchical approach needs to move blocks from one container to the other under the eyes of the adversarial server, which makes obliviousness more expensive to achieve. This is until the end of the P phases, when all server containers are merged into Main to re-establish the initial conditions. However the $O(N)$ cost of this final merge is amortized over the large number of queries that have taken place.

4.2 First Construction: LatORAM

In this section, we formally present LatORAM, our first construction using the lazy delaying of large shuffles in the framework of lateral stash growth outlined in Section 4.1. Formally, LatORAM is configurable by two parameters: the number of phases, P, and the number of blocks in client storage, M. The typical approach to choose parameters is to fix the client storage size M by the application's requirements and optimize P to minimize amortized communication over a *cycle* of PM queries. The pseudocode is presented in Algorithms 4, 5, 6 and 7. Note that LatORAM uses server containers Main and $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$.

Cryptographic Primitives. We use IND-CPA encryption scheme $\mathcal{E} = (\mathsf{Enc}, \mathsf{Dec})$ with pseudorandom ciphertexts and PRFs F, F_L and F_D to generate seeds, labels and dummy block contents, respectively. The client will store private keys K_E for encryption as well as PRF keys K_L and K_D. PRF F is used to generate the seeds for F_L to be used for each container. Specifically, the labels are generated using function F_L with seed $K_L^i \leftarrow F(K_L, i)$, for container $\mathsf{LatStash}_i$ and $i \in [P]$, and seed $K_L^0 \leftarrow F(K_L, 0)$ for Main.

Setup. In the setup, we assume that the client and server initialize the ORAM with each of the N blocks in `mem` being an arbitrary b-bit string (such as 0^b). The client will simply upload $N + D$ blocks in random order. These blocks consist of the N *real* blocks with index $\texttt{idx} \in [N]$ and initialized $\mathsf{blk} = \mathsf{Enc}(K_E, \texttt{idx} \| 0^b)$ with label with $F_L(K_L^0, \texttt{idx})$. We remind the reader that $K_L^0 = F(K_L, 0)$. Additionally, there will be $D = P \cdot M$ *dummy* blocks whose contents are generated using $F_D(K_D^0, \texttt{idx})$ for $\texttt{idx} \in [D]$ and label $\texttt{lab}F_L(K_L^0, \texttt{dmy} \| \texttt{idx})$. All N real and D dummy blocks will be stored in a random order (oblivious to the server) in Main. In our work, we will refer to blocks as *fresh* if they have not been accessed previously. So, all blocks after setup are considered fresh.

In some cases, you may want to setup the ORAM with some input memory `mem`. If we assume the client provides `mem` and it is stored in client memory already, the same algorithm works by replacing 0^b with $\texttt{mem}[\pi(i)]$ when uploading. In the case that `mem` is already stored in an encrypted manner on the server and client storage is restricted, then one must use the standard oblivious shuffling algorithms such as [24, 27] for setup.

Client State. The client stores the keys from setup: K_E, K_L and K_D. The client also stores counters for the phase, $0 \le c_p \le P$, and the number of queries performed in the c_p-th phase, $0 \le c_q \le M$. After each query, c_q is incremented. After M queries within a phase, we increment the phase counter c_p and reset the

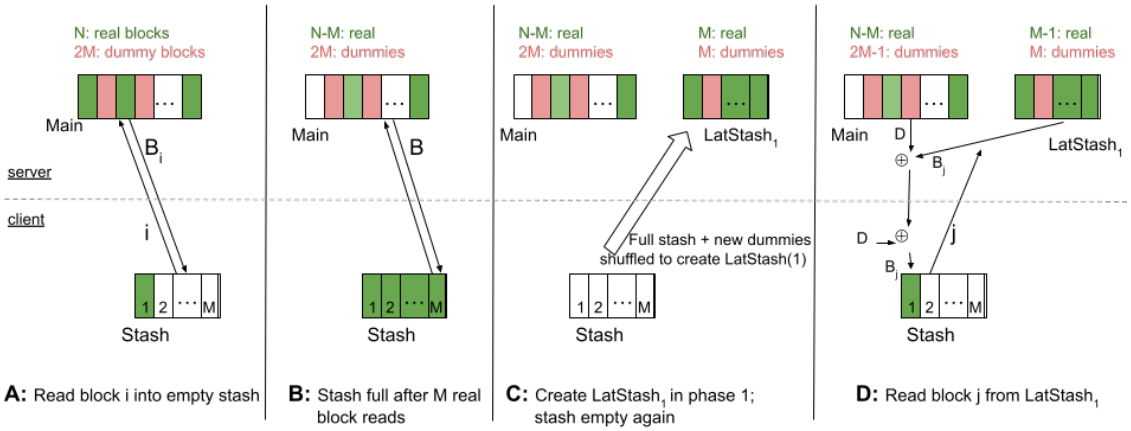

Figure 1: LatORAM with two phases ($P = 2$).

query counter c_q. If the client's phase counter is c_p, the server's storage consists of Main, $\mathsf{LatStash}_1$, ..., $\mathsf{LatStash}_{c_p-1}$. Additionally, the client stores a Stash of at most M real blocks consisting of the previously c_q queried blocks within this phase. The client also stores a position map Map that maps real block indices $q \in [N]$ to its corresponding container (if it not stored in the Stash or Main). If $\mathsf{Map}[q] = i$, then block B_q is stored in $\mathsf{LatStash}_i$. If $q \notin \mathsf{Map}$ and $q \notin \mathsf{Stash}$, then B_q is stored in Main. Note, the client can always find the container storing block B_q by using Map and Stash. Finally, the client also stores a map of counters c_d where $c_d[i]$ denotes the next dummy block that may be used in $\mathsf{LatStash}_i$. The dummy counter for Main will be $c_d[0]$.

Query: Client Request. To query index q, the client first looks at Stash and Map to determine the location of the desired block. If the $B_q \in \mathsf{Stash}$, then the client already has block B_q and will only access dummy blocks from the server. If $q \in \mathsf{Map}$, then we know that B_q is stored in $\mathsf{LatStash}_{\mathsf{Map}[q]}$. We will access block B_q from $\mathsf{LatStash}\mathsf{Map}[q]$ and access dummy blocks in all other containers. Finally, if $B_q \notin \mathsf{Stash}$ and $q \notin \mathsf{Map}$, then we know that B_q is stored in Main. We access block B_q from Main and dummy blocks from all other containers.

To access a block from any container, the client uploads the block's label. If one wishes to access a real block B_q from Main, then the client will upload label $F_L(K_L^0, q)$. Similarly, to access a real block B_q from $\mathsf{LatStash}_i$, the client uploads label $F_L(K_L^i, q)$. To access a dummy block from Main, the client uploads label $F_L(K_L^0, \mathtt{dmy} \,\|\, c_d[0])$. Afterwards, it increments $c_d[0]$ to avoid accessing the same dummy twice. For $\mathsf{LatStash}_i$, the client uploads $F_L(K_L^i, \mathtt{dmy} \,\|\, c_d[i])$ and increments $c_d[i]$ afterwards. In the c_p-th phase, each query will consist of a list L of exactly $c_p + 1$ labels: one for each of Main, $\mathsf{LatStash}_1$, ..., $\mathsf{LatStash}_{c_p}$ depending on the current location of B_q determined using Stash and Map.

Query: Server Response. After receiving the list L of c_p block labels, one for each of Main and $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_{c_p-1}$, the server computes the XOR of the blocks identified by the the labels and returns it to the client. After all these blocks are accessed, the server is free to delete them from memory.

Query: Process Response. First, if $B_q \in \mathsf{Stash}$, then we can simply ignore the server's response and retrieve B_q from Stash. If the queried block B_q is part of the server-returned XOR, then the XOR consists of exactly one real block and c_p dummy blocks. If a dummy of $\mathsf{LatStash}_i$ was accessed with label $\mathtt{lab}_i = F_L(K_L^i, \mathtt{dmy} \,\|\, c_d[i])$, then we know the dummy block's contents are $F_D(K_D, \mathtt{lab})$. As a result, the client can generate the contents of all the c_p dummy blocks and remove them from the XOR to retrieve real block B_q.

Once the client obtains B_q, we add B_q to Stash and delete entry $\mathsf{Map}[q]$ if it exists. If the operation was to overwrite B_q, we update B_q before adding it to Stash. Finally, we can return B_q to complete the query operation.

Query: Postprocessing. After the query, the client may need to perform additional postprocessing to

Algorithm 4 LatORAM.Setup$(\perp; N, P, M, 1^\lambda)$

Input: $N, P, M, 1^\lambda$: number of blocks, number of phases, client storage size, and security parameter.
Output: $\mathtt{Store}$, st: server storage and client state.

Client initialization (executed by client):

Randomly select $K_L, K_D \leftarrow \{0,1\}^\lambda$.
Initialize $\mathsf{Map} \leftarrow \emptyset$ and $\mathsf{Stash} \leftarrow \emptyset$.
Initialize phase and query counts $c_p \leftarrow 0$ and $c_q \leftarrow 0$.
Initialize dummy counts $c_d[i] \leftarrow 1$ for $i \in \{0, \ldots, c_p\}$.
Set $\mathsf{st} \leftarrow (K_E, K_L, K_D, \mathsf{Map}, \mathsf{Stash}, c_p, c_q, c_d)$.

Client initial upload (executed by client):

Set $K_L^0 \leftarrow F(K_L, 0)$.
Set total number of dummies $D \leftarrow P \cdot M$.
Generate random permutation $\pi : [N + D] \to [N + D]$.
for $i \in [N + D]$ **do**
  if $\pi(i) \leq N$ **then**
    Compute $\mathtt{lab} \leftarrow F_L(K_L^0, \pi(i))$.
    Compute $\mathsf{blk} \leftarrow \mathsf{Enc}(K_E, \pi(i) \| 0^b)$.
  else
    Compute $\mathtt{lab} \leftarrow F_L(K_L^0, \mathtt{dmy} \| \pi(i) - N)$.
    Compute $\mathsf{blk} \leftarrow F_D(K_D, \mathtt{lab})$.
  Upload $(\mathtt{lab}, \mathsf{blk})$ to server.

Server initialization (executed by server):

Server appends all uploaded blocks into Main.
Server sets $\mathtt{Store} \leftarrow \mathsf{Main}$.

prepare for the next query. First, we increment the query counter c_q. Once $c_q = M$ queries have been performed, we will prepare the next lateral stash. In particular, we increment c_p and execute SmallShuffle to upload Stash along with $(P - c_p) \cdot M$ dummy blocks to the server that will be stored in a new container $\mathsf{LatStash}_{c_p}$. We also update Map accordingly to reflect the recent shuffle. If any block $B_i \in \mathsf{Stash}$, then we update $\mathsf{Map}[i] = c_p$. Finally, we increment the phase counter c_p for the next phase.

Additional postprocessing may be necessary once we reach the last phase of $c_p = P$. In this case, the server's storage now consists of Main, with exactly N fresh blocks and $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$ each with M fresh blocks. The client and server jointly execute CacheShuffle to merge the P containers $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$ into a single container Tmp with MP blocks. Finally, the client and server execute TwoMerge on Main and Tmp to re-initialize Main. Note that the blocks in the two containers input to TwoMerge are randomly shuffled. Indeed, Main is randomly shuffled during Setup and Tmp is shuffled by CacheShuffle during the previous step of the postprocessing.

The phase counter is reset to $c_p \leftarrow 0$ and queries restart as if the setup phase had just completed (see Algorithm 7).

4.3 Analysis of LatORAM

We start by noting that LatORAM is correct unless one of its underlying algorithms aborts. We will thus instantiate the CacheShuffle and TwoMerge (SmallShuffle never aborts) with parameters that ensure negligible abort probabilities, and thus LatORAM is correct except with negligible probability as they are run a polynomial number of times.

The obliviousness property of LatORAM follows from the obliviousness of SmallShuffle, CacheShuffle and TwoMerge and from the fact that the access pattern is determined by the number of operations that were previously performed as well as the parameters N, P and M, quantities known to the adversary and the simulator. More specifically, the setup phase essentially uploads $N + PM$ blocks that are indistinguishable from b-byte random strings. Consider any query q during the c_p-th phase. The client requests

Algorithm 5 LatORAM.Query($\texttt{Store}; \mathsf{st}, \texttt{op}, q, \texttt{data}$)

Input: $\texttt{Store}, \mathsf{st}, \texttt{op}, q, \texttt{data}$: server and client storage, operation type, query index and (optionally) write data.
Output: $\texttt{Store}, \mathsf{st}, Z$: server and client state and output.

Client Request (executed by client):
- Initialize list L of size $c_p + 1$.
- Set $i \leftarrow 0$.
- **if** $q \in \mathsf{Map}$ **then**
  - Retrieve $i \leftarrow \mathsf{Map}[q]$ and delete $\mathsf{Map}[q]$.
- **else if** $q \in \mathsf{Stash}$ **then**
  - Set $i \leftarrow \perp$.
- Set $K_L^i \leftarrow F(K_L, i)$ and $L[i] \leftarrow F_L(K_L^i, q)$.
- **for** $j \in \{0, 1, \ldots, c_p\} \setminus \{i\}$ **do**
  - Set $K_L^j \leftarrow F(K_L, j)$.
  - Set $L[j] \leftarrow F_L(K_L^j, \texttt{dmy} \,\|\, c_d[j])$.
  - Increment dummy count $c_d[j] \leftarrow c_d[j] + 1$.
- Upload list L to the server.

Server Response (executed by server):
- Set $Z \leftarrow \mathsf{Main}[L[0]]$.
- **for** $j \in \{1, \ldots, |L| - 1\}$ **do**
  - Compute $Z \leftarrow Z \oplus \mathsf{LatStash}_j[L[j]]$.
- Return Z to client.

Process Response (executed by client):
- **if** $\texttt{op} = \texttt{Write}$ **then**
  - Set $Z \leftarrow \texttt{data}$
- **else if** $q \in \mathsf{Stash}$ **then**
  - Set $Z \leftarrow \mathsf{Stash}[q]$.
- **else**
  - **for** $j \in \{0, \ldots, c_p\} \setminus \{i\}$ **do**
    - Compute $Z \leftarrow Z \oplus F_D(K_D, L[j])$.
- Set $\mathsf{Stash}[q] \leftarrow Z$ and return Z.
- Execute $\mathsf{Postprocess}(\texttt{Store}; \mathsf{st})$.

to access a fresh (previously unaccessed) block with a pseudo-random label from each of the containers $\mathsf{Main}, \mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_{c_p}$. This access pattern is independent of the queried block ensuring obliviousness and can be simulated by knowing only N, M, and P. Finally, we note that SmallShuffle, CacheShuffle and TwoMerge are executed at fixed time points only depending on the number of queries that have been performed: SmallShuffle executes every M queries and CacheShuffle as well as TwoMerge executes every PM queries. Since all three shuffling algorithms are oblivious, we can conclude that LatORAM is an oblivious RAM.

We have thus the following theorem.

Theorem 2. LatORAM *is an ORAM according to Definition 1.*

Online Query Costs. Recall that one of main goals was to enable blazing fast online query costs consisting of a single roundtrip and one block of communication. We show that LatORAM successfully achieves this goal. Note, the client requests blocks to all containers in a single roundtrip as it can identify the desired block using information in Stash and Map. The request specifies at most P labels, which we ignore in this analysis as their cumulative size is much smaller than the typical size of a block. The server returns a single block: the XOR of all requested blocks. In terms of computation, the server accesses at most P blocks and XORs them together. The client only performs plaintext operations and fast symmetric-key cryptographic operations of encryption, decryption and PRF evaluation. As a result, online queries of LatORAM are blazing fast.

Algorithm 6 LatORAM.Postprocess($\texttt{Store}; \mathsf{st}$)

Input: $\texttt{Store}, \mathsf{st}$: server and client state.
Output: $\texttt{Store}, \mathsf{st}$: server and client state.

Stash Upload (executed by client):

Increment query count $c_q \leftarrow c_q + 1$.
If $c_q < M$, terminate algorithm.
Set query count $c_q \leftarrow 0$.
Increment phase count $c_p \leftarrow c_p + 1$.
Set $I \leftarrow [(P - c_p) \cdot M]$.
Set $K_L^{c_p} \leftarrow F(K_L, c_p)$ and $K_D^{c_p} \leftarrow F(K_D, c_p)$.
for $(i, B_i) \in \mathsf{Stash}$ **do**
    Set $\mathsf{Map}[i] \leftarrow c_p$.
Execute $\mathsf{SmallShuffle}(\perp; \mathsf{Stash}, I, K_E, K_L^{c_p}, K_D^{c_p})$.

Server Processing (executed by server):

Server receives $\mathsf{LatStash}_{c_p}$ as output of $\mathsf{SmallShuffle}$.
Server appends $\mathsf{LatStash}_{c_p}$ to $\texttt{Store}$.
If $c_p = P$, execute $\mathsf{MainReinit}(\texttt{Store}; \mathsf{st})$.

Amortized Communication. Amortized communication consists of labels and blocks that are typically much larger and dominate the cost. For example, blocks may be kilobytes (or even larger) in size while block labels will be much smaller (such as 16-20 bytes). Therefore here we concentrate on blocks and analyze the amortized communication of $\mathsf{LatORAM}$ over a cycle of PM queries. Each query costs only one block of communication for a total of PM blocks across all queries of the P phases. After each phase $1 \le c_p \le P$, $\mathsf{LatORAM}$ uploads M blocks from Stash and $(P - c_p) \cdot M$ dummy using $\mathsf{SmallShuffle}$. Across all P phases, the total cost is $M \cdot (1 + \ldots + (P-1))$.

At the end of the P-th phase, we note that Main contains exactly N blocks. Additionally, each of $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$ will have M blocks for a total of PM blocks. Across all $P+1$ containers, there are N real blocks and PM dummy blocks. The cost of executing $\mathsf{CacheShuffle}$ on $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$ is $4PM + o(PM)$ blocks of communication. The cost of $\mathsf{TwoMerge}$ is $2N + 2PM + o(N + PM)$. Altogether, the cost of these shuffle is $2N + 6PM + o(N + PM)$. The total cost across all P phases and $P \cdot M$ queries is $PM + M(1 + \ldots + P - 1) + 2N + 6PM$ that can be simplified to $2N + 7PM + MP(P+1)/2$. Amortizing across PM queries, we obtain $2N/(PM) + P/2 + 15/2 + o(1)$. The amortized bandwidth is minimized by picking $P^\star = 2\sqrt{N/M}$ which gives amortized bandwidth of $2\sqrt{N/M} + O(1)$. For $M = O(\sqrt{N})$, we have $P^\star = O(N^{1/4})$ and obtain $O(N^{1/4})$ amortized communication. Due to small constants, it turns out $\mathsf{LatORAM}$ outperforms prior works (see Section 6).

Roundtrips. It is clear that $\mathsf{LatORAM}$ requires exactly one roundtrip during online queries. Next we analyze and show that the amortized number of roundtrips is $1 + o(1)$ per query assuming the client stash size $M = \Theta(\sqrt{N})$ and the number of phases $P = o(M)$ and $P = \omega(1)$, and with $o(M)$ additional temporary block storage.

First, each query involves one roundtrip, contributing PM roundtrips across the PM queries. Each of the $\mathsf{SmallShuffle}$ to construct Stash_p involves uploading $(P - i)M$ blocks. Choose K such that $K = o(M)$ and $K = \omega(P)$. During $\mathsf{SmallShuffle}$, we upload the blocks in a batch of size K which results in at most $PM/K = o(1)M$ roundtrips. We use $\mathsf{CacheShuffle}$ on $\mathsf{Stash}_1, \ldots, \mathsf{Stash}_P$ to construct Tmp of PM blocks with a cost of $O(\sqrt{PM})$ roundtrips. In the $\mathsf{TwoMerge}$, we use $\mathsf{CacheShuffle}$ to construct SAux of size $o(N + PM)$, which costs $O(\sqrt{o(N + PM)})$ roundtrips. Finally, the rest of the $\mathsf{TwoMerge}$ costs $(N + PM)/M = N/M + P$ roundtrips. Summing everything up, we see that the total number of roundtrips during PM queries is $PM + o(PM) + O(\sqrt{PM}) + O(\sqrt{o(N + PM)}) + N/M + P$ roundtrips. Hence, assuming $M = \Theta(\sqrt{N})$ but and $P = o(M)$, the amortized number of roundtrips is $1 + o(1)$.

Note, apart from the M client stash blocks, we need an additional $K = o(M)$ temporary block storage to handle $\mathsf{SmallShuffle}$.

Algorithm 7 LatORAM.MainReinit($\texttt{Store}; \mathsf{st}$)

Input: $\texttt{Store}, \mathsf{st}$: server and client state.
Output: $\texttt{Store}, \mathsf{st}$: server and client state.

Client Re-initialization (executed by client):
Write $\mathsf{st} = (K_E, K_L, K_D, \mathsf{Map}, \mathsf{Stash}, c_p, c_q, c_d)$.
Randomly select $K_L^{\texttt{new}} \leftarrow \{0,1\}^\lambda$.
Re-initialize $\mathsf{Map} \leftarrow \emptyset$ and $\mathsf{Stash} \leftarrow \emptyset$.
Re-initialize $c_p, c_q \leftarrow 0$.
Re-initialize $c_d[i] \leftarrow 1$ for $i \in \{0, \ldots, c_p\}$.
Set $\mathsf{st} \leftarrow (K_E, K_L^{\texttt{new}}, K_D, \mathsf{Map}, \mathsf{Stash}, c_p, c_q, c_d)$.

Oblivious Shuffle (executed by client and server):
Server sets $X \leftarrow \mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$.
Set dummy indices $I \leftarrow [P \cdot M]$.
Randomly select $K_L' \leftarrow \{0,1\}^\lambda$.
Execute $\mathsf{Tmp} \leftarrow \mathsf{CacheShuffle}(X; I, K_E, K_L', K_D)$.
Set $K_L^0 \leftarrow F(K_L^{\texttt{new}}, 0)$.
Execute $\mathsf{Main} \leftarrow \mathsf{TwoMerge}(\mathsf{Main}, \mathsf{Tmp}; I, K_E, K_L^0, K_D)$.
Server sets $\texttt{Store} \leftarrow \mathsf{Main}$.

Client Storage. The largest components of client storage are Stash and Map. Note, the stash contains at most M blocks at any point in time. The Map stores the labels and container index for at most $P \cdot M$ blocks. For standard choices of $M = O(\sqrt{N})$ and $P = 2N^{1/4}$, we note that Map requires $O(N^{3/4}) = o(N)$ sublinear storage. Similar to communication, the client storage cost is dominated by blocks that are much larger than labels in most settings. Therefore, Stash is much larger than Map.

Both SmallShuffle, CacheShuffle and TwoMerge use M blocks on the client throughout their executions. For SmallShuffle, the input is the client stored Stash. When CacheShuffle and TwoMerge are executed, Stash has been emptied. Therefore, no extra block memory is needed when executing any of the shuffling algorithms. Finally, we note that the shuffling algorithms require client storage of indices for at most $D = P \cdot M$ dummies.

5 Lat^{2}ORAM: Further Delayed Merging

In this section, we present Lat^{2}ORAM that further delays large shuffles to reduce amortized bandwidth, yielding a strict improvement of LatORAM at the cost of a slightly larger position map (which, however, remains of size $o(N)$).

Recall that LatORAM performs a large merge over N blocks after a cycle of PM queries to combine all of the stashes $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$ with Main. At this stage, Main consists of N blocks while each of $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$ has M blocks (for a total of PM blocks). For typical settings of client storage of $M = \Theta(N^{1/2})$ blocks and by setting $P = \Theta(N^{1/4})$ so to minimize the amortized bandwidth of LatORAM, we have Main with N blocks while the remaining P containers account for $PM = \Theta(N^{3/4})$ blocks. Therefore the $O(N)$-cost shuffle when amortized over $PM = \Theta(N^{3/4})$ queries results in a $O(N^{1/4})$ amortized cost.

Expanding our ideas, we construct Lat^{2}ORAM that further delays the $O(N)$-cost shuffle with Main with the effect of amortizing the cost over a larger number of queries. Specifically, Lat^{2}ORAM is parameterized by E, the number of epochs, and by P, the number of phases in an epoch. The idea is to merge the PM blocks in $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$ into a distinct intermediate container Main'. We denote an *epoch* as consisting of P phases each of M queries. At the end of each epoch, the contents of the P containers $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$ are merged with Main' which thus grows by PM blocks. At the end of E epochs, a *cycle* of EPM queries is completed and Main' containing $O(EPM)$ blocks is merged into Main. This has the effect of amortizing the $O(N)$-cost shuffle with Main over the larger number of EPM queries.

Queries in Lat^{2}ORAM are similar to the one in LatORAM with the exception that the server additionally stores Main'. For obliviousness, each query accesses exactly one block from Main' like all other containers.

Algorithm 8 $\mathsf{Lat}^2\mathsf{ORAM.Setup}(\perp; E, P, M, 1^\lambda)$

Input: $N, E, P, M, 1^\lambda$: number of blocks, epochs and phases, client storage size and security parameter.
Output: $\mathtt{Store}$, st: server and client storage.

Client initialization (executed by client):

Randomly select $K_L, K_D \leftarrow \{0,1\}^\lambda$.
Initialize $\mathsf{Map} \leftarrow \emptyset$ and $\mathsf{Stash} \leftarrow \emptyset$.
Initialize epoch count $c_e \leftarrow 0$.
Initialize phase and query counts $c_p \leftarrow 0$ and $c_q \leftarrow 0$.
Initialize dummy counts $c_d[i] \leftarrow 1$ for $i \in \{0, \ldots, c_p\}$.
Initialize dummy count $c_d[-1] \leftarrow 1$.
Set $\mathsf{st} \leftarrow (K_E, K_L, K_D, \mathsf{Map}, \mathsf{Stash}, c_e, c_p, c_q, c_d)$.

Client initial upload (executed by client):

Set $K_L^0 \leftarrow F(K_L, 0)$.
Set total number of dummies $D \leftarrow EPM$.
Generate random permutation $\pi : [N + D] \rightarrow [N + D]$.
for $i \in [N + D]$ **do**
  if $\pi(i) \leq N$ **then**
    Compute $\mathtt{lab} \leftarrow F_L(K_L^0, \pi(i))$.
    Compute $\mathsf{blk} \leftarrow \mathsf{Enc}(K_E, \pi(i) \| 0^b)$.
  else
    Compute $\mathtt{lab} \leftarrow F_L(K_L^0, \mathtt{dmy} \,||\, \pi(i) - N)$.
    Compute $\mathsf{blk} \leftarrow F_D(K_D, \mathtt{lab})$.
  Upload $(\mathtt{lab}, \mathsf{blk})$ to server.

Server initialization (executed by server):

Server appends all uploaded blocks into Main.
Server sets $\mathtt{Store} \leftarrow \mathsf{Main}$.

With the introduction of Main', $\mathsf{Lat}^2\mathsf{ORAM}$ features some elements of the hierarchical approach. We would like to stress though that in $\mathsf{Lat}^2\mathsf{ORAM}$ the $\mathsf{LatStash}$'s are still created by simply uploading the blocks from the stash to the server and not by merging existing containers into a new one, like in the hierarchical approach.

5.1 Second Construction: Lat^{2}ORAM

In this section we formally present $\mathsf{Lat}^2\mathsf{ORAM}$ parameterized by the client storage M, the number of *epochs* E, and the number of *phases* P in each epoch. The formal pseudocode of $\mathsf{Lat}^2\mathsf{ORAM}$ is found in Algorithms 8, 9, 10, 11, and 12. The differences between $\mathsf{Lat}^2\mathsf{ORAM}$ and $\mathsf{LatORAM}$ are highlighted in blue in the pseudocode.

We note that the cryptographic primitives remain identical to $\mathsf{LatORAM}$. We next outline the differences in the setup, client state and query algorithm.

Setup. The only difference is that Main is initialized with EPM dummies (one for each query over the E epochs).

Client State. The client storage remains identical as $\mathsf{LatORAM}$ consisting of keys K_E, K_L and K_D as well as Stash. In addition to counters c_p and c_q, the client also stores a counter for epochs c_e. One difference for Map is that $\mathsf{Map}[q]$ may point to block B_q being stored in Main'.

Query. The query algorithm excluding postprocessing is nearly identical to $\mathsf{LatORAM}$. The only difference is that we need to access a block from Main' (if it exists). As before, the client uses Stash and Map to find the location of the queried block B_q. If $\mathsf{Map}[q]$ points to Main', then we access real block B_q from Main'. Otherwise, we access a dummy block from Main'.

Query postprocessing. The main differences occur during oblivious shuffling in postprocessing. At the

Algorithm 9 Lat^{2}ORAM.Query($\texttt{Store}; \mathsf{st}, \mathsf{op}, q, \texttt{data}$)

Input: $\texttt{Store}, \mathsf{st}, \mathsf{op}, q, \texttt{data}$: server and client storage, operation type, query index and (optionally) write data.
Output: $\texttt{Store}, \mathsf{st}, Z$: server and client state and output.

Client Request (executed by client):

Initialize list L of size $c_p + 1$.
Set $i \leftarrow 0$.
if $q \in \mathsf{Map}$ **then**
  Retrieve $i \leftarrow \mathsf{Map}[q]$ and delete $\mathsf{Map}[q]$.
else if $q \in \mathsf{Stash}$ **then**
  Set $i \leftarrow \perp$.
Set $K_L^i \leftarrow F(K_L, i \,||\, c_e)$.
if $i = 0$ **then**
  Set $K_L^i \leftarrow F(K_L, 0)$.
Set $L[i] \leftarrow F_L(K_L^i, q)$.
for $j \in \{0, 1, \ldots, c_p\} \setminus \{i\}$ **do**
  Set $K_L^j \leftarrow F(K_L, j \,||\, c_e)$.
  if $j = 0$ **then**
    Set $K_L^j \leftarrow F(K_L, 0)$.
  Set $L[j] \leftarrow F_L(K_L^j, \texttt{dmy} \,||\, c_d[j])$.
  Increment dummy count $c_d[j] \leftarrow c_d[j] + 1$.
Set $K_L' \leftarrow F(K_L, -1 \,||\, c_e)$.
if $i = -1$ **then**
  Append $F_L(K_L', q)$ to L.
else if $i \neq -1$ and $c_e > 0$ **then**
  Append $F_L(K_L', \texttt{dmy} \,||\, c_d[-1])$ to L.
  Increment dummy count $c_d[-1] \leftarrow c_d[-1] + 1$.
Upload list L to the server.

Server Response (executed by server):

Set $Z \leftarrow \mathsf{Main}[L[0]]$.
if $\mathsf{Main}' \neq \perp$ **then**
  Compute $Z \leftarrow Z \oplus \mathsf{Main}'[L[|L| - 1]]$.
  Pop last entry from L.
for $j \in \{1, \ldots, |L| - 1\}$ **do**
  Compute $Z \leftarrow Z \oplus \mathsf{LatStash}_j[L[j]]$.
Return Z to client.

Process Response (executed by client):

if $\mathsf{op} = \texttt{Write}$ **then**
  Set $Z \leftarrow \texttt{data}$
else if $q \in \mathsf{Stash}$ **then**
  Set $Z \leftarrow \mathsf{Stash}[q]$.
else
  for $j \in \{-1, 0, \ldots, c_p\} \setminus \{i\}$ **do**
    Compute $Z \leftarrow Z \oplus F_D(K_D, L[j])$.
Set $\mathsf{Stash}[q] \leftarrow Z$ and return Z.
Execute $\mathsf{Postprocess}(\texttt{Store}; \mathsf{st})$.

Algorithm 10 Lat^{2}ORAM.Postprocess($\mathtt{Store}$; st)

Input: $\mathtt{Store}$, st: server and client state.
Output: $\mathtt{Store}$, st: server and client state.

Stash Upload (executed by client):

Increment query count $c_q \leftarrow c_q + 1$.
If $c_q < M$, terminate algorithm.
Set query count $c_q \leftarrow 0$.
Increment phase count $c_p \leftarrow c_p + 1$.
Set $I \leftarrow [(P - c_p) \cdot M]$.
Set $K_L^{c_p} \leftarrow F(K_L, c_p \,||\, c_e)$.
for $(i, B_i) \in \mathsf{Stash}$ **do**
  Set $\mathsf{Map}[i] \leftarrow c_p$.
Execute $\mathsf{SmallShuffle}(\perp; \mathsf{Stash}, I, K_E, K_L^{c_p}, K_D)$.

Server Processing (executed by server):

Server receives $\mathsf{LatStash}_{c_p}$ as output of SmallShuffle.
Server appends $\mathsf{LatStash}_{c_p}$ to $\mathtt{Store}$.
if $c_p = P$ **then**
  Execute EpochReinit($\mathtt{Store}$; st).

end of each epoch, the server stores disjoint containers Main, Main$'$, $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$. The client and server jointly execute CacheShuffle to merge $\mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$ into a new container Aux. Afterwards, TwoMerge is executed to merge Main$'$ and Aux into the new Main$'$. Additionally, we ensure there are PM dummy blocks in Main$'$. If not, we add sufficient number of dummies during the shuffles. Note, at the end of the first epoch, Main$'$ does not exist. So, we can omit the TwoMerge there and only use CacheShuffle.

After all E epochs, the server now stores Main and Main$'$. The client and server execute TwoMerge to combine Main and Main$'$ into a newly re-initialized Main. Additionally, sufficient dummy blocks are added during the combine to ensure Main has EPM dummy blocks.

5.2 Analysis of Lat^{2}ORAM

Amortized Bandwidth. We start by calculating the total block communication across the EPM queries of E epochs, each of P phases. Each epoch of P phases costs PM for online queries and $M \cdot (1 + \ldots + P - 1)$ for shuffling. At the end of the i-th epoch, we execute CacheShuffle that costs roughly $4PM$ blocks. Then, we execute TwoMerge over Main$'$ containing $(i-1) \cdot P \cdot M$ blocks and the output of CacheShuffle that contains $P \cdot M$ blocks. Altogether, this shuffle requires approximately $2i \cdot P \cdot M$ communication. The last step combines Main of N blocks and Main$'$ of EPM blocks into a new Main that requires roughly $2N + 2EPM$ blocks. Across all E epochs, the total cost is roughly $EM(1 + \ldots + P) + 2PM(1 + 2 + \ldots + E) + 2N + 4EPM$. Averaging across EPM queries, we obtain amortized cost of approximately $P/2 + E + 2N/(EPM) + O(1)$. The amortized cost is thus minimized by picking $E^\star = (N/M)^{1/3}$ and $P^\star = 2 \cdot (N/M)^{1/3}$ for which we obtain amortized bandwidth $3 \cdot (N/M)^{1/3} + O(1)$. For the standard case of client storage $M = O(\sqrt{N})$ we obtain amortized bandwidth of $O(N^{1/6})$.

For the label communication, we note that each query will upload at most $P + 1$ labels. On average, this will be $P/2 + 1$ labels uploaded per query due to empty containers not being queried. For all other shuffles and merges, we can assume labels and simultaneously uploaded and downloaded with blocks meaning the amortized label communication is also $O(N^{1/6})$. As labels (16 bytes or less) are much smaller the blocks (up to KBs or more in size), the block communication will dominate the cost in most settings.

Online Query Costs. The online query of Lat^{2}ORAM is one roundtrip and the communication consists of at most $P + 2$ labels uploaded and a single block downloaded. As blocks are typically much larger than labels, we note the total communication cost is dominated by the single block. The server computational consists in computing the XOR of at most $P + 2$ blocks.

Algorithm 11 Lat^{2}ORAM.EpochReinit(Store; st)

Input: Store, st: server and client state.
Output: Store, st: server and client state.

Client Re-initialization (executed by client):

Write st as $(K_E, K_L, K_D, \mathsf{Map}, \mathsf{Stash}, c_e, c_p, c_q, c_d)$.
Randomly re-select $K_L \leftarrow \{0,1\}^\lambda$.
Increment $c_e \leftarrow c_e + 1$.
Re-initialize $\mathsf{Stash} \leftarrow \emptyset$.
Re-initialize $c_p \leftarrow 0$.
Re-initialize $c_d[i] \leftarrow 1$ for $i \in \{1, \ldots, c_p\}$.
Re-initialize $c_d[-1] \leftarrow 1$.

Oblivious Merge (executed by client and server):

Set dummy indices $I \leftarrow [P \cdot M]$.
Server sets $X \leftarrow \mathsf{LatStash}_1, \ldots, \mathsf{LatStash}_P$.
Execute $\mathsf{SAux} \leftarrow \mathsf{CacheShuffle}(X; I, K_E, K_L, K_D)$.
if $\mathsf{Main}' \neq \perp$ **then**
  Randomly re-select $K_L \leftarrow \{0,1\}^\lambda$.
  Set $\mathsf{Main}' \leftarrow \mathsf{TwoMerge}(\mathsf{Main}', \mathsf{SAux}; I, K_E, K_L, K_D)$.
else
  Set $\mathsf{Main}' \leftarrow \mathsf{Aux}$.
Server sets $\mathtt{Store} \leftarrow (\mathsf{Main}, \mathsf{Main}')$.
if $c_e = E$ **then**
  Execute MainReinit(Store; st).

Client Storage. The client storage of Lat^{2}ORAM is nearly similar to LatORAM. The stash size remains the same as well as the client storage cost of the oblivious shuffles. The only difference is that Map stores up to EPM blocks container indices (LatORAM only stores PM such indices).

Correctness. The construction of Lat^{2}ORAM is correct as long as the underlying shuffles do not abort early. If we pick parameters to ensure that each oblivious shuffles have negligible failures, then Lat^{2}ORAM is correct except with negligible failure as each shuffle is executed at most a polynomial number of times.

Obliviousness. We observe that the only difference with LatORAM is in the schedule for accessing containers and executing oblivious merging algorithm. However, the schedule is predetermined and only depends on the number of operations and public parameters.

Theorem 3. Lat^{2}ORAM *is an ORAM according to Definition 1.*

Algorithm 12 Lat^{2}ORAM.MainReinit(Store; st)

Input: Store, st: server and client state.
Output: Store, st: server and client state.

Client Re-initialization (executed by client):

Randomly re-generate $K_L \leftarrow \{0,1\}^\lambda$.
Re-initialize Map.
Re-initialize $c_d[0] \leftarrow 1$.
Re-initialize $c_e \leftarrow 0$.
Set $\mathsf{st} \leftarrow (K_E, K_L, K_D, \mathsf{Map}, \mathsf{Stash}, c_e, c_p, c_q, c_d)$.
Set $K_L^0 \leftarrow F(K_L, 0)$.
Set dummy indices $I \leftarrow [E \cdot P \cdot M]$.

Oblivious Merge (executed by client and server):

Execute $\mathsf{Main} \leftarrow \mathsf{TwoMerge}(\mathsf{Main}, \mathsf{Main}'; I, K_E, K_L^0, K_D)$.
Server sets $\mathtt{Store} \leftarrow \mathsf{Main}$.

Proof. We define $\mathsf{Sim}(1^N)$ as follows. In the setup, $\mathsf{Sim}(1^N)$ uploads $N + EPM$ random strings. For queries, it is not hard to determine the exact containers that will be accessed by $\mathsf{Lat}^2\mathsf{ORAM}$. For each container that would be accessed by $\mathsf{Lat}^2\mathsf{ORAM}$, $\mathsf{Sim}(1^N)$ will access a random block that was previously unaccessed. Finally, every time that LatORAM would execute one of SmallShuffle, TwoMerge or CacheShuffle, $\mathsf{Sim}(1^N)$ will execute the associated efficient simulator that exists since all of them are oblivious shuffling algorithms.

To prove that an adversary cannot distinguish interacting with $\mathsf{Lat}^2\mathsf{ORAM}$ and $\mathsf{Sim}(1^N)$, we consider the following sequence of hybrid games. First, we replace IND-CPA ciphertexts with random strings. Secondly, we replace the usage of PRFs with random functions. In either of these steps, an efficient adversary cannot distinguish between the games except with negligible probability. We note that the final game is identical to the distributed obtained by executing $\mathsf{Sim}(1^N)$ completing the proof. □

Comparison to LatORAM. As a comparison, we note that $\mathsf{Lat}^2\mathsf{ORAM}$ has smaller amortized block bandwidth asymptotically and concretely (see Section 6). In exchange, $\mathsf{Lat}^2\mathsf{ORAM}$ uses larger position maps storing up to EPM indices (only PM for LatORAM). In general, one should always use $\mathsf{Lat}^2\mathsf{ORAM}$ unless client storage is limited when LatORAM will be better suited.

6 Experimental Evaluation

In this section we report the results of experiments that aim to evaluate how the performance of our constructions depends on the number N of blocks, on the number of blocks M in the client stash, and on the block size. In addition, we provide a comparison with RingORAM and FutORAMa.

Experimental Setup. All our experiments are performed using Intel Xeon CPU @ 2.6 GHz with 32 GB of RAM with only single-threaded execution. The results are the average of at least 10 different executions with standard deviation less than 10% of the average.

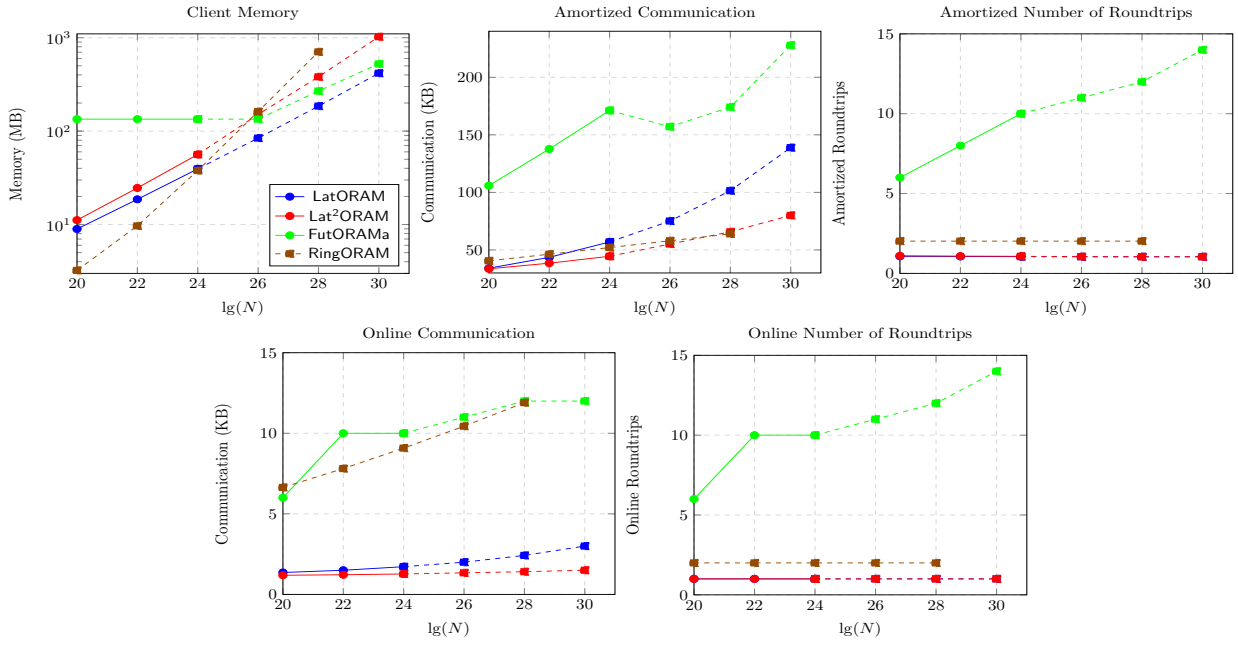

Figure 2: Costs for various ORAMs with databases of N blocks of 1KB each. LatORAM and $\mathsf{Lat}^2\mathsf{ORAM}$ are configured with $M = 8\sqrt{N}$ blocks in the stash and Client Memory accounts for stash and position map. The points on the dashed lines correspond to estimated costs. The Client Memory plot is a log plot with respect to the y-axis.

		LatORAM	Lat^{2}ORAM	FutORAMa	RingORAM
$\mathbf{N = 2^{20}}$	Client Memory	8.96 MB	11.15 MB	134.37 MB	**3.23 MB**
	Amortized (RT)	34.12 KB (1.07)	**33.62 KB (1.10)**	105.87 KB (6)	40.51 KB (2.02)
	Online (RT)	1.36 KB **(1)**	**1.19 KB (1)**	6 KB (6)	6.64 KB (2)
$\mathbf{N = 2^{22}}$	Client Memory	18.63 MB	24.66 MB	134.37 MB	**9.68 MB**
	Amortized (RT)	43.56 KB (1.06)	**38.49 KB (1.08)**	137.67 KB (8)	46.19 KB (2.02)
	Online (RT)	1.50 KB **(1)**	**1.22 KB (1)**	10 KB (10)	7.81 KB (2)
$\mathbf{N = 2^{24}}$	Client Memory	39.44 MB	56.25 MB	134.37 MB	**37.87 MB**
	Amortized (RT)	56.94 KB (1.05)	**44.50 KB (1.07)**	171.18 KB (10)	51.98 KB (2.02)
	Online (RT)	1.72 KB **(1)**	**1.27 KB (1)**	10 KB (10)	9.08 KB (2)
$\mathbf{N = 2^{26}}(*)$	Client Memory	**77.00 MB**	104.63 MB	134.37 MB	162.07 MB
	Amortized (RT)	73.50 KB (1.04)	**35.33 KB (1.05)**	157.09 KB (11)	57.85 KB (2.02)
	Online (RT)	2.00 KB **(1)**	**1.34 KB (1)**	11 KB (11)	10.44 KB (2)
$\mathbf{N = 2^{28}}(*)$	Client Memory	**167.60 MB**	275.88 MB	268.74 MB	706.26 MB
	Amortized (RT)	100.42 KB (1.04)	**44.42 KB (1.04)**	174.07 KB (12)	63.83 KB (2.02)
	Online (RT)	2.42 KB **(1)**	**1.41 KB (1)**	12 KB (12)	11.90 KB (2)
$\mathbf{N = 2^{30}}(*)$	Client Memory	**376.00 MB**	736.00 MB	524.88 MB	-
	Amortized (RT)	138.50 KB (1.04)	**53.52 KB (1.04)**	227.87 KB (14)	-
	Online (RT)	3.00 KB **(1)**	**1.50 KB (1)**	14 KB (14)	-

Figure 3: Costs for various ORAMs with databases of N blocks of 1 KB each. LatORAM and Lat^{2}ORAM are configured with $M = 8\sqrt{N}$ blocks in the stash and Client Memory accounts for stash and position map. Online (amortized) RT stands for online query (amortized) roundtrips. All numbers in the rows with asterisks are estimated costs.

Implementations. We implement both LatORAM and Lat^{2}ORAM in C++ using 2,500 lines of codes. Our implementations target error (abort) probability of 2^{-80} and 128 bits of computational security. We use SHA256 as the hash function and AES-GCM-256 as the encryption scheme with 32 byte keys. Block labels are 16 bytes. Since the position map was of manageable size we did not consider recursing to store it. For FutORAMa, we use the implementation found at [12]. For larger databases we use the counting mode provided by the implementation. We will compare with several instantiations of FutORAMa including shallow variants with less roundtrips but higher asymptotic costs similar to the ones in our constructions. We are unaware of any public implementation of RingORAM and we use formulas and analyses in [28].

Communication. We evaluate online and amortized (over a cycle) communication of various ORAM schemes with similar client memory sizes. In Figure 2, we consider 1 KB blocks across different numbers of entries N. LatORAM and Lat^{2}ORAM are configured using $M = 8\sqrt{N}$ blocks in the stash. The other constructions are configured to have as similar client memory size as possible. Online communication includes the one block of data sent by the server to the client and the labels of the blocks to be XORed (we make the conservative choice of 128-bit labels) sent by the client to the server; for RingORAM, the metadata that is transferred in the first roundtrip is included as well (see Table 6 of [28] for the sizes of the components of the metadata). Amortized communication is computed over a full cycle (of PM or EPM queries, for LatORAM and Lat^{2}ORAM respectively). We note that Lat^{2}ORAM has the smallest amortized and online communication across all settings (even if other constructions are given substantially more client memory) and its online communication is from 5x to 8x smaller than RingORAM. We also report numbers in Figure 3.

For larger databases of $N \in \{2^{28}, 2^{30}\}$, we see that both LatORAM and Lat^{2}ORAM are superior in all three dimensions of client memory, amortized and online communication. We expect this to continue to be true for even larger number of entries N as well.

In Figure 4, we fix $N = 2^{22}$ and the client stash size to $M = 8\sqrt{N}$, and evaluate LatORAM and Lat^{2}ORAM's client memory and amortized communication with varying block sizes. Client memory includes stash and position map. For all our test cases, the value of P and E are set equal to the optimal values $P^\star$ and $E^\star$ (see Section 4.3 and 5.2).

	LatORAM		Lat^{2}ORAM	
Block Size (Bytes)	Client Memory	Amortized Comm.	Client Memory	Amortized Comm.
2^8	6.63 MB	11.57 KB	12.66 MB	10.15 KB
2^9	10.63 MB	22.23 KB	16.66 MB	19.59 KB
2^{10}	18.63 MB	43.56 KB	24.66 MB	38.49 KB
2^{11}	34.63 MB	86.20 KB	40.66 MB	76.28 KB
2^{12}	66.63 MB	171.49 KB	72.66 MB	151.86 KB

Figure 4: $N = 2^{22}$ blocks, $M = 8\sqrt{N}$ and varying block sizes. Client memory includes stash and position map.

N	LatORAM	Lat^{2}ORAM	FutORAMa	RingORAM
2^{20}	8976	10 258	2034	3073
2^{22}	8138	10 006	1221	2486
2^{24}	7097	9619	1221	2048
2^{26}	6104	9110	1110	1719
2^{28}	5044	8658	1017	1467

Figure 5: Online query throughput (queries/second) for 100 Mbit/s network speeds with client block size as $M = 8\sqrt{N}$.

Computation. In Figure 6, we compare the computation and communication between LatORAM and FutORAMa [2]. We see that increasing the client memory size decreases the access time and the amortized communication cost. Our LatORAM outperforms FutORAMa in all aspects, even with relatively small client memory.

Roundtrips. LatORAM and Lat^{2}ORAM require only a single round for queries as opposed to the two roundtrips of RingORAM (one is used to download metadata to identify the blocks that must be XORed by the server). FutORAMa requires 9-28 roundtrips, a substantially larger number. In slow network settings, we expect LatORAM and Lat^{2}ORAM to have much faster online query latency due to fewer roundtrips. Similarly, LatORAM and Lat^{2}ORAM have smaller amortized roundtrips meaning their amortized latency will also be smaller (especially in slower network settings).

Sensitivity Analysis: Stash Size. In Figure 7, we fix the number of blocks to be $N = 2^{22}$. For 1 KB blocks, we have 4 GB of data. We conduct sensitivity analysis on the stash size, measured in number M of blocks. Specifically, we consider LatORAM and Lat^{2}ORAM with increasing stash size: from $M := \sqrt{N} = 2^{11}$ blocks to $M := 16\sqrt{N} = 2^{15}$ blocks., For each value of M, we pick the optimal value of $P = P^\star$ (for LatORAM) and the optimal values of $E = E^\star$ and $P = P^\star$ (for Lat^{2}ORAM) as determined by the analysis conducted in Section 4.3 and Section 5.2. As we can see, both LatORAM and Lat^{2}ORAM can make good used of increased stash sizes and exhibit reduced amortized communication cost with increasing stash size.

Network Cost. Figure 5 compares estimated calculations of online query throughput (queries served per second) for various constructions at 100Mbit per second network speed. Since publicly available implementations runnable in a client-server network setup were unavailable for FutORAMa and RingORAM, we performed manual calculations to ensure a fair comparison. Because LatORAM and Lat^{2}ORAM utilize smaller online query payloads than FutORAMa and RingORAM, they achieve higher calculated online query throughputs.

Server Memory. As we have seen, both LatORAM and Lat^{2}ORAM rely on server storage and, besides the actual blocks from the database, the server will host some blocks that derive from the client stash or are the result of merging operations. For LatORAM the number of extra blocks is at most PM and for $M = O(\sqrt{N})$ and for the value of P that minimizes the amortized bandwidth, we have $PM = O(N^{3/4}) = o(N)$. More specifically, for the test cases in Figure 2, the extra server memory does not exceed 18% of the database size and can be as small as 3%, whereas for the test cases in Figure 4 the extra server memory goes from 3%

Client Memory	Time	Comm. (RT)
FutORAMa		
32 MB	507 μs	34.48 KB (8)
LatORAM		
3.91 MB	195 μs	15.92 KB (1.12)
6.63 MB	143 μs	11.58 KB (1.06)
11.84 MB	108 μs	8.63 KB (1.03)
40.51 MB	64 μs	5.24 KB (1.01)

Figure 6: Comparisons of access time and the communication overhead between FutORAMa [2] and LatORAM with varying client memory size. We fix $N = 2^{22}$ and the block size to 256 bytes.

	LatORAM		Lat^{2}ORAM	
Stash Size (M)	Client Memory	Amortized Comm.	Client Memory	Amortized Comm.
2^{11}	5.09 MB	137.74 KB	7.53 MB	83.46 KB
2^{12}	5.28 MB	95.48 KB	10.46 MB	59.68 KB
2^{13}	9.86 MB	59.85 KB	19.05 MB	42.18 KB
2^{14}	18.63 MB	43.56 KB	25.09 MB	34.94 KB
2^{15}	35.84 MB	32.50 KB	44.63 MB	29.39 KB

Figure 7: $N = 2^{22}$ blocks of 1 KB with varying stash sizes. Client memory includes stash and position map.

to 13% of the database size. Similar consideration apply to Lat^{2}ORAM and for the test cases of Figure 2, the server expansion is between 13% and 40%. As a reference, we mention that, for the same test cases, RingORAM has a server expansion for around 6 times the size of the database.

7 Related Works

There is a huge literature on ORAM starting from the original works of Goldreich and Ostrovsky [13, 25, 14]. As we focus on practical efficiency, we consider prior works with similar efficiency goals.

Hierarchical ORAMs. The majority of hierarchical ORAMs were concerned with minimizing the asymptotic communication cost. There are two works from this line that focus on concrete efficiency. Zahur *et al.* [36] considered SQRT-ORAM, which is a hierarchical ORAM with only one level. SQRT-ORAM uses one roundtrip and block of communication during queries but $O(\sqrt{N})$ amortized communication. Even though our works also build upon SQRT-ORAM, we note our approach deviates from hierarchical ORAMs that build levels (server containers) of exponentially increasing sizes while our constructions use similar sized containers to avoid large shuffles. Additionally, by not mixing previously queried data together, our constructions avoid the need for a roundtrip per level as required by hierarchical ORAMs. FutORAMa [2] presented the first concretely efficient hierarchical ORAM with more than one level that successfully reduced amortized communication to logarithmic. However, this requires significantly more roundtrips during queries (9 to 28, depending on the size of the database). In contrast, our schemes obtain better amortized communication with a single roundtrip during queries. Rank ORAM [17] implements queries with only one round of communication by using a memory efficient position map. The space needed is linear, as opposed to $O(N \log N)$ needed by data structures used in previous constructions. In contrast, our schemes have $o(N)$ client memory and implement queries with one round and one block of communication.

We note that [19] also considered several containers within the same level for hierarchical ORAMs, but solely for asymptotic improvement.

Tree-based ORAMs. Tree-based ORAMs were introduced by Shi *et al.* [30] with a long line of follow-up work investigating its concrete efficiency (see [33, 6] and references therein). To our knowledge, RingORAM [28] is the most efficient tree-based construction to date. RingORAM uses only two roundtrips during queries where the first is used to retrieve metadata. Unfortunately, the downside of RingORAM is the need for large client storage storing $O(N)$ block identifiers. Note, this is still smaller than the memory of $O(N)$ blocks as

identifiers are typically much smaller than data blocks. It is possible to reduce client storage by recursively storing the client storage in progressively smaller ORAMs, but this increases the number of roundtrips and amortized communication.

Constant-Communication ORAMs. One line of work including [7, 4, 11] uses homomorphic encryption (fully or additive) to obtain constant-communication ORAMs for large enough block sizes, at the cost of increased server computation. In our work, we focus on constructing ORAMs for arbitrary block sizes (including small blocks) and we keep server computation at a minimum (the server only performs XOR computations). We leave it as future work to consider combining our techniques with homorphic encryption to obtain constant amortized communication for large block sizes.

ORAMs for MPC. We note that there is a line of work (see [16, 34, 36, 9] and references therein) concerned with optimizing ORAM usage in multi-party computation (MPC). There is also work in developing languages to compile algorithms obliviously using ORAMs [35].

8 Conclusions

We present a new framework for building ORAMs that obtain fast online queries of a single roundtrip and one block of communication while maintaining small amortized bandwidth and client storage. To do this, we introduce the usage of lateral stashes that delay oblivious shuffling of large amounts of data. We present two constructions, LatORAM and Lat^{2}ORAM, that obtain smaller concrete communication costs than RingORAM and FutORAMa while maintaining the same blazing fast online queries of SQRT-ORAM. Our constructions achieve the best of both worlds.

Acknowledgments

Part of this work has been done while the second author was visiting Google, New York. The research of the second author at the Università di Salerno is supported by a Google Sponsored Research Agreement.

References

[1] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 403–432. Springer, Cham, May 2020.

[2] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. FutORAMa: A concretely efficient hierarchical oblivious RAM. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 3313–3327. ACM Press, November 2023.

[3] Zhao Chang, Dong Xie, and Feifei Li. Oblivious RAM: A Dissection and Experimental Evaluation. *Proceedings of the VLDB Endowment*, 9(12):1113–1124, 2016.

[4] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 345–360. ACM Press, November 2019.

[5] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious Serializable Transactions in the Cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 727–743, 2018.

[6] Jonathan L. Dautrich, Jr., Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 749–764. USENIX Association, August 2014.

[7] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 145–174. Springer, Berlin, Heidelberg, January 2016.

[8] Samuel Dittmer and Rafail Ostrovsky. Oblivious tight compaction in O(n) time with smaller constant. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 253–274. Springer, Cham, September 2020.

[9] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

[10] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and experience*, 24(3):327–336, 1994.

[11] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth oblivious RAM. Cryptology ePrint Archive, Report 2015/1065, 2015.

[12] `https://github.com/cryptobiu/FutORAMa`.

[13] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.

[14] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[15] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part II*, volume 6756 of *LNCS*, pages 576–587. Springer, Berlin, Heidelberg, July 2011.

[16] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012.

[17] William L. Holland, Olga Ohrimenko, and Anthony Wirth. Single round-trip hierarchical ORAM via succinct indices. In Jianying Zhou, Tony Q. S. Quek, Debin Gao, and Alvaro A. Cárdenas, editors, *ASIACCS 24*. ACM Press, July 2024.

[18] Sajid Hussain, Hui Guo, Tuo Li, and Sri Parameswaran. Mp-oram: A novel oram design for multicore processor systems. *IEEE Transactions on Dependable and Secure Computing*, pages 1–15, 2023.

[19] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Yuval Rabani, editor, *23rd SODA*, pages 143–156. ACM-SIAM, January 2012.

[20] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Cham, August 2018.

[21] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGPLAN Notices*, 50(4):87–101, 2015.

[22] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 311–324. ACM Press, November 2013.

[23] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya V. Lokam, Elaine Shi, and Vipul Goyal. HOP: Hardware makes obfuscation practical. In *NDSS 2017*. The Internet Society, February / March 2017.

[24] Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *ICALP 2014, Part II*, volume 8573 of *LNCS*, pages 556–567. Springer, Berlin, Heidelberg, July 2014.

[25] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd ACM STOC*, pages 514–523. ACM Press, May 1990.

[26] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In Mikkel Thorup, editor, *59th FOCS*, pages 871–882. IEEE Computer Society Press, October 2018.

[27] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. CacheShuffle: A family of oblivious shuffles. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 161:1–161:13. Schloss Dagstuhl, July 2018.

[28] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 415–430. USENIX Association, August 2015.

[29] Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. Design and Implementation of the Ascend Secure Processor. *IEEE Transactions on Dependable and Secure Computing*, 16(2):204–216, 2017.

[30] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, Berlin, Heidelberg, December 2011.

[31] Signal. Technology deep dive: Building a faster ORAM layer for enclaves enclaves. https://signal.org/blog/building-faster-oram/. Accessed, January 20, 2024.

[32] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy*, pages 253–267. IEEE Computer Society Press, May 2013.

[33] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

[34] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, abhi shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 191–202. ACM Press, November 2014.

[35] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015.

[36] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy*, pages 218–234. IEEE Computer Society Press, May 2016.

[37] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, page 72–84, New York, NY, USA, 2004. Association for Computing Machinery.

A A General Approach

In this section we present the general notion of a D-Lateral ORAM and, as we shall see, LatORAM and Lat^{2}ORAM are two special cases of this more general notion.

A *D-Lateral ORAM* arranges its lateral stashes in D levels, numbered $1, \ldots, D$. Each level is described by two parameters: the maximum number l of lateral stashes of the level and the number of *waves* w needed for a lateral stash of the level to reach its maximum number of real blocks. A D-Lateral ORAM is thus described by the size M of client stash and by the sequence $((l_1, w_1), \ldots, (l_D, w_D))$ of the parameters of its D levels. For convenience, it would be useful to consider the client stash as a 0-level with parameters $l_0 = 1$ and $w_0 = M$; that is, one lateral stash that reaches its maximum size after M waves of one query each. In addition Main can be considered as level $(D + 1)$. We will denote by $L(i, d)$, for $d = 1, \ldots, D$ and $i = 1, \ldots, l_d$, the i-th lateral stash of level d.

Life of a Lateral Stash. All stashes undergo a creation-destruction cycle consisting of three states: `unborn`, `growing`, and `decreasing`. We start with an imprecise description of the life of a lateral stash that does not take into account the dummy blocks. We will revise the description after we describe how queries are performed and how dummy blocks are instrumental for the obliviousness of the queries.

All lateral stashes, except for Main and the client stash, start as `unborn` with no blocks. Lateral stash $L(i, d)$ with $i > 1$ goes from `unborn` to `growing` when $L(i-1, d)$ goes from `growing` to `decreasing`. Lateral stash $L(1, d)$, for $d = 1, \ldots, D$, instead, enters state `growing` when level d is created. This guarantees that at any given time, each level has at most one lateral stash in state `growing`.

Once lateral stash $L(i, d)$ has entered state `growing`, it remains in this state for w_d waves. Each wave takes place every q_{d-1} queries and brings q_{d-1} new blocks that are randomly merged with $L(i, d)$. After w_d waves $L(i, d)$ enters state `decreasing` and it has the its maximum size of $s_d := w_d q_{d-1}$ blocks.

Life of a Level. All levels, except for Main, start empty. Level d is `alive` for a *d-epoch*, consisting of $q_d := l_d s_d$ queries (these are the queries needed to make all l_d lateral stashes of level d reach their maximum size s_d). At the end of a d-epoch, that is after q_d queries, all the blocks in level d are removed from level d that ceases to be alive. If $d = D$, the blocks removed are randomly merged with Main. If $d < D$, instead, all the blocks removed are merged with the lateral stash of level $d + 1$ that is in state `growing`. If level $d + 1$ is not `alive`, it is created and $L(1, d + 1)$ enters state `growing` and receives the blocks removed.

Let us give a closed formula of the number q_d of queries in a d epoch. As we have seen, a d-epoch contains $q_d = l_d s_d$ queries where $s_d = w_d q_{d-1}$. Moreover, for $d = 0$, we can set $q_0 = M$ since a 0-epoch consists of M queries after which the client stash (the 0-level later stash) is emptied and the blocks are obliviously merged to a lateral stash of level 1. Therefore, we have

$$\begin{aligned}
q_d &= l_d s_d \\
&= l_d w_d q_{d-1} && \text{(by expanding } s_d\text{)} \\
&= l_d w_d l_{d-1} s_{d-1} && \text{(by expanding } q_{d-1}\text{)} \\
&= l_d w_d l_{d-1} w_{d-1} q_{d-2} && \text{(by expanding } s_{d-1}\text{)} \\
&= \cdots \\
&= q_0 \prod_{j=1}^{d} l_j w_j \\
&= M \prod_{j=1}^{d} l_j w_j. && \text{(by using } q_0 = M\text{)}
\end{aligned}$$

Life of a Query. Let us give an intuition of how a query for a block is answered. The client has a position map that gives, for each of the N real blocks, the lateral stash $L(d, i)$ that contains the current copy. As we have done for LatORAM and Lat^{2}ORAM, the client asks for one block from each lateral stash that is

not `unborn` (that is, from each lateral stash that is in state `growing` or `decreasing`). More specifically, the client asks for the actual block to the lateral stash specified by the position map and for a dummy block from every other lateral stash. All blocks selected by the client for a query are removed from the respective lateral stashes, XORed together, and the result is returned to the client. All blocks selected during a query are removed from their respective lateral stashes. For this to work, each lateral stash must contain enough dummy blocks to function for all the queries that will be performed during its life.

Life of a Dummy Block. As we have seen, dummy blocks are crucial for the obliviousness of the queries and therefore each lateral stash must be equipped with a sufficient number of dummy block so to be able to take part in all queries that it will take part in.

Therefore, we modify the construction by having each wave include q_{d-1} dummy blocks, sufficient for the q_{d-1} queries until the next wave. The dummy blocks, together with q_{d-1} blocks that come from level $d-1$, are merged with the content of the one lateral stash of level d that is in state `growing`. The last wave, that is wave w_d, will instead carry q_{d-1} blocks plus a number of dummy equal to the queries that will be performed for the remaining of the life of level d. That is, in the last wave w_d, lateral stash $L(d,i)$ will receive $(l_d - i)s_d$ dummy blocks. Note that number of dummies is set so that when level d is merged with level $d+1$ all lateral stashes of level d contain exactly s_d blocks (which could be real or dummy).

Mapping LatORAM and Lat^{2}ORAM to the general model. LatORAM with parameters P and M is a 1-Lateral ORAM with a single $(P,1)$ 1-level. Indeed level 1 contains at most P lateral stashes and each takes 1 wave of $q_0 = M$ queries to reach its maximum size. An epoch of LatORAM consists of $l_1 w_1 q_0 = PM$ queries after which Main is reshuffled.

Lat^{2}ORAM with parameters E, P, and M is instead a 2-Lateral ORAM with two levels: level 1 is a $(P,1)$ level as before, whereas level 2 is a $(1,E)$ level; that is, there exists one 2-level lateral stash (called Main$'$ in Section 5) and it takes $w_2 := E$ waves to reach its maximum size. Thus in Lat^{2}ORAM a 1-epoch consists of PM queries and a 2-epoch of $w_2 l_2 w_1 l_1 q_0 = EPM$ queries. When a 2-epoch is completed the Main is reshuffled.

B Oblivious Merging

In this section we give formal definitions for the problem of *oblivious merging*, a two-party protocol between a client Clt and an honest-but-curious server Srv. The server holds c containers, each storing encrypted blocks, for a total of N blocks. The client wants to merge the c containers into one container so that the server does not have any information about the initial container of each block in the output container. As a note, oblivious shuffling [24, 27] corresponds to the special case of $c = N$ containers each with one block.

The common input $\mathcal{I} = (((\mathsf{Blk}_{i,j})_{j\in[t_i]})_{i\in[c]}, M)$ for oblivious merging consists of the $N = \sum_i t_i$ encrypted blocks partitioned into c server-stored containers. Here, t_i is the number of encrypted blocks for container i, and M is the size of the Clt's stash. The client Clt receives as private inputs the decryption key for the blocks and an *arrangement* ϕ that is compatible with $\mathcal{I}$; specifically, a function $\phi : [N] \to [c]$ such that the number of $j \in N$ such that $\phi(j) = i$ is exactly t_i.

The execution of an oblivious merging is a sequence of Clt moves: *download* the next block from a server container to the private stash; or *upload* a block from the stash to any server container. Clt is allowed to create temporary server containers during the protocol, but it is required that, at the end of the protocol, all the blocks are in the same container.

Definition 2 (Correctness)**.** *We say that* Clt *is* correct *if on common input $\mathcal{I}$ and private input ϕ compatible with $\mathcal{I}$, we have that: 1) each of the input blocks appears in encrypted form exactly once in the output container; 2) for each $j \in [N]$, the block in the j-th position of the output container is from input container $\phi(j)$.*

In addition to the above correctness conditions, we want to enforce the *obliviousness* condition; that is, we want to hide the original container of each block in the output container. We assume Srv to have no visibility of the stash but it can observe the source container of every download move as well as the

destination container of every upload move. We denote by $\mathsf{View}_{\mathsf{Clt}}(\mathcal{I}, \phi)$ the distribution of the view of Srv when interacting with the Clt on common input $\mathcal{I}$ and Clt's private input ϕ. The view consists of the blocks that are downloaded from the containers, along with the container number, and of the blocks that are uploaded to the server, along with the server container to which they are uploaded.

To formalize obliviousness we consider a simulator Sim that, for every $\mathcal{I}$, receives a stripped-down version $\hat{\mathcal{I}} = (c, (t_i)_{i \in c}, M)$ of $\mathcal{I}$ and outputs a view indistinguishable from a real view from $\mathsf{View}_{\mathsf{Clt}}(\mathcal{I}, \phi)$, for a random ϕ. More formally, we have the following.

Definition 3 (Obliviousness)**.** *We say that* Clt *is* oblivious *if there exists a simulator* Sim*, such that for every* $\mathcal{I} = (((\mathsf{Blk}_{i,j})_{j \in [t_i]})_{i \in [c]}, M)$*, the following two distributions are indistinguishable*

$$R(\mathcal{I}) = \{\phi \leftarrow \Phi(\mathcal{I}) : (\phi, \mathsf{View}_{\mathsf{Clt}}(\mathcal{I}, \phi))\}$$

and

$$S(\mathcal{I}) = \{\phi \leftarrow \Phi(\mathcal{I}) : (\phi, \mathsf{Sim}(\hat{\mathcal{I}}))\},$$

where $\Phi(\mathcal{I})$ *is the set of all possible arrangements that are compatible with* $\mathcal{I}$ *and* $\hat{\mathcal{I}} = (c, (t_i)_{i \in [c]}, M)$*.*

Note that in the above definition the simulator does not receive the blocks and ϕ, which is chosen at random in both distributions. Therefore the notion of an oblivious merging guarantees the security for any sequence of blocks and for random ϕ.

In the design of our merging algorithm, we assume that the blocks are encrypted with an IND-CPA secure encryption scheme and that they are decrypted before being stored in the stash and encrypted using fresh randomness before being uploaded. This makes the blocks computationally indistinguishable to Srv. In addition, we assume that the arrangement ϕ that Clt receives as a private input has a succinct representation

Proving obliviousness. The following two simple observations will be used to prove the obliviousness of all our merging and shuffling algorithms. It is easy to see that if we execute two oblivious merging algorithms with randomly and independently sampled arrangement functions, then the composition is still oblivious. Also, it is straightforward to see that merging algorithm that exhibit a deterministic download and upload schedule; that is, at any given step of the algorithm it is prefixed whether the client uploads or downloads a block and, in case of downloads, the container of origin is also fixed.

Oblivious merging in ORAMs. We apply merging to to encrypted ORAM blocks (both dummy and real). In this case the blocks have identities and the adversary might even known which blocks are in each container. This does not harm obliviousness as all our ORAM constructions guarantee that the blocks of the containers to be merged are randomly permuted within the container. Therefore hiding the container of origin, which is what oblivious merging achieves, is sufficient to hide all information about the blocks in the output container. For this to hold it is crucial that all containers are created with randomly permuted blocks and this is achieved by the $\mathsf{SmallShuffle}$ algorithm that randomly shuffles the the blocks from the Stash into a newly created $\mathsf{LatStash}$.

C Further discussions on CacheShuffle

In this section, we present a description of the adapted $\mathsf{CacheShuffle}$ algorithm and its analysis. A crucial distinction from the original $\mathsf{CacheShuffle}$ algorithm [27] lies in the desired output. While the original algorithm takes both a source and a specific target permutation as input aiming to realize that target, our adaptation only needs to produce *any* uniformly random permutation of the blocks in the initial configuration.

C.1 Description

For the remainder of the algorithm description, all blocks are assumed to be re-encrypted prior to being uploaded to the server.

The $\mathsf{CacheShuffle}$ algorithm operates in two primary phases: Spray and $\mathsf{Recalibrate}$ that we describe below. A formal pseudocode of $\mathsf{CacheShuffle}$ is found in Algorithm 13. We note that Algorithm 13 uses

the MultiPermDS to implement the row assignment function π dynamically (i.e. on the fly) using only $O(r)$ memory.

Initialization. Before the Spray phase begins, the client initializes r empty queues, where each queue Q_i corresponds to the i-th row of the conceptual $r \times c$ matrix. These queues serve as temporary local storage for blocks destined for a specific row but not yet uploaded during the Spray phase. Concurrently, the server allocates a temporary cache matrix T, with dimensions $r \times \sqrt{N}$. The client also generates a random mapping $\pi : [N] \to [r]$, assigning each of the N logical block positions to a target row index. This mapping ensures each row index i is assigned exactly $c = N/r$ blocks, forming a uniformly random multi-permutation.

Phase 1: Spray. The Spray phase proceeds over $\sqrt{N}$ rounds (indexed $j = 1$ to $\sqrt{N}$). In each round j:

- The client downloads the next batch of $\sqrt{N}$ blocks from their original locations on the server. Let these blocks be $\mathsf{blk}_1^j, \ldots \mathsf{blk}_{\sqrt{N}}^j$ corresponding to the specific segment of the input blocks.
- For each download block blk_i^j, the client determines its target row index $z = \pi((j-1)\sqrt{N} + i)$. The client then pushes this blk_i^j to its local queue Q_z.
- The client attempts to pop one block from each of its r queues, $Q_1, \ldots, Q_r$. If the queue is non-empty, the client pops the queue and retrieves that block. Otherwise, the client generates a dummy block. The client prepares these r blocks to upload to the server.
- The client uploads the r blocks (which are either popped from Q_is or generated as dummies) to the server. The designated location of the i-th block (retrieved from Q_i or generated as dummy) is $T[i, j]$.

Phase 2: Recalibrate. After $\sqrt{N}$ rounds of Spray, the temporary matrix T is fully populated, and some blocks may remain in the client's queues. The Recalibrate phase finalizes the permutation row-by-row. The phase proceeds over r rounds, indexed $i = 1$ to r. In the i-th round:

- The client downloads the entire i-th row from the server's temporary cache matrix T, receiving $\sqrt{N}$ blocks.
- The client discards dummy blocks among the received $\sqrt{N}$ blocks. It then combines the remaining valid blocks with blocks still residing in its local queue Q_i. At this point, the client holds exactly c blocks in total for row i.
- The client performs a local uniform random shuffle of these c blocks.
- The client uploads the shuffled c blocks to their final positions corresponding to the i-th row of the permuted output arrangement on the server.

Upon completion of Recalibrate phase for all r rows, the blocks on the server represent a uniform random permutation of the N input blocks (e.g. flatten the matrix in a row major order).

C.2 Analysis

The entire protocol involves transferring $N + r \cdot \sqrt{N} + r \cdot \sqrt{N} + N = 2N + 2r \cdot \sqrt{N}$ blocks. Setting $r = (1+\epsilon)\sqrt{N}$, the total communication is therefore $\approx 4N$ blocks.

The analysis for bounding the size of the client stored blocks (specifically, the maximum total size of the local queues Q_is) presented in [27] also applies to our adapted algorithm. We refer the reader to [27] for the detailed derivation.

Theorem 4 ([27], adapted)**.** *Suppose that $r = (1+\epsilon)\sqrt{N}$ for some constant $\epsilon > 0$. Then* CacheShuffle *incurs $O(\sqrt{N})$ client storage except with negligible probability in N.*

By setting $r = (1+\epsilon)\sqrt{N}$, π requires $O(r) = O(\sqrt{N})$ memory. Therefore, the total client storage size is bounded by $O(\sqrt{N})$ except with negligible probability in N.

The next lemma bounds the number of roundtrips.

Algorithm 13 $\mathsf{CacheShuffle}(X; \perp, K_E, K_D, K_L)$

Input: Server-stored input blocks, encryption key and PRF keys for labels and dummies.
Output: Z: server-stored output.

Initialization (executed by client and server):

Denote $x \leftarrow \sqrt{|X|}$.
Client and server both agrees on $r \leftarrow (1+\epsilon)|X|$ and $c \leftarrow |X|/r$ for a constant $\epsilon > 0$.
Client locally initializes r empty queues $Q_1, \ldots, Q_r$.
Server locally initializes a temporary cache matrix T of dimension $r \times x$.

Spray Phase (executed by client and server):

Initialize $\mathsf{DS} \leftarrow \mathsf{MultiPermDS.Setup}(c, \ldots, c)$ (c repeating r times).
for $j \in [x]$ **do**
  Download x next blocks from X as $(\tilde{\mathtt{lab}}_1^j, \tilde{\mathsf{blk}}_1^j), \ldots, (\tilde{\mathtt{lab}}_x^j, \tilde{\mathsf{blk}}_x^j)$.
  for $i \in [x]$ **do**
    Compute $z \leftarrow \mathsf{DS.Next}()$.
    Push $(\tilde{\mathtt{lab}}_i^j, \tilde{\mathsf{blk}}_i^j)$ to Q_z.
  for $i \in [r]$ **do**
    if $|Q_i| > 0$ **then**
      Pop from Q_i as $(\tilde{\mathtt{lab}}, \tilde{\mathsf{blk}})$.
      Set $(k, B_k) \leftarrow \mathsf{Dec}(K_E, \tilde{\mathsf{blk}})$.
      Set $\mathtt{lab}_i^j \leftarrow F_L(K_L, k)$ and $\mathsf{blk}_i^j \leftarrow \mathsf{Enc}(K_E, k \,\|\, B_k)$.
    else
      Set $\mathtt{lab}_i^j \leftarrow F_L(K_L, \mathtt{cs} \,\|\, \mathtt{dmy} \,\|\, i \,\|\, j)$.
      Set $\mathsf{blk}_i^j \leftarrow F_D(K_D, \mathtt{lab}_i^j)$.
  Upload $(\mathtt{lab}_1^j, \mathsf{blk}_1^j), \ldots, (\mathtt{lab}_r^j, \mathsf{blk}_c^j)$ to the server.
  The server sets $T[i, j] \leftarrow (\mathtt{lab}_i^j, \mathsf{blk}_i^j)$.

Recalibrate Phase (executed by client and server):

The server initializes the final output Z of length $|X|$, initially unpopulated.
for $i \in [r]$ **do**
  Download the entire row of $T[i]$ as $(\tilde{\mathtt{lab}}_i^1, \tilde{\mathsf{blk}}_i^1), \ldots, (\tilde{\mathtt{lab}}_i^x, \tilde{\mathsf{blk}}_i^x)$.
  Initialize an empty list A.
  for $j \in [x]$ **do**
    if $\tilde{\mathsf{blk}}_i^j$ is real **then**
      Append $(\tilde{\mathtt{lab}}_i^j, \tilde{\mathsf{blk}}_i^j)$ to A.
  Append the remaining blocks in Q_i to A. ▷ $|A| = c$
  Randomly shuffle A.
  for $j \in [c]$ **do**
    Set $(\tilde{\mathtt{lab}}, \tilde{\mathsf{blk}}) \leftarrow A[j]$.
    Set $(k, B_k) \leftarrow \mathsf{Dec}(K_E, \tilde{\mathsf{blk}})$.
    Set $\mathtt{lab}_i^j \leftarrow F_L(K_L, k)$ and $\mathsf{blk}_i^j \leftarrow \mathsf{Enc}(K_E, k \,\|\, B_k)$.
  Upload $(\mathtt{lab}_i^1, \mathsf{blk}_i^1), \ldots, (\mathtt{lab}_i^c, \mathsf{blk}_i^c)$ to the server.
  The server sets $Z[(i-1) \cdot c + j] \leftarrow (\mathtt{lab}_i^j, \mathsf{blk}_i^j)$.

Lemma 1. *Let N be the number of items to shuffle in* CacheShuffle*, and suppose the client memory is $O(\sqrt{N})$. Then the number of roundtrips required in* CacheShuffle *is $(2+\epsilon)\sqrt{N}$.*

Proof. Let $s = \sqrt{N}$ and $q = (1+2/\epsilon)\sqrt{N}$. The Spray phase proceeds in q rounds, where each round consists of downloading s blocks and uploading q blocks. Each round involves only one roundtrip, and hence incurs q roundtrips. The Recalibrate phase also proceeds in q rounds, where each round consists of downloading s blocks and uploading some number of blocks. Each round involves only one roundtrip, and hence incurs q roundtrips. In total, the number of roundtrips in CacheShuffle is $2q = (2+\epsilon)\sqrt{N}$. □

Finally, obliviousness follows immediately from the fact that the patterns of downloads and uploads are deterministic and independent of the input arrangement.

D TwoMerge Analysis

Regarding the obliviousness of TwoMerge, we observe that it follows from the obliviousness of CacheShuffle and from the fact that, following the execution of CacheShuffle, the patterns of downloads and uploads are deterministic and independent of the input arrangement.

To analyze the failure probability of TwoMerge we will be using the well-known Chernoff bound.

Theorem 5. *Let $X_0, \ldots, X_{n-1}$ be independent 0/1 random variables and set $X = X_0 + X_1 + \ldots + X_n$. Then, for all $0 < \delta < 1$, $\text{Prob}\left[X \leq (1-\delta)\cdot \mathsf{E}[X]\right] \leq e^{-\frac{\delta^2}{2}\cdot \mathsf{E}[X]}$. Moreover, for $\delta > 0$, $\text{Prob}\left[X \geq (1+\delta)\cdot \mathsf{E}[X]\right] \leq e^{-\frac{\delta^2}{2+\delta}\cdot \mathsf{E}[X]}$.*

Throughout this section, we use $t_1 = |C_1|$ and $t_2 = |C_2|$ for convenience.

Theorem 6. *Suppose that the initialization phase of* $\mathsf{TwoMerge}_{q_1,q_2}$ *is successful. Then* TwoMerge *aborts with probability at most $\frac{t_1+t_2}{M}(e^{-(t_1+t_2)q_1^2/(2t_1M)} + e^{-(t_1+t_2)q_2^2/(2t_2M)})$.*

Proof. We can imagine that TwoMerge attempts to produce a random permutation of length t_1+t_2 with t_1 ones and t_2 zeroes. Note that the algorithm aborts if and only if there exists an interval $[(i-1)\cdot M+1, i\cdot M]$ such that the sub-permutation corresponding to this interval contains less than $l_1 = \lfloor \frac{t_1 M}{t_1+t_2} \rfloor - q_1$ ones or less than $l_2 = M - l_1 - q_1 - q_2$ zeroes.

Without loss of generality, choose an arbitrary interval, say the first interval $[1, M]$. We will first bound the probability that this interval contains less than l_1 ones in a random permutation. The other case of containing less than l_2 zeroes works similarly.

Let S_i be an indicator random variable denoting whether the i-th element of the permutation is a one. Let $X_1 = \sum_{i=1}^{M} S_i$, the number of ones in the interval. Since the collection of random variables $\{S_i\}_{i\in[t_1+t_2]}$ is negatively associated (in particular, any subset of them are also negatively associated), we can apply Chernoff bound on X_1. By symmetry, $E[S_i] = \frac{t_1}{t_1+t_2}$, and hence $E[X_1] = \mu_1 = \frac{t_1 M}{t_1+t_2}$. Now we have

$$\begin{aligned}
\Pr[X_1 < l_1] &= \Pr\left[X_1 < \left\lfloor \frac{t_1 M}{t_1+t_2} \right\rfloor - q_1\right] \\
&\leq \Pr[X_1 \leq \lfloor \mu_1 \rfloor - q_1 - 1] \\
&\leq \Pr[X_1 \leq \mu_1 - q_1] \\
&= \Pr[X_1 \leq (1 - q_1/\mu_1)\mu_1] \\
&\leq e^{-q_1^2/(2\mu_1)} = e^{-(t_1+t_2)q_1^2/(2t_1M)}
\end{aligned} \tag{1}$$

By an almost identical argument, if X_2 is the random variable denoting the number of zeroes in the interval, we have

$$\Pr[X_2 < l_2] \leq e^{-(t_1+t_2)q_2^2/(2t_2M)}$$

These bounds in fact apply to other intervals identically, and hence, by the union bound,

$$\Pr[Abort] \leq \frac{t_1+t_2}{M}\left(e^{-(t_1+t_2)q_1^2/(2t_1M)} + e^{-(t_1+t_2)q_2^2/(2t_2M)}\right)$$

completing the proof. □

We can plug in the correct parameters to complete the proof of the main analysis of TwoMerge presented in Section 3.3.

Proof of Theorem 1. Rearranging the inequality

$$\frac{t_1+t_2}{M}e^{-((t_1+t_2)q_i^2)/(2t_iM)} \leq \delta/2$$

in terms of q_i gives the desired bound. □

E Additional Experimental Data

In this appendix we give details on the numbers of Figure 3.

E.1 The parameters used for LatORAM and Lat²ORAM

The numbers for LatORAM and Lat²ORAM for $N = 2^{26}, 2^{28}, 2^{30}$ of Figure 3 have been computed using the analysis in the paper. Specifically, for LatORAM we use $P = 2\sqrt{N/M}$ which, for our choice of $M = 8\sqrt{N}$, gives $P = (N/4)^{1/4}$. We compute the size of the position map to be $PM\log_2(N)$ bits; that is, $\log_2(N)$ bits for each of the at most PM blocks in the lateral stashes. The online bandwidth consists of the at most P labels (we consider 128 bits per label) that the client sends to the server and the one block that is transferred from the server to the client. For the amortized bandwidth we consider the amortized cost of $P + 7.5$ blocks obtained from the analysis in Section 4.3 by neglecting the $o(1)$ term. For Lat²ORAM we set $E = (N/M)^{1/3}$ which, for our choice of $M = 8\sqrt{N}$, gives $E = 1/2N^{1/6}$ and $P = 2E = N^{1/6}$. Since we have at most EPM blocks in the lateral stashes, the position map is computed to be $EPM\log_2(N)$-bit long. The computation of the online bandwidth is similar to the one of LatORAM and for the amortized cost we consider the analysis of Section 5.2 which gives the amortized bandwidth to be $3E + 4$. Again, we ignore the $o(1)$ additive term. In Figure 8 we give the breakdown of the memory usage and of the bandwidth for LatORAM and Lat²ORAM along with the parameters used to instantiate the two constructions.

	LatORAM						Lat²ORAM						
		Client Memory			**Bandwidth**				**Client Memory**			**Bandwidth**	
N	P	Stash	PM	**Total**	Online	Amortized	E	P	Stash	PM	**Total**	Online	Amortized
2^{26}	64	64	13	77	2	73.5	10	20	64	40.63	104.63	1.33	35.33
2^{28}	91	128	39.60	167.60	2.41	100.42	13	26	128	147.88	275.88	1.42	44.42
2^{30}	128	256	120.00	376.00	3.00	138.5	16	32	256	480.00	736.00	1.52	53.52

Figure 8: The breakdown of memory usage and bandwidth for LatORAM and Lat²ORAM as per our analysis. Memory is in MB and bandwidth is blocks.

E.2 The parameters used for RingORAM

Let us start by giving a brief overview of the working of RingORAM and of the formulae that we have used to estimate the performance of RingORAM in Figure 3. For a more complete treatment we refer the reader to the original paper [28].

For a database of size N, RingORAM is instantiated by specifying 3 parameters, denoted A, Z and S. The server memory of RingORAM is organized in a binary tree of depth $L := \log(2N/A)$ and with $2N/A$ leaves and about $4N/A$ nodes, called *clusters*. Each cluster contains Z real blocks and S dummy blocks for a total of about $4N(Z+S)/A$ blocks. Each clusters, besides the $Z+S$ blocks, contains a small metadata buffer that gives information regarding the real and the dummy blocks of the cluster. According to Table 6 of [28], the metadata of each cluster consists of

$$Z \cdot (L + \log N + \log(Z+S) + 1) + S + \log S$$

bits.

The real block B_x of index $x \in [N]$ is hosted in a cluster that belongs to the path from the root to a leaf of the tree, denoted $\mathsf{leaf}(x)$. RingORAM does not keep in memory the actual position of B_x but only $\mathsf{leaf}(x)$. This requires $N \cdot \log(2N/A)$ bits. Therefore to perform a query for block B_x, RingORAM first downloads the metadata buffer of the $L+1$ clusters on the path from the root to $\mathsf{leaf}(x)$. This allows to locate B_x and one dummy block for each of the L remaining clusters along the path. Finally, the server returns one block obtained by XORing the $(L+1)$ blocks requested by the client. Once downloaded, the block B_x is assigned to a newly randomly chosen leaf and stored in the *overflow* buffer.

RingORAM performs two extra operations. Every A queries the paths are evicted in the reverse bitwise order and all blocks along the path and blocks in the overflow buffer are pushed as deep as possible in the path (compatibly with their assigned leaves). This *eviction* procedure has a bandwidth of $Z + 2S$ blocks and it requires a buffer, that we call the *eviction* buffer, of size $(L+1)Z$ blocks. In addition, if all the S dummy blocks of a cluster are touched by a query then the Z real blocks of a cluster are downloaded and $Z+S$ fresh blocks are uploaded again. This procedure is called *early shuffle*. Following the analysis in [28], the total amortized bandwidth contribution of eviction and early shuffle is upper bounded by

$$(L+1)(2Z+S) \cdot \frac{1 + \mathsf{Poisson}(A,S)}{A},$$

where $\mathsf{Poisson}(A,S)$ is an approximation that a cluster is touched by S queries, and thus needs to be early shuffled, between two successive eviction procedures.

Finally, the memory needed by RingORAM consists of three components:

- The *eviction buffer*, denoted by Ev, of size $Z \cdot (L+1)$ blocks that is used to store Z blocks for the $(L+1)$ clusters from the path being evicted;
- The *overflow* buffer, denoted by Of, used to store the blocks that are waiting to be stored on the tree. According to [28], by setting
$$R = \frac{\kappa - \ln(1 - e^{-q})}{\ln(2Z/A)}$$
where $q = Z \cdot \ln(2Z/A) + A/2 - Z - \ln(4)$, we have that the probability that the are more than R blocks in the overflow buffer is at most $2^{-\kappa}$. In our computation we took $\kappa = 80$.
- The *node to leaf* association, denoted by PM, requiring $N \cdot L$ bits;

	Client Memory				Bandwidth		
N	Ev	Of	PM	**Total**	Online	Ev + ES	**Amortized**
2^{20}	1500	17.81	1.75 MB	3.23 MB	6.64	33.87	40.51
2^{22}	1700	17.81	8.00 MB	9.68 MB	7.81	38.38	46.19
2^{24}	1900	17.81	36.00 MB	37.87 MB	9.08	42.90	51.98
2^{26}	2100	17.81	160.00 MB	162.07 MB	10.44	47.41	57.85
2^{28}	2300	17.81	704.00 MB	706.26 MB	11.90	51.93	63.83

Figure 9: The Ev and the Of as well as the bandwidth are measured in blocks. The Online bandwidth includes the metadata downloaded by the client as well as the one block resulting from the XOR. The bandwidth of Eviction and Early Shuffle are amortized. For all test cases we chose $Z = 100$ and $A = S = 168$. The server stores about $6N$ blocks, counting real and dummy but not counting metadata.