# Sub-logarithmic Distributed Oblivious RAM with Small Block Size[*]

Eyal Kushilevitz and Tamer Mour

Computer Science Department, Technion, Haifa 32000, Israel
{eyalk,tamer.mour}@cs.technion.ac.il

**Abstract.** *Oblivious RAM* (ORAM) is a cryptographic primitive that allows a client to securely execute RAM programs over data that is stored in an untrusted server. *Distributed Oblivious RAM* is a variant of ORAM, where the data is stored in $m > 1$ servers. Extensive research over the last few decades have succeeded to reduce the bandwidth overhead of ORAM schemes, both in the single-server and the multi-server setting, from $O(\sqrt{N})$ to $O(1)$. However, all known protocols that achieve a sub-logarithmic overhead either require heavy server-side computation (e.g. homomorphic encryption), or a large block size of at least $\Omega(\log^3 N)$.

In this paper, we present a family of distributed ORAM constructions that follow the hierarchical approach of Goldreich and Ostrovsky [GO96]. We enhance known techniques, and develop new ones, to take better advantage of the existence of multiple servers. By plugging efficient known hashing schemes in our constructions, we get the following results:

1. For any number $m \geq 2$ of servers, we show an $m$-server ORAM scheme with $O(\log N/\log\log N)$ overhead, and block size $\Omega(\log^2 N)$. This scheme is private even against an $(m-1)$-server collusion.
2. A three-server ORAM construction with $O(\omega(1) \cdot \log N/\log\log N)$ overhead and a block size almost logarithmic, i.e. $\Omega(\log^{1+\epsilon} N)$.

We also investigate a model where the servers are allowed to perform a linear amount of light local computations, and show that constant overhead is achievable in this model, through a simple four-server ORAM protocol. Through the theoretical lens, this is the first ORAM scheme with asymptotic constant overhead, and polylogarithmic block size, that does not use homomorphic encryption. Practically speaking, although we do not provide an implementation of the suggested construction, evidence from related work (e.g. [DS17]) makes us believe that despite the linear computational overhead, the construction can be potentially very efficient practically, in particular when applied to secure computation.

**Keywords:** Oblivious RAM, Multi-Server Setting, Secure Computation, Private Storage.

## 1 Introduction

Since it was first introduced by Goldreich and Ostrovsky [GO96], the *Oblivious RAM* problem has attracted a lot of attention (see, e.g. [SCSL11, KLO12,

---

[*] A first technical report was published on arXiv.org e-Print archive as arXiv:1802.05145 [cs.CR].

WCS15]). Throughout the past three decades, efficient ORAM protocols were constructed (e.g. [GM11, SvDS+13]), their various applications, such as secure storage [OS97, AKST14], secure processors [RYF+13], and secure multi-party computation [GKK+12, LO13], were studied, and their limits were considered [GO96, AFN+17, LN18].

*Standard Model.* The standard ORAM model considers a setting where a client outsources his data to an untrusted server that supports read and write operations only. The goal of an *ORAM simulation* is to simulate any RAM program that the client executes over the remote data, so that the same computation is performed, but the view of the server during the interaction would provide no information about the client's private input and the program executed, except their length. Clearly, encryption can be employed to hide the *content* of the data, but the sequence of reads and write locations might leak information as well. Thus, the focus of ORAM is to hide the *access pattern* made to the server. The main metric considered in ORAM research is the *bandwidth overhead* of an ORAM scheme (shortly referred to as "overhead"), which is the multiplicative increase in the amount of communication incurred by an oblivious simulation relative to a regular run of the simulated program. In this standard model, researchers have been able to improve the overhead from $O(\log^3 N)$ [GO96] to $O(\log N)$ [SvDS+13, WCS15, AKL+18], where $N$ is the number of data blocks in storage, and thus reaching the optimal overhead in that model due to the matching impossibility results of Goldreich and Ostrovsky [GO96] and Larsen and Nielsen [LN18].

In an attempt to achieve sub-logarithmic overhead, research has deviated from the standard model (e.g. [LO13, AKST14, WGK18]). For instance, by allowing the server to perform some local computation, multiple works [AKST14, FNR+15, DvF+16] could achieve a constant overhead. However, this improvement comes at a cost: the server performs heavy homomorphic encryption computation which practically becomes the actual bottleneck of such schemes.

*Distributed Oblivious RAM.* Another interesting line of work, often referred to as *Distributed Oblivious RAM* [ZMZQ16, AFN+17, WGK18, etc.], was initiated by Ostrovsky and Shoup [OS97] and later refined by Lu and Ostrovsky [LO13], and considers the multi-server setting. We denote by $(m, t)$-*ORAM* an ORAM scheme that involves $m > 1$ servers, out of which $t < m$ servers might collude. In the two-server setting, Zhang et al. [ZMZQ16] and Abraham et al. [AFN+17] construct $(2, 1)$-ORAMs with sub-logarithmic overhead. In order to achieve $O(\log_d N)$ overhead (for any $d \in \mathbb{N}$ using their construction, Abraham et al. require that the size of a memory block, i.e. the data unit retrieved in a single query to the RAM, is $\Omega(d \log^2 N)$ (with larger blocks the asymptotic overhead increases). For example, for an overhead of $O(\log N / \log \log N)$, one has to work with blocks of relatively large size of $\Omega(\log^3 N)$, which may be undesired in many applications. In the work of Zhang et al., a polynomial block size of $\Omega(N^\epsilon)$ is required for a constant bandwidth blowup. Other attempts to achieve low overhead in the multi-server setting [MBM15] were shown to be vul-

nerable to concrete attacks [AFN$^+$17]. These recent developments in distributed ORAM raise the following question, which we address in this paper:

*Can we construct a sub-logarithmic distributed ORAM with a small block size?*

Known sub-logarithmic ORAMs [AFN$^+$17, ZMZQ16] belong to the family of *tree-based ORAMs* [SCSL11]. One of the key components in tree-based ORAMs is a *position map* that is maintained through a recursive ORAM. Such a recursion imposes the requirement for a large block size (usually polylogarithmic)[1]. Thus, it seems that a positive answer to the question above will come, if at all, from constructions belonging to the other well-studied type of ORAMs, those based on the hierarchical solution of [GO96]. By applying the hierarchical approach to the distributed setting, Lu and Ostrovsky [LO13] succeeded to construct the first logarithmic *hierarchical* ORAM scheme. In this paper, we show how to take a further advantage of the multiple servers in order to beat the logarithmic barrier, and still use a relatively small block size, with constructions in both the two-server and three-server settings. In addition, we consider the case where $t > 1$, and show how to generalize our two-server solution to an $(m, m-1)$-ORAM, with the same asymptotic complexity, for any $m > 2$.

*Allowing Linear Server Work.* Although distributed ORAM schemes in the literature assume the servers are able to perform some server-side computations, most works limit them to be polylogarithmic in $N$ per read/write. An exception are two recent works [DS17, WGK18] that investigate oblivious RAM when the servers are allowed to perform computations that are linear in $N$. In this stronger variant, the relatively new cryptographic primitive of *function secret sharing* (FSS), introduced by Boyle et al. [BGI15], was shown to be useful for constructing schemes that are practically efficient [DS17], or that are of low interaction [WGK18]. However, none of the mentioned schemes achieve sub-logarithmic overhead, thus leaving us with the following question:

*How efficient can distributed ORAM be if the servers do linear work?*

We show that by allowing the servers to perform linear computations per RAM step, we can achieve a distributed ORAM scheme with a small constant overhead. We restrict ourselves to simple server-side computations which were shown to perform well in practical implementations [DS17]. Therefore, we believe that our construction has the potential to be very efficient in practical applications, such as those to secure computation, as discussed below.

*Application to Secure Computation.* Besides potentially being more efficient, distributed oblivious RAM was shown to be a useful tool in constructing secure computation protocols [OS97, GKK$^+$12, LO13]. By using our distributed

---

[1] To the best of our knowledge, the only tree-based ORAM that bypasses recursion belongs to Wang et al. [WGK18], which works in a different model where linear sever work is allowed (see preceding discussion).

| Scheme | $m$ | $t$ | Overhead | Block size $B$ | C. Strg. | S. Work |
|---|---|---|---|---|---|---|
| [Ost90, GO96] | 1 | - | $O(\log^3 N)$ | $\Omega(\log N)$ | $O(1)$ | - |
| [KLO12] | 1 | - | $O(\frac{\log^2 N}{\log\log N})$ | $\Omega(\log N)$ | $O(1)$ | - |
| [WCS15] | 1 | - | $O(\log N \cdot \omega(1))$ | $\Omega(\log^2 N)$ | $O(1)$ | - |
| [AKL+18] | 1 | - | $O(\log N)$ | $\Omega(\log N)$ | $O(1)$ | - |
| [LO13] | 2 | 1 | $O(\log N)$ | $\Omega(\log N)$ | $O(1)$ | polylog |
| [CKN$^+$18] | 3 | 1 | $O(\log^2 N)$ | $\Omega(\log N)$ | $O(1)$ | - |
| [ZMZQ16] | 2 | 1 | $O(1)$ | $\Omega(N^\epsilon)$ | $O(1)$ | polylog |
| [AFN$^+$17] | 2 | 1 | $O(\log_d N)$ | $\omega(d\log^2 N)$ | $O(1)$ | polylog |
| [DS17] | 2 | 1 | $O(\sqrt{N})$ | $\Omega(\log N)$ | $O(1)$ | linear |
| [WGK18] | 2 | 1 | $O(\log N)$ | $\Omega(\log N)$ | $O(1)$ | linear |
| our 4-server construction | | | | | | |
| Instantiation 1 | 4 | 1 | $O(1)$ | $\Omega(\lambda\log N)$ | $O(1)$ | linear |
| our 3-server construction | | | | | | |
| Instantiation 2 $d = \log^\epsilon N$ | 3 | 1 | $O(\log_d N \cdot \omega(1))$ $O(\frac{\log N}{\log\log N} \cdot \omega(1))$ | $\Omega(d\log N)$ $\Omega(\log^{1+\epsilon} N)$ | $O(1)$ | polylog |
| Instantiation 3 $d = \log^\epsilon N$ | 3 | 1 | $O(\log_d N)$ $O(\frac{\log N}{\log\log N})$ | $\Omega(d\log^{1.5} N)$ $\Omega(\log^{1.5+\epsilon} N)$ | $O(1)$ | polylog |
| our $m$-server construction | | | | | | |
| Instantiation 4 | $m \geq 2$ | $m-1$ | $O(\frac{\log N}{\log\log N})$ | $\Omega(\log^2 N)$ | $O(1)$ | polylog |

Table 1: Comparison of ORAM schemes. Columns by order: number of servers $m$, collusion size $t$, bandwidth overhead, block size $B$, client storage size in blocks (additive factors of the security parameter are omitted), and amortized server-side work per RAM step (computational overhead). $\lambda$ is a security parameter.

ORAM constructions as the underlying ORAM schemes in the elegant protocol from [LO13], we get better asymptotic parameters for secure RAM computation than any known single-server or multi-server ORAM solution. Furthermore, our constructions are the first to be applicable to multi-party computation protocols that are secure against collusions.

## 1.1 Our Contribution and Technical Overview

**Sub-logarithmic Distributed ORAM Constructions.** Our main contribution is a family of distributed hierarchical ORAM constructions with any number of servers. Our constructions make a black-box use of hashing schemes. Instantiating our constructions with known hashing schemes, that were previously used in ORAM constructions [GM11, KLO12, LO13, CGLS17], yields state-of-the-art results (see Table 1). We elaborate below.

*A Three-server ORAM Protocol.* By using techniques from [LO13] over the balanced hierarchy from [KLO12], and using two-server PIR [CGKS98] as a black box, we are able to construct an efficient $(3, 1)$-ORAM scheme. Instantiating the scheme with cuckoo hash tables (similarly to [GM11, KLO12, LO13]) achieves an

overhead of $O(\omega(1) \cdot \log_d N)$ with a block size of $B = \Omega(d \log N)$. Thus, for any $\epsilon > 0$, we achieve $O(\omega(1) \cdot \log N / \log \log N)$ overhead with $B = \Omega(\log^{1+\epsilon} N)$.

In the classic hierarchical solution from [GO96], the data is stored in $\log N$ levels, and the protocol consists of two components: *queries*, in which target virtual blocks are retrieved, and *reshuffles*, which are performed to properly maintain the data structure. Roughly speaking, in a query, a single block is downloaded from every level, resulting in $\log N$ overhead per query. The reshuffles cost $\log N$ overhead per level, and $\log^2 N$ overall. Kushilevitz et al. [KLO12] suggest to balance the hierarchy by reducing the number of levels to $\log N / \log \log N$. In the balanced hierarchy, however, one has to download $\log N$ blocks from a level in every query. Thus, balancing the hierarchy "balances", in some sense, the asymptotic costs of the queries and reshuffles, as they both become $\log^2 N / \log \log N$.

At a high level, we carefully apply two-server techniques to reduce the overhead, both of the queries and the reshuffles, in the single-server ORAM of [KLO12]. More specifically, to reduce the queries cost, we use two-server PIR to allow the client to efficiently read the target block from the $\log N$ positions, it had otherwise have to download, from every level. By requiring the right (relatively small) block size, the cost of PIRs can be made constant per level and, therefore, $\log N / \log \log N$ in total. To reduce the reshuffles cost, we replace the single-server reshuffles with the cheaper two-server reshuffles, that were first used by Lu and Ostrovsky [LO13], and that incur only a constant overhead per level.

So far, it sounds like we are already able to achieve $\log N / \log \log N$ overhead using two servers only. However, combining two-server PIR and two-server reshuffles is tricky: each assumes a different distribution of the data. In standard two-server PIR, the data is assumed to be identically replicated among the two servers. On the other hand, it is essential for the security of the two-server reshuffles from [LO13] that every level in the hierarchy is held only by one of the two servers, so that the other server, which is used to reshuffle the data, does not see the access pattern to the level. We solve this problem by combining the two settings using three servers: every level is held only by two of the three servers in a way that conserves the security of the two-server reshuffles and, at the same time, provides the required setting for two-server PIR.

*An $(m, m-1)$-ORAM Protocol.* We take further advantage of the existence of multiple servers and construct, for any integer $m \geq 2$, an $m$-server ORAM scheme that is private against a collusion of up to $m-1$ servers. Using oblivious two-tier hashing [CGLS17], our scheme achieves an overhead of $O(\log N / \log \log N)$, for which it requires $B = \Omega(\log^2 N)$ (see Theorem 4 and Instantiation 4).

We begin by describing a $(2, 1)$-ORAM scheme, then briefly explain how to extend it to any number of servers $m > 2$. Let us take a look back at our three-server construction. We were able to use both two-server PIR and two-server reshuffles only using a three-server setting. Now that we restrict ourselves to using two servers, we opt for the setting where the two servers store identical replicates of the entire data structure. Performing PIR is clearly still possible, but now that the queries in all levels are made to the same two servers, we cannot perform Lu and Ostrovsky's [LO13] two-server reshuffles securely. Instead, we

use *oblivious sort* (or, more generally, oblivious hashing) to reshuffle the levels. Oblivious sort is a sorting protocol in the client-server setting, where the server involved learns nothing about the obtained order of blocks. Oblivious sort is used in many single-server hierarchical ORAMs (e.g. [GO96,KLO12]), where it incurs $\log N$ overhead per level. Since we aim for a sub-logarithmic overhead, we avoid this undesired blowup by performing oblivious sort over the tags of the blocks only (i.e. their identities) which are much shorter, rather than over the blocks themselves. We require a block size large enough such that the gap between the size of the tags and the size of the blocks cancels out the multiplicative overhead of performing oblivious sort. Once the tags are shuffled into a level, it remains to match them with the blocks with the data. That is where the second server is used. We apply a secure two-server "matching procedure" which, at a high level, lets the second server to randomly permute the data blocks and send them to the server holding the shuffled tags. The latter can then match the data to the tags in an oblivious manner. Of course, the data exchange during the matching has to involve a subtle cryptographic treatment to preserve security.

The above scheme can be generalized to an $(m, m-1)$-ORAM, for any $m > 2$. The data is replicated in all servers involved, and $m$-server PIR is used. The matching procedure is extended to an $m$-server procedure, where all the servers participate in randomly permuting the data.

**Constant Overhead with Linear Server Work.** We also investigate the model where linear server work is allowed, and show that constant overhead is achievable in this model (see Table 1). The proposed scheme, described below, applies function secret sharing over secret-shared data, thus avoiding the need for encrypting the data using symmetric encryption (unlike existing schemes e.g. [DS17, WGK18]).

*A Simple Four-server ORAM Protocol.* Inspired by an idea first suggested in [OS97], we combine private information retrieval (PIR) [CGKS98], and PIR-write [OS97], to obtain a four-server ORAM. To implement the PIR and PIR-write protocols efficiently, we make a black-box use of *distributed point functions* (DPFs) [GI14, BGI15], i.e. function secret sharing schemes for the class of point functions. Efficient DPFs can be used to construct (i) a (computational) two-server PIR protocol if the data is replicated among the two servers, or (ii) a two-server PIR-write protocol for when the data is additively secret-shared among the two servers. These two applications of DPFs are combined as follows: we create two additive shares of the data, and replicate each share twice. We send each of the four shares (two pairs of identical shares) to one of the four servers. A read is simulated with two instances of PIR, each invoked with a different pair of servers holding the same share. A write is simulated with two instance of PIR-write, each invoked with a different pair of servers holding different shares.

We stress that the client in all of our constructions can be described using a simple small circuit, and therefore, our schemes can be used to obtain efficient secure multi-party protocols, following [LO13]. We elaborate on this in Appendix B.

## 1.2 Related Work

*Classic Hierarchical Solution.* The first hierarchical ORAM scheme appeared in the work of Ostrovsky [Ost90] and later in [GO96]. In this solution, the server holds the data in a hierarchy of levels, growing geometrically in size, where the $i^{th}$ level is a standard hash table with $2^i$ buckets of logarithmic size, and a hash function $h_i(\cdot)$, which is used to determine the location of blocks in the hash table: block of address $v$ may be found in level $i$ (if at all) in bucket $h_i(v)$. The scheme is initiated when all blocks are in the lowest level. An access to a block with a virtual address $v$ is simulated by downloading bucket $h_i(v)$ from every level $i$. Once the block is found, it is written back to the appropriate bucket in the smallest level ($i = 0$). As a level fills up, it is merged down with the subsequent (larger) level $i+1$, which is reshuffled with a new hash function $h_{i+1}$ using oblivious sorting. Thus, a block is never accessed twice in the same level with the same hash function, hence the obliviousness of the scheme. Using AKS sorting network [AKS83] for the oblivious sort achieves an $O(\log^3 N)$ overhead.

*Balanced Hierarchy.* Up until recently, the best known single-server ORAM scheme for general block size, with constant client memory, were obtained by applying an elegant "balancing technique" to the hierarchy of [GO96], that reduces the number of levels in the hierarchy, in exchange for larger levels. The technique was first suggested by Kushilevitz et al. [KLO12]. Their scheme achieves an overhead of $O(\log^2 / \log \log N)$, using *oblivious cuckoo hashing* (first applied to ORAM in [PR10, GM11]). An alternative construction, recently proposed by Chan et al. [CGLS17], follows the same idea, but replaces the relatively complex cuckoo hashing with a simpler oblivious hashing that is based on a variant of the two-tier hashing scheme from [ACMR95].

*Tree-based ORAM.* Another well-studied family of ORAM schemes is tree-based ORAMs (e.g. [SCSL11, SvDS$^+$13, WCS15, DvF$^+$16, AFN$^+$17]). Tree-based ORAMs, as the name suggests, are oblivious RAM schemes where the data is stored in a tree structure. The first ORAM constructions with a logarithmic overhead, in the single-server model, were achieved following the tree-based approach [SvDS$^+$13, WCS15]. However, tree-based ORAMs require relatively large block size of at least $B = \Omega(\log^2 N)$ (see Table 1).

*Optimal ORAM with General Block Size.* The recent work of Asharov et al. [AKL+18], which improves upon the work of Patel et al. [PPRY18], succeeds to achieve optimal logarithmic overhead with general block size (due to known lower bounds [GO96, LN18]). Both results are based on the solution from [GO96] and use non-trivial properties of the data in the hierarchy to optimize the overhead.

*Distributed ORAM Constructions.* Ostrovsky and Shoup [OS97] were the first to construct a distributed private-access storage scheme (that is not read-only). Their solution is based on the hierarchical ORAM from [GO96]. However, their model is a bit different than ours: they were interested in the amount of communication required for a single query (rather than a sequence of queries), and

they did not limit the work done by the servers. Lu and Ostrovsky [LO13] considered the more general ORAM model, defined in Section 2.1. They presented the first two-server oblivious RAM scheme, and achieved a logarithmic overhead with a logarithmic block size by bypassing oblivious sort, and replacing it with an efficient reshuffling procedure that uses the two servers.

The tree approach was also studied in the multi-server model. Contrary to the hierarchical schemes, known distributed tree-based ORAMs [ZMZQ16, AFN+17] beat the logarithmic barrier. The improvement in overhead could be achieved by using $k$-ary tree data structures, for some parameter $k = \omega(1)$. However, these constructions suffer from a few drawbacks, most importantly, they require a large polylogarithmic (sometimes polynomial) block size.

*Amortized vs. Worst-Case Overhead.* Although overhead is defined as the amortized blowup in bandwidth, several works also considered the worst-case overhead per query. While tree-based schemes typically achieve low worst-case overhead [SCSL11, SvDS+13, WCS15], hierarchical ORAMs might need additional work to turn their amortized overhead into worst-case overhead. Ostrovsky and Shoup [OS97] were the first to propose a method to de-amortize hierarchical constructions, which was adapted in some consequent works (e.g. [GM11, KLO12]).

*ORAMs with Linear Server-side Computational Overhead.* The work of Ostrovsky and Shoup [OS97], as well as some recent works [DS17, WGK18] have considered the model where the servers are allowed to perform a linear amount of light computations. Both the works of Doerner and Shelat [DS17] and Wang et al. [WGK18] elegantly implement techniques from the standard model (square-root construction, and tree structure, respectively), and use the efficient PIR protocol from [BGI15], to construct practically efficient two-server ORAM schemes with linear server-side computational overhead and bandwidth overhead matching their analogues in the single-server setting (see Table 1).

## 1.3   Paper Organization

Section 2 contains a formal definition of the model and problem, as well as a description of the cryptographic tools we use in our constructions. In Section 3, we present our simple four-server ORAM with linear server work. In Section 4, we provide a high-level description of the hierarchical ORAM framework, on which our main distributed ORAM constructions are based. In Sections 5 and 6, we present these constructions, beginning with a high-level overview of each of them, followed by the full details and analysis. We finish with a few open questions. Due to page limit, de-amortization of our constructions, and a discussion of their application to secure computation, are left to Appendix A and B (resp.), together with some additional complementary material.

## 2 Preliminaries

### 2.1 Model and Problem Definition

*The RAM Model.* We work in the RAM model, where a RAM machine consists of a CPU that interacts with a (supposedly remote) RAM storage. The CPU has a small number of local registers, therefore it uses the RAM storage for computations over large data, by performing a sequence of reads and writes to memory locations in the RAM. A sequence of $\ell$ queries is a list of $\ell$ tuples $(op_1, v_1, x_1), \ldots, (op_\ell, v_\ell, x_\ell)$, where $op_i$ is either Read or Write, $v_i$ is the location of the memory cell to be read or written to, and $x_i$ is the data to be written to $v_i$ in case of a Write. For simplicity of notation, we unify both types of operations into an operation known as an *access*, namely "Read then Write". Hence, the *access pattern* of the RAM machine is the sequence of the memory locations and the data $(v_1, x_1), \ldots, (v_\ell, x_\ell)$.

*Oblivious RAM Simulation.* A *(single-server) oblivious RAM simulation*, shortly *ORAM simulation*, is a simulation of a RAM machine, held by a client as a CPU, and a server as RAM storage. The client communicates with the server, and thus can query its memory. The server is untrusted but is assumed to be *semi-honest*, i.e. it follows the protocol but attempts to learn as much information as possible from its view about the client's input and program. We also assume that the server is not just a memory machine with I/O functionality, but that it can perform basic local computations over its storage (e.g. shuffle arrays, compute simple hash functions, etc.). We refer to the access pattern of the RAM machine that is simulated as the *virtual* access pattern. The access pattern that is produced by the oblivious simulation is called the *actual* access pattern. The goal of ORAM is to simulate the RAM machine correctly, in a way that the distribution of the view of the server, i.e. the actual access pattern, would look independent of the virtual access pattern.

**Definition 1 (ORAM simulation, informal).** *Let RAM be a RAM machine. We say that a (probabilistic) RAM machine ORAM is an* oblivious RAM simulation *of RAM, if (i) (correctness) for any virtual access pattern $\boldsymbol{y} := ((v_1, x_1), \ldots, (v_\ell, x_\ell))$, the output of RAM and ORAM at the end of the client-server interaction is equal with probability $\geq 1 - negl(\ell)$, and (ii) (security) for any two virtual access patterns, $\boldsymbol{y}, \boldsymbol{z}$, of length $\ell$, the corresponding distribution of the actual access patterns produced by ORAM, denoted $\tilde{\boldsymbol{y}}$ and $\tilde{\boldsymbol{z}}$, are computationally indistinguishable.*

An alternative interpretation of the security requirement is as follows: the view of the server, during an ORAM simulation, can be simulated in a way that is indistinguishable from the actual view of the server, given only $\ell$.

*Distributed Oblivious RAM.* A *distributed oblivious RAM simulation* is the analogue of ORAM simulation in the multi-server setting. To simulate a RAM machine, the client now communicates with $m$ semi-honest servers, rather than with a single server only.

**Definition 2 (Distributed ORAM Simulation, informal).** *An $(m,t)$-ORAM simulation $(0 < t < m)$ is an oblivious RAM simulation of a RAM machine, that is invoked by a CPU client and m remote storage servers, and that is private against a collusion of t corrupt servers. Namely, for any two actual access patterns $\boldsymbol{y}, \boldsymbol{z}$ of length $\ell$, the corresponding combined view of any t servers during the ORAM simulation (that consists of the actual access queries made to the t servers) are computationally indistinguishable.*

With the involvement of more servers, we can hope to achieve schemes that are more efficient as well as schemes that protect against collusions of servers.

*Parameters and Complexity Measures.* The main complexity measure in which ORAM schemes compete is the *bandwidth overhead* (or, shortly, *overhead*). When the ORAM protocol operates in the "balls and bins" manner [GO96], where the only type of data exchanged between the client and servers is actual memory blocks, it is convenient to define the overhead as the amount of actual memory blocks that are queried in the ORAM simulation to simulate a virtual query to a single block. However, in general, overhead is defined as the blowup in the number of information bits exchanged between the parties, relative to a non-oblivious execution of the program. Following the more general definition, the overhead is sometimes a function of the block size $B$. Clearly, we aim to achieve a small asymptotic overhead with block size as small as possible.

Other metrics include the size of the server storage and the client's local memory (in blocks), and the amount and type of the computations performed by the servers (e.g. simple arithmetics vs. heavy cryptography). We note that all of these notions are best defined in terms of overhead, compared to a non-oblivious execution of the program, e.g. storage overhead, computational overhead, etc..

## 2.2 Private Information Retrieval

*Private information retrieval (PIR)* [CGKS98] is a cryptographic primitive that allows a client to query a database stored in a remote server, without revealing the identity of the queried data block. Specifically, an array of $n$ blocks $X = (x_1, \ldots, x_n)$ is stored in a server. The client, with input $i \in [n]$, wishes to retrieve $x_i$, while keeping $i$ private. PIR protocols allow the client to do that while minimizing the number of bits exchanged between the client and server. PIR is studied in two main settings: single-server PIR, where the database is stored in a single server, and the multi-server setting, where the database is replicated and stored in all servers, with which the client communicates simultaneously. More specifically, an $(m,t)$-PIR is a PIR protocol that involves $m > 1$ servers and that is secure against any collusion of $t < m$ servers. It was shown in [CGKS98] that non-trivial single-server PIRs cannot achieve information-theoretic security. This is made possible with two servers already in multi-server PIR. Moreover, many known two-server PIRs (both information theoretic and computational, e.g. [CGKS98, BIW07, DG16, BGI15]) do not involve heavy server-side computation, like homomorphic encryption or number theoretic computations, as opposed to known single-server protocols (e.g. [KO97, CMS99, GR05]).

# 3 A Simple Four-Server ORAM with Constant Overhead

We present our four-server ORAM protocol with constant bandwidth overhead and linear server-side computational overhead. The protocol bypasses the need for symmetric encryption as it secret-shares the data among the servers. We use distributed point functions [GI14] (see Section 3.1 below) as a building block.

**Theorem 1 (Four-server ORAM).** *Assume the existence of a two-party DPF scheme for point functions $\{0,1\}^n \rightarrow \{0,1\}^m$ with share length $\Lambda(n,m)$ bits. Then, there exists a $(4,1)$-ORAM scheme with linear$^2$ server-side computation overhead and bandwidth overhead of $O(\Lambda(\log N, B)/B)$ for a block size of $B = \Omega(\Lambda(\log N, 1))$.*

Instantiating our scheme with the DPF from [BGI15] obtains the following.

**Instantiation 1** *Assume the existence of one-way functions. Then, there exists a $(4,1)$-ORAM scheme with linear server-side computation overhead and constant bandwidth overhead for a block size of $B = \Omega(\lambda \log N)$, where $\lambda$ is a security parameter.*

## 3.1 Building Block: Distributed Point Functions

*Distributed Point Functions* (DPF), introduced by Gilboa and Ishai [GI14], are a special case of the broader cryptographic primitive called *Function Secret Sharing* (FSS) [BGI15]. Analogous to standard secret sharing, an FSS allows a dealer to secret-share a function $f$ among two (or more) participants. Each participant is given a share that does not reveal any information about $f$. Using his share, each participant $p_i$, for $i \in \{0,1\}$, can compute a value $f_i(x)$ on any input $x$ in $f$'s domain. The value $f(x)$ can be computed by combining $f_0(x)$ and $f_1(x)$. In fact, $f(x) = f_0(x) + f_1(x)$. Distributed point function is an FSS for the class of point functions, i.e., all functions $P_{a,b} : \{0,1\}^n \rightarrow \{0,1\}^m$ that are defined by $P_{a,b}(a) = b$ and $P_{a,b}(a') = 0^m$ for all $a' \neq a$. Boyle et al. [BGI15] construct a DPF scheme where the shares given to the parties are of size $O(\lambda n + m)$, where $\lambda$ is a security parameter, that is the length of a PRG seed. We are mainly interested in the application of DPFs to PIR and PIR-write [GI14, BGI15].

## 3.2 Overview

Similarly to the schemes of [DS17, WGK18], we apply DPF-based PIR [BGI15] to allow the client to efficiently read records from a replicated data. If we allow linear server-side computational overhead, the task of oblivious reads becomes trivial by using DPFs. The remaining challenge is how to efficiently perform oblivious writes to the data.

The core idea behind the scheme is to apply DPFs not only for PIR, but also for a variant of *PIR-write*. PIR-write (a variant of which was first investigated

---

$^2$ Up to polylogarithmic factors.

in [OS97]) is the write-only analog of PIR. We use DPFs to construct a simple two-server PIR-write where every server holds an additive share of the data. Our PIR-write protocol is limited in the sense that the client can only modify an existing record by some difference of his specification (rather than specifying the new value to be written). If the client has the ability to read the record in a private manner, then this limitation becomes irrelevant.

We combine the read-only PIR and the write-only PIR-write primitives to obtain a four-server ORAM scheme that enables both private reads and writes. In the setup, the client generates two additive shares of the initial data, $X^0, X^1$ s.t. $X = X^0 \oplus X^1$, and replicates each of the shares. Each of the four shares obtained is given to one of the servers. For a private read, the client retrieves each of the shares $X^0, X^1$, using the DPF-based PIR protocol, with the two servers that hold the share. For a private write, the proposed PIR-write protocol is invoked with pairs of servers holding different shares of the data (see illustration in Figure 1).

We remark that our method to combine PIR and PIR-write for ORAM is inspired by the 8-server ORAM scheme presented in [OS97], in which an elementary 4-server PIR-write protocol was integrated with the PIR from [CGKS98].
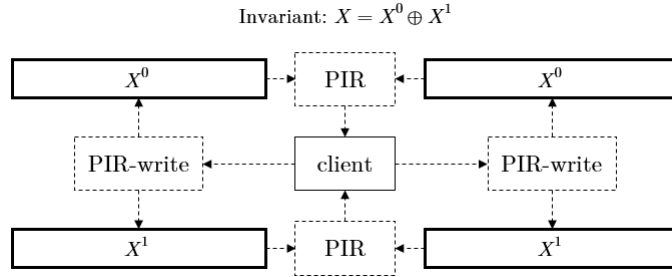
Invariant: $X = X^0 \oplus X^1$



Fig. 1: an illustration of the four-server construction.

### 3.3 Oblivious Read-only and Write-only Schemes with Two Servers

*Basic PIR and PIR-write.* Recall the classic two-server PIR protocol, proposed in [CGKS98]. To securely retrieve a data block $x_i$ from an array $X = (x_1, \ldots, x_N)$ that is stored in two non-colluding servers $\mathcal{S}_0$ and $\mathcal{S}_1$, the client generates two random $N$-bit vectors, $e_i^0$ and $e_i^1$ such that $e_i^0 \oplus e_i^1 = e_i$, where $e_i$ is the $i^{th}$ unit vector, and sends $e_i^b$ to $\mathcal{S}_b$. In other words, the client secret-shares the vector $e_i$ among the two servers. Then, each server, computes the inner product $x_i^b := X \cdot e_i^b$ and sends it to the client. It is easy to see that $x_i = x_i^0 \oplus x_i^1$.

The same approach can be used for two-server PIR-write. However, now we require that the data is shared, rather than replicated, among the two servers. Namely, server $\mathcal{S}_b$ holds a share of the data $X^b$, such that $X^0 \oplus X^1 = X$. In order to write a new value $\hat{x}_i$ to the $i^{th}$ block in the array, the client secret-shares the vector $(\hat{x}_i \oplus x_i)e_i$ to the two servers. Each of the servers adds his share to $X^b$, and obtains a new array $\hat{X}^b$. After this update, the servers have additive shares of $X$ with the updated value of $x_i$. Notice that we assume that the client already read and knows $x_i$; this is not standard in the PIR-write model.

12

*Efficient PIR and PIR-write via DPFs.* In the heart of the PIR and PIR-write protocols described above is the secret sharing of vectors of size $N$. Applying standard additive secret sharing yields protocols with linear communication cost. Since we share a very specific type of vectors, specifically, unit vectors and their multiples, standard secret sharing is an overkill. Instead, we use DPFs. The values of a point function $P_{i,x} : [N] \to \{0,1\}^m$ (that evaluates $x$ at $i$, and zero elsewhere) can be represented by a multiple of a unit vector $v_{i,x} := xe_i$. Hence, one can view distributed point functions as a means to "compress" shares of unit vectors and their multiples. We can use DPFs to share such a vector among two participants $p_0$ and $p_1$, as follows. We secret-share the function $P_{i,x}$ using a DPF scheme, and generate two shares $P_{i,x}^0$ and $P_{i,x}^1$. For $b \in \{0,1\}$, share $P_{i,x}^b$ is sent to participant $p_b$. The participants can compute their shares of the vector $v_{i,x}$ by evaluating their DPF share on every input in $[N]$. Namely, $p_b$ computes his share $v_{i,x}^b := (P_{i,x}^b(1), \ldots, P_{i,x}^b(n))$. From the correctness of the underlying DPF scheme, it holds that $v_{i,x}^0 \oplus v_{i,x}^1 = v_{i,x}$. Further, from the security of the DPF, the participants do not learn anything about the vector $v_{i,x}$ except the fact that it is a multiple of a unit vector. Using the DPF construction from [BGI15], we have a secret sharing scheme for unit vectors and their multiples, with communication complexity $O(\lambda \log N + m)$, assuming the existence of a PRG $G : \{0,1\}^\lambda \to \{0,1\}^m$.

### 3.4 Construction of Four-Server ORAM

*Initial Server Storage.* Let $\mathcal{S}_0^0, \mathcal{S}_1^0, \mathcal{S}_0^1$ and $\mathcal{S}_1^1$ be the four servers involved in the protocol. Let $X = (x_1, \ldots, x_N)$ be the data consisting of $N$ blocks, each of size $B = \Omega(\Lambda(\log N, 1))$ bits. In initialization, the client generates two additive shares of the data, $X^0 = (x_1^0, \ldots, x_N^0)$ and $X^1 = (x_1^1, \ldots, x_N^1)$. That is, $X^0$ and $X^1$ are two random vectors of $N$ blocks, satisfying $X^0 \oplus X^1 = X$. For $b \in \{0,1\}$, the client sends $X^b$ to both $\mathcal{S}_0^b$ and $\mathcal{S}_1^b$. Throughout the ORAM simulation, we maintain the following invariant: for $b \in \{0,1\}$, $\mathcal{S}_0^b$ and $\mathcal{S}_1^b$ have an identical array $X^b$, such that $X^0$ and $X^1$ are random additive shares of $X$.

*Query Protocol.* To obliviously simulate a read/write query to the $i^{th}$ block in the data, the client first reads the value $x_i$ via two PIR queries: a two-server PIR with $\mathcal{S}_0^0$ and $\mathcal{S}_1^0$ to retrieve $x_i^0$, and a two-server PIR with $\mathcal{S}_0^1$ and $\mathcal{S}_1^1$ to retrieve $x_i^1$. The client then computes $x_i$ using the two shares. Second, to write a new value $\hat{x}_i$ to the data (which can possibly be equal to $x_i$), the client performs two *identical* invocations of two-server PIR-write, each with servers $\mathcal{S}_0^b$ and $\mathcal{S}_1^b$ for $b \in \{0,1\}$. It is important that $\mathcal{S}_0^b, \mathcal{S}_1^b$ (for $b \in \{0,1\}$) receive an identical PIR-write query, since otherwise, they will no longer have two identical replicates.

### 3.5 Analysis

The security of the scheme follows directly from the security of the underlying DPF protocol from [BGI15]. It remains to analyze the bandwidth cost. To simulate a query, the client sends each of the servers two DPF shares: one for reading

of length $\Lambda(\log N, 1)$ bits, and another for writing of length $\Lambda(\log N, B)$. With a block size of $B = \Omega(\Lambda(\log N, 1))$ this translates to $O(\Lambda(\log N, B)/B)$ bandwidth overhead. Each of the servers, in return, answers by sending two blocks.

# 4 The Balanced Hierarchical ORAM Framework

In this section, we lay the groundwork for our constructions in the standard distributed ORAM model, that are presented later in Sections 5 and 6.

## 4.1 Main Building Block: Hashing

Hashing, or more accurately, oblivious hashing, has been a main building block of hierarchical ORAM schemes since their first appearance in [Ost90]. Various types of hashing schemes, each with different parameters and properties, were plugged in ORAM constructions in an attempt to achieve efficient protocols (e.g. [GO96, GM11, CGLS17]). Hashing stands at the heart of our constructions as well. However, since we make a generic black-box use of hashing, we do not limit ourselves to a specific scheme, but rather take a modular approach.

We consider an $(n, m, s)$-*hashing scheme*[3], $H$, to be defined by three procedures: Gen for key generation, Build for constructing a hash table $T$ of size $m$ that contains $n$ given data elements, using the generated key, and Lookup for querying $T$ for a target value. The scheme may also use a stash to store at most $s$ elements that could not be inserted into $T$. In a context where a collection of hashing schemes operate simultaneously (e.g. ORAMs), a *shared stash* may be used by all hash tables. We denote by $C_{\mathsf{Build}}(H)$ and $C_{\mathsf{Lookup}}(H)$, the build-up complexity and the query complexity of $H$ (resp.) in terms of communication (in the client-server setting).

An *oblivious* hashing scheme is a scheme whose Build and Lookup procedures are oblivious of the stored data and the queried elements (respectively). In Appendix C, we provide formal definitions and notation for the above, and survey a few of the schemes that were used in prior ORAM works.

## 4.2 Our Starting Point: The Single-server Scheme from [KLO12]

**Overview.** The starting point of our distributed ORAM constructions in Sections 5 and 6 is the single-server scheme from [KLO12]. In standard hierarchical ORAMs, the server stores the data in $\log N$ levels, where every level is a hash table, larger by a factor of 2 than the preceding level. Kushilevitz et al. changed this by having $L = \log_d N$ levels, where the size of the $i^{th}$ level is proportional to $(d-1) \cdot d^{i-1}$. Having less levels eventually leads to the efficiency in overhead, however, since level $i + 1$ is larger by a factor of $d$ (no longer constant) than level $i$, merging level $i$ with level $i + 1$ becomes costly (shuffling an array of size $(d-1) \cdot d^i$ every $(d-1) \cdot d^{i-1}$ queries). To solve this problem, every level is stored in $d - 1$ separate hash tables of equal size in a way that allows us to reshuffle every level into a single hash table in the subsequent level.

---

[3] Implicitly stated parameters may be omitted for brevity.

**Theorem 2 ([KLO12, CGLS17]).** *Let $d$ be a parameter, and define $L = \log_d N$. Assume the existence of one-way functions, and a collection $\{H_i\}_{i=1}^L$, where $H_i$ is an oblivious $(d^{i-1}k, \cdot, \cdot)$-hashing scheme, with a shared stash of size $s$. Then there exists a single-server ORAM scheme that achieves the following overhead for block size $B = \Omega(\log N)$.*

$$O\left(k + s + \sum_{i=1}^L d \cdot C_{\mathsf{Lookup}}(H_i) + \sum_{i=1}^L \frac{C_{\mathsf{Build}}(H_i)}{d^{i-1}k}\right)$$

A special variant of the theorem was proven by Kushilevitz et al. [KLO12]. In their work, they use a well-specified collection of hashing schemes (consisting of both standard and cuckoo hashing [PR04]), and obtain an overhead of $O(\log^2 N / \log \log N)$. The modular approach to hierarchical ORAM was taken by Chan et al. [CGLS17], in light of their observations regarding the conceptual complexity of cuckoo hashing, and their construction of a simpler oblivious hashing scheme that achieves a similar result. Our results in the distributed setting fit perfectly in this generic framework, as they are independent of the underlying hashing schemes. Below, we elaborate the details of the construction from [KLO12], as a preparation towards the following sections.

*Data Structure.* The top level, indexed $i = 0$, is stored as a plain array of size $k$. As for the rest of the hierarchy, the $i^{th}$ level ($i = 1 \ldots L$) is stored in $d - 1$ hash tables, generated by an oblivious $(d^{i-1}k, \cdot, \cdot)$-hashing scheme $H_i$. For every $i = 1, \ldots, L$ and $j = 1, \ldots, d - 1$, let $T_i^j$ be the $j^{th}$ table in the $i^{th}$ level, and let $\kappa_i^j$ be its corresponding key. All hashing schemes in the hierarchy share a stash $S^4$. The keys $\kappa_i^j$ can be encrypted and stored remotely in the server. Also, the client stores and maintains a counter $t$ that starts at zero, and increments by one after every virtual access is simulated. The ORAM simulation starts with the initial data stored entirely in the lowest level.

*Blocks Positioning Invariant.* Throughout the ORAM simulation, every data block in the virtual memory resides either in the top level, or in one of the hash tables in the hierarchy, or in the shared stash. The blocks are hashed according to their virtual addresses. The data structure does not contain duplicated records.

*Blocks Flow and Reshuffles.* Once a block is queried, it is inserted into the top level, therefore the level fills up after $k$ queries. Reshuffles are used to push blocks down the hierarchy and prevent overflows in the data structure. Basically, every time we try to insert blocks to a full level, we clear the level by reshuffling its blocks to a lower level. For instance, the top level is reshuffled every $k$ queries.

In every reshuffle, blocks are inserted into the first empty hash table in the highest level possible, using the corresponding Build procedure, with a freshly

---

[4] In the scheme of [KLO12], the shared stash is 'virtualized', and is re-inserted into the hierarchy. We intentionally roll-back this optimization in preparation to our distributed constructions.

generated key. Thus, the first time the top level is reshuffled (after round $k$), its blocks are inserted to the first table in the next level, i.e. $T_1^1$, which becomes full. The top level fills up again after $k$ queries. This time, the reshuffle is made to $T_1^2$, as $T_1^1$ is not empty anymore. After $d-1$ such reshuffles, the entire first level becomes full, therefore, after $d \cdot k$ queries, we need to reshuffle both the top level and the first level. This time, we insert all blocks in these levels into $T_2^1$.

Observe that this mechanism is analogous to the process of counting in base $d$: every level represents a digit, whose value is the number of full hash tables in the level. An increment of a digit with value $d-1$, equivalently - insertion to a full level, is done by resetting the digit to zero, and incrementing the next digit by 1, that is, reshuffling the level to a hash table in the next level. A demonstration of this analogy is given in Figure 2. Using this observation, we formalize the process as follows: in every round $t = t' \cdot k$, levels $0, \dots, i$ are reshuffled down to hash table $T_{i+1}^j$, where $i$ is the maximal integer for which $d^i \mid t'$, and $j = (t' \bmod d^{i+1})/d^i$. Notice that level $i$ is reshuffled every $k \cdot d^i$ queries.

$d = 6$
$t = 360 \cdot k = 1400_6 \cdot k$



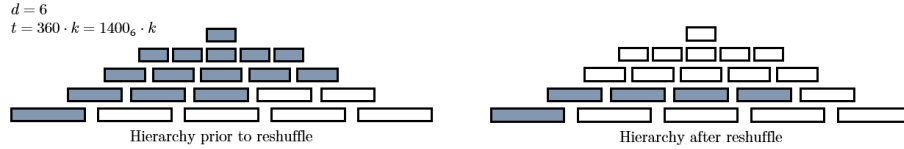| Hierarchy prior to reshuffle | Hierarchy after reshuffle |

Fig. 2: a demonstration of the flow of blocks during an ORAM simulation with $d = 6$. A gray cell indicates a full hash table, a white one is an empty table.

*Query.* In order to retrieve a data block with virtual address $v$, the client searches for the block in the top level and the stash first. Then, for every level $i$, the client scans hash tables $T_i^j$ using $H_i$.Lookup procedure, *in reverse order*, starting with the table that was last reshuffled into. Once the target block was found, the scan continues with dummy queries. This is important for security (see Claim 4). Appendix **??** contains a detailed pseudo-code for the query algorithm.


## 5 A Three-server ORAM Scheme

Below, we formally state our first result in the standard distributed ORAM model: an efficient three-server ORAM scheme.

**Theorem 3 (Three-server ORAM using regular hashing).** *Let $d$ be a parameter, and define $L = \log_d N$. Assume the existence of one-way functions, and a collection $\{H_i\}_{i=1}^L$, where $H_i$ is a $(d^{i-1}(k+s), m_i, s)$-hashing scheme. Then, there exists a $(3,1)$-ORAM scheme that achieves an overhead of*

$$O\left(k + L + \sum_{i=1}^{L} \frac{m_i}{d^{i-1}k}\right)$$

*for block size $B = \Omega(\alpha d \log N + s \log d)$, where $\alpha := \max_i C_{\mathsf{Lookup}}(H_i)$.*

16

We propose two different instantiations of our construction, each with a different collection of hashing schemes that was used in prior ORAM works [GM11, KLO12, CGLS17]. Both instantiations yield sub-logarithmic overhead, and their parameters are very close. However, Instantiation 3 may be conceptually simpler (due to [CGLS17]). More details about the used hashing schemes can be found in Appendix C.

First, we plug in the collection of hashing schemes used by Goodrich and Mitzenmacher [GM11], and later by Kushilevitz et al. [KLO12]. The collection mainly consists of cuckoo hashing schemes, however, since stashed cuckoo hashing was shown to have a negligible failure probability only when the size of the hash table is polylogarithmic in $N$ (specifically, $\Omega(\log^7 N)$) [GM11], standard hashing with bucket size $\log N / \log \log N$ is used in the first $\Theta(\log_d \log N)$ levels. We point out that in both mentioned works [GM11, KLO12], the stash size for cuckoo hashing is logarithmic. In our instantiation, we use a stash of size $\Theta(\omega(1) \cdot \log N / \log \log N)$. Although [GM11] proved that failure probability is negligible in $N$ when the stash is of size $s = \Theta(\log N)$ and the size of the table is $m = \Omega(\log^7 N)$ (by extending the proof for constant stash size from [KMW09]), their proof works whenever the value $m^{-\Theta(s)}$ is negligible in $N$, and in particular, when we choose $s = \Theta(\omega(1) \cdot \log N / \log \log N)$.

**Instantiation 2 (Three-server ORAM using cuckoo hashing)** *Assume the existence of one-way functions. Let $d$ be a parameter at most polylogarithmic in $N$. Then, there exists a three-server ORAM scheme that achieves overhead of $O(\log_d N \cdot \omega(1))$ for $B = \Omega(d \log N)$.*

When $d = \log^\epsilon N$ for a constant $\epsilon \in (0, 1)$, we achieve an overhead of $O(\omega(1) \cdot \log N / \log \log N)$ with $B = O(\log^{1+\epsilon} N)$.

Alternatively, we can use the simple two-tier hashing scheme from [ACMR95], with buckets of size $\log^{0.5+\epsilon} N$, to achieve the following parameters.

**Instantiation 3 (Three-server ORAM using two-tier hashing)** *Assume the existence of one-way functions. Let $d$ be a parameter at most polylogarithmic in $N$. Then, there exists a three-server ORAM scheme that achieves overhead of $O(\log_d N)$ for block size $B = \Omega(d \log^{1.5+\epsilon} N)$.*

For $d = \log^\epsilon N$, we obtain an overhead of $O(\log N / \log \log N)$ with $B = O(\log^{1.5+2\epsilon} N)$.

### 5.1 Overview

Our three-server scheme is based on the single-server balanced hierarchical structure of Kushilevitz et al. [KLO12] (described in Section 4). We take advantage of the existence of multiple servers and reduce the overhead as follows.

*Reduce query cost using PIR.* One of the consequences of balancing the hierarchy is having multiple hash tables in a level, in any of which a target block can reside. More specifically, if $T_i^1, \ldots, T_i^{d-1}$ are the hash tables at level $i$, then a block with

address $v$ can possibly reside in any of the positions in $T_i^j[H_i.\mathsf{Lookup}(v, \kappa_i^j)]$ for $j = 1, \cdots, d - 1$. To retrieve such a block, we could basically download all blocks in these positions, i.e. $\sum_{i=1}^{L}(d - 1)C_{\mathsf{Lookup}}(H_i)$ blocks in total. This already exceeds the promised overhead. Instead, we use PIR to extract the block efficiently without compromising the security of the scheme. For every level $i$, starting from the top, we invoke a PIR protocol over the array that consists of the $(d - 1)C_{\mathsf{Lookup}}(H_i)$ possible positions for $v$ in the level.

Performing PIR queries requires that the client knows the exact position of the target block in the queried array, namely, in which bucket, out of the $d - 1$ possibilities, block $v$ resides, if at all. Therefore, the client first downloads the addresses of all blocks in the array, and only then performs the PIR query. Although some PIR protocols in the literature (e.g. [BGI15]) do not impose this requirement, we still need to download the addresses since it is essential for the security of the protocol that the client re-writes the address of the queried block.

An address of a block can be represented using $\log N$ bits. Thus, downloading the addresses of all possible positions in all levels costs us $\sum_{i=1}^{L}(d - 1)C_{\mathsf{Lookup}}(H_i)\log N$ bits of communication. If we choose $B = \Omega(\alpha d \log N)$ for $\alpha = \max_i C_{\mathsf{Lookup}}(H_i)$, this cost translates to the desirable $O(L)$ overhead. Two-server PIRs work in the model where the data is replicated and stored in two non-colluding servers. Thus, every level in the hierarchy, except the top level, will be stored, accessed, and modified simultaneously in two of the three servers.

*Reduce reshuffles cost by bypassing oblivious hashing.* We use a variant of the reshuffle procedure suggested by Lu and Ostrovsky [LO13]. Their protocol works in a model with two non-colluding servers, where one server stores the odd levels, and the other stores the even levels. Before reshuffling a level, the servers gather all blocks to be reshuffled, permute them randomly, and exchange them through the client, who re-encrypts them and tags them with pseudorandom tags. The level is then reshuffled by one server using some *regular* hashing scheme (not necessarily oblivious), and is sent to the other server, record by record, through the client. The security of their scheme follows from the following observations:
 (i) the blocks are re-encrypted and permuted randomly before the reshuffle, eliminating any dependency on prior events,
 (ii) the blocks are hashed according to pseudorandom tags, hence their order is (computationally) independent of their identities,
(iii) the server that holds a level cannot distinguish between dummy queries and real ones since he was not involved in the reshuffle, and
(iv) the server that reshuffles the level (and can tell a dummy query) does not see the accesses to the level at all.
Applying this method naively when each of the servers holds the entire hierarchy might reveal information about the access pattern since (iii) and (iv) no longer hold. Therefore, we should adapt their method wisely, while having two replicates of every level, to allow performing PIR queries. A straight forward implementation would require four servers: two holding replicates of the odd levels, and two holding replicates of the even levels. However, this can be done

using three servers only by having every pair of servers (out of the three possible pairs) hold every third level.

## 5.2   Full Construction

**Data Structure.** The data is virtually viewed as an array of $N$ blocks, each of size $\Omega(\alpha d \log N)$ bits. Every block therefore has a virtual address in $[N]$.

*Distributed Server Storage.* The data structure is identical to that from [KLO12], however, our scheme uses three servers, $\mathcal{S}_0, \mathcal{S}_1$, and $\mathcal{S}_2$, to store the data. The top level is stored in all servers. Every other level is held by two servers only: for $j = 0, \ldots, \lfloor \frac{L}{3} \rfloor$, $\mathcal{S}_0$ and $\mathcal{S}_1$ share replicates of levels $i = 3j$, $\mathcal{S}_1$ and $\mathcal{S}_2$ share replicates of levels $3j + 1$, and $\mathcal{S}_2$ and $\mathcal{S}_0$ both hold all levels $i = 3j + 2$.

*Dummy Blocks.* Dummy blocks are blocks that are not "real" (not part of the virtual memory), but are treated as such, and assigned dummy virtual addresses. From the point of view of the 'reshuffler' server, a dummy block, unlike an empty block, cannot be distinguished from a real block. We use two types of dummy blocks, both essential for the security of the scheme.
(i) *Dummy Hash Blocks.* Dummy hash blocks replace real blocks once they are read and written to the top level. The security of our scheme relies on the fact that all blocks in the hierarchy are of distinct addresses, hence the importance of this replacement.
(ii) *Dummy Stash Blocks.* Dummy stash blocks are created by the client to fill in empty entries in the hierarchy. Since our scheme uses a stash to handle overflows, the number of blocks in the stash and in each of the hash tables is not deterministic and is dependent on the access pattern. To hide this information from the server that performs the reshuffling of a level, we fill all empty entries in the stash, and some of the empty entries in the hash tables, with dummy stash blocks.

*Block Headers.* To properly manage the data, the client needs to know the identity of every block it downloads (i.e. its virtual address). Therefore, every entry in the server storage contains, besides the data of the block, a header that consists of the virtual address of the block, which can be either an address in $[N]$, a numbered dummy address, such as '$dummyHash \circ t$' or '$dummyStash \circ r$', or just '$empty$'. The length of the header is $O(\log N)$ bits, thus does not affect the asymptotic block size. Unless explicitly stated otherwise, the headers are downloaded, uploaded and re-encrypted together with the data. An entry with a block of virtual address $v$ and data $x$ is denoted by the tuple $(v, x)$.

*Tags.* Since we use the servers for reshuffling the levels, we wish to hide the virtual addresses of the blocks to be reshuffled. We use pseudorandom tags to replace these addresses, as first suggested in [LO13]. The tags are computed using a keyed PRF, $F_s$, that is known to the client only. When generating a new hash table, the server hashes the blocks according to their tags (rather than their

virtual addresses). Furthermore, to eliminate any dependency between tags that are seen in different reshuffles, the client keeps an *epoch* $e_i^j$ for every hash table $T_i^j$ in the hierarchy. The epoch of a table is updated prior to every reshuffle, and is used, together with $i$ and $j$, to compute fresh tags for blocks in the table. The epochs can be stored remotely in the servers to avoid large client storage.

**Protocol.** Again, we refer to the balanced hierarchy of [KLO12] (see Algorithm 4) as our starting point.

*Query.* We replace the reads performed by the client with PIR protocols that are executed over arrays in the data. Specifically, the first PIR is performed over the stash to retrieve the target block if it is found there. The top level can be downloaded entirely since it has to be re-written anyway. The search continues to the other levels in the hierarchy in the order specified in Section 4. The target block can possibly reside in any of the $d - 1$ hash tables in a level, therefore, the client invokes a PIR protocol to extract the target block out of the many possible positions. Every PIR in the procedure is preceded by downloading the headers in the queried array, using which the client knows the position of the target block. A technical detailed description is provided in Algorithm 1.

*Reshuffles.* Let $\mathcal{S}_a$ and $\mathcal{S}_b$ be the two servers holding level $i + 1$, and let $\mathcal{S}_c$ be the other server. Reshuffling levels $0, \ldots, i$ into hash table $T_{i+1}^j$ is performed as follows. As a first step, we send all non-empty blocks that should be reshuffled (including stash) to $\mathcal{S}_c$, by having the servers exchange the blocks they hold in levels $0, \ldots, i$ and the stash, through the client, one block at a time, in a random order. Besides forwarding the blocks to $\mathcal{S}_c$, the client also re-encrypts every block and re-tags it with a fresh tag (using epochs, as already mentioned). Once $\mathcal{S}_c$ has all tagged blocks, he can create a new hash table and stash using the appropriate Build procedure. He then sends the hash table and stash, one record at a time, to the client. The client re-encrypts all records, and forwards them to the other two servers, who store the hash table in $T_{i+1}^j$, and the stash to its place. The client uses dummy stash blocks to replace as many empty blocks as needed to get a full hash table, and a full stash. This is important since we do not want to reveal the load of the stash to the server that does the next reshuffle. The reshuffle procedure is described in full details in Algorithm 2.

### 5.3   Analysis

**Complexity.** We begin with analyzing the complexity of the described scheme.

*Server storage.* The combined server storage contains a stash of size $s$, a top level of size $k$, and two duplicates of every other level $i$, consisting of $d - 1$ hash tables of size $m_i$ each. In total, we have $O\left(s + k + \sum_{i=1}^{L} dm_i\right)$.

---
**Algorithm 1** Three-Server Construction: Query
---
1: Allocate a local register of the size of a single record.
2: Initialize a flag found $\leftarrow 0$.
3: Download the top level, one record at a time. If $v$ is found at some entry $(v, x)$ then store $x$ in the local register, and mark $found \leftarrow 1$.
4: Download all headers from $S$. If $v$ was found among these headers, let $p$ be its position, and mark found $\leftarrow 1$. Otherwise, let $p$ be a position of a random entry in the stash. Invoke $\text{PIR}(S, p)$ to fetch $(v, x)$ with any two of the three servers, and store $x$ in the register.
5: **for** every level $i = 1 \dots L$ **do**
6:      $t' \leftarrow \lfloor t/k \rfloor$
7:      $r \leftarrow \lfloor (t' \bmod d^i)/d^{i-1} \rfloor$
8:      $headers \leftarrow \emptyset$
9:      **for** every hash table $j = r \dots 1$ **do**
10:          If found $= false$, compute the corresponding tag of $v$, $\tau \leftarrow F_s(i, j, e_i^j, v)$. Otherwise, assign $\tau \leftarrow F_s(i, j, e_i^j, dummy \circ t)$.
11:          $Q_i^j \leftarrow H_i.\text{Lookup}(\tau, \kappa_i^j)$
12:          Download all headers of entries in $T_i^j[Q_i^j]$, and append them to $headers$. If one of the headers says $v$, mark found $\leftarrow true$.
13:      **end for**
14:      Let $p$ be the position of $v$ in $headers$ if it was found there, or a random value in $\{1, \dots, |headers|\}$ otherwise.
15:      Let $A$ be the array of entries corresponding to headers in $headers$.
16:      Invoke $\text{PIR}(A, p)$ to fetch $(v, x)$ with the two servers holding level $i$, and store $x$ in the register (if $v$ was not found in $headers$ this would be a dummy PIR).
17:      Re-encrypt $headers$, and upload it back to the two servers, while changing $v$ to $dummyHash \circ t$.
18: **end for**
19: If the query is a write query, overwrite $x$ in the register.
20: Read each entry of the entire top level from both servers one at a time, re-encrypt it, then write it back, with the following exception: if the entry $(v, x)$ was first found at the top level, then overwrite $x$ with the (possibly) new value from the register, otherwise, write $(v, x)$ in the first empty spot of the form $(empty, \cdot)$.
21: Increment the counter $t$, and reshuffle the appropriate levels.
---

*Client storage.* The client uses constant working memory as it only receives and forwards records, one record at a time. Notice that the client does not need to keep all the headers he downloads prior to executing PIR queries, and it would be sufficient to keep only the position of the target block among them.

*Overhead.* We now analyze the cost of performing a single query. First, consider the communication cost of downloading the headers for the PIRs. The PIRs are performed over the stash and each of the levels $i = 1, \dots, L$. The number of headers downloaded amounts to $s + \sum_{i=1}^{L} (d-1)C_{\text{Lookup}}(H_i) \leq s + \alpha L(d-1)$, which is equivalent to $O(L)$ blocks of the required minimum size. Overall, $L + 1$ PIR queries are invoked. For levels $i = 1, \dots, L$, the PIR queries are performed over arrays of size at most $(d-1)C_{\text{Lookup}}(H_i)$. By using even the classic two-server PIR from [CGKS98], this costs $(d-1)C_{\text{Lookup}}(H_i) < \alpha d$ bits and a single

---

**Algorithm 2** Three-Server Construction: Reshuffle

---

**Reshuffling into table $T_{i+1}^j$**

Let $\mathcal{S}_a$ and $\mathcal{S}_b$ be the servers holding level $i+1$, and let $\mathcal{S}_c$ be the other server.

1: Every server of the three allocates a temporary array. For every level $\ell$ between levels 1 and $i$, let $\mathcal{S}^\ell$ be the server with the smallest id that holds level $\ell$. For every such $\ell$, $\mathcal{S}^\ell$ inserts all records in level $\ell$ to its temporary array. In addition, one of the servers, say $\mathcal{S}_0$, inserts all stash records into its temporary array.

2: $\mathcal{S}_c$ applies a random permutation on its temporary array, and sends the records one by one to the client. The client re-encrypts each record and sends it to $\mathcal{S}_b$. $\mathcal{S}_b$ inserts all records it receives to its array. $\mathcal{S}_b$ permutes its array randomly, and forwards it to $\mathcal{S}_a$ through the client (who re-encrypts them). $\mathcal{S}_a$, in his turn, also inserts all received records, applies a random permutation, and sends them one by one to the client.

3: The client re-encrypts every non-empty record $(v, x)$ and sends it to $\mathcal{S}_c$, together with a tag, which is the output of the PRF $F_s(i+1, j, e_{i+1}^j, v)$, where $e_{i+1}$ is the new epoch of $T_{i+1}^j$. Note that $v$ may be a virtual memory address, or a dummy value. In this step, dummy records are treated as real records and only empty records are discarded.

4: $\mathcal{S}_c$ receives $d^i(k+s)$ tagged records, which are all records that should be reshuffled into $T_{i+1}^j$. It generates a new key $\kappa_i^j \leftarrow H_i.\mathsf{Gen}(N)$, and constructs a hash table and a stash $(T_i^j, S) \leftarrow H_i.\mathsf{Build}(\kappa_i^j, Y)$, where $Y$ is the set of tagged records received from the client. If the insertion fails, a new key is generated (this happens with a negligible probability). $\mathcal{S}_c$ then informs the client about the number of elements inside the stash, $\sigma$, and the key $\kappa_i^j$, then sends both the hash table $T_i^j$ and the stash one record at a time to the client.

5: As the client receives entries from $\mathcal{S}_c$ one at a time, it re-encrypts each record and sends it to both $\mathcal{S}_a$ and $\mathcal{S}_b$ without modifying the contents except:

   (a) The first $\sigma$ empty records in the table the client receives from $\mathcal{S}_c$ are encrypted as $(dummyStash \circ r, \cdot)$, incrementing $r$ each time.

   (b) Subsequent empty records from the table are encrypted as $(empty, \cdot)$.

   (c) Every empty record in the stash is re-encrypted as $(dummyStash \circ r, \cdot)$, incrementing $r$ each time.

6: $\mathcal{S}_a$ and $\mathcal{S}_b$ store the table records in level $i+1$ in the order in which they were received, and store the stash records at the top level.

---

block per level. The stash adds $s$ bits and a block. All of this sums up to no more than $O(L)$ data blocks. As for the actual blocks the client downloads, these are the blocks of the top level, $O(k)$ in number.

It remains to add the overhead caused by the reshuffles made between queries. Blocks are reshuffled down to some hash table in the $i^{th}$ level if $i$ is the smallest integer for which $(t/k) \bmod d^i \neq 0$. This occurs whenever $t/k$ is a multiple of $d^{i-1}$, but not of $d^i$, i.e., at most once every $k \cdot d^{i-1}$ queries. One can clearly see that during the reshuffle of a hash table $T_i^j$, the number of blocks transmitted is asymptotically bounded by the size of $T_i^j$ and the size of the stash, which is $m_i + s = O(m_i)$. Hence, the amortized overhead of the reshuffles is $O(\sum_{i=1}^{L} \frac{m_i}{d^{i-1}k})$.

**Security.** Next, we present the security proof for our construction. We prove that the access pattern to any of the servers in the scheme is oblivious and independent on the input. We describe a simulator $\mathsf{Sim}_a$ (for $a \in \{0, 1, 2\}$), that produces an output that is computationally indistinguishable from the view of server $\mathcal{S}_a$ during the execution of the protocol, upon any sequence of virtual queries $v_1, \ldots, v_\ell$, given only its length $\ell$.

**Lemma 1 (Security of the three-server ORAM).** *Let* $\mathsf{View}_a(\boldsymbol{y})$ *be the view of server* $\mathcal{S}_a$ *during the execution of the three-server ORAM protocol, described in Algorithms 1 and 2, over a virtual access pattern* $\boldsymbol{y} = ((v_1, x_1), \ldots, (v_\ell, x_\ell))$. *There exist simulators* $\mathsf{Sim}_0, \mathsf{Sim}_1, \mathsf{Sim}_2$, *such that for every* $\boldsymbol{y}$ *of length* $\ell$, *and every* $a \in \{0, 1, 2\}$, *the distributions* $\mathsf{Sim}_a(\ell)$ *and* $\mathsf{View}_a(\boldsymbol{y})$ *are computationally indistinguishable.*

*Proof Sketch.* As in all previous works, we assume that the client uses one-way functions to encrypt and authenticate the data held in the servers, and therefore, encrypted data is indistinguishable by content (notice that the client re-encrypts every piece of data before sending it). We replace the keyed tagging functions, that are modeled as PRFs in the scheme, with random functions. These preliminary steps can be formalized using proper standard hybrid arguments, which we avoid for brevity.

We begin by inspecting the view of the servers during the reshuffle procedure. The procedure starts with the servers exchanging all blocks stored in levels $1, \ldots, i$ and in the stash, and sending them to $\mathcal{S}_c$. It is essential for security that the number of these blocks is independent of the input, as we show below.

**Claim 1** *Throughout the ORAM simulation, the stash is always full (contains $s$ records).*

*Proof.* In the reshuffle procedure (Algorithm 2, Step 5), every empty slot in the stash is filled with a dummy stash block, thus the claim holds. ☐

**Claim 2** *Let $t$ be a multiple of $k$, and denote $t' = t/k$. For every $1 \leq i \leq L$, define $r_i^t := \lfloor (t' \bmod d^i)/d^{i-1} \rfloor$. Then,*

*(i) the top level is full prior to the reshuffle at round $t$, and is empty afterwards.*
*(ii) for every other level $1 \leq i \leq L$, once the reshuffle is completed, the first $r_i^t$ tables in level $i$ (i.e., $T_i^1, \ldots, T_i^{r_i^t}$) are full (contain $d^i(k + s)$ records each), and all other tables in level $i$ are empty.*

Recall the analogy of the reshuffles to counting in base $d$ (see Section 4). Notice that $r_i^t$ can be also defined as the $i^{th}$ digit in the base $d$ representation of $t'$. Claim 2 follows from these two observations. Due to space limitation, a complete proof for this claim is provided in Appendix E.

Having shown that the amount of data exchanged during the first steps of the reshuffling procedure depends only on $t$, we can simulate the view of any of the

23

servers by a sequence of arbitrary encrypted data of the appropriate length. Next, $\mathcal{S}_c$ receives $(k+s) \cdot d^i$ tagged encrypted records (Claim 2). In Claim 3 below, we show that these records have unique addresses, therefore their tags will also be unique (with overwhelming probability). Furthermore, these tags are computed using a random function that has not been used before (fresh epoch). Hence, the view of $\mathcal{S}_c$ can be simulated as a sequence of $(k+s) \cdot d^i$ arbitrary encrypted records with random distinct tags. Once $\mathcal{S}_c$ successfully creates the hash table, it sends it to $\mathcal{S}_a$ and $\mathcal{S}_b$ via the client. The size of the hash table is fixed. Again, since the entries of the hash tables are encrypted, they are indistinguishable from any arbitrary sequence of encrypted records, and can be simulated as such.

**Claim 3** *At all times during the execution, any non-empty record of the form $(v, \cdot)$ will appear at most once in all hash tables in the hierarchy.*

*Proof.* A label $v$ can be either a virtual address or a dummy label of some type. A real virtual address can be added to the top level of the hierarchy once it is queried, in such a case, it is removed from its prior location and replaced by a dummy hash block. Dummy blocks, of any type, are labeled with a counter that is incremented after the creation of each block, thus cannot reoccur.  □

**Corollary 1.** *The tagging function $F_s(\cdot)$ will not be computed twice on the same input throughout the executions of Algorithm 2 during the ORAM simulation.*

*Proof.* Otherwise, the same $v$ would have appeared twice during the pass in Step 3 of Algorithm 2, in contradiction to Claim 3.  □

To summarize, to simulate the view of the servers during the reshuffling phase, $\mathsf{Sim}_a(\ell)$ and $\mathsf{Sim}_b(\ell)$ output a sequence of encrypted arbitrary records of the appropriate length (which is fixed due to Claims 1 and 2), whereas $\mathsf{Sim}_c(\ell)$ outputs a sequence of encrypted arbitrary records that are tagged using distinct uniform values ($a, b, c$ alternate between 0,1,2 throughout the phases). From Corollary 1 and the security of the underlying symmetric encryption and PRFs, these outputs are indistinguishable from the views of the servers at the reshuffles.

We proceed to simulating the access pattern during queries. A query for a block $v$ begins, independently of $v$, with downloading all blocks in the top level, and all headers in the stash. Next, a PIR is invoked over the stash. From the assumed security of the underlying PIR scheme, there exist two simulators $\mathsf{Sim}_0^{\mathrm{PIR}}(m), \mathsf{Sim}_1^{\mathrm{PIR}}(m)$, that simulate the individual views of the two servers (resp.) involved in the protocol, given only the size of the queried array, $m$. We use these simulators to simulate the view of the servers involved in the PIR over the stash, and in the other PIR invocations to follow.

It remains to show that the identity of the blocks over which the PIRs are called, i.e. the values $Q_i^j$ that a server $\mathcal{S}_a$ sees during the execution of Algorithm 1, can be simulated as well. Recall that, at every execution of the algorithm, $Q_i^j$ is computed, for every $i, j$, as $H_i.\mathsf{Lookup}(\tau, \kappa_i^j)$, where $\tau$ is a tag computed using $F_s$, and $\kappa_i^j$ is the used hash key. We denote by $\langle Q_i^j \rangle_a$ the sequence of $Q_i^j$ values seen by $\mathcal{S}_a$ at *all* executions of Algorithm 1 during the

ORAM simulation (these values correspond to levels $i$ that are stored in $\mathcal{S}_a$). We also denote by $\langle \tau \rangle_a$ and $\langle \kappa_i^j \rangle_a$ the values used to compute $\langle Q_i^j \rangle_a$.

**Claim 4** *The same $v$ will not be queried upon twice at the same hash table (in Algorithm 1) between two reshuffles of the table during the ORAM execution.*

*Proof.* For a dummy address $v = $ '$dummy \circ t$', the lemma is trivial, since it can be queried only at access $t$. If a virtual address $v$ is queried at hash table $T_i^j$, it will be written to the top level. If $v$ reaches level $i$ before $T_i^j$ is shuffled down to level $i + 1$, then it would be placed in a hash table $T_i^{j'}$ for $j' > j$. Therefore, subsequent queries to $v$ will find $v$ either at some level above level $i$, or at hash table $T_i^{j'}$. In both cases, the client will perform a dummy query at hash table $T_i^j$ (hence the importance of the reverse order scan over the hash tables). $\qquad\square$

**Corollary 2.** *The tagging function $F_s$ will not be computed twice on the same input throughout the executions of Algorithm 1 during the ORAM simulation.*

*Proof.* Consider two different inputs $(i, j, e_i^j, v), (i', j', e_i^{j'}, v')$. If $i = i', j = j'$ and $e_i^j = e_i^{j'}$, then, due to Lemma 4, $v \neq v'$. $\qquad\square$

**Claim 5** *The sequence $\langle \tau \rangle_a$, defined above, is computationally indistinguishable from a uniform sequence of unique tags, given the view of $\mathcal{S}_a$ during the reshuffles.*

*Proof.* The claim follows from Corollary 2, and from the fact that the tags $\langle \tau \rangle_a$ correspond to levels that are not hashed by $\mathcal{S}_a$. The view of $\mathcal{S}_a$ during the reshuffles of such levels consist of encrypted data only, and is computationally independent of $\langle \tau \rangle_a$ (from the security of the underlying encryption scheme). The view of $\mathcal{S}_a$ during the reshuffles of any other level are also independent of $\langle \tau \rangle_a$ since they consist of encrypted data, and random tags that are computed for different values of $i$. $\qquad\square$

**Claim 6** *The sequence $\langle \kappa_i^j \rangle_a$, defined above, is computationally indistinguishable from a uniform sequence of hash keys, given the view of $\mathcal{S}_a$ during the reshuffles.*

*Proof.* The values in $\langle \kappa_i^j \rangle_a$ are uniformly chosen by a server other than $\mathcal{S}_a$ under the constraint that keys do not cause a failure in the build up of the corresponding hash tables. From the definition of a hashing scheme (Definition 3), a failure occurs with a negligible probability over the choice of the key. The distribution of $\langle \kappa_i^j \rangle_a$ is clearly computationally independent of the view of $\mathcal{S}_a$ during the reshuffles, which consists of encrypted data, and random tags at levels in which the hash keys $\langle \kappa_i^j \rangle_a$ are not used. $\qquad\square$

To conclude, in Claims 5 and 6, we show that $\langle \tau \rangle_a$ and $\langle \kappa_i^j \rangle_a$ (resp.) are indistinguishable from sequences of uniformly chosen values, given the view obtained at the reshuffles. Therefore, to simulate the values $\langle Q_i^j \rangle_a$, the simulator $\mathsf{Sim}_a(\ell)$ computes the output of $H_i.\mathsf{Lookup}$ for uniformly random sequence of tags and hash keys. This completes the proof of Lemma 1, and hence Theorem 3.

# 6  A Family of Multi-Server ORAM Schemes

In this section, we show how to use oblivious hashing, rather than regular hashing, to obtain our sub-logarithmic $(m, m-1)$-ORAM, for any $m > 2$.

**Theorem 4 (Multi-server ORAM using oblivious hashing).** *Let $d$ be a parameter, and define $L = \log_d N$. Assume the existence of one-way functions, and a collection $\{H_i\}_{i=1}^L$, where $H_i$ is an oblivious $(d^{i-1}(k+s), m_i, s)$-hashing scheme. Then, for any $m \geq 2$, there exists an $(m, m-1)$-ORAM scheme that achieves the following overhead for block size $B = \Omega(\beta \log N + \alpha d \log N)$*

$$O\left(k + L + \sum_{i=1}^{L} \frac{m_i}{d^{i-1}k}\right)$$

*where $\alpha := \max_i C_{\mathsf{Lookup}}(H_i)$ and $\beta := \max_i \frac{C_{\mathsf{Build}}(H_i)}{d^{i-1}k}$.*

For instantiating the multi-server construction, we suggest using the oblivious variant of the hashing schemes from [GM11, KLO12, CGLS17] (see Appendix C). Here, in contrary to the construction in Section 5, two-tier hashing obtains (slightly) better results, both in overhead and in the minimal required $B$ (see Table 1).

**Instantiation 4 ($m$-server ORAM using two-tier hashing)** *Assume the existence of one-way functions. Then, for any $m \geq 2$, there exists a $(m, m-1)$-ORAM scheme that achieves overhead of $O(\log N / \log \log N)$ for block size of $B = \Omega(\log^2 N)$.*

For clarity of presentation, we first present the special case of our construction in the two-server setting. We then show how to generalize the construction to the setting where more servers are involved. In Appendix A, we show how to de-amortize the construction to obtain a worst-case ORAM scheme.

## 6.1  Two-Server ORAM: Overview

Our two-server ORAM solution is based on the three-server scheme from Section 5. We make the following modifications to reduce the number of servers.

*Back to oblivious hashing.* Now that we limit ourselves to using two servers only, each of which has to hold a replicate of the data for the PIR queries, we lose the ability to perform the reshuffles through a "third-party". Hence, we require now that the underlying hashing schemes are oblivious, and the build-up of the hash table is done using the oblivious Build procedures, where the client is the CPU, and one of the servers takes the role of the RAM.

Recall that the tags were essential for the security of the three-server scheme since the reshuffles were made by one of the servers, to which we did not want to reveal the identity of the blocks being reshuffled. Now that the reshuffling is done using oblivious hashing that hides any information about the records that are being hashed, or the hash keys used to hash them, using tags is not necessary anymore. Instead, the blocks are hashed, and accessed, by their headers.

*Optimizing the reshuffles.* Creating a hash table at level $i$ using $H_i.\mathsf{Build}$, incurs an overhead of $C_{\mathsf{Build}}(H_i)$ when naively applied. We observe that in any hashing scheme (by Definition 3), the only input relevant for the build-up of a hash table is the tags or, in our case, the headers of the blocks being reshuffled. Based on this observation, we suggest the following solution. The reshuffles are modified so that the build-up of the hash tables is given, as input, the set of headers, rather than the blocks themselves. Since the headers are smaller than the blocks by a factor of at least $\beta := \max_i \frac{C_{\mathsf{Build}}(H_i)}{d^{i-1}k}$ (see Theorem 4), the overhead incurred by the build-ups is cut by $\beta$, making it linear in $d^{i-1}k$.

*Finalizing with a matching procedure.* As the headers are hashed, we still have to move the data to the new hash table. To securely match the data to the hashed headers, we suggest a method that involves tagging the data elements, and permuting them randomly by the servers. Thus, tags are still used, however, in a totally different context.

## 6.2   Two-Server ORAM: Full Construction

*Data Structure.* We start with the scheme from Section 5. The server storage remains as is, except the data is now distributed among two servers, rather than three. All levels in the hierarchy, as well as the stash, are duplicated and stored in $\mathcal{S}_0$ and $\mathcal{S}_1$. The protocol guarantees that, at the end of every round, the data in the two servers will be identical.

*Query.* Every virtual access is simulated as described in Algorithm 1, with the exception that the target block is queried upon in the hash tables by its virtual address, rather than its tag: $H_i.\mathsf{Lookup}(v, \kappa_i^j)$ rather than $H_i.\mathsf{Lookup}(\tau, \kappa_i^j)$. Also, all reads and writes, as well as the PIR queries, are made now to $\mathcal{S}_0$ and $\mathcal{S}_1$.

*Reshuffles.* The key modification made to the scheme lays in the reshuffling procedure. The reshuffles are still performed in the same frequency. However, the roles of the servers change, as only two servers participate in the protocol. First, $\mathcal{S}_0$ prepares all headers of blocks that have to be reshuffled into the destination hash table, and, together with the client, invokes the appropriate oblivious $\mathsf{Build}$ procedure to hash the blocks into a new hash table.

Now that the headers are shuffled, it remains to match them to the data. The matching procedure begins with tagging the headers. $\mathcal{S}_0$ sends the shuffled headers, one by one to the client. The client decrypts every header, computes its tag using a new epoch, and sends the tag back to $\mathcal{S}_0$. The headers corresponding to empty slots in the hash table are tagged using numbered values, e.g. '$empty \circ 1$'. Notice that the number of empty slots in the hash table and stash, combined, is fixed and independent of the input. Next, $\mathcal{S}_1$ sends the records (headers and data) that correspond to the shuffled headers, one by one, in a random order. Among the actual records, $\mathcal{S}_1$ also sends as many (numbered) empty records as required to match the number of empty records in the newly-reshuffled hash table. The client tags every record he receives from $\mathcal{S}_1$, and forwards it $\mathcal{S}_0$ together with its

tag. $\mathcal{S}_0$ can easily match every record he receives to a header in the hash table or stash, according to the tags. Once this is complete, $\mathcal{S}_0$ sends the new hash table and stash to $\mathcal{S}_1$, through the client. Refer to Algorithm 3 for full details.

---

**Algorithm 3** Two-Server Construction: Reshuffle

---

**Reshuffling headers into table $T_{i+1}^j$**

1: $\mathcal{S}_0$ sends all records in levels $1, \ldots, i$ and the stash, one by one, to the client. The client re-encrypts every record he receives and forwards it to $\mathcal{S}_1$, while eliminating all empty records. $\mathcal{S}_1$ inserts every record he receives to a temporary array $Y$. Server $\mathcal{S}_1$ now sends every header in $Y$ back to $\mathcal{S}_0$, through the client.

2: Let $\hat{Y}$ be the array of encrypted headers received by $\mathcal{S}_0$. The client generates a fresh hashing key $\kappa_i^j \leftarrow H_i.\mathsf{Gen}(N)$, and, together with $\mathcal{S}_0$, invokes $(\hat{T}, \hat{S}) \leftarrow H_i.\mathsf{Build}(\kappa_i^j, \hat{Y})$ to obliviously hash the headers into a hash table and stash.

**Matching data to headers.**

3: $\mathcal{S}_0$ sends $(\hat{T}, \hat{S})$, record by record, to the client. The client decrypts every header $v$ he receives, and computes a tag $\tau \leftarrow F_s(i+1, j, e_{i+1}^j, v)$. If the header is empty, then $\tau \leftarrow F_s(i+1, j, e_{i+1}^j, empty \circ z)$, where $z$ is a counter that starts at 1 and and increments after every empty header. Notice that the number of empty headers, denoted by $Z$, depends only on $i$. The client sends the tag back to $\mathcal{S}_0$.

4: $\mathcal{S}_1$ inserts $Z$ empty records $(empty \circ 1, \cdot), \ldots, (empty \circ Z, \cdot)$ to $Y$. Server $\mathcal{S}_1$ permutes $Y$ randomly, and sends it, one record at a time, to the client.

5: The client re-encrypts every record $(v, x)$ it receives, and sends it to $\mathcal{S}_0$ with a tag $\tau$, that is the output of $F_s$ on $v$ with the appropriate epoch.

6: $\mathcal{S}_0$ matches every tagged record it receives to one of the tags it received in Step 3, and inserts the corresponding record to its appropriate slot (either in $\hat{T}$ or $\hat{S}$).

7: At this point, $\mathcal{S}_0$ holds the newly reshuffled hash table and stash, headers and data. The tags are discarded. $\mathcal{S}_0$ sends both the table and the stash to $\mathcal{S}_1$, via the client. Both servers replace the old stash and $T_{i+1}^j$ with the new data.

---

### 6.3 From Two Servers to $m$ Servers

We generalize the ideas behind the two-server construction from Section 6.1 to construct a family of $(m, m-1)$-ORAM schemes. The construction of the two-server scheme consists of two main parts: queries and reshuffles. In order to transform the construction to the multi-server setting, we transform each of these components while maintaining their security.

*Query using multi-server PIR.* To obliviously simulate a query to a block, the client follows the protocol used in the two-server construction (Algorithm 1). However, now that we want to achieve privacy against any colluding subset of corrupt servers, we use an $m$-server PIR protocol which guarantees such a privacy. That is, instead of invoking two-server PIRs to query blocks from the stash and hierarchy levels, the client now uses an $(m, m-1)$-PIR protocol involving all $m$ servers, where the joint view of any $m-1$ servers is (computationally) independent of the target index. In particular, we can use the straight-forward

$m$-server generalization of the basic PIR protocol from [CGKS98]. Since this protocol, as well as many known $m$-server PIRs, follow the standard PIR setting where the data is assumed to be replicated in all of the servers, the servers during the ORAM execution will hold identical replicates of the same data structure.

*Extending the matching procedure.* Reshuffles of levels are done in the same frequency, and in a very similar manner as in the two-server protocol. The only change we make is in matching procedure. To match the content to the tags, we cannot rely only on two servers, since they might be both corrupt. Instead, we let all servers participate. The reshuffling procedure from Algorithm 3 is followed up to Step 5. After the client receives the permuted records from $\mathcal{S}_1$, he re-encrypts them and forwards them to $\mathcal{S}_2$. $\mathcal{S}_2$, in its turn, randomly permutes the records it receives, and forwards them to $\mathcal{S}_3$ (if it exists), through the client. This continues until all servers, except $\mathcal{S}_0$, have received the records and permuted them on their own. Once they all had, the client tags the records and sends them to $\mathcal{S}_0$, who matches them to the shuffled headers, as described in Steps 5 and 6. Lastly, the new final hash table and stash are sent to all servers.

We note that different servers can be used to reshuffle different levels, thus distributing the load of work and communication equally among all servers.

## 6.4  Analysis

**Complexity.** The query complexity of the $m$-server scheme is identical to that of the three-server construction, and is equal to $O(k + L)$. To obliviously construct a hash table and a stash for a level $i$, the client and the servers exchange $C_{\mathsf{Build}}(H_i) = O(\beta d^{i-1}k)$ records (recall $\beta := \max_i \frac{C_{\mathsf{Build}}(H_i)}{d^{i-1}k}$). However, since the build-up is done over tags of size $\log N$ bits, rather than whole blocks of size $\Omega(\beta \log N)$, the cost of $O(\beta d^{i-1}k)$ tags translates to $O(d^{i-1}k)$ overhead in blocks. The matching procedure also has a linear cost in the size of the level, that is $O(m_i)$. Since the reshuffling of the level occurs every $d^{i-1}k$ rounds, this amortizes to $O(1 + m_i/d^{i-1}k)$ overhead per level, and $O(L + \sum_{i=1}^{L} \frac{m_i}{d^{i-1}k})$ overall.

**Security.** Following Definition 2, it suffices to prove the following Lemma.

**Lemma 2 (Security of the $m$-server ORAM).** *Let* $\mathsf{View}_a(\boldsymbol{y})$ *be the view of server* $\mathcal{S}_a$ *during the execution of the m-server ORAM protocol, described in Section 6, over a virtual access pattern* $\boldsymbol{y} = ((v_1, x_1), \ldots, (v_\ell, x_\ell))$. *For any subset of servers* $A \subseteq \{0, \ldots, m-1\}$ *of size* $|A| \leq m-1$, *there exist a simulator* $\mathsf{Sim}_A$, *such that for every* $\boldsymbol{y}$ *of length* $\ell$, *the distributions* $\mathsf{Sim}_A(\ell)$ *and* $\langle \mathsf{View}_a(\boldsymbol{y}) \rangle_{a \in A}$ *are computationally indistinguishable.*

We make the same assumptions taken in the analysis of the three-server scheme: encryption is secure, and the tagging functions are random.

First, consider the view of the servers at the reshuffles in the two-server case. Claims 1 and 2 are true for the multi-server scheme as well, therefore, the amount of encrypted data exchanged in Step 1 of Algorithm 3 is oblivious, and therefore

so is the view of the servers. From Definition 4, the view seen in Step 2 can be also simulated by producing an access pattern for an arbitrary execution of the oblivious $H_i$.Build procedure. As for the matching procedure (Steps 3-7), the view of $\mathcal{S}_1$ consists of the newly constructed hash table and stash, both encrypted and of fixed size, and therefore can be simulated arbitrarily. $\mathcal{S}_0$ receives a sequence of tags computed using $F_s$. We show below that these tags are computed for unique headers.

**Claim 7** *The tagging function $F_s(\cdot)$ will not be computed twice on the same input in Step 3 of Algorithm 3 throughout the executions of the algorithms during the ORAM simulation.*

*Proof.* For headers $v$ that are of the form $'empty \circ z'$ the claim is trivial. It is also easy to verify that Claim 3 is true for the $m$-server ORAM construction as well, thus implying the claim for non-empty headers. □

Since unique, the sequence of tags seen by the server is indistinguishable from uniform distinct values, and $\mathsf{Sim}_0$ simulates them as such. Lastly, $\mathcal{S}_0$ receives a sequence of tagged records. The records themselves are encrypted, and therefore can be simulated. The tags were obtained by tagging the same set of headers already tagged previously, however, in an order that is generated by $\mathcal{S}_1$. Since $\mathcal{S}_1$ applies a uniformly random permutation, that is not known to $\mathcal{S}_0$ and is *independent of previous reshuffles*, over the headers, the view of $\mathcal{S}_0$ in Step 5 of the algorithm can be simulated by taking the sequence of tags generated previously by $\mathsf{Sim}_0$, and permute them randomly.

In the multi-server setting, we take the seemingly hardest case in which $\mathcal{S}_0$ is among the $m - 1$ corrupt servers. To simulate the view of $\mathcal{S}_0$, up to Step 5 of the algorithm, we follow the steps taken above. The combined view of all other corrupt servers consists of sequences of encrypted records of fixed lengths. We argue that the sequence of tagged records, received by $\mathcal{S}_0$ once all servers took their turn, can be simulated by randomly permuting the tags that were seen previously in Step 3. The key observation to make here is that, since there is at least one honest server that permutes the records randomly, using a permutation unknown to the adversary, the order of the tagged records is random from the adversary's point of view, and is independent of the input. More specifically, every permutation that the honest server may choose yields a different order of the records. Since the permutation is chosen uniformly at random, the order of the records distributes uniformly.

The access pattern seen by the servers during the queries is even easier than the three-server construction from Section 5, since now we rely on the obliviousness of the Lookup procedure (see Definition 4), rather than on the randomness of the tags (as in Claim 5). From Definition 4, the sequence of $H_i.\mathsf{Lookup}(v, \kappa_i^j)$ values is indistinguishable from a sequence generated for an arbitrary sequence of addresses $v$ using random hash keys. Thus, $\mathsf{Sim}_A$ simulates the queries to the hash tables by generating random keys using Gen, and computing Lookup over an arbitrary sequence of addresses. The transcripts of the $(m - 1)$-private PIR protocols, can be simulated from the security definition of PIR.

# 7 Conclusion and Open Questions

In this paper, we presented a family of efficient distributed ORAM schemes, that achieve both sub-logarithmic overhead, and commit to a small block size. Although our parameters match the lower bound proven in [AFN$^+$17] for PIR-based ORAM, we use techniques that are not captured in the model considered in their proof. It remains an open question whether we can further develop these techniques to surpass their bound, without using additively homomorphic encryption (AHE), or linear server work.

By allowing the servers to perform a linear amount of computation, we have shown that constant overhead ORAM is achievable. Our 4-server construction is simple, and does not use a complex data structure for the server storage, but rather stores it as a plain array. The simplicity of the construction opens the possibility to use it as a building block, to further reduce the overhead of distributed ORAM protocols, in exchange for increasing server-side computations.

# References

ACMR95. M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen. Parallel randomized load balancing. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 238–247, New York, NY, USA, 1995. ACM.

AFN$^+$17. I. Abraham, C. W. Fletcher, K. Nayak, B. Pinkas, and L. Ren. *Asymptotically tight bounds for composing ORAM with PIR*, volume 10174 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 91–120. Springer Verlag, Germany, 2017.

AIK06. B. Applebaum, Y. Ishai, and E. Kushilevitz. Cryptography in $NC^0$. *SIAM J. Comput.*, 36(4):0097-5397, December 2006.

AKL+18. G. Asharov, I. Komargodski, W-K. Lin, K. Nayak, and E. Shi. Optorama: Optimal oblivious RAM. Cryptology ePrint Archive, Report 2018/892, 2018.

AKS83. M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

AKST14. D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In Hugo Krawczyk, editor, *Public-Key Cryptography – PKC 2014*, pages 131–148, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

BGI15. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 337–367, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

BIW07. O. Barkol, Y. Ishai, and E. Weinreb. On locally decodable codes, self-correctable codes, and t-private PIR. In M. Charikar, K. Jansen, O. Reingold, and J. D. P. Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 311–325, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

CGKS98. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.

CGLS17.  T-H. H. Chan, Y. Guo, W-K. Lin, and E. Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. Cryptology ePrint Archive, Report 2017/924, 2017.

CKN$^+$18.  T-H. H. Chan, J. Katz, K. Nayak, A. Polychroniadou, and E. Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. Cryptology ePrint Archive, Report 2018/851, 2018.

CMS99.  C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 402–414, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

DG16.  Z. Dvir and S. Gopi. 2-server PIR with subpolynomial communication. *J. ACM*, 63(4):39:1–39:15, September 2016.

DS17.  J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 523–535, New York, NY, USA, 2017. ACM.

DvF$^+$16.  S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. *Onion ORAM: A constant bandwidth blowup oblivious RAM*, volume 9563 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 145–174. Springer Verlag, Germany, 1 2016.

FNR$^+$15.  C. W. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth oblivious ram. *IACR Cryptology ePrint Archive*, 2015:1065, 2015.

GI14.  N. Gilboa and Y. Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 640–658, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

GKK$^+$12.  S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 513–524, New York, NY, USA, 2012. ACM.

GM11.  M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Automata, Languages and Programming*, pages 576–587, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

GO96.  O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.

Gol87.  O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM.

Goo11.  M. T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27:1–27:26, December 2011.

GR05.  C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming*, pages 803–815, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

IKOS08. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with constant computational overhead. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 433–442, New York, NY, USA, 2008. ACM.

KLO12. E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

KMW09. A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, December 2009.

KO97. E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, pages 364–, Washington, DC, USA, 1997. IEEE Computer Society.

LN18. K. G. Larsen and J. B. Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology CRYPTO 2018*, pages 523-542, Cham, 2018. Springer International Publishing.

LO13. S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In Amit Sahai, editor, *Theory of Cryptography*, pages 377–396, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

MBM15. T. Moataz, E. Blass, and T. Mayberry. Chf-oram: A constant communication ORAM without homomorphic encryption. Cryptology ePrint Archive, Report 2015/1116, 2015.

OS97. R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 294–303, New York, NY, USA, 1997. ACM.

Ost90. R. Ostrovsky. Efficient computation on oblivious RAMs. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 514–523, New York, NY, USA, 1990. ACM.

PPRY18. S. Patel, G. Persiano, M. Raykova, and K. Yeo. Panorama: Oblivious RAM with logarithmic overhead. Cryptology ePrint Archive, Report 2018/373, 2018.

PR04. R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

PR10. B. Pinkas and T. Reinman. Oblivious RAM revisited. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 502–519, Berlin, Heidelberg, 2010. Springer-Verlag.

RYF+13. L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. *SIGARCH Comput. Archit. News*, 41(3):571–582, June 2013.

SCSL11. E. Shi, T. H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with O((logn)3) worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 197–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

SvDS+13. E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious RAM protocol. In

Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

WCS15. X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 850–861, New York, NY, USA, 2015. ACM.

WGK18. X. Wang, D. Gordon, and J. Katz. Simple and efficient two-server ORAM. Cryptology ePrint Archive, Report 2018/005, 2018.

ZMZQ16. J. Zhang, Q. Ma, W. Zhang, and D. Qiao. Mskt-oram: A constant bandwidth ORAM without homomorphic encryption. Cryptology ePrint Archive, Report 2016/882, 2016.

# A   From Amortized Overhead to Worst-Case Overhead

Up to this point, in all of our constructions and analysis, we referred to the amortized overhead an ORAM protocol incurs. Although we have achieved an overall sub-logarithmic overhead, some operations, specifically reshuffles of large levels, might require sending $\Omega(N)$ blocks in a round. This might be unwanted in several applications, in which we may have to guarantee that every query, without exceptions, incurs a bounded worst-case amount of overhead. Ostrovsky and Shoup [OS97] were the first to construct a worst-case ORAM scheme. They achieved a $O(\log^3 N)$ worst-case overhead by de-amortizing the classic hierarchical solution from [GO96]. Since then, many works have tackled the worst-case version of the ORAM problem, either by de-amortizing existing schemes (e.g. [KLO12]), or by constructing new tree-based ORAMs [SCSL11, SvDS$^+$13].

In this section, we show how to de-amortize our two-server construction from Section 6.1 in order to obtain a two-server oblivious RAM scheme with an equal *worst-case* overhead. The same method can be used also to de-amortize the $(m, m-1)$-ORAM schemes, however, it fails when applied to the three-server scheme from Section 5.

## A.1   Overview

We follow the techniques used in [OS97] and [KLO12]. The idea is to 'spread' the reshuffles and perform them throughout many rounds, rather than in a single round. Once a level has to be reshuffled, it is not reshuffled immediately, instead, a *reshuffling process* starts:

1. The full level is put aside and marked as 'inactive', while the protocol continues as usual with a new empty 'active' instance of the level.
2. In the round when the process starts, and in every round to follow, a fixed amount of $\rho$ blocks is exchanged between the client and servers as part of the reshuffle. The blocks are shuffled into an empty hash table in a larger active level. Up until the completion of the process, the destination hash table is not accessed and is treated as if it was empty.

34

During the process, the client might have to query the inactive levels for blocks that reside there. Therefore, during the search for a block, PIR protocols are performed over the inactive levels as well.

3. The process ends when the reshuffle is complete and all blocks reside in a newly created hash table in both servers.

The rate of the process, i.e. $\rho$, has to be chosen wisely. On the one hand, $\rho$ has to be small enough so the overall amortized overhead, caused by the reshuffling of all levels, does not exceed $O(k + L)$. On the other hand, very small $\rho$ means slow long-lasting reshuffling processes, which might lead to storage overflow and undesired overhead, as reshuffles will overlap and stack over time. In fact, for every level $i = 1, \ldots, L$, we choose a different rate $\rho_i$, that depends on the parameters of the hashing scheme $H_i$, and that guarantees that a reshuffling process of level $i$ ends before a new one starts. Thus, it would be sufficient to have only two instances of every level, alternating between active and inactive.

This approach introduces few obstacles, for which we suggest the following solutions.

*Reshuffles are Unproportional to Level Size.* Take the top level as an example. The top level has to be reshuffled every $k$ queries. The level is not always reshuffled into the next level, but it may be reshuffled, together with other blocks, to much larger ones (levels of size up to $\Theta(N)$). Reshuffles to large levels clearly require large amount of communication, therefore require large number of rounds to complete, during which, the top level is reshuffled many times. This situation forces us to maintain lots of inactive instances of the top level, as it is reshuffled very frequently, while the reshuffles last for unproportionally long periods.

To solve this problem, we reshuffle levels separately of each other: level $i$ is always reshuffled into the next empty hash table in level $i + 1$ (when all its hash tables are full). Furthermore, we add an extra hash table to every level in the hierarchy, making it $d$ tables in a level. Conceptually, these two modifications do not change much in the scheme: levels are still reshuffled in the same frequency, and the reshuffling of levels $1, \ldots, i$ to level $i + 1$ is replaced by a cascading series of reshuffles, starting from the top level, and ending with level $i$ reshuffled into level $i + 1$. However, this change is necessary for the de-amortization.

*Inactive Levels are Immutable.* In the amortized two-server scheme, whenever a block is found and read from the hierarchy, it is wiped out and marked as a dummy block, before it is re-written to the top level. To maintain the integrity of the reshuffling process, we clearly cannot alter the content of an inactive level while it is being reshuffled. Therefore, we allow having duplicated blocks in the hierarchy, and we avoid wiping out blocks after they are queried.

*Duplicates Compromise Security.* The uniqueness of tags in the hierarchy played an essential role in the security of the scheme in Section 6.1. Now that we allow duplicates, the scheme is no longer secure, as repeated tags, which indicate identical block addresses, might appear. We suggest the following solution. We

add a single bit to the header of every block that indicates whether its content is up to date. The blocks inserted to the top level are marked as *up-to-date*. Every time a block is read, it is marked as *outdated*. Before the start of the reshuffling process of any level, the client eliminates outdated blocks, and thus we guarantee uniqueness of tags in every hash table, which is, in fact, sufficient for the security of the scheme.

## A.2   De-amortization in Details

We modify the scheme from Section 6.1 as follows.

*Data Structure.* The following changes are made to the data structure:

– An additional hash table is added to every level.
– We double every level in the hierarchy into two instances: one initially marked as 'active', the other as 'inactive'. The two instances are identical in structure, but they operate separately, and each has its own hashing key.

*Block Headers.* In addition to the virtual address, we require that the header of every block contains an *up-to-date* bit, that indicates whether the block is up-to-date and its content is relevant. The bit is encrypted together with the virtual address, however, it is not involved in the computation of the tags. The notion of a block is extended to $(v, u, x)$, with $u \in \{up\text{-}to\text{-}date, outdated\}$ being the up-to-date bit.

*Queries.* The query procedure (Algorithm 1) is worst-case efficient. Nonetheless, in order to support the new reshuffling method, we make a couple of modifications:

– The target block is searched for in the inactive levels as well. Clearly, out-to-date blocks are ignored during the scan.
– Instead of wiping out blocks that are found in the hierarchy, and replacing them with dummy hash blocks (Algorithm 1, Step 17), the client marks them as unread, by toggling the *up-to-date* bit off.

*Reshuffles.* As a first step towards de-amortized reshuffles, we make the following changes.

– A level is reshuffled once its all hash tables are full.
– A level is always reshuffled to the next empty hash table in the subsequent level (the first bullet guarantees that there is at least one empty hash table).

Notice that, as a consequence to these changes, the last hash table of every level is never accessed while it is active, as the level is immediately reshuffled once it is filled. Thus, the purpose of adding an extra hash table is to 'make room' for blocks that are pending to be reshuffled to a larger level.

The reshuffling of a level begins with marking it as inactive, and replacing it by the other instance of the level, which is guaranteed to be clear and already reshuffled to the next level, as we show later on (Lemma **??** below).

The reshuffling proceeds following the reshuffling procedure of the scheme in Section 6.1, and is performed over many rounds. In every round of the reshuffling procedure of level $i$, $\rho_i$ blocks are exchanged between the client and servers. We choose $\rho_i = \Theta(\frac{m_i}{d^{i-1}k})$ (exact value is provided in the analysis section).

Lastly, the reshuffling procedure is adjusted to handle outdated blocks. Just before the reshuffling begins, the client replaces all outdated blocks, that reside in the level, with numbered dummy hash blocks. More specifically, $\mathcal{S}_0$ (or $\mathcal{S}_1$) sends all records in the level, one by one, to the client. The client decrypts every record $(v, u, x)$ it receives. If $u = up\text{-}to\text{-}date$, then the client re-encrypts the record and sends it back to both $\mathcal{S}_0$ and $\mathcal{S}_1$. Otherwise, the client replaces the record with $(dummyHash \circ r, \cdot, \cdot)$, where $r$ is a counter that increments whenever a dummy hash record is created (this is the only place it happens, as we do not create dummy blocks in the queries), and sends it back to the servers. The servers, in their turn, replace the received records with the records in the level, and proceed with the reshuffling.

Although the reshuffling processes of a level do not overlap, reshuffling processes of different levels might be running simultaneously. The chosen values of $\rho_i$ guarantee that this does not cause an undesired blow-up in the overhead of the protocol. However, now that the client has to maintain the state of many reshuffling processes in parallel, his working memory is no longer constant. To overcome this issue, the client encrypts the state of every ongoing reshuffling process, and stores it in one of the servers. When a reshuffling process is resumed, the client downloads its state, and proceeds with the reshuffling. Once $\rho_i$ blocks were exchanged, the client pauses the process, and uploads the new state to the server. Thus, at every moment, the client holds the state of a single reshuffling process, which is of constant size.

### A.3 Analysis

**Claim 8** *For a carefully chosen value $\rho_i \in \Theta(\frac{m_i}{d^{i-1}k})$, the reshuffling processes of level $i - 1$ in the de-amortized two-server scheme do not overlap.*

*Proof.* During the reshuffling into a hash table in level $i$, $O(d^{i-1}k + m_i)$ blocks are exchanged between the client and servers (see analysis in Section 6.4), specifically, there exists a constant $c_i$, s.t. the cost of the reshuffling is bounded by $c_i(d^{i-1}k + m_i)$. Therefore, the reshuffling process into level $i$ lasts less than $\frac{c_i}{\rho_i}(d^{i-1}k + m_i)$ rounds, and is initiated every $d^{i-1}k$ rounds. Hence, for $\rho_i > \frac{c_i m_i}{d^{i-1}k} + 1$ the processes do not overlap. $\square$

*Query Overhead.* It was shown in Section 6.1 that the worst-case overhead incurred by the queries in the amortized two-server scheme is $O(k+L)$, the changes made to the query procedure do not change that.

*Reshuffling Overhead.* The reshuffling of level $i - 1$ to a hash table in level $i$ contributes an amount of $\rho_i$ blocks to the worst-case overhead. Summing over all levels, we get $O(\sum_{i=1}^{L} \frac{m_i}{d^{i-1}k})$.

# B   Application to Secure Multi-Party Computation

In this section, we briefly describe how our distributed ORAM constructions can be used in the setting of secure multi-party computation of RAM programs to achieve efficient protocols. The idea of using oblivious RAM for secure computation was suggested in prior work [OS97, LO13, GKK$^+$12]. Ostrovsky and Shoup [OS97] were the first to propose a method to use oblivious RAM protocols with simple client circuits (e.g. [KLO12]), to construct secure computation protocols for functions that are naturally presented as RAM programs, rather than circuits, and thus avoiding the heavy cost caused by unrolling such programs into circuits. The main drawback of their approach is the fact that they use single-server ORAM as a building block, over which they apply two-server PIR [CGKS98]. That is, they do not take full advantage of the existence of two servers. Lu and Ostrovsky [LO13] make a better use of the two servers and achieve a better overhead.

The main idea behind using distributed ORAM for secure computation is as follows. To securely compute a RAM program in an $m$-party setting, we use an $m$-server ORAM protocol. Each of the servers is simulated by one of the parties. From the security of the underlying ORAM, the secrecy of the inputs is protected from the views of the servers. However, it is not guaranteed that the client's view is oblivious in the inputs, and therefore, the client has to be simulated securely. Specifically, the client's state (its local memory) is secret-shared among the $m$-parties. To simulate the next query in the ORAM protocol, the parties invoke a secure computation protocol to simulate the client's circuit, and compute the query to the servers, and the client's next state, given the current state. In all of our constructions, the local computations performed by the client can be described by circuits of size linear in the required word size, therefore, using a secure computation protocol that incurs a constant overhead in the size of the circuit (e.g. [IKOS08]) does not cause an undesired blowup.

By using any of our constructions, together with a secure circuit computation protocol with the appropriate number of parties and privacy, we achieve the following results. We refer the reader to [OS97, LO13] for further details.

**Theorem 5.** *Suppose there exists a symmetric-key encryption scheme and a hash function modeled as a random function or an efficient PRF. Suppose that for any constant $m \geq 2$, there exists an $m$-party secure circuit computation protocol with constant overhead, that is private against an adversary corrupting an arbitrary subset of parties (e.g. [IKOS08]). Then, to securely compute a RAM program that runs in $T(n)$ time with access to $S(n)$ space, with program size (including inputs) bounded by $\Lambda(n)$, for input of size $n$ words, there exist*

- a three-party secure RAM computation protocol in the semi-honest model with $O(\omega(1) \cdot \log(T + \Lambda)/\log\log(T + \Lambda))$ multiplicative overhead in communication and computational complexity. To achieve optimal overhead, the protocol requires that the program runs over data of $\Omega(\log^{1+\epsilon} n)$-bit words (or, alternatively, $O(\log(T+\Lambda)/\log\log(T+\Lambda))$ overhead with $\Omega(\log^{1.5+\epsilon} n)$-bit words).
- for every constant $m \geq 2$, an $m$-party secure RAM computation protocol in the semi-honest model, that is private against an adversary corrupting an arbitrary subset of parties, with $O(\log(T+\Lambda)/\log\log(T+\Lambda))$ multiplicative overhead in communication and computation complexity. To achieve optimal overhead, the protocol requires that the program runs over data of $\Omega(\log^2 n)$-bit words.
- a four-party secure RAM computation protocol in the semi-honest model with constant multiplicative overhead in communication and $O(S(n))$ overhead in computation complexity. To achieve optimal overhead, the protocol requires that the program runs over data of $\Omega(\lambda \log n)$-bit words, where $\lambda$ is a security parameter. In this construction we also assume the existence of constant-depth PRGs (with minimal expansion) [AIK06].

all protocols incur an additive one-time cost of $O(\Lambda \log \Lambda / \log\log \Lambda)$ for setup.

## C    Hashing

We hereby formally define a hashing scheme, and an oblivious hashing scheme. As will be shown below, these definitions capture many of the known hashing schemes, especially those applicable to ORAM.

**Definition 3.** A $(n, m, s)$-Hashing Scheme is a tuple $H = (\mathsf{Gen}, \mathsf{Build}, \mathsf{Lookup})$, where $\mathsf{Gen}$ and $\mathsf{Lookup}$ are PPT algorithms, and $\mathsf{Build}$ is a program in the standard RAM model (see Section 2). The scheme operates in two phases:

1. **Build-up.**

$$k \leftarrow H.\mathsf{Gen}(N)$$
$$(T, S) \leftarrow H.\mathsf{Build}(k, X)$$

The input to $H.\mathsf{Build}$ is a key, $k$, generated by $H.\mathsf{Gen}$, and a set of $n$ distinctly tagged values $X = \{(\tau, x) \mid \tau \in [N]\}$. The output consists of:
  - An array $T$ of size $m$, referred to as the hash table. Every position in $T$ is either empty or contains some record $(\tau, x)$.
  - A stash $S$, that contains at most $s$ tagged values,
  The build-up might fail with probability at most $N^{-\omega(1)}$ over the choice of $k$. The **build-up complexity** of $H$, denoted by $C_{\mathsf{Build}}(H)$ is the amount of data blocks exchanged between the CPU and RAM in an $H.\mathsf{Build}$ invocation.
2. **Queries.**

$$Q \leftarrow H.\mathsf{Lookup}(\tau, k)$$

*The input to H.Lookup is a tag $\tau \in [N]$, and a key $k$. The output of H.Lookup is a set of positions $Q \subseteq [m]$. Let $(T, S, k)$ be the output of a previously performed construction phase on input $(k, X)$. If $(\tau, x) \in X$ for some $x$, then either $(\tau, x)$ resides in $T[j]$ for some $j \in Q$, or $(\tau, v)$ resides in the stash $S$. The **query complexity** of $H$, denoted by $C_{\mathsf{Lookup}}(H)$, is the maximal size of a set $Q$ returned by H.Lookup.*

**Definition 4.** *An* Oblivious Hashing Scheme $H$ *is a hashing scheme where*

- $H$.Build *is oblivious in $k, X$. That is, for any two distinct inputs, $(k, X)$ and $(k', X')$ of the same length, the corresponding access patterns produced by $H$.Build are computationally indistinguishable.*
- *For any two sequences of distinct tags $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_\ell)$ and $\boldsymbol{\tau}' = (\tau_1', \ldots, \tau_\ell')$, the outputs of $H$.Lookup on $\boldsymbol{\tau}$ and $\boldsymbol{\tau}'$ are computationally indistinguishable.*

For completeness, we provide below an overview of the hash schemes that were considered in the oblivious RAM context.

*Standard hashing.* To store $n$ tagged values, standard hashing generates a hash table that is a plain array of $m = cn$ buckets (e.g. $c = 2, 4$), each of size $O(\log n)$, and a hash function $h_k : [N] \to \{1, \ldots, m\}$. A record $(\tau, x)$ is stored in bucket $h_k(\tau)$ in the array. When $h_k$ is thought of as a random oracle, the probability of an overflow is negligible in $N$. In their classic hierarchical solution, Goldreich and Ostrovsky [GO96] constructed an oblivious variant of standard hashing with build-up complexity of $O(n \log n)$, by using PRFs as hash functions, and oblivious sort [AKS83, Goo11] to obliviously store the data in the hash table. Lu and Ostrovsky [LO13] showed that by using a (shared) stash of size $O(\log N)$, the bucket size can be reduced to $O(\log N / \log \log N)$ while maintaining a negligible failure probability.

*Cuckoo hashing.* Cuckoo hashing [PR04] is a variant of standard hashing, that was found to be useful in some ORAM schemes [PR10, GM11, KLO12, LO13]. In cuckoo hashing, every bucket may contain at most one record, and two hash functions $h_k^0, h_k^1$ are used. A record $(\tau, x)$ is initially stored in $h_k^0(\tau)$, and is moved to $h_k^1(\tau)$ once another record is inserted into $h_k^0(\tau)$. Notice that this strategy might cause a long chain, or even a cycle of block relocations. In such case, new functions are chosen, and the structure is updated. It was shown that the amortized insertion time in cuckoo hashing is $O(1)$. A variant of cuckoo hashing is stashed cuckoo hashing, presented by Kirsch et al. [KMW09]. In stashed cuckoo hashing, a stash of size $s$ is used to handle overflows. They show that using a constant size stash, inserting $n$ elements into a table of size $\Theta(n)$ will fail with probability $O(n^{-s})$. Goodrich and Mitzenmacher [GM11] showed that a table of size $\Omega(\log^7 N)$ with a stash of size $O(\log N)$ fails with probability negligible in $N$. Oblivious sorting was shown to be useful also to construct oblivious cuckoo hashing scheme with build-up complexity of $O(n \log n)$ [PR10, GM11, CGLS17].

*Two-tier hashing.* In two-tier hashing [ACMR95], as the name suggests, the data is stored in two standard hash tables, each with a different hash function. Every table contains $O(n/b)$ buckets, each of capacity $b = \log^\epsilon N$ for $\epsilon \in (0.5, 1)$. A record $(\tau, v)$ is inserted to the appropriate bucket in the first tier, if the bucket is full, then the record is inserted to the second tier. Chan et al. [CGLS17] presented an oblivious variant of two-tier hashing with build-up complexity of $O(n \log n)$, and used it to construct oblivious RAM. Although their ORAM scheme has an overhead asymptotically equal to the ORAM based on cuckoo hashing [KLO12], its advantage lays in the simplicity of oblivious two-tier hashing, both in construction and analysis, relative to oblivious cuckoo hashing.

## D  Single-server Query Procedure [KLO12]

Below is a pseudo-code of the query procedure of the single-server ORAM scheme from [KLO12], which is described in a high level in Section 4.

---
**Algorithm 4** Single-Server Balanced Hierarchy: Query
---
1: Allocate a local register of the size of a single entry.
2: Initialize a flag found $\leftarrow 0$.
3: Download the top level and stash $S$, record by record. If a record $(v, x)$ was found, then store it in the local register, and mark found $\leftarrow 1$.
4: **for** every level $i = 1 \ldots L$ **do**
5:     $t' \leftarrow \lfloor t/k \rfloor$
6:     $r \leftarrow \lfloor (t' \bmod d^i)/d^{i-1} \rfloor$
7:     **for** every hash table $j = r \ldots 1$ **do**
8:         If not found, $Q \leftarrow H_i.\mathsf{Lookup}(v, \kappa_i^j)$. Else, $Q \leftarrow H_i.\mathsf{Lookup}(dummy \circ t, \kappa_i^j)$.
9:         Download records $T_i^j[Q]$, record by record. If a record $(v, x)$ was found, then store it in the local register, and mark found $\leftarrow 1$.
10:     **end for**
11: **end for**
12: If the query is a write query, overwrite $x$ in the register.
13: Read each entry of the entire top level from both servers one at a time, re-encrypt it, then write it back, with the following exception: if the the entry $(v, x)$ was first found at the top level, then overwrite $x$ with the (possibly) new value from the register, otherwise, write $(v, x)$ in the first empty spot of the form $(empty, \cdot)$.
14: Increment the counter $t$, and reshuffle the appropriate levels.
---

## E  Proof of Claim 2

In this section, we prove Claim 2 from Section 5.3. We re-state it here for convenience.

**Claim 2** *Let $t$ be a multiple of $k$, and denote $t' = t/k$. For every $1 \leq i \leq L$, define $r_i^t := \lfloor (t' \bmod d^i)/d^{i-1} \rfloor$. Then,*

*(i) the top level is full prior to the reshuffle at round $t$, and is empty afterwards.*

*(ii) for every other level $1 \leq i \leq L$, once the reshuffle is completed, the first $r_i^t$ tables in level $i$ (i.e., $T_i^1, \ldots, T_i^{r_i^t}$) are full (contain $d^i(k+s)$ records each), and all other tables in level $i$ are empty.*

*Proof.* We prove the lemma using induction. For $t = 0$, we have $i^* = 0$ and $r_i^t = 0$ for every $i$, the entire storage is empty, therefore the lemma is trivially correct. Suppose the lemma holds true for some $t = t' \cdot k \geq 0$. Let $i^*$ be the smallest value of $i$ for which $r_i^{t+k} > 0$. Notice that $i^*$ is also the largest integer $i$ for which $d^{i-1} \mid t' + 1$ (the level to which the reshuffle of round $t+k$ is done). We proceed by proving the lemma is correct at round $t + k$ for four different groups of levels:

1. *The top level*: from the induction hypothesis, the top level is empty when the reshuffle at round $t$ is finished. Performing queries $t, \ldots, t+k$ fills up the top level entirely (as it is not reshuffled down during this time). The reshuffle after query $t + k$ empties the top level, thus, (i) holds.
2. *Levels $1 \leq i < i^*$*: the reshuffle at round $t + k$ evicts all blocks from levels smaller than $i^*$ to hash table $T_{i^*}^{r_{i^*}}$. Therefore, once the reshuffle is done, levels $i = 1, \ldots, i^* - 1$ are empty, matching the fact that $r_i^{t+k} = 0$ and proving the lemma for such $i$'s.
3. *Level $i^*$*: from the choice of $i^*$, there exists some $0 < r < d$ for which $t' + 1 = q \cdot d^{i^*} + r \cdot d^{i^*-1}$. It also holds that $r_{i^*}^{t+k} = r$, and $r_{i^*}^t = \lfloor (t' \bmod d^{i^*})/d^{i^*-1} \rfloor = \lfloor (r \cdot d^{i^*-1} - 1)/d^{i^*-1} \rfloor = r - 1$. These give $r_{i^*}^{t+k} = r_{i^*}^t + 1$. By induction, we can imply that after the reshuffle of round $t$, $T_{i^*}^r$ is empty. Furthermore, all hash tables in levels $i < i^*$ are all full prior to the reshuffle, i.e. contain $d^{i-1}(k+s)$ blocks, since $r_i^t = d - 1$ for such $i$'s. The reshuffle procedure in round $t + k$ takes all blocks in levels $i = 0, \ldots, i^* - 1$ and stash, and inserts them into $T_{i^*}^r$ and the stash. The count of these blocks is $s + k + \sum_{i=1}^{i^*-1} d^{i-1}(k+s) = d^{i^*-1}(k+s)$, making $T_{i^*}^r$ full (notice that blocks that are sent to the stash are replaced by dummy stash blocks). Since the other tables in level $i^*$ are untouched in rounds $t$ and $t + k$, the lemma holds for $i^*$.
4. *Levels $i^* < i \leq L$*: these levels are not involved in the reshuffle at round $t+k$, and remain unchanged during queries $t, \ldots, t + k$. Thus, it suffices to show that $r_i^t = r_i^{t+k}$ for every such $i$. Again, we rewrite $t' + 1 = q \cdot d^{i^*} + r \cdot d^{i^*-1}$ for some $0 < r < d$. Take some $i > i^*$. Using the fact that $r > 0$, we get $r_i^t = \lfloor (t' \bmod d^i)/d^{i-1} \rfloor = \lfloor ((q \cdot d^{i^*} \bmod d^i) + r \cdot d^{i^*-1} - 1)/d^{i-1} \rfloor = \lfloor ((q \cdot d^{i^*} \bmod d^i) + r \cdot d^{i^*-1})/d^{i-1} \rfloor = r_i^{t+k}$.

□