FutORAMa: A Concretely Efficient Hierarchical Oblivious RAM

Gilad Asharov
Gilad.Asharov@biu.ac.il
Bar-Ilan University
Ramat-Gan, Israel

Ilan Komargodski
ilank@cs.huji.ac.il
Hebrew University and NTT Research
Jerusalem, Israel

Yehuda Michelson
Yehuda.Michelson@gmail.com
Bar-Ilan University
Ramat-Gan, Israel

ABSTRACT

Oblivious RAM (ORAM) is a general-purpose technique for hiding memory access patterns. This is a fundamental task underlying many secure computation applications. While known ORAM schemes provide optimal asymptotic complexity, despite extensive efforts, their concrete costs remain prohibitively expensive for many interesting applications. The current state-of-the-art practical ORAM schemes are suitable only for somewhat small memories (Square-Root ORAM or Path ORAM).

This work presents a novel concretely efficient ORAM construction based on recent breakthroughs in asymptotic complexity of ORAM schemes (PanORAMa and OptORAMa). We bring these constructions to the realm of practically useful schemes by relaxing the restriction on constant local memory size. Our design provides a factor of at least 6 to 8 improvement over an implementation of the original Path ORAM for a set of reasonable memory sizes (e.g., 1GB, 1TB) and with the same local memory size. To our knowledge, this is the first practical implementation of an ORAM based on the full hierarchical ORAM framework. Prior to our work, the belief was that hierarchical ORAM-based constructions were inherently too expensive in practice. We implement our design and provide extensive evaluation and experimental results.

1 INTRODUCTION

Cloud services are becoming more and more popular. One of the earliest and most basic applications of cloud computing is for cloud storage, where a client can offload a database to a remote server and later access the database and query it. Such services have been in wide use already since the early 1980s [29]. A cloud-based paradigm has various advantages, ranging from improved energy consumption, flexible resource utilization, and various other optimized workflows and environmental considerations. However, all these advantages entail a significant cost in data security and privacy.

It is by now well-known and widely accepted that merely encrypting the entries of the database before uploading them to the cloud does not guarantee privacy (e.g., [8, 20, 21, 50]). Indeed, the access patterns themselves may reveal non-trivial information about the underlying data or program being executed on the data. To mitigate these kinds of attacks, we would like to be able not only to encrypt the underlying data but also to "scramble" the observed access patterns so that they look unrelated to the data. The algorithmic tool that achieves this goal is called *Oblivious RAM* (ORAM).

ORAM, introduced in the seminal work of Goldreich and Ostrovsky [16, 17, 34], is a (probabilistic) RAM machine whose memory accesses do not reveal anything about the program or data on which it is executed. An ORAM construction accomplishes this by permuting data blocks stored on the server and periodically reshuffling them around. While the original use-case of Goldreich and Ostrovsky was for software protection, ORAMs found their way to become a central tool in designing various cryptographic systems, including cloud computing design, secure processor design, multi-party computation protocols, and more [6, 13–15, 28, 30, 31, 35, 39, 42, 43, 46, 47, 49].

To be useful, ORAMs have to be "efficient". Whether an ORAM is efficient or not is typically measured by its *overhead in bandwidth*: that is, how many data items must be accessed in the oblivious simulation as compared to the original non-oblivious implementation. The original work of [17] showed that logarithmic overhead is necessary for a restricted class of ORAM constructions (ones in the balls and bins model and information-theoretically secure). These restrictions were lifted in a breakthrough result of Larsen and Nielsen [24], showing that logarithmic overhead is unavoidable, for any construction.

Existing approaches. Considerable effort has been invested in developing efficient ORAM schemes [3–5, 9, 17, 18, 23, 36, 40, 44, 45], resulting in two main approaches for designing ORAMs. First is the method originating from the original work of Goldreich and Ostrovsky [16, 17], called *hierarchical ORAM*. Along with many breakthrough non-trivial ideas and optimizations of PanORAMa [36] and OptORAMa [3], this technique can be used to get *asymptotically* optimal ORAM constructions. Concretely, for a memory of size N, OptORAMa uses a local memory consisting of $O(1)$ memory words, each of size $O(\log N)$, and their ORAM has $O(\log N)$ overhead. However, the current folklore understanding is that hierarchical ORAM-based construction *are practically inefficient*. Indeed, the hidden constant in the construction of [3] is $\gg 2^{228}$ (according to [11]) and even with various optimizations, like [11], it remains in the order of tens of thousands which makes it prohibitively inefficient for any reasonable memory size. Degenerate version of hierarchical ORAM, like the square root ORAM [12], were implemented but because of the large bandwidth overhead, their construction can only support rather small memory sizes in practice (e.g., $N = 2^{14}$).

The other method, originated in the groundbreaking work of Stefanov et al. [44], is called *Path ORAM*. This approach is considered practical because the hidden constants are very small [37]. However, it uses a poly-logarithmic local memory size and the overhead is $O(\log^2 N)$. Due to its concrete efficiency for moderate memory sizes, Path ORAM-style systems have been widely implemented and deployed. The Ascend secure processor [13, 38], the Phantom secure processor [32], the GhostRider system [27], and the practical obfuscation project [33], all adopt versions of Path ORAM. More recently, the secure messaging company Signal announced using Path ORAM for a faster layer for enclaves [41].

This work. The current folklore belief is that (a) Path ORAM-based schemes are practically efficient, albeit being sub-optimal asymptotically, but (b) Hierarchical ORAM-based schemes can be made asymptotically optimal but behave practically poorly. In this

work, we change this folklore belief by showing that **Hierarchical ORAM-based schemes can be made practical**. Our main contribution is a practically-efficient implementation of a Hierarchical ORAM, using ideas from the asymptotically optimal ORAM scheme of [3]. According to our experiments, our ORAM implementation outperforms an implementation of the original Path ORAM [44].[1] of To the best of our knowledge, this is the first time that (full) Hierarchical ORAM-based ORAMs have proven to be practically useful, let alone being better than a basic implementation of Path ORAM.

The challenge. Achieving a practical version of [3]'s ORAM scheme is highly non-trivial because the latter strongly relies on heavy machinery that we currently do not know how to bring to the realms of real-life systems. The most obvious such barrier is the *tight compaction* algorithm they develop and use. This algorithm relies on the existence of an expander graph with particular properties. They show the existence of such an expander but its concrete size is (roughly) proportional to the number of atoms in the universe. Another seemingly inherent barrier is their reliance (in multiple places) on *oblivious* Cuckoo hashing [9, 18]. While non-oblivious cuckoo hashing is practical, the building of oblivious cuckoo hashing is much more challenging and is very far from being practically relevant. We know how to design such a hashing paradigm with reasonable *asymptotic* cost; however, they employ numerous invocations of oblivious sorts used to perform non-trivial graph algorithms in an oblivious manner. Implementations of such procedures will be too costly in practice.

Recall that Path ORAM-based schemes have an extra logarithmic factor in efficiency but rather small hidden constants; In order to compete with Path ORAM, we must design a scheme where the hidden constants are smaller than a logarithmic factor in memory size, for reasonable memory sizes (e.g., to obliviously simulate a memory of size 1GB, we would need a constant < 30). Looking forward, we will achieve this goal and obtain a significant speedup over a basic implementation of Path ORAM.

Our approach. We first make the very useful observation that much of the effort in OptORAMa was due to the restriction that the client has only $O(1)$ blocks of (secure) local memory. In many motivating scenarios, such a strong requirement is not needed. For instance, when a user delegates 1TB (=2^{40} bytes) of data to the cloud, it is reasonable to require, say 1MB (=2^{20} bytes) of local storage. We introduce a parameter Z, which captures the size of the local memory and think of it as a small polynomial in $\log N$, where N is the size of the memory. We also argue that assuming Z to be roughly the square root of the memory size is reasonable. 1MB (2^{20} bytes) suffices for storing a terabyte (2^{40} bytes), 32MB (2^{25} bytes) suffice for a petabyte (2^{50} bytes), and 1GB (2^{30} bytes) suffices for an exabyte (2^{60} bytes). We find such ratios reasonable and appropriate for many settings.

A curious reader can already jump to the evaluation section (Section 7) to see concrete efficiency measures of our implementation and how they compare to the a basic implementation of Path ORAM. To get a flavor, we give a numerical example. Assuming our memory is 1TB ($N = 2^{40}$) and the local memory is only 8MB ($Z = 2^{23}$), we get an ORAM where each access takes $\approx$ 5KB of bandwidth (amortized). Basic Path ORAM, in a similar setting of parameters, consumes about $\times 6$ more bandwidth. Thinking of N and Z as parameters, the overhead of our scheme is $\leq 15 \cdot \log(N/Z)$; see the formal statement in Theorem 6.1.

We emphasize that we theoretically analyze our scheme by providing concrete constants, leaving no unspecified hidden terms. Such a thorough analysis of a Hierarchical ORAM has never been done before (mostly because they have never been considered practically interesting), and we view this as an independent contribution. The experimental results match the theoretical bounds that appear in this paper. Our implementation of both schemes exemplifies that our scheme outperforms a basic implementation of Path ORAM in all settings of parameters that we consider.

Techniques. The major advancement in ORAM constructions is due to PanORAMa [36], who observed that prior works were lossy in the way they introduced randomness into the system. PanORAMa was sub-optimal asymptotically, achieving $O(\log N \cdot \log\log N)$ overhead. At the expense of significantly enlarging hidden constants, OptORAMa [3] shaved off the extra $\log\log N$ factor. We relax the optimality of OptORAMa in one dimension, i.e., the size of local storage, allowing us to make hidden constants tiny. We now describe some of our key ideas.

We have already mentioned that one central building block in OptORAMa is an oblivious tight compaction algorithm. In this problem, first studied in the oblivious setting by [25], the input is an array where some elements are distinguished, and the goal is to output an array with the same set of elements, but where all distinguished elements appear at the beginning. This can be seen as a relaxation of an oblivious sort where elements are associated with 1-bit keys. It has been shown [26] that if the elements are indivisible, then stable oblivious tight compaction requires $\Omega(n \log n)$ work for an input of size n. The work of [26] shows a non-stable oblivious tight compaction that works in $O(n \log\log n)$ work and negligible probability of error. OptORAMa [3] shows a deterministic oblivious algorithm that is asymptotically optimal and requires only $O(n)$ work; however, the hidden constant is huge (at least 2^{228}, according to [11]). The algorithm of OptORAMa is rather involved and uses expander graphs and packing tricks. The constant was improved in [11] but is still in the tens of thousands, making it far from being practically useful.

As our first novel result, we show an oblivious tight compaction algorithm with linear overhead and assuming the client can store poly $\log n$ words (as opposed to $O(1)$ as in [3, 26]). Not only that we get asymptotically optimal linear overhead, but our hidden constant is relatively small, around 18. We do not stop there and make the following central contribution: Our ORAM is designed in such a way that we *never use oblivious tight compaction in its full generality*. Specifically, our whole ORAM only uses compaction for arrays where (1) exactly half of the elements are distinguished (and this fact is known to the adversary, and the algorithm can safely "leak" it); and (2) the input array is randomly shuffled using a permutation that is hidden from the adversary. Compaction with both relaxations has never been studied before because it was not known to suffice for an

[1] We emphasize that we compared our construction against a rather basic implementation of Path ORAM, as described in [44] in pseudocode. This version is *not* a fully optimized, as there are known "tricks" to shave off constants in performance (see, e.g., [37] for details).

ORAM. Inspired by the general tight compaction algorithm of [26], we obtain a new, extremely efficient tight compaction algorithm that is secure under the above relaxations. Concretely, it requires $4.2n$ bandwidth for an input array of size n. See Section 3 for details.

Oblivious hash table. Recall that a Hierarchical ORAM consists of a hierarchy of hash tables, a.k.a "levels", of geometrically increasing sizes. The largest layer can hold around N elements, the size of the memory. Access to the ORAM translates into a lookup into each of the levels, and the found item is moved up to the smallest level. Whenever a level gets full, its content is essentially moved down the hierarchy. Without going into too much detail, an important fact is that levels are merged and rebuilt via a fixed schedule that depends on the number of accesses. We mostly follow the procedure of OptORAMa, but make several key modifications to how the above process works, some of which we highlight next.

A level in OptORAMa is built as follows. Let n denote the number of elements in the input array. The n elements are routed (in the clear) to random bins, and then a (secret) sublinear portion of elements from each bin is routed to a secondary structure called *the overflow pile*. The remaining bins are called *the major bins*. Each major bin is poly-logarithmic in size and the overflow pile is of sublinear size in n. OptORAMa made a significant effort to implement access to the major bins obliviously, and their solution involved very practically expensive tools (including oblivious Cuckoo hashing, compaction, and various SIMD packing tricks). We completely avoid this complication by leveraging the fact that our local storage is large enough to store a whole major bin. We also use oblivious cuckoo hashing, but the build procedure is handled in the local memory and therefore is concretely efficient.

The overflow pile is largely handled similarly to OptORAMa. Since it is sublinear in size, we can actually afford to use known hashing techniques with logarithmic overhead. The main and most notable difference from OptORAMa is in the extract operation of each hash table. Recall that once in a while we need to merge some levels; for this, we first need to collect the elements that remain in each table. The extract operation does exactly that. It becomes nontrivial to implement this operation when we recall that we want to make sure that the extracted elements are randomly permuted–this is an invariant that should be maintained throughout the construction. The extract operation of a hash table, consisting of the major bins and overflow pile is easy if we do not care about concrete efficiency. Indeed, in OptORAMa the authors simply invoke general compaction and a shuffling algorithm (called intersperse). However, these primitives are concretely inefficient and so we want to avoid them. To this end, we use a concretely efficient extraction procedure which, at a very high level, reverses the build operation. This idea appeared in a similar form PanORAMa [36] but we optimize it even further, taking advantage of the fact that our local memory is large.

Experiments. We implement our scheme and compare it to an implementation of the original Path ORAM [44]. Firstly, our experiments show that our implementation behaves as the expected theoretical analysis. Secondly, in comparison to Path ORAM, our construction requires bandwidth that is about 6 to 8 times smaller. For instance, for a memory of size 1TB, accessing a block (of 32 bytes) requires an overhead of 68KB by Path ORAM, whereas our construction requires accessing only 9.2KB (amortized). We emphasize that we implemented both schemes with similar parameters, i.e., the size of the local storage is the same (specifically, roughly 8MB local storage), and we optimized basic implementation of Path ORAM to utilize this fact (specifically, by stopping the recursion at a higher level). More details appear in Section 7.

As for running times, for 1GB memory, the overhead of having an oblivious memory over a non-oblivious memory is roughly $x75$ (515 vs. 6.8 microseconds). Path ORAM requires an overhead of $x200$.

We remark that Path ORAM achieves statistical security while our scheme only gives computational security (since we rely on pseudo-random functions, implemented using AES). We analyze our scheme and bound the error probability assuming that the PRF is modeled as an ideal cipher. Moreover, Path ORAM is easier to implement (specifically, our scheme requires $\approx \times 6$ lines of code).

Summary of our contributions. We finish this section with a conclusion of our main contributions.

- To the best of our knowledge, our work is the first to ever implement a practically useful ORAM based on the (full) hierarchical framework, capable of supporting very large memory sizes and with small local memory.
- Our implementation results in an ORAM that is extremely efficient, to the extent that it beats the basic implementation of Path ORAM in analog settings of parameters. This is the first time that a hierarchical construction is better in practice for (moderately) large memory sizes.
- En route to our ORAM, we implement several underlying building blocks, some of which, we believe, are of independent interest (e.g., oblivious tight compaction, oblivious intersperse, oblivious hash tables).
- The theoretical analysis of our construction is aligned with the implementation. Our bandwidth overhead is $c \cdot \log N/Z$ where N is the size of the overhead, Z is the size of the internal memory, and c, the constant is $\approx 11.5 - 15$ (depending on several parameters), which is concretely very small.
- Our implementation is open source: https://github.com/cryptobiu/FutORAMa.

1.1 Related Work

Due to the wide range of their applications, there has been significant effort in developing and deploying efficient ORAM systems. Because existing schemes (including Path ORAM-based) were somewhat inefficient for various concrete tasks, the systems side of this endeavor focused on considering ORAM constructions with worse asymptotics but smaller hidden constants. They showed that for small memory sizes, this trade-off could lead to better practical efficiency [12, 49]. ORAMs have also been used as central building blocks in efficient secure multiparty computation protocols (in theory and practice). For example, Gordon et al. [19] showed that ORAMs could be used to perform two-party computation in sub-linear (amortized) time. Also, various improvements on the asymptotics of ORAM for multiparty computation were presented in [7, 10, 45, 46]. To ease deployment, new languages and compilers were developed to help design and transform algorithms into oblivious in a seamless manner [48].

The work which is closest in nature to ours is due to Zahur et al. [49], where a degenerate version of [17]'s hierarchical ORAM was optimized and implemented. Concretely, they implement a single level of a hierarchical ORAM, which allows to obliviously access a size N memory using $O(\sqrt{N})$ local memory and the same bandwidth overhead. Zahur et al. show that for relatively small values of N their scheme beats the state-of-the-art variant of Path ORAM [45] by making their hidden constants much smaller than in the latter. However, because their system has $O(\sqrt{N})$ bandwidth overhead, it does not fit applications where a large memory size is needed. (Indeed, in their experiments, the maximal considered memory size is $N \leq 2^{14}$.) In comparison, our ORAM has only logarithmic bandwidth overhead, so it performs very well even when the memory size is very large (e.g., 1TB or even 1PB; see Section 7).

Paper organization. The paper is organized as follows. In Section 2, we provide definitions and preliminary results needed for the rest of the paper. In Section 3, we describe and analyze our tight compaction algorithm, and in Section 4, we use it to get the intersperse algorithm. In Section 5, we present out oblivious hash table. Section 6 presents the complete ORAM construction and theoretical efficiency analysis. In Section 7, we discuss various implementation choices, evaluate our scheme and compare it to the basic implementation of Path ORAM. Section 8 provides a summary and a conclusion.

2 PRELIMINARIES AND BUILDING BLOCKS

We use λ to denote the security parameter. A function $\mu(\cdot)$ is *negligible* if for every constant $c > 0$ there exists an integer N_c such that $\mu(\lambda) < \lambda^{-c}$ for all $\lambda > N_c$.

Oblivious machines and simulation. A RAM is an interactive Turing machine that consists of a memory and a CPU. The memory is denoted as $\mathsf{mem}[N, w]$, and is indexed by the logical address space $[N] = \{1, \ldots, N\}$. We refer to each memory word also as a *block* and we use w to denote the bit-length of each block. We assume the block size is $O(\log N)$, and in our implementation, we use 64 bits. The memory supports standard read/write instructions.

We consider machines that interact with the memory via read and write operations. We consider RAM program that is reactive; namely, it supports several commands and interacts with the memory to implement that commands. Each command has some input and output, and the program stores some state between the different commands in the memory. We say that a (reactive) RAM program $\mathcal{G}$ is *oblivious* if its access pattern can be simulated by a simulator that receives only the type of the commands but not the inputs of the commands. See details in Definition A.4.

Additional preliminaries. We overview the concept of oblivious computation and the necessary definitions of oblivious machines, simulation, hybrid model, input assumptions, and the definition of oblivious RAM in Appendix A. Additionally, there we provide various probability bounds that are needed for our analysis.

Here, we briefly explain the efficiency measures we use throughout this work. We use the terminology *bandwidth* to denote the total number of read/write operations of memory cells in the RAM. We use *bandwidth* in the amortized sense. We also count round-trips; we allow sending several read/write operations in parallel to the memory. The round-trips are the number of distinct non-parallel accesses to the memory. We give a concrete example: reading T blocks to the internal memory, processing them, and writing them back is considered as a single round-trip and $2T$ bandwidth.

Lastly, we provide in Appendix E a high-level overview of the structure of OptORAMa since the construction given in this paper is highly related.

2.1 Basic Building Blocks

In our construction, we utilize several standard building blocks. Here, we explain, at a high level, what these objects achieve and defer definitional and implementation details to Appendix B.

Oblivious 2-key dictionary. An oblivious dictionary is a dynamic data structure that allows the inserting of elements (k, ℓ, v) where k is the key (i.e., in $[N]$), ℓ is a label (in $\{0,1\}^w$), and v is the value (in $\{0,1\}^w$). It is assumed that the same key k appears at most once in the dictionary. One can pop an element with the key k by using $\mathsf{PopKey}(k)$– the operation removes the element with the key k. It also supports $\mathsf{PopLabel}(\ell)$, which returns (and removes) an element with the label ℓ, if exists in the dictionary.

Oblivious bin placement. Oblivious bin placement receives as input an array X containing n real elements, each element is marked with some destination bin. The goal of the functionality is to move all elements into their destination bin (obliviously). We assume some parameter Z (which is half of the target bin size). It is assumed that the number of elements assigned to some destination bin is bounded by $2Z$. Our implementation of this functionality is based on [2].

Sampling secret loads. The goal of the algorithm is to sample loads of throwing n' balls into b bins in such a way that the access pattern does not reveal the loads. That is, our goal is to sample from the multinomial distribution with parameters n', b. Explicitly, the output is $(\ell_1, \ldots, \ell_b)$ such that $\sum_{i=1}^{b} \ell_i = n'$ with probability $\binom{n'}{\ell_1,\ldots,\ell_b} \cdot b^{-n'}$. We provide a slightly faster implementation than that of [3], see Appendix B.

3 OBLIVIOUS TIGHT COMPACTION

Oblivious tight compaction is a method for sorting an array according to a given mark. That is, it implements the following abstraction:

- **Input:** An array of size n where some of the inputs are distinguished;
- **Output:** An array of size n where all distinguished elements are moved to the front.

This abstraction turns out to be central for the design of asymptotically optimal ORAM [3]. Specifically, it allows us to avoid expensive oblivious sort [1]. However, known constructions of oblivious tight compaction are either asymptotically sub-optimal (e.g., [26]) or highly inefficient due to huge hidden constants [3]. In this section, we present yet another compaction algorithm that is both very practically efficient and theoretically optimal, assuming a poly $\log \lambda$-size client local memory. Our algorithm, which is inspired in part by the compaction algorithm of [26], is made highly efficient due to the following novel observations:

(1) **Number of distinguished elements is public:** We design our system in such a way that all invocations of tight compaction are applied on arrays where the number of distinguished elements is exactly half of the size of the array. For instance, compaction is used on an input array of size n in which $n/2$ elements are real, and $n/2$ are dummies; We invoke compaction to move all dummy elements to the end and get rid of them. This design allows us to avoid hiding the number of distinguished elements, which relaxes the security requirement and results in an efficiency improvement.
(2) **Input assumption:** We also make sure that in all invocations to tight compaction, the input array is *randomly shuffled* using a permutation that is not known to the adversary. This allows us to simplify the algorithm and skip a few of the first steps of the construction of [26].
(3) **Client's local memory:** We utilize the fact that the client's memory consists of poly $\log \lambda$ words (as opposed to $O(1)$ words [3, 26]). Concretely, our client's local memory is only 1.4MB; see more at the end of this section. This allows us to turn parts of the algorithm to be "non-oblivious" by performing them locally on the client's side.
(4) **Error probability:** The compaction of [3] has deterministic access pattern and perfect correctness. Our construction's correctness holds with overwhelming probability (i.e., statistical security).

Utilizing all those facts allows us to design a very fast algorithm, which consumes only $4.2n$ bandwidth. (c.f., the bandwidth of [3]'s compaction algorithm is at least 2^{228}.)

As mentioned, we will only utilize tight compaction where the number of distinguished elements is exactly half. To complement, we show in Appendix C a variant where the number of distinguished elements is secret and should be kept secret from the adversary. Naturally, the latter variant is much less efficient (2x to 3x slower). Finally, we note that without the input assumption, we have to slightly randomize the input, which results in another slowdown (another factor of roughly 3x). See a comparison at the end of this section.

The algorithm. The algorithm interprets the input array as n/Z bins of size Z, where $Z = \text{poly}\log\lambda$. The client sorts each one of the bins locally. Since the number of distinguished elements in the entire array is exactly half of the entire size of the array, and since we assume that the input is randomly shuffled, we expect to see the same ratio of distinguished elements in each bin. Using concentration bounds, we can show that in each bin, after sorting it locally, we are guaranteed (with overwhelming probability) to have at least $Z/4$ elements marked 0s and $Z/4$ elements marked 1s. Thus, there are $Z/2$ elements that are left to compact. The algorithm then combines all the bins and runs compaction recursively on the remaining elements to compaction - an array of size $n/2$ with $n/4$ distinguished elements. See Algorithm 3.1.

To argue that the number of elements in each bin has a similar ratio as the entire input, we need some randomization process and concertation bound. We utilize the fact that the input is randomly shuffled to derive this conclusion in the first few iterations. However, in later recursive calls, the input assumption does not necessarily hold and we have to introduce some random process. This is RandCyclicShift that is described in the algorithm, which turns out to be the bottleneck since it is not performed in place. It, in essence, places elements in random bins (but does not completely shuffle the array). Yet, RandCyclicShift will be applied on an array $1/64$ factor smaller than the input array, and so the added complexity will be small, in practice (roughly $0.2n$ in bandwidth).

Algorithm 3.1 (`CompactArrayByHalf`(A))**:**

Input: An array A of size n in which exactly half of the elements are marked as distinguished (i.e., marked 0, and the rest are marked 1).
Public input: The number of distinguished elements, $n/2$, and the total number of elements n.
Input assumption: The elements are randomly shuffled. (The input assumption holds only for the first level of the recursion.)
The algorithm:

(1) Let $Z = \text{poly}\log\lambda$, and let $b = \lceil n/Z \rceil$.
(2) If $b = 1$, then read the array to the local memory, compact it locally, and return the result as output.
(3) Let $\mathsf{Bin}_i = [A_i, A_{b+i}, A_{2b+i}, \ldots, A_{(Z-1)b+i}]$ for every $i \in [1, b]$. Note that $|\mathsf{Bin}_i| = Z$.
(4) Let depth be the depth of the recursion (initially, 0). If depth > 6, then run RandCyclicShift:
   (a) Initialize an array $W[1, \ldots, b]$.
   (b) For $i = 1, \ldots, Z$, choose a random $\mathsf{r} \leftarrow [b]$, assign $(W[1], \ldots, W[n]) \leftarrow (\mathsf{Bin}_1[i], \ldots, \mathsf{Bin}_b[i])$, and write back $(\mathsf{Bin}_1[i], \ldots, \mathsf{Bin}_b[i]) \leftarrow (W[r{+}1 \bmod b], \ldots, W[r{+}b \bmod b])$.
(5) For every $i \in [1, b]$ read bin Bin_i to the local memory, and compact it locally. If $\mathsf{Bin}_i[1, \ldots, Z/4]$ is not all 0s, or $\mathsf{Bin}_i[3Z/4{+}1, \ldots, Z]$ is not all 1s, then abort and output fail. Otherwise, write Bin_i back.
(6) Run the algorithm recursively on $A[n/4, \ldots, 3n/4]$ with $n/4$ as the number of distinguished elements.

A very useful property of the above algorithm is that, apart from Step 4, the algorithm works completely *in place*; that is, there is no need to copy the elements to a different working array and then write them back (as we do when we have to hide the number of distinguished elements, see Section C).RandCyclicShift is not in place, but is invoked on an array which is of size $1/64$-factor of the original input array. The execution of one of the first 6 recursive calls in the algorithm is depicted below.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
A = [	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	]
Bin_1 = [	1				1				0				1				]
Bin_2 = [		0				0				0				1			]
Bin_3 = [			1				0				0				0		]
Bin_4 = [				1				0				1				1	]
compact each bin locally																	
Bin_1 = [	0				1				1				1				]
Bin_2 = [		0				0				0				1			]
Bin_3 = [			0				0				0				1		]
Bin_4 = [				0				1				1				1	]
A = [	0	0	0	0	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	1	1	1	1	]
recursive call on input																	
A =				[	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	]				
Bin_1 =				[	1		0		1		0		]				
Bin_2 =				[		0		1		0		1	]				
compact each bin locally...																	

The following theorem is proven in Appendix D.1.

Theorem 3.2. *Algorithm 3.1 is an oblivious tight compaction for arrays where exactly half of the elements are distinguished. Its bandwidth is bounded by* $4.2n$ *where* n *is the number of elements in the input array. The number of roundtrips is* $2.1n/Z$*. The error probability is bounded by* $2n/Z \cdot \exp(-Z/256)$*.*

To get some intuition regarding efficiency, without RandCyclicShift, the algorithm just reads each bin, locally compacts it, and writes it back (total – $2n$ bandwidth). The recursive form is therefore $B(n) = 2n + B(n/2)$, i.e., a total of $4n$. In a variant where we have to run RandCyclicShift in each iteration, we have in addition to copy all elements to a working buffer ($2n$); then we read from the working buffer and write them back to their place, shifted (another $2n$). In total, we obtain an additional $4n$ in each iteration, i.e., $B(n) = 6n + B(n/2)$ which results in $12n$. Since we have RandCyclicShift only on an input of size $n/64$, our total bandwidth is bounded by $4n + \frac{1}{64}12n \leq 4.2n$.

Other variants. As mentioned, we give in Appendix C a variant where the number of distinguished elements is secret and should be kept secret from the adversary. In fact, we can have four different variants - depending on whether we have to compact exactly by half or have general compaction, and whether the input is randomly shuffled or not (where if not shuffled we have to run RandCyclicShift in all iterations). We get the following bandwidth for those four variants. We emphasize again that we designed our ORAM such that only the most efficient variant is used.

Algorithm	Randomly Shuffled Input?	Bandwidth
`CompactArrayByHalf`	YES	$4.2n$
`CompactArrayByHalf`	NO	$12n$
General compaction	YES	$10.3n$
General compaction	NO	$18n$

Concrete instantiation of the parameters. As we saw, the error probability can be bounded by $2ne^{-Z/256}$. Since we have many instances of compaction in the overall ORAM structure, we use the error probability of 2^{-100}. This is guaranteed when $Z > 256 \cdot (\log 2n + 100)$. Since we can surely bound n by 2^{70} (allow more than zettabyte...), we must take $Z \geq 43{,}740$ elements. Taking a block size of 32 bytes, the local memory needed is just 1.4MB.

4 INTERSPERSE

Another central building block in [3]'s asymptotically optimal ORAM construction is the ability to intersperse two randomly shuffled arrays into one randomly shuffled array. More formally, we would like to realize the following abstraction:

(1) **Input:** An array $A = A_0 \| A_1$ of size n_0, n_1, respectively. We assume that each element in the input array fits in $O(1)$ memory words.
(2) **Output:** An array B of size $n_0 + n_1$ containing a random permutation of the elements in $A_0 \| A_1$.

Naïvely, obliviously shuffling an array of size n takes $O(n \log n)$ work. However, since the two arrays are already randomly shuffled, we do not have to reshuffle them. Instead, we randomly combine them ("intersperse") and get a randomly shuffled array; this is done in $O(n)$ work. To implement this primitive efficiently, OptORAMa uses the following recipe, originated in PanORAMa [36]:

(1) Sample an auxiliary array Aux uniformly at random among all arrays of size $n_0 + n_1$ with n_0 zeros and n_1 ones:
   (a) Initialize $m_0 = n_0$ and $m_1 = n_1$; For every position $1, 2 \ldots, n_0 + n_1$ flip a random coin that results in head with probability $m_1/(m_0 + m_1)$. If head, write down 1 and decrement m_1; Else, write down 0 and decrement m_0.[2]
(2) Run `Compaction` on Aux while recording all move balls operations.
(3) Reverse route all operations on the input array $A_0 \| A_1$.

We sometimes omit the parameters n_0, n_1 when calling intersperse, and just write $\texttt{intersperse}(A_0 \| A_1)$.

The idea behind the above procedure is that there is no need to generate all $(n_0 + n_1)!$ possible outputs; instead, the algorithm generates only $\binom{n_0+n_1}{n_0}$ outputs as the number of possible Aux arrays. It then moves the balls obliviously according to the Aux array; If $\text{Aux}[i] = 0$, then the ith position in the output array would have an element from A_0 (without replacement). Likewise, if $\text{Aux}[i] = 1$, then in the ith position we would have an element from A_1. The number of possible Aux arrays is $\binom{n_0+n_1}{n_0}$; Given the input assumption, wherein the two input arrays are randomly shuffled, we get that the actual number of possible outputs is $\binom{n_0+n_1}{n_0} \cdot n_0! \cdot n_1! = (n_0 + n_1)!$, as required.

Interspersing two arrays of the same size. In our application, in all invocations of Intersperse, the adversary is aware of n_0, n_1 and it is always the case that $n_0 = n_1$. Thus, we get a highly efficient version by utilizing `CompactArrayByHalf` (Algorithm 3.1) instead of a general compaction algorithm. We denote our algorithm as $\texttt{interspserseTwoHalves}(A_0 \| A_1)$, and it obliviously implements the functionality $\mathcal{F}_{\text{shuffle}}$, which is defined as follows. The functionality receives as input an array A of size n. It chooses a random permutation $\pi : [n] \to [n]$ uniformly at random, and outputs an array B where $B[i] = A[\pi(i)]$.

Claim 4.1. *[3, Claim 6.3] Let* A_0 *and* A_1 *be two arrays of size* n_0 *and* n_1*, respectively. Under the input assumption in which* A_0 *is randomly shuffled, and* A_1 *is randomly shuffled, then the algorithm* $\texttt{intersperse}(A_0 \| A_1, n_0, n_1)$ *obliviously implements* $\mathcal{F}_{\text{shuffle}}$*.*

[2] We implement that in bulk fashion, writing and sampling Z positions at a time.

General intersperse. Plugging in a generic compaction algorithm would result in a generic intersperse procedure, where n_0 might be different from n_1 and they could be secret. This is obtained by using the general case of compaction (see Appendix C)in the above template. We call the resulting algorithm "General `intersperse`". This variant will not be used in our final system but we mention it for completeness, as this is the main variant used in OptORAMa.

Interspersing multiple arrays. We will need to intersperse multiple arrays: $A_1, \ldots, A_k$ of different lengths, rather than just two of the same size. In the final ORAM construction, the different arrays are always the different levels (plus a level that is stored at the client size). That is, we always invoke this intersperse at inputs of size $X, X, 2X, 4X, 8X, \ldots,$. It is assumed that each array is randomly shuffled. We denote this algorithm as $\texttt{intersperseMulti}(A_1, \ldots, A_k)$. The algorithm first shuffles $A_1 \| A_2$, then it shuffles the result with the array A_3, and so on. Note that $|A_1| = |A_2|$, and that $|A_3| = |A_1 \| A_2|$, thus we can use `interspserseTwoHalves` as the underlying procedure, instead of general `intersperse`, which would have increased the associated costs significantly.

Efficiency. We conclude this section by comparing the three algorithms and providing an efficiency analysis. The main observation is that the Aux array works on bits and not blocks. This implies that the size of Aux is n/w blocks (each block is of size w). Thus, compaction of the auxiliary array is performed on an input of size n/w, not n. Therefore, step 1 costs n/w read/write operations, and step 2 is the invocation of `Compaction` of an input of size n/w ($4.2n/w$ bandwidth instead of $4.2n$). The only costly operation is Step 3, which reverse routes all *real* elements as in the compaction, i.e., $4.2n$. In our implementation, we use $w = 32$ bytes. Thus, for `interspserseTwoHalves`, the addition of Steps 1 and 2 is no more than $4.2n/32 = 0.13n$ (note that here we do not even pack 8 elements in bytes but allocate a full byte per bit). The error probability is just the error probability of the underlying compaction. To that, we have to add $4.2n$ of Step 3. The overall cost of the three variants of intersperse is given in Table 1. Note that `intersperseMulti` is just $\log(n/Z)$ invocations of `interspserseTwoHalves`, i.e., its costs is $4.5 \cdot \sum_{i=1}^{\log(n/Z)} 2^i Z \leq 9n$. As for roundtrips, using a similar analysis, we can see that in all three algorithms we can read, process, and write Z elements in one roundtrip which gives us a result of a fraction of $\frac{1}{2Z}$ from the bandwidth. However in the general `intersperse` the number of roundtrips is slightly better than that due to the improvement in the inner compaction (see Appendix C).

	Roundtrips	Bandwidth
`interspserseTwoHalves`	$2.25\frac{n}{Z}$	$4.5n$
General `intersperse`	$4.45\frac{n}{Z}$	$11.3n$
`intersperseMulti`	$4.5\frac{n}{Z}$	$9n$

Table 1: Comparison of the different intersperse algorithms for an input array of size n. `intersperseMulti` works on arrays of size $Z, Z, 2Z, \ldots, n/2$.

5 NON-RECURRENT HASH TABLES

Here, we overview our implementation of each level in the hierarchical ORAM. The main inspiration is the hash table construction from OptORAMa called CombHT, which in turn combines two other hash tables called BigHT and SmallHT. Intuitively, a hash table construction should implement the functionality $\mathcal{F}_{\mathsf{HT}}$, which supports the following commands (see [3, Functionality 4.7] for a precise description of the functionality):

- $\mathcal{F}_{\mathsf{HT}}.\texttt{Build}(A)$: Receives an input array of length n, containing real and dummy elements. A real element is a pair (k, v), of a key $k \in [N]$ and $v \in \{0,1\}^w$. It stores the elements in a private state.
- $\mathcal{F}_{\mathsf{HT}}.\texttt{Lookup}(k)$: where $k \in [N] \cup \{\perp\}$. The algorithm looks in the private state for the key k. If found (i.e., $k \in A$), it returns the associated v and marks it as accessed. Otherwise, return $\perp$. It is guaranteed that the same k will not be queried more than once.
- $\mathcal{F}_{\mathsf{HT}}.\texttt{Extract}()$: Returns a shuffled array of size n, containing all elements in A that were not queried by `Lookup` (those that were queried are replaced with dummies).

Our algorithm for implementing $\mathcal{F}_{\mathsf{HT}}$ uses several of the ideas underlying the construction of OptORAMa, but we make many modifications and optimizations and thereby obtain a practically efficient oblivious hash table construction. Most of our optimizations leverage the fact that we have at our disposal a larger local memory; this fact has a dramatic effect, mostly in the implementation of ShortHT.

Overview of the construction. We briefly overview the construction of [3]. Essentially, to build a hash table, we have to place the elements "obliviously" in some structure. Classically, this was done using oblivious sort, incurring $O(n \log n)$ overhead. However, as pointed out in PanORAMa [36] (and later optimized in [3]), we can do better if we assume that the elements are randomly shuffled and build the structure with linear work. The hash table would have the following components:

(1) **Major bins:** Bins are small hash tables of size Z, i.e., $\text{poly}\log\lambda$. We show how bins are implemented in Section 5.1. As opposed to OptORAMa, here we utilize the fact that the client has large local storage, and we have a much simpler construction.
(2) **Overflow pile:** The construction also uses a secondary hash, called an overflow pile, which we describe in Section 5.2. In this secondary hash, we do not assume that the elements are randomly shuffled, and the cost to build it is $O(n' \cdot \log n')$ where n' is the number of elements in the overflow pile. But, n' will be $n/\log\lambda$, so the cost will be linear in n.
(3) **Level:** We show how to implement the level (i.e., the entire hash table), which contains major bins and overflow piles, in Section 5.3. Here, `Build` starts with the input assumption that all elements are randomly shuffled, and the goal is to build and destruct the hash table with constant overhead.

We present the level in a bottom-up fashion. A reader who is interested in a top-down design can first read section 5.3, then section 5.2, and conclude with section 5.1.

5.1 Implementing Bins

Bin is a small hash table, i.e., it works on $\text{poly}\log\lambda$ size. The main challenge is to design a hash table with linear time for `Build` and

Extract, and constant overhead for Lookup. While we can fit an entire bin into the local memory, we still do not want to spend too much time on every lookup operation. In particular, reading the entire bin per lookup would incur a blowup proportional to the size of a bin, i.e., $\text{poly}\log\lambda$ blowup. However, we are okay with fetching the whole bin upon Build and Extract. This allows us to execute non-oblivious operations during Build and Extract. We implement the bin using a Cuckoo hash with a stash [22]. Given an array B of total size at most Z (in which we expect to see $Z/2$ real elements), each real element (k, v) is placed in either $B_0[\mathsf{PRF}_{\mathsf{sk}}(k, \mathsf{id}, 0)]$, $B_1[\mathsf{PRF}_{\mathsf{sk}}(k, \mathsf{id}, 1)]$, or in a stash, where id is the identity of the bin (see below). Dummy elements are ignored. We denote the stash size by stashBound. We denote this procedure as $(B_0, B_1, \mathsf{stash}) \leftarrow \mathsf{BuildCuckooHash}(B, \mathsf{sk})$. The procedure is non-oblivious but is executed on the local memory on a small input (i.e., a bin). The stash is implemented using a dictionary. For ease of presentation, for now we assume access to an ideal implementation of a dictionary, and so describe the implementation of a bin in the $\mathcal{F}_{\mathsf{dict}}$-hybrid model (see appendix B.1).

Conventions. Looking ahead, each level in the hash table consists of an array of bins, where the number of bins is known in advance. We, therefore, associate each bin with a unique id, consisting of the level of the hash table, whether it is part of the overflow pile or the major bins, and the index of the bin. We assume that each Bin has id hardwired into it. We also use a secret key sk for a PRF for all bins on the same level (in each Build the entire level is being rebuilt, i.e., including all its bins). We, therefore, also assume that Bin internally holds a secret key sk, which is the secret key of the level where the bin resides.

Construction 5.1 (Oblivious Bin Using $\mathcal{F}_{\mathsf{dict}}$)**:**

We assume that the bin has a known id (consisting of the index of the level and the index of the bin within the level) and a PRF key sk (same as the entire level). We also assume a global parameter stashBound.

Bin.Build(B): B is of size at most Z (the expected number of reals is at most $Z/2$).

(1) Run $(B_0, B_1, \mathsf{stash}) \leftarrow \mathsf{BuildCuckooHash}(B, \mathsf{sk})$.
(2) Write B_0, B_1 to the memory, each is of size $Z/2$.
(3) For $i = 1, \ldots,$ stashBound: read the next element in stash as (k, v) (if it does not exist, use $(\perp, \perp)$). Invoke $\mathcal{F}_{\mathsf{dict}}$.Insert$(k, \mathsf{id}, v)$.

Bin.Lookup(k):

(1) If $k \neq \perp$, then let $\beta_0 = \mathsf{PRF}_{\mathsf{sk}}(k, \mathsf{id}, 0)$, $\beta_1 = \mathsf{PRF}_{\mathsf{sk}}(k, \mathsf{id}, 1)$. Otherwise, choose β_0, β_1 uniformly at random from $[Z/2] \times [Z/2]$.
(2) Access $B_0[\beta_0]$, $B_1[\beta_1]$, and if k is found there, then replace it with dummy; Moreover, access the stash using $\mathcal{F}_{\mathsf{dict}}$.PopKey$(k, \mathsf{id})$. Return the found value, or $\perp$ if not found anywhere.

Bin.Extract():

(1) Read B_0, B_1 into the local memory into an array X.
(2) For $i = 1, \ldots,$ stashBound, perform $\mathcal{F}_{\mathsf{dict}}$.PopLabel(id) and append the result to X.
(3) Locally compact X to size Z (while removing some dummy elements that might be introduced due to the stash), and randomly permute it. Return X.

Looking ahead, our construction guarantees that the expected number of reals in the input array is half of the elements. This enables us to allocate the two tables B_0, B_1 of size Z each. Otherwise, we would have needed to allocate larger tables.

Claim 5.2. *Construction 5.1 obliviously implements the $\mathcal{F}_{\mathsf{HT}}$ functionality in the $\mathcal{F}_{\mathsf{dict}}$-hybrid model. The costs are as follows:*

Algorithm	*Bandwidth*	*Round-trips*	*Error Probability*
Build	$2Z$	1	$Z^{-\mathsf{stashBound}}$
Lookup	4	1	*0*
Extract	$2Z$	1	*0*

In addition, each algorithm invokes $\mathcal{F}_{\mathsf{dict}}$ stashBound times.

Proof: For a cuckoo-hash of size n, the probability of having more than s elements in the stash is bounded by $O(n^{-s})$, see [22]. Letting $\log\lambda$ be the size of the stash, we get that the probability that there are more than $\log\lambda$ elements in the stash is bounded by $O(n^{-\log\lambda})$ which is negligible for $n = Z = \text{poly}\log\lambda$. Yet, we leave the parameter of the size of the stash unspecified and write $Z^{-\mathsf{stashBound}}$. Since BuildCuckooHash is run locally, we get that Bin.Build is clearly oblivious. As for lookup, assuming non-recurrent lookups, the simulator can simply access two random elements in B_0, B_1 and simulate a call to $\mathcal{F}_{\mathsf{dict}}$.PopKey. Moreover, for Extract the simulator can just run stashBound calls to $\mathcal{F}_{\mathsf{dict}}$.PopLabel. The rest is local computation. □

5.2 Implementing the Overflow Pile

Our overflow pile also implements $\mathcal{F}_{\mathsf{HT}}$. We utilize the fact that the overflow pile needs to support only $n \cdot \epsilon = n/\log\lambda$ elements. We simply assign each element to a random bin, obliviously place the elements into their respective bins, and then implement each bin similarly to the major bins, i.e., as in Section 5.1. The oblivious bin placement takes $O(n \cdot \log n)$ bandwidth for an input of size n; here, we use the fact that our input is rather small ($n/\log\lambda$), which results in a construction with total bandwidth $n \cdot (\log n/\log\lambda)$.

Main differences from OptORAMa. We highlight a few key optimizations we employ. The main difference from OptORAMa is in the Extract procedure. The Extract in OptORAMa involves extracting each bin ($2n$ bandwidth), (general) tight compaction on the entire resulted array ($10.3 \cdot 2n$ bandwidth), moving all real elements to the front, truncating the array to be of size n, and then running intersperse on the result to shuffle the reals and dummies in the resulted array ($8.9n$ bandwidth). The original cost is, therefore, at least $30n$. We reduce it to $3n$, i.e., a $\times 10$ improvement. Specifically, the above is used to spread the dummy elements (that were introduced during lookup) among the real elements; We provide a different implementation. First, we record with each lookup whether the element was found, and then was marked as removed. In Extract, we utilize the fact that the elements already reside in bins, and we throw dummy elements randomly into the bins, where the number of elements we throw is the number of real elements that were

found during lookup. We then compact each bin locally and shuffle it. This guarantees a random permutation.

Another difference from OptORAMa is that in the Build procedure. We use the lighter primitive $\mathcal{F}_{\text{BinPlacement}}$ instead of a full oblivious sort, as in OptORAMa. This gives us at least a ×2 improvement, where we use the implementation that is due to [2].

Conventions. Just like in Bin, we assume that we use the same secret PRF key sk as in the entire level. Moreover, we assume an implicit id, consisting of the level index in which the overflow pile resides. We present the construction in the $\mathcal{F}_{\text{HT}}$-hybrid model to implement the underlying bins. As such, we do not count for their efficiency and error probability; we will do that when analyzing the entire level.

Construction 5.3 (Overflow Pile Implementation)**:**

overflowPile.Build(X, sk):
Input: Array X of n real elements.
(1) Set $Z = \text{poly}\log\lambda$, $b = 2\lceil n/Z \rceil$.
(2) For every element (k_i, v_i) in X, choose a random bin $d_i = \text{PRF}_{\text{sk}}(k_i, \text{overflow})$ and save it as (k_i, v_i, d_i).
(3) Call $\mathcal{F}_{\text{BinPlacement}}$ (Functionality B.2) on X where each element (k_i, v_i, d_i) is (obliviously) placed in the destination bin d_i. Let Y be the result, i.e., $Y_1, \ldots, Y_b$ bins of size Z each.
(4) Call $\text{ObvB}_\beta \leftarrow \text{Bin.Build}(Y_\beta, \text{sk})$ for $\beta \in [b]$, where the internal id is (id, "overflowPile").
(5) Initialize numFound = 0.
(6) Store $\text{ObvB}_1, \ldots, \text{ObvB}_b$ and the counter numFound.

overflowPile.Lookup(k):
(1) Let $\beta = \text{PRF}_{\text{sk}}(k)$.
(2) Return ObvB_β.Lookup(β). If found a real element, then increment numFound.

overflowPile.Extract():
(1) Sample secret loads by calling Algorithm B.5 $(\ell_1, \ldots, \ell_b) \leftarrow \text{SampleSecretLoads}(\text{numFound}, b)$.
(2) Let $Y = \text{ObvBin}_1.\text{Extract}() \| \ldots \| \text{ObvBin}_b.\text{Extract}()$. Recall that Y is of size $2n$.
(3) Initialize an output array X (which will eventually be of size n).
(4) For $i \in [b]$, read ObvBin_i to the local memory. Let L_i be the number of real elements remaining in ObvBin_i. If $L_i + \ell_i > Z$ then halt and output fail. Compact ObvBin_i to be of size $L_i + \ell_i$, containing the real elements in addition to ℓ_i dummies. Randomly shuffle ObvBin_i and append the result to the array X.
(5) Return X.

We prove the following theorem in Appendix D.2.

Claim 5.4. *Construction 5.3 implements $\mathcal{F}_{\text{HT}}$ assuming the existence of pseudorandom functions and that* Bin *implements $\mathcal{F}_{\text{HT}}$. The bandwidth is as follows:*

Algorithm	*Bandwidth*	*Roundtrips*	*Error*
Build	$4n \log n/Z$	$n/Z \log n/Z$	$\frac{2n}{Z} e^{-Z/6}$
Lookup	4	1	0
Extract	$3n$	$2n/Z$	$\frac{2n}{Z} e^{-Z/6}$

5.3 Level

We are now ready to present our final hash table, i.e., a level in the ORAM hierarchy. Note that in this hash table, we assume that the input is randomly shuffled, and the goal is to provide a Build procedure that requires just $O(n)$ bandwidth.

Overview. According to our input assumption, the input array is randomly shuffled; thus, it seems safe to place the elements in the bins "in the clear", i.e., we place them non-obliviously. More specifically, we sample a secret PRF key sk, and place the element k_i in the bin $\text{PRF}_{\text{sk}}(k_i)$ mod b, where b is the number of bins. If the element is a dummy, we sample a random bin to place it. Whenever we lookup for a real element k', we access the bin $\text{PRF}_{\text{sk}}(k')$. We might either find the element there (if it was indeed in the input array) or not. If we wish to perform a dummy lookup, we choose a random bin and pretend searching in that bin. Observe that loads of the bins are public and known to the adversary.

While this results in a Build procedure that can be simulated (since the input array is randomly shuffled), the joint distribution of Build and Lookup cannot be simulated. Specifically, n lookups of the real elements will result in accessing each bin with exactly the same amount of times as its public load after Build. On the other hand, the result of n dummy lookups (or n elements that are not part of the hash table) will result in a fresh sample from a n balls into b bins distribution.

To solve this, the idea is to sample fresh secret loads $\ell_1, \ldots, \ell_b$, of throwing $n \cdot (1 - \epsilon)$ balls into b bins for some parameter ϵ. We should think about ϵ as $1/\text{poly}\log\lambda$, but we leave it as an explicit parameter for the construction for now. The secret loads $\ell_1, \ldots, \ell_b$ are not revealed to the adversary. With high probability, the secret load of each bin is smaller than its public load, and we move all elements that exceed the secret load to an overflow pile overflowPile. Overall among all bins combined, exactly $n \cdot \epsilon$ elements are moved to the overflow pile. When looking up for an element k, we first search for it in the overflow pile, and then, if not found, we look for it in the major bins according to $\text{PRF}_{\text{sk}}(k)$, as before. If found, we instead visit a random bin. The crux of the security proof in [3] is showing that the access pattern of Lookup looks like a fresh sampling of throwing n balls into b bins and is independent of the loads that were revealed to the adversary during Build.

In our implementation, we use essentially the same construction as of OptORAMa for Build and Lookup, while we mainly change the way overflowPile is implemented, as well as the bins (and some other implementation details). The proof of OptORAMa is modular and works with any implementation of the underlying building blocks, as long as they implement $\mathcal{F}_{\text{HT}}$. Since we have already shown that Bin and the overflow pile implement $\mathcal{F}_{\text{HT}}$, we can directly use the security claim from [3]. Note that all underlying hash tables that required a dictionary and were implemented in the $\mathcal{F}_{\text{dict}}$-hybrid model share the same dictionary. Next, we highlight the key differences between our construction and that of OptORAMa:

- *Building the overflow pile*: In OptORAMa, the construction takes all bins into a designated working buffer and performs general oblivious tight compaction to take the elements that reside at the top of each bin (elements in $[\ell_i, L_i]$, see the description of the construction). This requires running tight compaction on the

entire structure – i.e., $18n$ according to our improved implementation of compaction, which becomes the dominant factor in the construction. We, instead, copy the top $2\epsilon Z$ blocks in each bin to the working array, resulting in an array of total size $2\epsilon n$. Moreover, we apply the optimized `CompactArrayByHalf` procedure. This costs just $4.2 \cdot 2\epsilon n$.
- *Extract*: In OptORAMa the implementation of the extract procedure was concretely expensive. In particular, after extracting each major bin and the overflow pile, they invoked tight compaction to get rid of extra dummy elements and then ran an intersperse procedure to shuffle the reals and the remaining dummies. We, on the other hand, use an idea of PanORAMa [36]: we first reverse the Build procedure by obliviously placing each element into its original bin and locally removing extra dummies. This approach is significantly more efficient in practice. (Roughly $2n + 2\epsilon n \log \frac{\epsilon n}{Z}$ instead of $\geq 18n$ (running general compaction).

In the analysis of level we instantiate the overflow pile with the construction of overflow pile as in the construction of Section 5.2, and the underlying bins as in Section 5.1. The cost, therefore, contains also the build, lookup and extract of these underlying primitives. We still leave $\mathcal{F}_{\mathsf{dict}}$ (which is used in the implementation of Bin) as an ideal implementation. The analysis also counts the total number of invocations of $\mathcal{F}_{\mathsf{dict}}$. We show the full construction and the proof of the following theorem in Appendix D.3.

Theorem 5.5. *Assuming that* PRF *is a pseudorandom function, Construction D.3 obliviously implements* $\mathcal{F}_{\mathsf{HT}}$. *Moreover,*

Algorithm	*Bandwidth*	*Roundtrips*
`Build`	$6n + 14.4\epsilon n + 4\epsilon n \log \frac{\epsilon n}{Z}$	$\left(3n + 6.2\epsilon n + \epsilon n \log \frac{\epsilon n}{Z}\right)/Z$
`Lookup`	8	2
`Extract`	$3n + 7\epsilon n + 4\epsilon n \log \frac{\epsilon n}{Z}$	$\left(2n + 4\epsilon n + \epsilon n \log \frac{\epsilon n}{Z}\right)/Z$

In addition, `Build` *and* `Extract` *perform* $\frac{2(1+\epsilon)n}{Z} \cdot \mathsf{stashBound}$ *calls to* $\mathcal{F}_{\mathsf{dict}}$, *and* `Lookup` *perform* $\mathsf{stashBound}$ *calls to* $\mathcal{F}_{\mathsf{dict}}$. *The error is bounded by* $4n/Z \cdot \exp(-\min\{\ln(Z) \cdot \mathsf{stashBound}, \epsilon^2 Z/16\})$.

5.4 Dictionary Implementation

We finally explain how we implement the dictionary in an oblivious manner (recall the definition of the primitive in appendix B.1). For our purpose, we utilize the fact that we have a rather large local memory and so as long as we do not need to store too many items in the dictionary, we can store it completely in the local memory. When the number of elements needed to be stored grows beyond a certain threshold, we will start delegating the dictionary to the server. A delicate point is that we do not want the event of moving to the external memory to leak information, and so we do this at a fixed time. Specifically, we will move to store the dictionary in the external memory after we have seen more than $Z/2$ different labels. When we move to store the dictionary on the server, we use OptORAMa to get obliviousness [3]. The base, non-oblivious implementation is straightforward: we instantiate two balanced binary search trees (e.g., red-black tree), where the first tree orders elements according to the key k, and the second tree orders elements by the given time t. To distinguish whether sufficiently many different labels have arrived, we maintain a simple counter. This construction incurs a logarithmic cost for each operation. Applying OptORAMa on top of this structure (to get an oblivious analog) incurs another logarithmic factor in overhead.

From an asymptotic point of view, our construction is as good as using the server and invoking OptORAMa from the very beginning. However, our approach of "delaying the usage of OptORAMa" has a significant advantage in practice, as we elaborate next.

We use the dictionary to maintain stashed elements, i.e., elements that we are unable to hash to a main hash table and therefore need to place elsewhere. Concretely, we are using Cuckoo hashing which allows to hash Z elements into two main tables each of size (say) $2Z$ such that with probability $n^{\Omega(-\sigma)}$ more than σ elements need to be stashed (and otherwise they all fit in the main tables). We argue that a stash of size Z, which is poly-logarithmic in the memory size and can be maintained in the local memory, suffices to store the stash of many small Cuckoo hashing tables. It turns out that in our parameter setting this actually suffices.

Concretely, consider $B = Z/\log\lambda$ Cuckoo hashing tables and let X_i be an indicator random variable that is set if at least one element needs to be stashed when hashing the i-th table. Let $X = \sum_{i=1}^{B} X_i$. It holds that $\mathbb{E}[X_i] \leq O(1/Z)$ and so $\mathbb{E}[X] \leq O(B/Z) = O(\log\lambda)$. By Hoeffding's bound, $\Pr[|X - \mathbb{E}[X]| \geq Z/2] \leq 2\exp(-Z^2/(2B^2))$, where the latter is negligible in λ. In other words, we will never need to invoke OptORAMa as long as we hash less than $B \approx Z$ tables. Since we are hashing N/Z and the latter is $\leq B$ in our parameter choices, indeed, it is very likely that we will handle the dictionary completely in memory.

6 ORAM

We are now ready to describe our oblivious RAM construction. At a high level, the construction follows the hierarchical framework of ORAM constructions. Our structure consists of:
- $O(\log N)$ levels. Fix $L = \log N$ and $\ell = \log Z$. Each level $i \in \{\ell, \ldots, L\}$ is a hash table with a capacity of 2^i elements.
- A small structure (of capacity of Z blocks) which we denote as localMem that is stored in the local memory.

Each level is associated with an additional bit, marking whether the level is empty (if the table is not allocated) or full. It is assumed that initially, all levels are marked empty. In addition, there is a global counter, denoted counter, initialized to 0.

The following is proven in Appendix D.4.

Theorem 6.1. *Assuming a secure PRF, Algorithm 6.2 obliviously implements the ORAM functionality (Functionality A.5). Each access consumes (amortized):*
- *Bandwidth:* $\log \frac{N}{Z} \cdot (11(1+\epsilon) + 8\epsilon \log(\epsilon N/Z)) + 9$,
- *Roundtrips:* $\left(\log \frac{N}{Z} \cdot (6(1+\epsilon) + \epsilon \log(\epsilon N/Z)) + 4.5\right)/Z + \log \frac{N}{Z}$.

The error probability for m accesses is bounded by $\frac{4m}{Z} \log \frac{N}{Z} \exp(-\min\{\ln(Z) \cdot \mathsf{stashBound}, \epsilon^2 Z/16, Z/256\})$.

Algorithm 6.2 (ORAM – `Access(op, addr, data)`)**:**

Input: `op` $\in \{$`Read`, `Write`$\}$, `addr` $\in [N]$ and `data` $\in \{0,1\}^w$.
The algorithm:
(1) Initialize `found` = `False` and `result` = $\perp$.
(2) Search for `addr` in the localMem. If found, set `found` = `True`.
(3) For $i \in \{\ell, \ldots, L\}$:

(a) If found = True and T_i is not marked as empty, then perform T_i.Lookup($\perp$).
(b) If found = False and T_i is not marked as empty, then result $\leftarrow T_i$.Lookup(addr). If result $\neq \perp$ then set found = True.

(4) If op = Write then result = data.
(5) Insert result into localMem and increment counter.
(6) If counter mod Z then:
(a) Shuffle localMem. Let i^* be the first table that is marked as empty; if all levels are full, then set $i^* = L$.
(b) Let $X \leftarrow$ localMem$\|T_\ell$.Extract()$\| \ldots \|T_{i^*}$.Extract(). If $i^* = L$ then X = CompactArrayByHalf(X) (i.e., Algorithm 3.1).
(c) Run $Y \leftarrow$ intersperseMulti(X), see section 4.
(d) Run T_i.Build(Y); mark T_i as full and $T_\ell, \ldots, T_{i^*-1}$ as empty.
(7) Return result.

7 EVALUATION AND IMPLEMENTATION

In this section, we evaluate our construction. We first apply the different parameters of our construction and see how they affect the performance by analyzing the terms obtained in the statement of Theorem 6.1. We then discuss our implementation and see how it relates to the theoretical bounds. Lastly, we compare our scheme to that of Path ORAM.

7.1 Our Construction

We let the block size in our construction be 32 bytes. We have several parameters in our construction, which we recall briefly:

(1) The total size of the logical memory, which represents the total capacity of the ORAM. Since a block is of 32 bytes, a total logical memory of size X means that the number of balls, N, is, in fact, the total size divided by X.
(2) Z: The size of bin. We use the same size throughout our building blocks (e.g., bins in bin compaction, intersperse, bin placements, etc.). We remark that Z represents the number of elements. The actual memory consumption is $Z \cdot 32$ bytes.
(3) ϵ: represents the fraction of elements that are moved to the overflow pile in a hash table (see Section 5.3).
(4) stashBound: This is the size of the stash we use for elements that did not fit into their main Cuckoo hash tables, within Bin (see Section 5.1). As we explain in Section 5.3, due to our choice of bin size, Z, relative to N, we can provably guarantee that at most very few elements will go to the stash with very high probability. Concretely, in our experiments, we use stashBound = 9.

On the size of ϵ. Recall that on the overflow pile we run an algorithm with running time proportional to $n \cdot \epsilon \log(n\epsilon/Z)$. Therefore, in theory, we have to use ϵ of roughly $1/\log\lambda \approx 1/\log n$ to actually get rid of the $\log(n\epsilon/Z)$ factor. However, in Table 2, we see that while we make ϵ smaller, we indeed see a smaller overhead, but the error probability increases. We show for various sizes of total logical memory size the overhead when $\epsilon = 1/10, 1/15$ and $\epsilon = 1/20$. As for roundtrips, the rounds that are due to the lookups (i.e., $\log N/Z$) are the dominant factor, and we see almost negligible effect on the build procedure (the term $\left(\log\frac{N}{Z} \cdot (6(1+\epsilon) + \epsilon\log(\epsilon N/Z)) + 4.5\right)/Z$). The table is obtained using the terms in Theorem 6.1.

The error probability. The error probability that is reported in Table 2 is *per access*. The effective error should also consider the size of the memory and the total number of accesses. For instance, an error of 2^{-148} in a memory of 1GB (2^{30}) enables accessing the memory 2^{88} times in total (say 2^{58} accesses to each memory cell), while guaranteeing overall error 2^{-60}. An error of 2^{-131} enables running a program that has $\approx 2^{70}$ accesses to $1GB$ memory while still having error in the regime of 2^{-60}, but 2^{-65} enables running a program that has only 2^5 accesses before reaching the 2^{-60} zone.

On the size of Z. According to Eq. (1), the best we can hope for is an overhead of $11.5 \cdot \log\frac{N}{Z}$. We, therefore, also show for what parameters of Z, ϵ for which this overhead can be reached. The tradeoff is, of course, increasing the local memory. Table 3 shows various parameters for logical memory of 1TB (2^{40}), while aiming to reach overhead of ≈ 11.5. We see that the local memory increases but is still $\leq 0.05\%$ than the size of the logical memory. The table

is again obtained using the terms in Theorem 6.1. Note that an overhead of $1.79KB$ is for obtaining a block of size 32 bytes. This means that our overhead is roughly 56 blocks per block the client accesses.

7.2 Implementation

We implement our solution, and we first compare our results to the theoretical bounds.

Experimental setup. Our experiments were conducted on a computer equipped with an AMD Ryzen 7 5800H processor running at a speed of 3.20 GHz and 32 GB of RAM. Python 3.10 was used in conjunction with the pycryptodomex open-source library for AES implementation, leveraging its ability to interact with the Intel AES-NI hardware extensions. We have two modes for the implementation:

(1) Full: In this mode, we run the ORAM implementation, including all memory allocations and accesses. I.e., we allocate all requested memory in the physical memory, and run the full implementation of our construction.
(2) Counting mode: This mode is used to benchmark the construction in terms of the number of accesses and rounttrips. Specifically, in this mode, we do not allocate the external memory and we only count accesses for each read/write to the external memory. Counting mode enables us to run faster simulations and scale them to larger memory sizes.

In all experiments that only report bandwidth and round-trips, we use the counting mode. For experiments that measure "time", e.g., the last experiment, we run in the full mode. (Of course, the two modes give the same results in terms of rounds and bandwidth.)

Conducting the tests. The test that we report were conducted as follows (1) We initialize the ORAM to allocate N blocks; (2) Access N random blocks from the ORAM for read/write operations. This ensures that the bottom level is accessed at least N times, and is rebuilt at least once.

In Table 4 we show an experiment where we used $Z = 131,220$ and $\epsilon = 1/10$. The table shows the number of blocks being accessed (in average) on the server when accessing an element on the logical memory. We then compare the result of the implementation vs. the expected theoretical analysis.

In the actual implementation of ORAM, we observed slight variations from the theoretical formula, where the difference is at most 13%. The difference is mainly due to rounding the number of levels, the number of bins in each level, and other implementation details. E.g., when throwing the balls into bins in the clear in `Build` of a level (see Construction D.3)we have to read how many elements are in the bin in order to write the element in the correct place in the bin. When the size of the memory is large, the implementation is even slightly better than the mathematical analysis since we used many upper bounds in the mathematical analysis.

Comparison to Path ORAM. We evaluate the performance of our ORAM construction by comparing it to an implementation of the celebrated Path-ORAM construction [44]. To ensure a fair comparison, we implement Path ORAM with the same local memory size (admittedly, further clever optimization for Path ORAM are possible; see [37]). Note that in path ORAM we use the local memory to cut-off the recursion of the position map at an earlier stage.

Table 5 presents the results of the comparison for varying memory sizes. In the experiments where we used $Z = 131,220$ and $\epsilon = 1/10$ for our construction. And for Path ORAM we used a bucket size of 4 as recommended in [44]. Our construction outperforms Path ORAM in terms of bandwidth by a significant factor.

Running time. We have ignored running times so far, and our main metric is bandwidth as it is the main bottleneck. In Table 6 we compare the average time to retrieve an element in our ORAM, compared to Path ORAM. For reference, we also report the average running time of just accessing an element in the memory non-obliviously. When profiling the code, we saw that computing the PRF consumes the most time. Still, as expected, the most dominating factor is reading/writing from the disk.

Various block sizes and round-trips. OptORAMa's optimality is when the block size is small, in particular, $O(\log N)$ bits. Path ORAM improves and gets closer to the asymptotics of OptORAMa when blocks become large ($\Omega(\log^2 N)$). Table 7 compares the two schemes with various memory sizes and with different block sizes. In short, as block sizes grow, the advantage in bandwidth of our construction compared to Path ORAM becomes less dramatic, but our construction is still superior (i.e., our leading constants are smaller). Nevertheless, we note that in terms of round-trips, Path ORAM becomes better and better as block sizes grow while a hierarchical ORAM (ours included) still requires logarithmic number of levels/hash tables and therefore logarithmic round-trips.

Varying local memory size. In some applications, the user might wish to use a larger local memory size, and reduce the round-trips and bandwidth. Table 8 describes different local memory sizes and how they affect the bandwidth and roundtrips.

8 CONCLUSION

Our results confirm that the hierarchical ORAM framework, with appropriately adapted algorithms and optimizations, is practically useful to execute algorithms with data-dependent memory accesses at scale. Concretely, our ORAM design provides a general-purpose oblivious memory that can be used to execute any algorithm in a privacy-preserving manner. Our approach is novel, based on a paradigm that has been largely considered practically irrelevant. Thereby, we demonstrate that full hierarchical ORAMs can be implemented efficiently. For many settings of natural parameters, our construction beats all previously known constructions and provides the only feasible solution. We believe that this advancement represents a significant step towards wider adoption of ORAM as a tool, and for large scale MPC applications. Technically, we design, optimize, and implement several oblivious building blocks that are of independent interest. We provide a thorough theoretical analysis for our construction that aligns with the implementation.

[3]Recall that (by Eq. 1) the total bandwidth is $c \cdot \log N/Z + 9$ (blocks) , where $c = 11(1+\epsilon) + 4\epsilon \log(\epsilon N/Z)$. The table explicitly this term.

ϵ	Error Probability	Bandwidth constant[3]	Bandwidth Total (blocks)	Bandwidth Total (KB)	Roundtrips Lookup	Roundtrips Build	Roundtrips Total
Total memory: 1GB (2^{30} bytes), $Z = 131,220$, local memory: 8MB							
1/5	2^{-148}	18.74	177	5.72	9	0.001	9.001
1/10	2^{-131}	14.47	139	4.48	9	0.001	9.001
1/15	2^{-65}	13.31	129	4.15	9	0.001	9.001
Total memory: 1TB (2^{40} bytes), $Z = 262,440$, local memory: 16MB							
1/5	2^{-156}	25.48	476	15.3	18	0.001	18.001
1/10	2^{-156}	18.07	334	10.75	18	0.001	18.001
1/15	2^{-118}	15.7	291	9.38	18	0.001	18.001
Total memory: 1PB (2^{50} bytes), $Z = 393,660$, local memory: 24MB							
1/5	$\leq 2^{-161}$	33.94	959	30.19	28	0.001	28.001
1/10	2^{-161}	22.07	626	19.73	28	0.001	28.001
1/15	2^{-161}	18.11	516	16.25	28	0.001	28.001

Table 2: Error probability (per access), bandwidth and roundtrips of our construction. In all rows, the blocksize is 32 bytes and stashBound is 9.

Z	Local Memory (MB)	ϵ	Error Probability	Bandwidth constant[3]	Bandwidth Total (blocks)	Bandwidth Total (KB)	Roundtrips
262,440	16.52	1/15	2^{-118}	15.71	292	9.38	18
1,049,760	66.07	1/30	2^{-120}	12.95	216	6.95	16
7,217,100	454.26	1/80	2^{-119}	11.48	169	5.16	14

Table 3: Various values of Z and ϵ for logical memory of 1TB (2^{40}). The blocksize is **32** bytes and **stashBound** is **9**.

Memory Size	Implementation	Mathematical Analysis
1 GB	157	139
10 GB	236	212
100 GB	296	285
1 TB	351	359
10 TB	455	461

Table 4: The number of blocks being accessed (in average) on the server when accessing an element on the logical memory. Comparison of our implementation vs. the expected bandwidth according to the analysis.

Logical Memory Size	Path-ORAM	**Our-construction**
1 GB	28KB	4.9 KB
10 GB	46 KB	7.3 KB
100 GB	68 KB	9.2 KB
1 TB	84 KB	10.9 KB
10 TB	116 KB	14.2 KB

Table 5: Comparison to Path-ORAM. Reporting the average number of bytes accessed when accessing a single element in the logical memory. In both constructions, the local memory is 8MB.

Memory Size	Non-Oblivious μs	Path-ORAM μs	**Our construction** μs
100 MB	6.5	996	378
1 GB	6.8	1374	515

Table 6: Comparison of average running time per access for Path-ORAM, our construction, and non-oblivious access using microseconds as the unit of measurement. The local memory is 8MB. The results are average of 5 runs.

Block-size	Ours Roundtrips	Ours Bandwidth	Path ORAM Roundtrips	Path ORAM Bandwidth
32B	21.7	10.9KB	20	84KB
256B	17.7	67KB	10	135KB
1KB	15.7	229.6KB	6	393KB
4KB	13.7	772KB	4	1.2MB
256KB	7.7	24.4 MB	2	48MB
1MB	6.2	111MB	2	172MB

Table 7: Bandwidth as a function of different blocksize, for an ORAM size of 1TB, with $Z = 131,220$ and $\epsilon = 1/10$.

ACKNOWLEDGEMENTS

We thank Ling Ren, Elaine Shi, and Xiao Wang for useful feedback about Path ORAM.

Asharov and Michelson are sponsored by the Israel Science Foundation (grant No. 2439/20). Asharov is sponsored in addition by JPM

ϵ	Error Probability	Bandwidth			Round-trips		
		constant[4]	Total (blocks)	Total (KB)	Lookup	Build	Total
Total memory: 1TB (2^{40} bytes), $Z = 262,440$, local memory: 16MB							
1/5	2^{-156}	25.48	476	15.3	18	0.001	18.001
1/10	2^{-156}	18.07	334	10.75	18	0.001	18.001
1/15	2^{-118}	15.7	291	9.38	18	0.001	18.001
Total memory: 1TB (2^{40} bytes), $Z = 524,880$, local memory: 33MB							
1/5	2^{-164}	25.14	436	14.03	17	0.0006	17.0006
1/10	2^{-164}	17.67	309	9.95	17	0.0005	17.0005
1/15	2^{-164}	15.44	271	8.7	17	0.0005	17.0005
Total memory: 1TB (2^{40} bytes), $Z = 2,099,520$, local memory: 132MB							
1/5	2^{-183}	23.54	362	11.64	15	0.0001	15.0001
1/10	2^{-183}	16.87	262	8.42	15	0.0001	15.0001
1/15	2^{-183}	14.9	232	748	15	0.0001	15.0001
Total memory: 1TB (2^{40} bytes), $Z = 8,398,080$, local memory: 528MB							
1/5	2^{-201}	21.94	294	9.4	13	0.00002	13.002
1/10	2^{-201}	1607	217	7	13	0.00002	13.002
1/15	2^{-201}	14.38	195	6.29	13	0.00002	13.002
Total memory: 1TB (2^{40} bytes), $Z = 16,796,160$, local memory: 1GB							
1/5	2^{-210}	21.14	262	8.44	12	0.00001	12.00001
1/10	2^{-210}	15.67	197	6.33	12	0.00001	12.00001
1/15	2^{-210}	14.11	178	5.73	12	0.00001	12.00001

Table 8: Bandwidth and round-trips as a function of different local memory sizes.

Faculty Research Award, and by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 891234. Komargodski is the incumbent of the Harry & Abe Sherman Senior Lectureship at the School of Computer Science and Engineering at the Hebrew University, supported in part by an Alon Young Faculty Fellowship, by a grant from the Israel Science Foundation (ISF Grant No. 1774/20), and by a grant from the US-Israel Binational Science Foundation and the US National Science Foundation (BSF-NSF Grant No. 2020643).

REFERENCES

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An $O(n \log n)$ Sorting Network. In *STOC*. 1–9.

[2] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket Oblivious Sort: An Extremely Simple Oblivious Sort. In *3rd Symposium on Simplicity in Algorithms, SOSA 2020*. SIAM, 8–14.

[3] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2023. OptORAMa: Optimal Oblivious RAM. *J. ACM* 70, 1 (2023), 4:1–4:70. https://doi.org/10.1145/3566049

[4] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. 2022. Optimal Oblivious Parallel RAM. In *SODA*. 2459–2521.

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. 2021. Oblivious RAM with Worst-Case Logarithmic Overhead. In *Advances in Cryptology - CRYPTO*. 610–640.

[6] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *CCS*. 837–849.

[7] Elette Boyle, Kai-Min Chung, and Rafael Pass. 2016. Oblivious Parallel RAM and Applications. In *Theory of Cryptography - 13th International Conference, TCC*. 175–204.

[8] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *CCS*. 668–679.

[9] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. 2017. Oblivious Hashing Revisited, and Applications to Asymptotically Efficient ORAM and OPRAM. In *ASIACRYPT*. 660–690.

[10] T.-H. Hubert Chan and Elaine Shi. 2017. Circuit OPRAM: Unifying Statistically and Computationally Secure ORAMs and OPRAMs. In *TCC*. 72–107.

[11] Samuel Dittmer and Rafail Ostrovsky. 2020. Oblivious Tight Compaction In $O(n)$ Time with Smaller Constant. In *SCN*. 253–274.

[12] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*. 523–535.

[13] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. 2012. A secure processor architecture for encrypted computation on untrusted programs. In *STC*. 3–8.

[14] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2015. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In *ASPLOS*. 103–116.

[15] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. 2015. Private database access with he-over-oram architecture. In *CANS*. 172–191.

[16] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *STOC*. 182–194.

[17] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431–473.

[18] Michael T. Goodrich and Michael Mitzenmacher. 2011. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In *ICALP*. 576–587.

[19] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security, CCS*. 513–524.

[20] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. 2016. Breaking Web Applications Built On Top of Encrypted Data. In *CCS*. 1353–1364.

[21] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.

In *NDSS*.
[22] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM J. Comput.* 39, 4 (2009), 1543–1561.
[23] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2012. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*. 143–156.
[24] Kasper Green Larsen and Jesper Buus Nielsen. 2018. Yes, There is an Oblivious RAM Lower Bound!. In *CRYPTO*. 523–542.
[25] Tom Leighton, Yuan Ma, and Torsten Suel. 1995. On Probabilistic Networks for Selection, Merging, and Sorting. In *SPAA* (Santa Barbara, California, USA) *(SPAA '95)*. ACM, New York, NY, USA, 106–118. https://doi.org/10.1145/215399.215429
[26] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. 2019. Can We Overcome the $n \log n$ Barrier for Oblivious Sorting?. In *SODA*.
[27] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS*. 87–101.
[28] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *S&P*. 359–376.
[29] Bill Louden. 1983. Increase Your 100's Storage with 128K from Compuserve. *Portable* 100, 1 (1983), 1.
[30] Steve Lu and Rafail Ostrovsky. 2013. Distributed oblivious RAM for secure two-party computation. In *TCC*. 377–396.
[31] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: practical oblivious computation in a secure processor. In *CCS*. 311–324.
[32] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: practical oblivious computation in a secure processor. In *CCS*. 311–324.
[33] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya V. Lokam, Elaine Shi, and Vipul Goyal. 2017. HOP: Hardware makes Obfuscation Practical. In *NDSS*.
[34] Rafail Ostrovsky. 1990. Efficient Computation on Oblivious RAMs. In *STOC*. ACM, 514–523.
[35] Rafail Ostrovsky and Victor Shoup. 1997. Private Information Storage. In *STOC*. 294–303.
[36] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. 2018. PanORAMa: Oblivious RAM with Logarithmic Overhead. In *FOCS*. 871–882.
[37] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security Symposium*. 415–430.
[38] Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2019. Design and Implementation of the Ascend Secure Processor. *IEEE Trans. Dependable Secur. Comput.* 16, 2 (2019), 204–216.
[39] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. 2013. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*. 571–582.
[40] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT*. 197–214.
[41] Signal. 2022. Technology Deep Dive: Building a Faster ORAM Layer for Enclaves. https://signal.org/blog/building-faster-oram. Accessed: 2023-03-30 14.
[42] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In *S&P*. 253–267.
[43] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. 2012. Towards Practical Oblivious RAM. In *NDSS*.
[44] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*. 299–310.
[45] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*. 850–861.
[46] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. 2014. SCORAM: Oblivious RAM for Secure Computation. In *CCS*. 191–202.
[47] Peter Williams, Radu Sion, and Alin Tomescu. 2012. PrivateFS: A Parallel Oblivious File System. In *CCS*. 977–988.
[48] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptol. ePrint Arch.* (2015), 1153.
[49] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation. In *S&P*. 218–234.
[50] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX*. 707–720.

A PRELIMINARIES (CONT.)

A.1 Concentration Bounds

We define a hypergeometric distribution.

Definition A.1 (Hypergeometric Distribution). *The hypergeometric distribution $H(n, R, m)$ describes the number of red balls drawn in an experiment where m balls are sampled without replacement from a universe containing n balls, R of which are red.*

We state a concentration of measure theorem for hypergeometric distributions (see [**?**], Chapter 7).

Proposition A.2. *Let $H = H(n, R, m)$ be a hypergeometric distribution as in Definition A.1. Let X be a random variable distributed according to H. Letting $\epsilon > 0$ and assuming that $n > 2$, it holds that*

$$\Pr\left[\left|X - \frac{R}{n}m\right| > \epsilon\frac{R}{n}m\right] \leq \exp\left(-\epsilon^2\frac{R^2 m}{n(n-m)}\right).$$

We state also Chernoff's bound:

Proposition A.3 (Chernoff's bound). *Suppose $X_1, \ldots, X_n$ are independent random variables taking values from $\{0, 1\}$. Let X denote their sum and let $\mu = E[X]$ denote the sum's expected value. Then, for every $0 \leq \delta \leq 1$:*

$$\Pr\left[|X - \mu| > \delta \cdot \mu\right] < 2e^{-\delta^2\mu/3}$$

A.2 Oblivious Computation

Oblivious machines. We adopt standard definitions of Oblivious RAM machines and refer the readers to [3] for further reading. We define oblivious simulation of (possibly randomized) functionalities. A RAM is an interactive Turing machine that consists of a memory and a CPU. The memory is denoted as $\mathsf{mem}[N, w]$, and is indexed by the logical address space $[N] = \{1, \ldots, N\}$. We refer to each memory word also as a *block* and we use w to denote the bit-length of each block. In this work, we follow the standard setting where the block size is of $O(\log N)$, and specifically, we use the same word-size as standard computer, namely, 64-bit. The memory supports read/write instructions $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ where $\mathsf{op} \in \{\mathtt{Read}, \mathtt{Write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^w \cup \{\perp\}$. If $\mathsf{op} = \mathtt{Read}$, then $\mathsf{data} = \perp$ and the returned value is $\mathsf{mem}[\mathsf{addr}]$. If $\mathsf{op} = \mathtt{Write}$ then we write $\mathsf{mem}[\mathsf{addr}] = \mathsf{data}$.

Oblivious simulation. We consider machines that interact with the memory via $\mathtt{Read}/\mathtt{Write}$ operations. We consider RAM program that is reactive; namely, it supports several commands and interacts with the memory to implement that commands. Each command has some input and output, and the program stores some state between the different commands in the memory. Let $\mathcal{G}$ be such a (reactive) RAM program; We say that $\mathcal{G}$ is *oblivious* if its access pattern can be simulated by a simulator who receives only the type to the commands but not the inputs of the commands. Moreover, we say that a reactive machine $M_{\mathcal{F}}$ obliviously simulates the reactive machine $\mathcal{F}$ if it has the exact same input/output behavior as the program $\mathcal{F}$, but in addition it is also oblivious; namely, its access pattern can be simulated. Formally:

Definition A.4 (Oblivious simulation of a reactive RAM program). *We say that a reactive machine $M_{\mathcal{F}}$ is an oblivious implementation of the reactive functionality $\mathcal{F}$ if there exists a probabilistic polynomial time (PPT) simulator* Sim*, such that for any non-uniform PPT adversary $\mathcal{A}$, the view of the adversary $\mathcal{A}$ in the following two experiments $Real_{\mathcal{A},M}(\lambda)$ and $Ideal^{\mathcal{F}}_{\mathcal{A},\mathsf{Sim}}(\lambda)$ is computationally-indistinguishable:*

$\underline{Real_{\mathcal{A},M}(\lambda)}$:
Let $(\mathsf{command}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}(1^\lambda)$.
Loop while $\mathsf{command}_i \neq \perp$:
  $\mathsf{out}_i, \mathsf{addr}_i \leftarrow M(1^\lambda, \mathsf{command}_i, \mathsf{inp}_i)$.
  $(\mathsf{command}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_i, \mathsf{addr}_i)$.
$\underline{Ideal^{\mathcal{F}}_{\mathcal{A},\mathsf{Sim}}(\lambda)}$:
Let $(\mathsf{command}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}(1^\lambda)$.
Loop while $\mathsf{command}_i \neq \perp$:
  $\mathsf{out}_i \leftarrow \mathcal{F}(\mathsf{command}_i, \mathsf{inp}_i)$
  $\mathsf{addr}_i \leftarrow \mathsf{Sim}(1^\lambda, \mathsf{command}_i)$.
  $(\mathsf{command}_i, \mathsf{inp}_i) \leftarrow \mathcal{A}(1^\lambda, \mathsf{out}_i, \mathsf{addr}_i)$.

Public inputs, hybrid model, and input assumptions. We sometimes use the notion of "public inputs" (e.g., the size of the array). This is a public parameter to the command, which the simulator also receives as input when simulating the command. We also describe executions in a hybrid model, i.e., a program M in the $\mathcal{F}$-hybrid model, denoted as $M^{\mathcal{F}}$. In that model, the program M can invoke the function $\mathcal{F}$ and the access pattern is not given to the adversary (i.e., the adversary is only notified that $\mathcal{F}$ is invoked). See a formal definition in [3]. We sometimes also have some input assumptions (for instance, hash tables assume that the input is randomly shuffled using a permutation that is not known to the adversary). In that case, we assume that the input assumption X is implemented using some functionality $\mathcal{F}_X$ and analyze the construction in the $\mathcal{F}_X$-hybrid model, without charging the cost of implementing $\mathcal{F}_X$.

Oblivious RAM. An ordinary ORAM is essentially a functionality that implements logical memory. The functionality (given in Functionality A.5) is reactive in which the adversary can choose the next command (i.e., either `Read` or `Write`) as well as the address and data according to the access pattern it has observed so far.

Functionality A.5 (Oblivious RAM)**:**

The functionality is reactive and holds as an internal state – N memory of blocks, each of size w, $X[1, \ldots, N]$. Initially, each $X[\mathtt{addr}] = 0$ for every $\mathtt{addr} \in [N]$.
  Access($\mathtt{op}, \mathtt{addr}, \mathtt{data}$): where $\mathtt{op} \in \{\mathtt{Read}, \mathtt{Write}\}$, $\mathtt{addr} \in [N]$ and $\mathtt{data} \in \{0,1\}^w$:
  (1) If $\mathtt{op} = \mathtt{Read}$ then set $\mathtt{data}^* = X[\mathtt{addr}]$.
  (2) If $\mathtt{op} = \mathtt{Write}$ then set $X[\mathtt{addr}] = \mathtt{data}$ and $\mathtt{data}^* = \mathtt{data}$.
  (3) Output $\mathtt{data}^*$.

Efficiency measures. We use the terminology *bandwidth* to denote the total number of read/write operations of size w the machine uses. We use bandwidth in the amortized sense. We also count round-trips; we allow sending several read/write operations in parallel to the memory (and count each as part of the bandwidth). The round-trips are the number of rounds of accesses to the memory.

B DETAILS ON BUILDING BLOCKS

This section provides full details for the building blocks mentioned in Section 2.

B.1 Oblivious 2-Key Dictionary

An oblivious dictionary is a dynamic data structure that allows inserting of elements (k, ℓ, v) where k is the key (i.e., in $[N]$), ℓ is a label (in $\{0,1\}^w$), and v is the value (in $\{0,1\}^w$). It is assumed that the same key k appears at most once in the dictionary. One can pop an element with the key k by using $\mathsf{PopKey}(k)$– the operation removes the element with the key k. It also supports $\mathsf{PopLabel}(\ell)$, which returns (and removes) an element with the label ℓ, if exists in the dictionary. We formalize the primitive in Functionality B.1. We remark that the following is not our implementation of the primitive but just a description of the input-output relation. Details of our implementation of this primitive are given in Section 5.3.

Functionality B.1 ($\mathcal{F}_{\mathsf{dict}}$ – 2-Key Dictionary Functionality)**:**

Initialization: M is an set.
$\mathcal{F}_{\mathsf{dict}}.\mathsf{Insert}(k, \ell, v)$: on input a key k, label ℓ and value v:
  If $k \neq \perp$, and there is no $(k, \cdot, \cdot)$ in M, then add (k, ℓ, v) to M.
$\mathcal{F}_{\mathsf{dict}}.\mathsf{PopKey}(k, \ell)$: on input a key k and a label ℓ:
  Find an element $(k, \ell', v') \in M$. If $\ell = \ell'$ then return v', and delete (k, ℓ', v') from M. Otherwise, return $\perp$.
$\mathcal{F}_{\mathsf{dict}}.\mathsf{PopLabel}(\ell)$: on input label ℓ:
  Return the lexicographicaly first k such that $(k, \ell, v) \in M$ for some v. If exists, then delete the triple (k, ℓ, v) and return v.

B.2 Oblivious Bin Placement

Oblivious bin placement recieves as input an array X contianing n real elements, each element is marked with some destination bin. The goal of the functionality is to move all elements into their destination bin (obliviously). We assume some parameter Z (which is half of the target bin size). It is assumed that the number of elements assigned to some destination bin is bounded by $2Z$. We show how to implement this functionality based on [2].

Functionality B.2 ($\mathcal{F}_{\mathsf{BinPlacement}}$: Oblivious Bin Placement)**:**

Input: An array X containing n real elements. Each element (k, v, d) is marked with destination bin $d \in [b]$. It is assumed that for every $d \in [b]$, the number of elements assigned to it is bounded by $2Z$.
(1) Initialize an array Y of size $2n$ containing b bins, each is of size $2Z$, denoted as $Y_1, \ldots, Y_b$.
(2) For every $d \in [b]$ assign to Y_d all real elements that are marked with destination d. Append to Y_d dummies such that its size is $2Z$, and shuffle them.
(3) Return Y.

For completeness, we describe the implementation of this functionality.

Algorithm B.3 (Oblivious Bin Placement)**:**

Input: An array X of size n, and a secret PRF key sk.

(1) Set $Z = \text{poly}\log\lambda$. Let $b = 2n/Z$. Interpret X as b bins of size $Z/2$ each, $X_1, \ldots, X_b$.
(2) Define $(\log b+1)$ arrays, each containing b bins of size Z. Denote the jth bin of the ith array $A^{(i)}_j$.[5]
(3) For every real element (k, v) in X, define its key as $\mathsf{PRF}_{\mathsf{sk}}(k) \bmod b$.
(4) Initialize an array Y of total size $2n$. For $i = 1, \ldots, b$, read X_i and append to Y the bin X_i followed by $Z/2$ dummy elements.
(5) Define $A^{(0)} = Y$
(6) For $i = 0, \ldots, \log b - 1$:
   (a) For $j = 0, \ldots, b/2 - 1$:
      (i) Read bins $A^{(i)}_{j'+j}, A^{(i)}_{j'+j+2^i}$ where $j' = \lfloor j/2^i \rfloor \cdot 2^i$ into the local memory.
      (ii) Set B_0 as all real elements in $A^{(i)}_{j'+j} \cup A^{(i)}_{j'+j+2^i}$ where the $(i+1)$-st MSB of the key is 0.
      (iii) Set B_1 as all real elements in $A^{(i)}_{j'+j} \cup A^{(i)}_{j'+j+2^i}$ where the $(i+1)$-st MSB of the key is 1.
      (iv) Pad each B_0, B_1 to be of size Z with dummies.
      (v) Write back B_0 into $A^{(i+1)}_{2j}$ and B_1 into $A^{(i+1)}_{2j+1}$.

Claim B.4. *Algorithm B.3 [[2]] obliviously implements Functionality B.2, with bandwidth $4n\log\frac{n}{Z}$ and $\frac{n}{Z}\log\frac{n}{Z}$ roundtrips; the error probability is bounded by $e^{-Z/6}$.*

Proof: We just go over the efficiency analysis here; security and error probability [2]. The cost of one iteration in terms of bandwidth is $4n$ ($2n$ for reading each bin and $2n$ for writing). since there are $\log\frac{n}{Z}$ iterations the total bandwidth cost is $4n\log\frac{n}{Z}$. As for roundtrips, since we read and write two bins in one roundtrip, the cost of one iteration is $b/2 = n/Z$ roundtrips. And so the total roundtrip cost is $\frac{n}{Z}\log\frac{n}{Z}$. □

B.3 Sample Secret Loads

The algorithm samples loads that remain secret from the adversary, of throwing n' balls into b bins. That is, our goal is to sample from the multinomial distribution with parameters n', b. Explicitly, the output is $(\ell_1, \ldots, \ell_b)$ such that $\sum_{i=1}^{b} \ell_i = n'$ with probability $\binom{n'}{\ell_1,\ldots,\ell_b} \cdot b^{-n'}$.

Denote by $\mathsf{Binomial}(n', p)$ the process of tossing n coins, each with probability p to be head, and counting how many coins resulted in head. The output is $k \le n$ with probability $\binom{n'}{k}p^k(1-p)^{n'-k}$.

Algorithm B.5 ($\mathsf{SampleSecretLoads}(n', b)$)**:**

Input: n' is the total number of balls, and b is the number of bins.
The algorithm:
(1) If $b = 1$ then return n'.
(2) Let $\ell = \mathsf{Binomial}(n', 1/b)$.
(3) Return $(\ell, \mathsf{SampleSecretLoads}(n' - \ell_1, b - 1))$.

[5] It is enough to allocate only two arrays - one is the current array, and the other is a working array. At the end of each iteration, we swap between the current array and the working array.

Claim B.6. *The output of Algorithm B.5 is identically distributed to the multinomial distribution with parameters (n', b).*

Proof: First, if $b = 1$ then the multinomial distribution assigns probability 1 to the output n', and probability 0 to all other possibilities. The algorithm always outputs n'.

For $b > 1$, fix some output loads $(\ell_1, \ldots, \ell_b)$ such that $\sum_{i=1}^{b} \ell_i = n'$. The probability to obtain output $(\ell_1, \ldots, \ell_b)$ by the algorithm is:
(1) First we sample $\mathsf{Binomial}(n', 1/b)$, which is ℓ_1 with probability $\binom{n'}{\ell_1} \cdot (1/b)^{\ell_1} \cdot (1-1/b)^{n'-\ell_1}$.
(2) Then, the algorithm calls recursively to $\mathsf{SampleSecretLoads}(n' - \ell_1, b-1)$. This results with output $(\ell_2, \ldots, \ell_b)$ with probability:

$$\binom{n'-\ell_1}{\ell_2,\ldots,\ell_b} \cdot (b-1)^{n'-\ell_1}.$$

Overall, we obtain output $(\ell_1, \ldots, \ell_b)$ with probability:

$$\binom{n'}{\ell_1} \cdot \left(\frac{1}{b}\right)^{\ell_1} \cdot \left(\frac{b-1}{b}\right)^{n'-\ell_1} \cdot \binom{n'-\ell_1}{\ell_2,\ldots,\ell_b} \cdot (b-1)^{n'-\ell_1} = \binom{n'}{\ell_1,\ldots,\ell_b} \cdot \left(\frac{1}{b}\right)^{n'},$$

as required. □

We describe the implementation as a recursion for the purpose of the proof. In our actual implementation, we run it iteratively and not recursively; This enables us to save roundtrips. By storing locally the index of the current bin, and the number of balls already thrown to previous bins, we can implement the sampling non-interactively and at no cost.

C COMPACTION - THE GENERAL CASE

In the general case, we have to hide the number of distinguished elements. Note that this also implies that we cannot necessarily apply the recursion on $A[1/4n, \ldots, 3/4n]$ since it is not guaranteed, for instance, that the number of 0s in the input array exceeds $1/4n$. Let x be the number of zeros in the entire input, and assume for simplicity that $3n/4 > x > n/2$. We expect that within each bin the first x/b rows are all zeros, and the rest are ones. However, the number of zeros and ones in each bin might be slightly different than x/b. Yet, with high probability the first $x/b - Z/4$ rows are all 0s. Moreover, all the rows in $x/b + Z/4, \ldots, Z$ inside each bin are all 1s. Thus, we have to operate recursively only at the range of $(x/b - Z/4, x/b + Z/4)$. To hide x, we obliviously copy only those elements to some working array and operate there recursively.

Algorithm C.1 (Oblivious Compaction - General Case)**:**

Input: An array A of size n in which x elements are marked as 0 and the rest are marked 1. The number x is given also as input.
Public input: The total number of elements n.
Input assumption: The elements are randomly shuffled. (The input assumption holds only for the first level of the recursion.)
The algorithm:
(1) Run Steps 1-4 of Algorithm 3.1.
(2) Initialize a working array B of size $n/2$ as b bins of size $Z/2$ each, $B = (B_1, \ldots, B_b)$.

(3) Set $(\alpha, \beta) = \begin{cases} (0, Z/2) & \text{if } x/b \in [0, Z/4)\,, \\ (x/b - Z/4, x/b + Z/4) & \text{if } x/b \in [Z/4, 3Z/4)\,, \\ (Z/2, Z) & \text{if } x/b \in [3Z/4, Z]\,. \end{cases}$

(4) For every $i \in [1, b]$ read bin Bin_i to the local memory, compact it locally, and write $\mathsf{Bin}_i[\alpha, \beta]$ into B_i.

(5) Run the algorithm recursively on B with $x - b\alpha$ as the number of distinguished elements.

(6) Read Bin_i; Write locally B_i into $\mathsf{Bin}_i[\alpha, \beta]$ and write back Bin_i.

Theorem C.2. *Algorithm C.1 is an oblivious tight compaction. Its bandwidth is bounded by* $10.3n$ *where* n *is the number of elements in the input array. The number of roundtrips is bounded by* $4.2n/Z$*. The error probability is bounded by* $2n/Z \cdot \exp(-Z/256)$.

Proof: We start with the efficiency analysis. Similarly to the proof of Theorem 3.2, we have that within each recursive call, we read a bin locally, write it back, write half of it also to some working array (this can be done in one roundtrip while taking $2.5Z$ bandwidth). Later, after the recursive call, we read the half from the working array, read the bin again, and write it back. Overall, we read and write each bin twice and read and write its half once, resulting in $5n$ without considering the recursive call. Overall, we get $10n$, ignoring the RandCyclicShift. For RandCyclicShift we pay additional $0.3n$, resulting in $10.3n$ bandwidth. Similar analysis shows that the roundtrip is bounded by $4.2n/Z$.

Obliviousness is clear. As for correctness, a similar analysis to that of Theorem 3.2 shows that the in each bin, the elements in the range $[0, \alpha]$ are all 0s and the elements in $[\beta, Z]$ are all 1s with the same error probability bound as in Theorem 3.2. □

D OMITTED PROOFS

D.1 Proof of Theorem 3.2

Theorem D.1 (Theorem 3.2, restated)**.** *Algorithm 3.1 is an oblivious tight compaction for arrays where exactly half of the elements are distinguished. Its bandwidth is bounded by* $4.2n$ *where* n *is the number of elements in the input array. The number of roundtrips is* $2.1n/Z$*. The error probability is bounded by* $2n/Z \cdot \exp(-Z/256)$.

Proof: We start with the efficiency analysis. When $b = 1$, then the bandwidth is simply reading the bucket and then writing it back. For the general case, we read the bucket exactly once, write it back once, and run recursively on half of the array. Denoting $B(n)$ the bandwidth of the algorithm on an input of length n, we have that $B(n) = B(n/2) + 2n$ for the first six iterations (without RandCyclicShift). This results in a total of $4n$. When applying RandCyclicShift, the cost is $B(n) = B(n/2) + 6n$, i.e., a total of $12n$. We run the first 6 iterations without RandCyclicShift (paying at most $4n$), and then continue with running RandCyclicShift on the input of size $n/64$, paying at most $0.2n$. The total is bounded by $4.2n$. As for roundtrips, we read/write Z elements at a time, and so using a similar analysis we have $2.1n/Z$ roundtrips.

Simulation of RandCyclicShift is trivial - just moving elements according to some random shift. All the access pattern is deterministic: The access pattern is simply reading a bin and writing it back, and which elements reside in each bin is public. The only input-dependent operations happen within the sorting within each bin, which happens in the local memory.

We now turn to correctness. After shuffling the elements (as part of the input assumption), within the first six recursive calls, there is no further randomness involved in the process. As such, whether the algorithm will abort or not is already determined by the input assumption. For the following recursive calls, we do introduce new randomness and so we will need to take it into account to argue that the algorithm succeeds.

First six iterations. Focusing on the first six iterations, we have that:

(1) In the first level of the recursion, there are n elements, $b = \lceil n/Z \rceil$ bins, each of size Z. Denote the bins by $\mathsf{Bin}_1, \ldots, \mathsf{Bin}_b$. It is necessary that for every $j \in [b]$ the bin Bin_j contains at least $Z/4$ elements marked as 0s and $Z/4$ elements marked as 1s, as otherwise processing Bin_j would fail.

(2) In the second level of the recursion, there are $n/2$ elements and $b_2 = (n/2)/Z$ bins, each is of size Z. Denote the jth bin as Bin^2_j for $j \in [b']$. Note that Bin^2_j receives elements from exactly two bins from the previous recursion level, specifically, from Bin_j and Bin_{j+b_2}. If we reach the second level of the recursion, and in addition $\mathsf{Bin}_j \cup \mathsf{Bin}_{j+b_2}$, which is of a total size $2Z$ contains $Z - Z/4$ elements marked as zero, and at least $Z - Z/4$ elements marked as ones, then the processing of $\mathsf{Bin}^{(2)}_j$ is guaranteed to succeed.

(3) More generally, in the kth level of the recursion for k in 1 to 6, there are $n/2^{k-1}$ elements, and $b_k = (n/2^{k-1})/Z$ bins, each is of size Z. The bin Bin^k_j receives its elements from exactly two bins from the previous recursion level, and thus from exactly 2^{k-1} bins from the first recursion level - i.e., $\cup_{j \in S} \mathsf{Bin}_j$, for some set S of cardinality 2^{k-1} (explicitly, $S = \{j, j + b_k, j + 2b_k, \ldots, j + 2^{k-2}b_k\}$). If we reach the kth level of the recursion, and if $\cup_{j \in S} \mathsf{Bin}_j$ (which contain in total $2^{k-1}Z$ elements) contains at least $2^{k-2}Z + Z/4$ elements marked as zero and $2^{k-2}Z + Z/4$ elements marked as one, then processing the jth bin in the kth level of recursion will succeed.

Thus, it is enough to show that for every S of cardinality X it holds that

$$\Pr\left[\#\text{ zeros in } \cup_{j \in S} \mathsf{Bin}_j = \frac{|S|Z}{2} \pm \frac{Z}{4}\right]$$

We'll use proposition A.2:

$$\Pr\left[\left|X - \frac{1}{2}|S|Z\right| > \epsilon \frac{|S|Z}{2}\right] \leq \exp\left(-\epsilon^2 \frac{(n/2)^2 |S|Z}{n(n - |S|Z)}\right)\,.$$

Set $\epsilon = 1/(2|S|)$:

$$\Pr\left[\left|X - \frac{1}{2}|S|Z\right| > \frac{Z}{4}\right] \leq \exp\left(-\frac{(n/2)^2 |S|Z}{4|S|^2 n(n - |S|Z)}\right)\,.$$

Observe that the above probability grows with $|S|$, but since we only apply it with $|S| = 2^5$ (in the worst case), we can bound the above by $e^{(-Z/256)}$. Thus, for sufficiently large Z a high success probability is guaranteed.

The other iterations: Here, we leverage the fact that we perform RandCyclicShift before fixing the content of the bins. Let us fix a bin and let $X := \sum_{i=1}^{Z} X_i$ be the total number of distinguished

elements in that bin, where X_i is the bit associated with the i-th ball in that bin, indicating whether its distinguished or not. Observe that X_i is distributed according to a Bernoulli distribution $B(1, p_i)$, where p_i is the fraction of distinguished elements in the i-th row $(\mathsf{Bin}_1[i], \ldots, \mathsf{Bin}_b[i])$ and the X_i's are independent. Furthermore, $\mathbb{E}[X] = \sum_{i=1}^{Z} p_i$. Thus, by Chernoff's inequality,

$$\Pr\left[|X - \mathbb{E}[X]| > \frac{Z}{4}\right] \leq 2e^{-Z/24}.$$

Thus, except with negligible probability of error, the total number of distinguished elements in each bin must be within the range $(\mathbb{E}[X] - Z/4, \mathbb{E}[X] + Z/4)$.

By a union bound over all n/Z bins in all iterations, the above holds for all bins simultaneously with probability $1-(2n/Z)e^{-Z/256}$. □

D.2 Proof of Claim 5.4

Claim D.2 (Claim 5.4, restated)**.** *Construction 5.3 implements $\mathcal{F}_{\mathsf{HT}}$ assuming the existence of pseudorandom functions and that* Bin *implements $\mathcal{F}_{\mathsf{HT}}$. The bandwidth is as follows:*

Algorithm	*Bandwidth*	*Roundtrips*	*Error*
`Build`	$4n \log n/Z$	$n/Z \log n/Z$	$\frac{2n}{Z} e^{-Z/6}$
`Lookup`	4	1	0
`Extract`	$3n$	$2n/Z$	$\frac{2n}{Z} e^{-Z/6}$

Proof: We start with the efficiency analysis. The input of `Build` is an array of size n. The output of $\mathcal{F}_{\mathsf{BinPlacement}}$ is an array of size $2n$, and the cost is $4n \log(n/Z)$ in bandwidth. `Lookup` requires just a `Lookup` in the respective bin, and `Extract` calls to extract of each bin locally and then writes back roughly half of the bin (with a total of precisely half of what we read). This is therefore $3n$.

Correctness of `Build`. `Build` simply assigns a random bin for each real element. Thus, the expected load of each bin is Z. The load of a bin is distributed according to the binomial distribution $\mathsf{Bin}(n, 1/b)$ (i.e., the probability to have load t is $\binom{n}{t}(1/b)^t(1-1/b)^{n-t}$). According to Chernoff bound, the probability that a bin is assigned with more than Z elements is bounded by $e^{-Z/6}$, which is negligible. As such, the input assumption of Functionality B.2 is preserved, and thus `Build` succeeds with overwhelming probability. Given that `Build` succeeds, it is straightforward that the lookups in a real execution return the same results as the functionality.

Correctness of `Extract()`. We argue that the output of the `Extract()` in the real execution results in a random shuffle of all elements that were not queried in the hash table, combining with numFound dummies. To see that, we can assume that we replace all real elements that are found with dummies, while for simplicity both in the real and ideal experiments, we tag each dummy with some unique identifier (this simplifies counting the different possible outputs). In the real experiment, we "throw" all remaining real elements into the bins (according to the multi-nomial distribution), throw the numFound dummy elements into the b bins, shuffle each bin, and concatenate the bins. Fix a particular load $(n_1, \ldots, n_b)$ with $\sum_{i=1}^{b} n_i = n$. Out of the n balls, there are exactly $\binom{n}{n_1,\ldots,n_b}$ ways to distribute the n elements into the b bins; then, we shuffle each bin, and in the ith bin we have $(n_i!)$ different possible orderings. Overall, the total number of possible outputs with the loads $(n_1, \ldots, n_b)$ is

$$\binom{n}{n_1, \ldots, n_b} \cdot (n_1)! \cdot \ldots (n_b)! = n! \,.$$

That is, conditioned on loads $(n_1, \ldots, n_b)$, all permutations are equally likely. In the functionality, again assuming that all dummy elements replaced real balls that were queried receive some unique identifiers, we also have $n!$ possible outputs and each is equally likely. This shows that the real algorithm provides the same output as the functionality.

Obliviousness. We describe the simulator Sim: In `Build`, it simulates an invocation of $\mathcal{F}_{\mathsf{BinPlacement}}$. For each `Lookup`, it simulates an access to a random bin and performs $\mathcal{F}_{\mathsf{HT}}$.`Lookup` at that bin. In `Extract`, it simulates $\mathcal{F}_{\mathsf{HT}}$.`Extract()` for each one of the b bins; Then, it samples fresh loads of n balls into b bins – i.e., loads $(n_1, \ldots, n_b) \leftarrow \mathsf{Multinomial}(n, b)$, where Multinomial denotes a sampler of the multinomial distribution (see more in Section B.3). Then, for every $i \in [b]$, it simulates reading the next bin (of size $2Z$) and appends to the output array n_i elements.

The fact that the simulator simulates the access pattern of `Build` and `Lookup` is obvious by construction and we argue that the simulation of the access pattern of `Extract` is identical to the access pattern of the real algorithm. Indeed, in the simulation, we leak bin loads $(n_1, \ldots, n_b)$, where $\sum n_i = n$. The real-world algorithm leaks $L_i + \ell_i$ for each bin i. Why are these distributed identically? (1) the distribution of accessed real elements is sampled via a balls-to-bins (multinomial) distribution (because this is how they were thrown, to begin with); and (2) because throwing n balls into b can be done either directly or by first throwing $n - \mathsf{numFound}$ balls to b bins and then numFound balls, for every possible $0 \leq \mathsf{numFound} \leq n$.

Error probability of `Extract()`. The `Extract` operation fails if by throwing the elements according to the secret load we have a bin which overflows. As explained in the previous paragraph, each bin is distributed as a balls-to-bins (multinomial) distribution, and so the probability for the above to occur for a particular bin is at most $e^{-Z/6}$. By a union bound over the number of bins, we get an error probability of $(2n/Z) \cdot e^{-Z/6}$. □

D.3 Construction and Proof of Theorem 5.5

Construction D.3 (Level – implementing $\mathcal{F}_{\mathsf{HT}}$)**:**

`HT.Build`(A, id)**:**
Input: Array A of n elements, containing real elements (of the form (k, v)) and dummy elements, and the identity of the hash table (i.e., the level).
Public input: The total number of elements, n, and a global parameter ϵ.
Input assumption: The elements in A are randomly shuffled.
The algorithm:
(1) Set $Z = \mathrm{poly}\log\lambda$, $b = 2n/Z$.
(2) Initialize an array B of size $2n$, interpreted as b bins $B_1, \ldots, B_b$, each of size Z. Initialize `overflowPile` of total size $2\epsilon n$.

(3) Sample a random PRF key $\mathsf{sk} \leftarrow \{0,1\}^\lambda$.
(4) **Non-oblivious balls into bins:**
  (a) Read the next $2Z$ elements from A.
  (b) Locally assign each element (k_i, v_i) to bin $\beta_i = \mathsf{PRF}_{\mathsf{sk}}(k_i)$. If the element is a dummy, assign a random bin $\beta_i \leftarrow [b]$. Append the ith element to bin B_{β_i}.
(5) **Sample secret loads and move elements to overflow pile:**
  (a) $(\ell_1, \ldots, \ell_b) \leftarrow \mathsf{SampleSecretLoads}(n - n\cdot\epsilon, b)$ (Algorithm B.5).
  (b) For every $i \in [b]$: From each B_i, let L_i be its public load, i.e., $|L_i| = |B_i|$. Read to the local memory the top $2\epsilon Z$ blocks of the bin, i.e., elements in $[L_i - 2\epsilon Z, L_i]$ from B_i. (Actually, since the local memory is of size $2Z$, we can do the above on $1/\epsilon$ bins in parallel.)
  (c) If $\ell_i \le L_i - 2\epsilon Z$, or $\ell_i \ge L_i$ then abort and output fail.
  (d) Move $2\epsilon Z$ elements to `overflowPile` where all elements in $[L_i - 2\epsilon Z, \ell_i]$ are replaced with dummies.
  (e) Write back $2\epsilon Z$ blocks to the top of B_i, where all elements in $[\ell_i, L_i]$ are replaced with dummies.
(6) Initialize $\mathcal{F}_{\mathsf{dict}}$.
(7) **Build main bins:**
  (a) Read each bin B_i to the local memory.
  (b) Run $\mathtt{ObvBin}_i \leftarrow \mathtt{Bin.Build}(B_i)$ (see Section 5.1) with access to $\mathcal{F}_{\mathsf{dict}}$.
(8) **Build overflow pile:**
  (a) Run `CompactArrayByHalf` (Algorithm 3.1) on `overflowPile` moving dummy elements to the end and truncate them.
  (b) Run Construction 5.3, with access to $\mathcal{F}_{\mathsf{dict}}$: $\mathtt{ObvOverflowPile} \leftarrow \mathtt{overflowPile.Build}(\mathtt{overflowPile})$.
(9) $(\mathtt{ObvBin}_1, \ldots, \mathtt{ObvBin}_b, \mathtt{ObvOverflowPile})$ are stored in the memory. The secret key sk is stored locally.

`HT.Lookup`(k)

Input: $k_i \in [N]$.

The algorithm: Using $(\mathtt{ObvBin}_1, \ldots, \mathtt{ObvBin}_b, \mathtt{ObvOverflowPile})$ that are stored in the memory and sk that is stored locally:

(1) Run $\mathtt{result} \leftarrow \mathtt{ObvOverflowPile.Lookup}(k)$.
(2) If result is dummy, then set $\beta \leftarrow \mathsf{PRF}_{\mathsf{sk}}(k)$ and run $\mathtt{ObvBin}_\beta.\mathtt{Lookup}(k)$. If found, increment the counter `numFound` and return the element that was found.
(3) Otherwise, i.e., result is not dummy and is not marked as found in `ObvOverflowPile`, set $\beta \leftarrow [b]$, run $\mathtt{ObvBin}_\beta.\mathtt{Lookup}(\perp)$ and return result.

`HT.Extract`$()$

(1) Let $X = \mathtt{ObvBin}_1.\mathtt{Extract}() \| \ldots \| \mathtt{ObvBin}_b.\mathtt{Extract}() \|$.
(2) Let $Y = \mathtt{ObvOverflowPile.Extract}()$.
(3) For every element (k_i, v_i) in Y, compute its origin bin $d_i = \mathsf{PRF}_{k_i}$.[6] Call $\mathcal{F}_{\mathsf{BinPlacement}}$ (Functionality B.2) on Y where each element is obliviously placed in its origin bin, $Y_1, \ldots, Y_b$.
(4) Locally: for every $i \in [b]$, append $W_i = X_i \| Y_i$, truncate it to size L_i (its origin public load) by removing dummy elements, and shuffle it.
(5) Return $W_1, \ldots, W_b$.

[6] We assume that elements that were found in the overflow pile and replaced with dummies retain their original key so that they can be returned to their origin bins.

Theorem D.4 (Theorem 5.5, restated)**.** *Assuming that* PRF *is a pseudorandom function, Construction D.3 obliviously implement* $\mathcal{F}_{\mathsf{HT}}$. *Moreover,*

Algorithm	*Bandwidth*	*Roundtrips*
`Build`	$6n + 14.4\epsilon n + 4\epsilon n \log \frac{\epsilon n}{Z}$	$\left(3n + 6.2\epsilon n + \epsilon n \log \frac{\epsilon n}{Z}\right)/Z$
`Lookup`	8	2
`Extract`	$3n + 7\epsilon n + 4\epsilon n \log \frac{\epsilon n}{Z}$	$\left(2n + 4\epsilon n + \epsilon n \log \frac{\epsilon n}{Z}\right)/Z$

In addition, `Build` *and* `Extract` *perform* $\frac{2(1+\epsilon)n}{Z} \cdot \mathsf{stashBound}$ *calls to* $\mathcal{F}_{\mathsf{dict}}$, *and* `Lookup` *perform* stashBound *calls to* $\mathcal{F}_{\mathsf{dict}}$. *The error is bounded by* $4n/Z \cdot \exp(-\min\{\ln(Z) \cdot \mathsf{stashBound}, \epsilon^2 Z/16\})$.

The bandwidth cost is as follows:

(1) `Build`: the non-oblivious balls into bins costs $2n$ bandwidth and n/Z roundtrips. Moving the elements into the overflow pile obliviously takes $6\epsilon n$ bandwidth and $2\epsilon n/Z$ roundtrips - reading $2\epsilon Z$ from each bin and writing it to both places - $4\epsilon Z$, but in the same roundtrip, utilizing the $2Z$ local-memory and reading from $1/\epsilon$ different bins in the same roundtrip. We run tight compaction on the overflow pile which costs $4.2 \cdot 2\epsilon n$ bandwidth and $2.1 \cdot 2\epsilon n/Z$ roundtrips. The build of the overflow pile costs $4\epsilon n \log \frac{\epsilon n}{Z}$ bandwidth and $\frac{\epsilon n \log \frac{\epsilon n}{Z}}{Z}$ roundtrips, and the build of the major bins costs $4n$ and $2n/Z$ roundtrips.
(2) `Lookup` we perform lookup in the overflow pile and in a major bin, therefore we only access 4 blocks in two roundtrips, read them and writing them back (looking in the overflow pile in one roundtrip, and in the major bins in the second roundtrip).
(3) `Extract`: The extract of the overflow pile is $3\epsilon n$, and $2\epsilon n/Z$ roundtrips (see Claim 5.4). Extract has the reverse access pattern of `Build`, where now we run extract of the overflow pile and of the major bins instead of building them. This means running oblivious bin placement on the overflow pile ($4\epsilon n \log \frac{\epsilon n}{Z}$ bandwidth, since we read two bins at a time, this requires $\frac{\epsilon n \log \frac{\epsilon n}{Z}}{Z}$ roundtrips), moving the elements from the overflow pile into the major bins (reading $2\epsilon n$ and writing them into the major bins – total of $4\epsilon n$ bandwidth and $2\epsilon n/Z$ roundtrips), extracting all bins – $3n$ bandwidth (reading total $2n$ but writing back just n). However, this still requires $2n/Z$ roundtrips (we have $2n/Z$ bins, in each round we read a bin and write back approximately half of it).

Error probability. We first bound the probability that a bin exceeds Z. The expected number of elements in each bin is $Z/2$. Thus, we can bound this error by $e^{-Z/6}$.

Moreover, we sample throwing n elements into the major bins. The expected value of the load in each bin is $\mu = Z/2$. We now bound the probability that $L_i < \mu - 0.5\epsilon Z/2$. Plugging in $\delta = 0.5\epsilon$ to the following Chernoff bound

$$\Pr[X \le (1-\delta)\mu] \le \exp(-\delta^2\mu/2),$$

we reach an error bound of $\exp(-\epsilon^2 Z/16)$.

Similarly we sample throwing $n - n\cdot\epsilon$ elements into the major bins. The expected load in each bin is $\mu_s = Z/2 - \epsilon Z/2$. We now

bound the probability that $\ell_i > \mu_s + 0.5\epsilon Z/2$. Plugging in $\delta = 0.5\frac{\epsilon}{1-\epsilon}$ to the following Chernoff bound

$$\Pr[X \geq (1+\delta)\mu] \leq \exp(-\delta^2\mu/(2+\delta)),$$

we reach an error bound of $\exp(-Z\epsilon^2/(16-12\epsilon))) \leq \exp(-\epsilon^2 Z/16)$. By a union bound over all bins, we get that the error probability is bounded by $2n/Z \cdot \exp(-\epsilon^2 Z/16)$.

We evaluate the error probability of the `Build` operation. To this end, we apply a union bound to analyze the error probability of each of the underlying operations. Recall that we invoke `overflowPile.Build`, `Bin.Build` (both in the major bins and in the overflow pile), `Compaction` on the overflow pile, Balls into bins, and SampleSecretLoads. Thus, we get error at most

$$\begin{aligned}
&\frac{2\epsilon n}{Z}e^{-Z/6} + (2n(1+\epsilon)/Z)\cdot Z^{-\mathsf{stashBound}} + \frac{2\epsilon n}{Z}\cdot\exp(-Z/256) + \\
&\frac{2n}{Z}e^{-Z/6} + 2n/Z\cdot\exp(-\epsilon^2 Z/16) \\
\leq &\frac{3n}{Z}\cdot Z^{-\mathsf{stashBound}} + \frac{3\epsilon n}{Z}\cdot\exp(-Z/256) + \frac{3n}{Z}\cdot\exp(-\epsilon^2 Z/16) \\
\leq &\frac{3n}{Z}\cdot\exp(-\ln(Z)\cdot\mathsf{stashBound}) + \frac{4n}{Z}\cdot\exp(-\epsilon^2 Z/16) \\
\leq &\frac{4n}{Z}\cdot\exp(-\min\{\ln(Z)\cdot\mathsf{stashBound}, \epsilon^2 Z/16\}).
\end{aligned}$$

D.4 Proof of Theorem 6.1

Theorem D.5 (Theorem 6.1, restated)**.** *Assuming a secure PRF, Algorithm 6.2 obliviously implements the ORAM functionality (Functionality A.5). Each access consumes (amortized):*

- *Bandwidth:* $\log\frac{N}{Z}\cdot(11(1+\epsilon)+8\epsilon\log(\epsilon N/Z))+9$,
- *Roundtrips:* $\left(\log\frac{N}{Z}\cdot(6(1+\epsilon)+\epsilon\log(\epsilon N/Z))+4.5\right)/Z+\log\frac{N}{Z}$.

The error probability for m accesses is bounded by $\frac{4m}{Z}\log\frac{N}{Z}\exp(-\min\{\ln(Z)\cdot\mathsf{stashBound}, \epsilon^2 Z/16, Z/256\})$.

Proof: Our ORAM is the standard hierarchical ORAM from [3] so we refer there for the exact proof. Here, we provide a proof sketch. Assuming that the hash tables are implemented as an ideal $\mathcal{F}_{\mathsf{HT}}$, we obtain that the access pattern is deterministic. Therefore, we can separately consider the access pattern and correctness. It is clear that the access pattern can be simulated - we just call to the underlying primitives at some public schedule. For correctness -we replace all invocations of the underlying primitives with their ideal implementation. To do so, just show that the input assumptions of those primitives are preserved:

(1) In Step 6c, we have that the input array to `intersperseMulti` consists of shuffled arrays since each one of the arrays is an output of `Extract`.
(2) In Step 6d, we run hash table build, which assumes that the input is randomly shuffled.
(3) It is also straightforward that the construction guarantees that no item is searched twice on the same table between different `Builds`.

As such, we can replace the hash table with $\mathcal{F}_{\mathsf{HT}}$, `intersperseMulti` with $\mathcal{F}_{\mathsf{shuffle}}$. At this point, correctness follows from standard hierarchical ORAM (see [17]).

Efficiency analysis. We conclude with the efficiency analysis of the whole ORAM. Each access `Access(op, addr, data)` results with a single lookup in each level and we also rebuild each level once in a while. Precisely, we rebuild level i with the content of all levels up to it every 2^i accesses. The content of all level $\leq i$ is collected by extracting all these levels and then performing intersperseMulti. We perform an analysis in an amortized sense where we consider the total cost within a sequence of n operations.

Lookup: Lookup within each level costs bandwidth 4 and 2 round trips. In average, half of the levels are not built, and thus we get expected $2\log\frac{N}{Z}$ and $\log\frac{N}{Z}$, respectively.

Rebuild: We distinguish between rebuilding the last level and each one of the other levels.

Fix a level $i < L$ and recall that it holds up to 2^i elements. Rebuilding it requires extracting all levels up to $i-1$, which costs $E_{\mathsf{bandw}} = 3\cdot 2^i + 7\epsilon\cdot 2^i + 4\epsilon\cdot 2^i\log\frac{\epsilon\cdot 2^i}{Z}$ in bandwidth and $E_{\mathsf{rndt}} = \left(2\cdot 2^i + 4\epsilon 2^i + \epsilon 2^i\log\frac{\epsilon 2^i}{Z}\right)/Z$ in roundtrips. Then, we shuffle all extracted lists using intersperseMulti, which costs $9\cdot 2^i$ bandwidth and $4.5\cdot 2^i/Z$ roundtrips. Lastly, we rebuild the ith level which costs $B_{\mathsf{bandw}} = 6\cdot 2^i + 14.4\epsilon\cdot 2^i + 4\epsilon\cdot 2^i\log\frac{\epsilon\cdot 2^i}{Z}$ and $B_{\mathsf{rndt}} = (3\cdot 2^i + 6.2\epsilon 2^i + \epsilon 2^i\log\frac{\epsilon 2^i}{Z})/Z$, respectively. Upon m accesses, we rebuild level i around $m/2^{i+1}$ times. Summing over all levels $i < L = \log\frac{N}{Z}$ we get overall bandwidth:

$$\begin{aligned}
\sum_{i=\ell}^{L}\frac{m}{2^{i+1}}\cdot\left(18\cdot 2^i + 21.4\epsilon 2^i + 8\epsilon 2^i\log\frac{\epsilon\cdot 2^i}{Z}\right) \leq \\
m\cdot\sum_{i=\ell}^{L}\left(9 + 10.7\epsilon + 4\epsilon\log\frac{\epsilon\cdot N}{Z}\right) \leq \\
m\log\frac{N}{Z}\cdot(9 + 10.7\epsilon + 4\epsilon\log(\epsilon N/Z)).
\end{aligned}$$

A similar calculation gives the following bound to roundtrips:

$$\frac{m}{Z}\log\frac{N}{Z}\cdot(4.75 + 5.1\epsilon + \epsilon\log(\epsilon N/Z))$$

In every $2N$ accesses, we also rebuild level L from its own content. We already counted the cost of extracting all levels and rebuilding the level L, we need in addition to count for interspersing all levels with level L (additional $9N$) and compacting level L from $2N$ into N. This adds at most $18N$. Since this happens once in $2N$ accesses, it contributes additional 9 blocks per access (and $4.5/Z$ roundtrips). Summing it all together, the amortized cost per access over all levels, including all the lookups, is bounded by:

$$\mathsf{Bandwidth} = \log\frac{N}{Z}\cdot(11(1+\epsilon) + 4\epsilon\log(\epsilon N/Z)) + 9\,. \tag{1}$$

The error probability of a level i is $\frac{4\cdot 2^i}{Z}\cdot\exp(-\min\{\ln(Z)\cdot\mathsf{stashBound}, \epsilon^2 Z/16\})$. Let $E_1 := \min\{\ln(Z)\cdot\mathsf{stashBound}, \epsilon^2 Z/16\}$ and $E_2 = Z/256$, and let $E_{\mathsf{HT}}(2^i)$ denote the error associated with level of size 2^i, and $E_{\mathtt{intersperseMulti}}(2^i)$ be the error for `intersperseMulti` for input of size 2^i. Over the course of m accesses, the total error

probability is:

$$\begin{aligned} ERR &= \sum_{i=\ell}^{L} \frac{m}{2^{i+1}} \left(E_{\mathsf{HT}}(2^i) + E_{\mathtt{intersperseMulti}}(2^i) \right) \\ &= \sum_{i=\ell}^{L} \frac{m}{2^{i+1}} \left(\frac{2^{i+2}}{Z} \exp(-E_1) + \frac{2^{i+1}}{Z} \exp(-E_2) \right) \\ &\leq \frac{2m}{Z} \log \frac{N}{Z} \left(\exp(-E_1) + \exp(-E_2) \right) \\ &\leq \frac{4m}{Z} \log \frac{N}{Z} \exp(-\min\{E_1, E_2\}) \ . \end{aligned}$$

□

E OVERVIEW OF OPTORAMA

OptRAM [3] follows the hierarchical paradigm established by Goldreich and Ostrovsky [16, 17]. An ORAM scheme in the hierarchical paradigm can be viewed as a technique to reduce the task of constructing ORAM to constructing an oblivious hash table. Specifically, a hierarchical ORAM typically consists of $\log_2 N + 1$ levels numbered $0, 1, 2, \ldots$. Each level i is an *oblivious hash table* that can contain at most 2^i elements. An oblivious hash table is a data structure that supports the following operations:

- `Build` takes an input array containing (key, value) pairs and creates the data structure (we also say a pair is an element, a block, or an item);
- `Lookup` receives a key k, and returns the value corresponding to the key k contained in the data structure, or returns $\perp$ if not found or if the key looked up is dummy (denoted $\perp$).
- `Extract` is called when the data structure is destructed, and returns a list of all the elements in the data structure that were never looked-up.

Optimizing oblivious hash table construction. The original oblivious hash table implementation suggested by Goldreich and Ostrovsky [16, 17] is slow and takes $O(n \log n)$ time to build for an input array of size n. This would result in a non-optimal ORAM scheme of at least $\Omega(\log^2 N)$. Asharov et al. (following Patel et al. [36]) showed an oblivious hash table with $O(n)$ build time $O(1)$ lookup overhead, in addition to a scan of a stash where few elements may end up at. The stashes of all hash tables are later merged into one large table so the cost of looking up the stashes does not affect the overall complexity. The security of their construction relies on an input assumption: the input array of `Build` must be randomly shuffled. As such, to benefit from the reduction of the extra $\log n$-factor in `Build`, the construction must guarantee that whenever a hash table is being built, the input is randomly shuffled.

Technically, the hash table construction is obtained by hashing elements into a set of smaller hash tables, each supporting poly-logarithmically many elements in the input. The smaller hash tables are implemented, at a high level, by optimizing oblivious Cuckoo hash to obtain optimal overhead.

The ORAM, at a high level. As mentioned, for a logical memory of N blocks, the ORAM consists of a hierarchy of hash tables, henceforth denoted $T_1, \ldots, T_L$ where $L = \log N$. Each T_i stores 2^i memory blocks. We refer to table T_i as the i-th level. When receiving an access request to `Read`/`Write` some logical memory address `addr`, the ORAM proceeds as follows:

- **Read phase.** Access each level $T_1, \ldots, T_L$ in order and perform `Lookup` for `addr`. If the item is found in some level T_i, then when accessing all levels $T_{i+1}, \ldots, T_L$ look for dummy.
- **Write back.** If this operation is `Read`, then store the found data in the read phase and write back the data value to T_1. If this operation is `Write`, then ignore the associated data found in the read phase and write the value provided in the access instruction in T_1.
- **Rebuild:** Every 2^ℓ for $\ell \geq 1$ accesses, merge all levels $\{T_j\}_{1 \leq j \leq \ell}$ into level ℓ.

Since we use the efficient oblivious hash table construction mentioned above, we know that the `Read`/`Write` operations will consume only logarithmic overhead. The remaining challenge is to implement the `Rebuild` procedure efficiently. To this end, they again utilize the fact that the (remaining) elements in each level are randomly shuffled. This allows them to perform the merge in essentially linear time by running a procedure they call *intersperse*. Intersperse merges two randomly shuffled arrays into a joint shuffled array in linear time. This allows to guarantee that the input assumption is maintained when coming to build a hash table.