Distribution-Aware Oblivious RAM*

Marilyn George, Tarik Moataz, and Andrew Park

MongoDB Research, 1633 Broadway, New York City, NY 10019, USA

Abstract. Oblivious RAM (ORAM) is a cryptographic primitive that enables a trusted client to hide its memory access patterns from an untrusted memory. The standard ORAM definition enforces worst-case security and correctness over all possible access sequences, which captures the fact that ORAM security holds for any client access distribution, skewed or not. However, in many real-world settings, access sequences are known to follow skewed, power-law distributions, whether they correspond to data structure accesses, database queries, or filesystem accesses. In this work, we introduce *distribution-aware oblivious RAM*. Its security guarantees match those of standard ORAM, except that it also reveals the access distribution $\mathcal{D}$; correctness, however, is only guaranteed when accesses follow $\mathcal{D}$. Our core result is that, when accesses are drawn from a distribution, the communication overhead of our constructions depends on the expected number of distinct addresses accessed per fixed-size epoch, which for naturally skewed workloads can be orders of magnitude smaller than the RAM size N.

We give three constructions. The first, DAS, builds on the classic square-root ORAM and shows that the cache size can be decoupled from the epoch length, allowing it to be set to $O(\mu + \log k)$, where μ is the expected number of distinct accesses per epoch and k is the security parameter. Our main construction, DAT, adapts tree-based ORAM to the distribution-aware setting via a hybrid tree whose height is tuned to the access distribution; we show that the height is $O(\log(\mu + \log k))$ and that the communication complexity is $O(\log(\mu + \log k) + \log k / \log\log k)$, compared to $O(\log N)$ for non-recursive Path ORAM. Finally, DAB is a black-box construction that builds on any standard ORAM, achieving sublinear communication at the cost of a super-logarithmic delay, under certain assumptions.

We evaluate all of our constructions and show that, for a Zipf distribution, the per-access communication and computation costs can be up to 6.1× and 25× smaller, respectively, than those of Path ORAM, yielding significant efficiency gains.

1 Introduction

Oblivious RAM (ORAM), first introduced by Goldreich and Ostrovsky [23], is a cryptographic primitive that hides a client's access patterns to a memory held

* © IACR 2026. This article is the final version submitted by the author(s) to the IACR and to Springer-Verlag on 14 Sep 2026.

by an untrusted server. ORAM effectively hides all information, revealing only the number of words (or blocks) in the memory and the length of the RAM program (number of accesses) being executed. Since its introduction, ORAM has been a fundamental building block to many applications including outsourced storage [57], multiparty computation [24], trusted hardware [17], searchable symmetric encryption (SSE) [56], structured encryption (STE) [30], and encrypted databases [28].

The logarithmic barrier. Informally, the security of ORAM requires that all access sequences of the same length be identically distributed from the view of the untrusted server. This security guarantee imposes a logarithmic lower bound in the size of the RAM on the communication overhead of oblivious RAM, as shown in [23]. Since the original lower bound was proven in the balls-and-bins model, several works have explored alternative variants in the hope of circumventing the logarithmic barrier, including extending the balls-and-bins model to the cell probe model [38], moving from the single-server setting to the multi-server setting [39], relaxing ORAM security to differential privacy [47], and restricting the adversary's view to as little as observing only two accesses in a given access sequence [48]. All of these works showed that the logarithmic lower bound still holds. This lower bound also matches the asymptotic communication overhead of existing constructions such as Path ORAM [54] for certain block-size regimes, and was more recently matched for any block-size regime in OptORAMa [2]. Since we now have matching upper and lower bounds, it appears that we have reached the optimal communication overhead for oblivious RAM, and this holds under many relaxations of the model. However, one setting that has not been investigated is the one where accesses are sampled from a known distribution, which is the focus of this work.

Access distributions. The definition of ORAM enforces worst-case security and correctness over *all possible* access sequences. This, in essence, captures the fact that ORAM security holds for any client access distribution, whether skewed or not. However, in many real-world settings, access sequences are known to follow skewed, power-law distributions such as the Zipf distribution [6, 36, 42]. Skewed distributions arise in a wide range of applications, including query workloads to relational and non-relational databases, file accesses in operating systems, and web traffic. This is also well reflected in industry-standard benchmarks, including the `YCSB` key-value benchmark [11], the online transaction processing (OLTP) benchmarks `TPC-C` and `TPC-E` [41, 10], and the storage benchmark `SPC-1` [58], to name a few.

Given the prevalence of skewed distributions in the plaintext world, the study of distribution-aware data structures can be traced back to the 1950s, when Huffman [26] introduced a prefix code that exploits symbol frequencies: the underlying structure is a binary tree in which frequent symbols are placed along shorter paths, minimizing the expected codeword length. The first explicit distribution-aware search structure was Knuth's optimal binary search tree [35], which minimizes expected search time given known access probabilities. The re-

sulting tree reduces it from logarithmic in the size of the tree to logarithmic in the inverse frequency of the entry. Many follow-up works extended this idea to balanced trees, dynamic self-adjusting trees, and the study of the optimality of such self-adjustments [43, 52, 14, 13].

Given this rich body of work in the plaintext setting, one would naturally wonder whether the distribution-aware paradigm can be extended to the encrypted setting; the answer is yes. To the best of our knowledge, two areas have explored constructions that take query distributions into account: structured encryption where George, Kamara, Moataz and Espiritu [22] propose a distribution-aware dictionary encryption scheme with the same security guarantees as an oblivious dictionary (with the exception of revealing the access distribution); and, more recently, private information retrieval (PIR) where Lehmkuhl, Henzinger and Corrigan-Gibbs [40] introduce a distribution-aware PIR construction that maintains the same cryptographic privacy as standard PIR while achieving better overhead when accesses are skewed. Although neither line of work circumvents the corresponding lower bounds, the resulting constructions achieve meaningful practical gains in certain settings. It is therefore a natural next step to extend the distribution-aware theme to oblivious RAM, leading us to the following question:

> *Can we improve the efficiency of oblivious RAM for skewed access distributions?*

We answer this question in the affirmative. When accesses are drawn from a skewed distribution, the communication overhead of our constructions becomes a function of the expected number of distinct addresses accessed per sequence, rather than of the RAM size alone. Our contributions are as follows.

1.1 Our Contributions

Definitions for distribution-aware ORAM. We define the notion of distribution-aware oblivious RAM, which uses a fixed access distribution $\mathcal{D}$ at setup. Notably, the security definition still allows the adversary to choose any sequence of accesses, not just those sampled from $\mathcal{D}$. In addition to the RAM size and the number of accesses, a distribution-aware ORAM also reveals the distribution $\mathcal{D}$, which we capture in our definition by providing $\mathcal{D}$ as input to the simulator. We also define two notions of correctness for the distribution-aware setting. The first, which we call *distribution-aware correctness*, guarantees that the scheme returns a correct response for each access sampled from $\mathcal{D}$ in an online setting: that is, for each access, the ORAM outputs the correct response before executing the next access operation. The second, which we call *distribution-aware correctness with n-length delay*, guarantees that the scheme outputs correct responses after at most $n \in \mathbb{N}_{>0}$ subsequent operations when accesses are sampled from $\mathcal{D}$. The former is the standard, more realistic notion of correctness and is achieved by our concrete constructions; the latter is achieved by our black-box design.

Distribution-aware square-root ORAM. We begin with DAS, a simpler construction that introduces a core idea used in our main results. DAS builds on the classic square-root oblivious RAM of Goldreich and Ostrovsky, making small but important changes to leverage the access distribution $\mathcal{D}$. Recall that square-root ORAM uses a cache that, once full, triggers an expensive reinitialization step (an oblivious sort) whose cost is amortized over the accesses. The cache holds λ entries and supports exactly λ accesses before reinitialization, so the cache size and the epoch length are tied together. In DAS, our first modification is to decouple these two parameters: we set the cache size λ to be the expected number of distinct addresses accessed during an epoch of length t, rather than to t itself. For skewed distributions, this expectation is much smaller than t, allowing a smaller cache without sacrificing correctness. We provide more technical intuition, and describe our other modifications, in Section 1.2. From an efficiency standpoint, given a distribution $\mathcal{D} = \{p_1, \ldots, p_N\}$, DAS achieves amortized communication complexity

$$O\left(\mu + \log k + \frac{(N+t)\log(N+t)}{t}\right),$$

where k is the security parameter and $\mu = \sum_{i=1}^{N}(1-(1-p_i)^t)$ is the expected number of distinct addresses accessed independently during an epoch of length t. This compares to $O(\sqrt{N}\log N)$ for classic square-root ORAM: the cache size drops from $\sqrt{N}$ to $O(\mu+\log k)$. Concretely, for a Zipf distribution with parameter $s > 1$, the epoch length becomes linear in N (rather than $\sqrt{N}$) and the cache size becomes

$$O\left(\left(\frac{N}{H_{N,s}}\right)^{1/s} + \log k\right),$$

where $H_{N,s}$ is the generalized harmonic number. For highly skewed Zipf distributions, e.g., $s > 2$, this is substantially smaller than $\sqrt{N}$. Throughout the paper, our correctness and efficiency analyses assume that accesses are sampled independently from $\mathcal{D}$.

Distribution-aware tree-based ORAM. Our main construction DAT builds on a new hybrid tree structure in which the top of the tree is a binary tree with k-ary leaves. DAT conceptually deviates from standard tree-based ORAMs in that the height of the tree is no longer a function of the size of the RAM, but rather a function of the access distribution. Intuitively, the upper tree (excluding the leaf level) is only large enough to hold the frequently-accessed addresses, while the leaves hold the less frequently-accessed addresses. Realizing this intuition involves several technical challenges, including the choice of the upper tree height, the size of internal and leaf nodes, the eviction process, and the choice of the stash size, which we discuss in Section 1.2. We show that DAT, in its non-recursive form, is distribution-aware correct and achieves communication complexity

$$O\left(\log(\mu + \log k) + \frac{\log k}{\log\log k}\right),$$

compared to the $O(\log N)$ communication complexity of non-recursive Path ORAM. In essence, the communication cost now depends on μ, the expected number of distinct addresses accessed during an epoch of length N.[1] This comes at the cost of additional storage:

$$O\left(N \cdot \frac{\log k}{\log \log k}\right),$$

compared to $O(N)$ for Path ORAM. For a Zipf distribution with parameter $s > 1$, we have $\mu = O\big((N/H_{N,s})^{1/s}\big)$, which gives DAT communication complexity scaling roughly as $(1/s) \cdot \log(N/H_{N,s})$ instead of $\log N$; thus, while the improvement is not asymptotic, it is substantial in concrete terms. As a concrete reference, for $N = 2^{20}$ and Zipf with parameter $s = 2$, we have $\mu \approx 1{,}415$, which is almost three orders of magnitude smaller than N.

Distribution-aware black-box ORAM. Contrary to DAS and DAT, which are two concrete instantiations, DAB is a black-box distribution-aware oblivious RAM scheme that builds on an underlying ORAM scheme Σ_{RAM}. Our approach is to use the access probabilities to partition the input RAM into two smaller RAMs: (1) RAM^h, containing the addresses with high access probabilities, and (2) RAM^ℓ, containing the remaining addresses, with low access probabilities. On every client access, we access RAM^h and maintain a client-side queue for any addresses that are not found there. Once every n accesses, we perform λ accesses to the low-probability RAM^ℓ to flush the queue. As a consequence, the responses to some low-probability accesses are delayed by up to n operations due to the queueing.

For this to work, there are several questions to resolve, including how to set the partition point, as well as a more subtle question regarding certain properties we require from the underlying ORAM; we discuss these further in Section 1.2. In terms of efficiency, assuming that the underlying ORAM is the recursive variant of Path ORAM and the distribution is Zipf with parameter $s > 1$, the communication complexity of DAB is

$$O\left(\log^2\left(\frac{\log^2 N}{H_{N,s}}\right)\right),$$

and the delay is

$$n = \omega\left(\frac{\log k \cdot \log^2 N}{\log \log N}\right).$$

While this result may appear to circumvent the lower bound, the correctness relaxation introduces delay, and we can still observe a logarithmic dependency (now shifted to the delay term rather than the per-access communication cost).

[1] For the recursive variant of DAT, as is standard for all tree-based ORAMs, there is an additional cost of $O(\log^3 N/B)$, where B is the main RAM block size, as well as an increase in round complexity from $O(1)$ to $O(\log N)$. The advantage is that the client state is reduced to $\omega(\log k)$.

Evaluation. We implemented a simulation of our tree-based protocols, DAT and DAB, as well as Path ORAM [54], in C++. Throughout our experiments, we consider the Zipf distribution with parameter $s \in \{1.5, 2, 3\}$ and a RAM size of $N = 2^{17}$. In terms of computation, in the non-recursive setting, DAT achieves a 16–25× speedup over Path ORAM, and DAB instantiated with Path ORAM achieves a 3.7–8.8× speedup. In the recursive setting, DAT achieves a 4.4–4.7× speedup over recursive Path ORAM, while DAB instantiated with Path ORAM achieves a 4.7–11× speedup. In terms of communication, DAT achieves comparable or lower communication, with up to a 1.46× improvement, and DAB delivers up to a 4.3× improvement over Path ORAM. In the recursive setting, DAT has comparable or slightly smaller communication, with an improvement of up to 1.28× over recursive Path ORAM, while DAB instantiated with the recursive variant of Path ORAM reduces bandwidth by up to 6.1×.

1.2 Technical Overview

We give a technical overview of our three constructions. We begin with DAS, a simple modification of the classic square-root ORAM that already captures one of our key ideas; we then describe DAT, our main tree-based construction; and finally DAB, our black-box construction. Each subsection explains the design intuition and the technical challenges encountered.

1.2.1 Distribution-aware Square-Root ORAM

DAS is nearly identical to the square-root ORAM construction [23], except that we now separate the cache size λ from the epoch length t (i.e., the number of dummies). The technical challenges here are twofold.[2]

Correctness challenge. Given an access distribution $\mathcal{D}$, we need to ensure that the cache does not exceed λ with all but negligible probability when sampling t accesses independently. Accesses can be viewed through the lens of the non-uniform balls-and-bins problem, where the bins represent the addresses in the address space and their weights represent the probability masses. For instance, if an address a has a high probability of being sampled, then thrown balls have a high chance of landing in the bin corresponding to a. The size of the cache can then be viewed as the number of non-empty bins in this experiment. In particular, we show that it suffices to set $\lambda = \Theta(\mu + \log k)$, where μ is the expected number of non-empty bins after throwing t balls. The more skewed the distribution, the smaller λ.

Security challenge. A subtle issue arises when the cache reaches capacity (i.e., it contains exactly λ real blocks) while the number of performed accesses is strictly below the epoch length t. In the standard square-root ORAM, if the

[2] For readers unfamiliar with the construction in [23], we provide an overview in Section 5.

client accesses a new address that does not appear in the cache, the client retrieves it from the main structure and adds it to the cache; this works because the cache is guaranteed to have as many available slots as the number of accesses in the epoch. This invariant no longer holds in DAS, since the cache size, for skewed distributions, is significantly smaller than the epoch length. One possible workaround would be to still access this address from the main structure but not add it to the cache; this way, the cache remains at capacity λ and the client retrieves the intended block. However, this approach would violate obliviousness: if the same address is later re-accessed, the client would not find it in the cache and would re-access the same location in the main structure, thereby revealing that the address was accessed twice. To address this, we instead access a dummy in place of the intended address. Accessing a dummy does not extend the cache size, but it results in an incorrect access. If the client deviates from $\mathcal{D}$, obliviousness is still preserved (since the access pattern is independent of the deviation), but correctness is no longer guaranteed.

1.2.2 Distribution-Aware Tree-based ORAM

DAT is based on a hybrid tree structure in which the top of the tree is a full binary tree with k-ary leaves. At a high level, the fundamental technique DAT relies on is tying the height of the upper tree (the full binary tree) to the underlying access distribution, so that the more skewed the distribution, the smaller the height of this tree and the more efficient the construction. Of course, for this to work, there are several technical challenges to solve, which we discuss below; but let us start with the intuition behind the design.

Consider a full binary tree, as is standard in the majority of tree-based ORAMs. The main design principle of these structures (starting with the construction of Shi, Chan, Stefanov, and Li [51]) is that recent accesses are placed at the root of the tree; but as more accesses are made, the root will overflow. To address this, they introduce the elegant notion of *eviction*, which in principle, and regardless of its specific form, consists of pushing real blocks as far down the tree as possible. This eviction creates more empty space at the top of the tree, and real blocks percolate toward the leaves as additional accesses occur. In standard tree-based ORAM constructions, however, since the structure is designed to support any access distribution, the occupancy of the lower part of the tree is roughly similar to that of the top as real blocks move toward the leaves. Consequently, in most tree-based ORAM stash analyses, the occupancy of every internal node (with the leaves sometimes treated separately) is the same; we refer the reader to the node occupancy analysis in [54]. One can view this uniform occupancy across the tree as an invariant resulting from the unpredictability of the accesses; see the leftmost tree in Figure 1 for an illustration.

In contrast, in DAT, this behavior is different and, more precisely, depends on the access distribution. For the sake of example, suppose that the access distribution $\mathcal{D}$ is extremely skewed, such that only three addresses $(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3)$ have a high probability of being sampled, and suppose we have a full binary tree as in standard tree-based ORAMs. Whenever the client samples an access, there

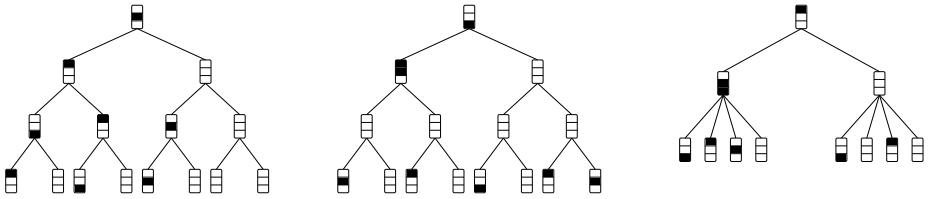


Fig. 1: The left tree illustrates the real block distribution in a standard tree-based ORAM structure; the center tree illustrates the real block distribution when addresses are sampled from a skewed distribution; and the right tree illustrates the structure of the hybrid tree.

is a high probability that the address comes from $\{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3\}$, which means that after every access, the address is added to the root, and then a single eviction occurs that pushes the address down the tree. One can then observe that these three addresses will remain in the top part of the tree with high probability, even though eviction pushes them (on average) slightly toward the leaves; see the center tree in Figure 1 for an illustration. One can also observe that the lower parts of this binary tree are likely to be unoccupied.

This is the main technical observation our design relies on: whenever the distribution is skewed, the addresses that are likely to be accessed will remain at the top of the tree, while the lower parts of the tree will be mostly empty. We therefore change the structure of the tree as a function of the skewness of the distribution, keeping the upper part of the tree as is and compacting the lower part into a set of leaves; see the rightmost tree in Figure 1 for an illustration. Each such leaf can be viewed as conceptually representing a (part of the) subtree condensed into a single node. Of course, two main questions arise in making such a design more efficient than traditional tree-based structures: (1) how do we determine the height of the upper tree, and (2) the idea of compacting a subtree into a set of leaves does not, by itself, reduce communication, since a leaf is now as large as a non-trivial portion of the subtree. We now discuss how we resolved each of these questions, which will also highlight the main proof challenges.

Height of the tree. Determining the height of the tree first relies on the observation that, given an access distribution, one can determine the expected number of distinct addresses λ accessed in an m-length access sequence. To determine λ, we view the problem through a non-uniform balls-and-bins experiment, where the balls are the accesses and the bins are the addresses in the address space. Thus, λ corresponds to the expected number of non-empty bins in this setting. Using a concentration bound, one could then choose a height equal to $\log \lambda$, since we know we can only operate on λ distinct addresses. However, this approach is not viable in general: even for a (very) skewed distribution, λ can be fairly large if the sequence length is long, which in this paper we take to be polynomial in the security parameter k. In other words, for large m, we may end up with a height as large as $\log N$, yielding no significant gain.

To address this issue, which is central to our overflow analysis, we rely on the observation that when we leverage a deterministic eviction process; specifically, a reverse-lexicographic eviction; we can bound the number of distinct addresses over much smaller intervals, as small as the size of the RAM N (which is significantly smaller than m in real-world settings; when $m < N$, we reduce to the simple case above). In particular, in Lemma 2, we show that the number of distinct addresses at *any point in time* throughout the m accesses is at most 2λ, where λ is the number of distinct addresses accessed within N consecutive accesses. The proof builds on an inductive argument: we show that any address either survives into the subsequent N-long sequence or is evicted by the end of the current one.

Leveraging this lemma, we show that (1) we can reason about an upper tree that, in the case of skewed distributions, is much smaller than the height of a standard tree, and (2) the overflow analysis of this upper tree reduces to the stash analysis of Ring ORAM [49] over every N-length window. Applying the union bound yields a result similar to that of Ring ORAM, except for a logarithmic-in-m loss in security (resulting in a slightly larger theoretical stash size). Furthermore, contrary to standard tree-based ORAMs, where the stash overflow probability depends only on the size of the stash and the capacity of the nodes, in our case it also depends on the event that the tree has a specific size, which introduces an additional (slight) loss in security; we detail this in Lemma 2.

Compacting the subtrees. As discussed above, the leaves in our hybrid structure can be viewed as *compacting* an entire subtree into some number of leaves. In particular, if h is the height of the upper tree, then we have 2^h subtrees, each of size (prior to compaction) $2^{\log N - h}$, that get compacted into $\lceil N/2^h \rceil$ leaves. This can be conceptually viewed as completely removing the internal nodes of each subtree and keeping only the leaves. Clearly, if we perform this compaction (i.e., reduce the number of nodes) while increasing the capacity of the leaf nodes, there is no gain from an efficiency standpoint. To demonstrate a gain, we need to show that the size of the leaf node is significantly smaller than $z \cdot (\log N - h)$, which we achieve as discussed below, where z is the node size in [49].

A modified setup and eviction. One could argue that our upper-tree analysis is missing an important point: *setup*. In most tree-based ORAM schemes, the standard way to load all real blocks into the tree is to first initialize an empty full binary tree and then perform N distinct write accesses. Such an approach would invalidate our analysis above, which relies on the fact that accesses to the tree follow a given access distribution $\mathcal{D}$; in particular, our height analysis would no longer hold. To circumvent this issue, we proceed differently in the setup phase. Specifically, we assign each block to a leaf uniformly at random, doing this for all real blocks. With this approach, we know that the maximum load is logarithmic in N, which violates our efficiency target stated above. To address this, we introduce a *leaf stash* and truncate every leaf to have capacity

z_L. We are now in a setting where, after setup, the upper tree is empty and all blocks reside in the leaves or in the leaf stash.

When accesses are made, they can be viewed as drawing a ball from a bin and rethrowing it uniformly at random into another bin. (While it is true that eviction happens gradually, and blocks may take several steps to land in a given bin, assuming that a block lands directly in the leaf captures a worst case that simplifies our analysis.) The setup and accesses can therefore be modeled as the *repeated* balls-and-bins problem; we show in Lemma 4 that the leaf size is $\Theta(\log k/\log\log k)$, where k is the security parameter, with a leaf stash of size $\Theta(\log k)$.

The eviction process is the same as standard reverse-lexicographic eviction, except that when we evict a path ℓ and a block is assigned to this path, we move that block to the corresponding leaf, or to the leaf stash if the leaf is full. In other words, no block with leaf ℓ remains in the upper tree (or in the stash) after the eviction. This is critical both for our main Lemma 2 and for Lemma 4 to hold.

1.2.3 Black-Box Distribution-Aware ORAM

Contrary to DAS and DAT, DAB builds a distribution-aware ORAM from any ORAM in a black-box way (modulo some technicalities that will become clear below). The key idea is that, instead of using a single ORAM to hold all addresses, we partition the address space into two sets: one holding the most frequent addresses, $\mathbb{A}^h$, and the other holding the least frequent addresses, $\mathbb{A}^\ell$. We then store the blocks corresponding to each set in two separate ORAMs. While the idea is simple, two key challenges arise.

Obliviousness challenge. When the distribution $\mathcal{D}$ is skewed, most of the mass is concentrated within a small number of addresses, i.e., $\#\mathbb{A}^h$ is small, so the complexity of accessing the first ORAM is small. However, we cannot simply access the corresponding ORAM based on which set a sampled address belongs to: this would violate obliviousness, since one could trivially infer whether an access targets $\mathbb{A}^h$ or $\mathbb{A}^\ell$. To address this, the access pattern must be independent of the distribution. Accessing both ORAMs on every operation would clearly solve this issue, but the resulting construction would be less efficient than the underlying ORAM.

Instead, we introduce the following deterministic approach: given an epoch of length n, we always access the first ORAM, regardless of which set the sampled address belongs to. If the address belongs to $\mathbb{A}^h$, this access retrieves the correct block; if not, the address is added to a queue. At the end of the n-th access, the queue is flushed by accessing the second ORAM. Note that DAB introduces delays: any address added to the queue receives a response only at the end of the n operations.

While this deterministic approach brings us close to our goal, it still does not preserve obliviousness, for two reasons. First, simply flushing the queue would reveal the number of low-frequency accesses made during the epoch, violating

obliviousness. To address this, we always flush a fixed number of elements λ (if the queue contains fewer than λ addresses, we pad with dummies; if it contains more, we stop at λ). Second, during an epoch the client has no knowledge of whether an address belongs to $\mathbb{A}^h$ or $\mathbb{A}^\ell$ (storing this information locally would require linear client state, which we want to avoid), so the client cannot decide on the fly which ORAM to access, because (1) many ORAM constructions fail if one accesses an address outside their domain, and (2) the client cannot add a low-frequency address to the queue without knowing the distribution. To address this, we require an additional property of the underlying ORAM scheme: address-space agnosticism. Specifically, the scheme must preserve obliviousness even when the accessed address does not exist in its address space. This property is satisfied by most tree-based ORAM constructions.

Correctness and efficiency challenges. For correctness, the challenge is to show that the queue is large enough so that the number of low-frequency addresses sampled in each epoch is at most some bound λ that we will define. The number of low-frequency addresses in an epoch can be viewed as the number of successes in a binomial experiment, where the success probability is the total mass of low-frequency queries. We then use a Chernoff bound to bound the overflow probability, ensuring that the queue does not exceed λ in any epoch over a sequence of length polynomial in k. For efficiency, we model the partitioning as an optimization problem in which we choose the size of $\mathbb{A}^h$ so that the (amortized) communication complexity is minimized.

2 Related Work

ORAM has received considerable attention since its introduction in the seminal paper by Goldreich and Ostrovsky [23], with many subsequent works focused on improving the communication complexity with the goal of matching the communication complexity lower bound [51, 55, 49, 2, 45, 3]. These constructions fall into one of two paradigms: the hierarchical design paradigm or the more recently introduced tree-based paradigm. In the former, the OptORAMa construction [2] was the first to match the logarithmic lower bound. In the latter, Path ORAM [54] (and its variants such as Ring ORAM [49]) remains arguably one of the fastest designs in practice, even though it matches the lower bound only for certain (yet realistic) block-size regimes. Many recent works have looked at improving other dimensions, including round complexity [18, 7], the parallel RAM setting [5], and resizability [44, 4].

ORAM applications. Alongside these continuous design improvements, ORAM has been used as a building block to hide data access patterns in a variety of applications. For instance, ORAM has been used to design more efficient multi-party computation when the underlying function is better represented as a RAM program [24, 59, 34]. ORAM has also served as a building block for more complex encrypted data structures, including searchable symmetric encryption (SSE) [56, 19], structured encryption (STE) [30, 21], and encrypted

databases [28]. ORAM is often integrated with trusted hardware systems to protect the privacy of external memory accesses [50, 61]. These integrated systems have been used for a variety of applications, including federated learning [31], and database systems [15].

Distribution-aware designs and cryptanalysis. The closest works to ours are those mentioned in the introduction on distribution-aware STE and PIR [22, 40]. However, knowledge of the data or query distribution has also been used for other purposes. For instance, parameter-hiding order-revealing encryption is a primitive that hides the mean and variance of the distribution underlying the encrypted plaintext values [8]. Volume-hiding STE schemes [29, 46] hide the response volumes, which can be viewed as hiding the response-length distribution of a data structure. Pancake [25] is an encrypted key-value store that hides the query distribution in the passive setting (i.e., when the adversary does not choose the accesses). Additionally, concurrent work has considered heuristically optimizing ORAM based on temporal locality. Temporal locality is the property of an access sequence whereby an item that was accessed recently is more likely to be accessed again in the near future than an item that has not been accessed for a long time [1]. Further studies of both temporally-correlated access patterns and dynamically evolving access patterns are promising directions for future work. From a cryptanalysis perspective, the access distribution is fundamental to many leakage attacks analysis and evaluation, especially those that exploit algebraic structure, such as attacks on encrypted search schemes supporting expressive queries like range search. Initially, several leakage attacks assumed a uniform distribution [33, 37], but more recent works have studied more realistic, skewed query distributions [27, 16].

3 Preliminaries

Notation. We denote the security parameter by k, and all algorithms run in time polynomial in k. The set of all binary strings of length n is denoted by $\{0,1\}^n$, and the set of all finite binary strings by $\{0,1\}^*$. We write $[n]$ for the set of integers $\{1,\ldots,n\}$. The output x of an algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$. Given a sequence $\mathbf{v}$ of n elements, we refer to its ith element as v_i or $\mathbf{v}[i]$. For a set S we write $\#S$ to refer to its cardinality. For a string s we write $|s|_2$ to refer to its bit length.

The word RAM. Our computational model is the word RAM. In this model, memory consists of an unbounded sequence of words, each of size w bits, and arithmetic, logical, read, and write operations on words all run in $O(1)$ time. We denote by $|x|_w$ the word length of an item x, that is, $|x|_w = |x|_2/w$. We assume throughout that $w = \Omega(\log k)$.

RAM data structures. A plaintext RAM data structure maps addresses in an address space $\mathbb{A}$ to data values in a value space $\mathbb{V}$. It supports read and write

operations from an operation space $\mathbb{O}$, which we define as $\{\mathtt{read}, \mathtt{write}\} \times \mathbb{A} \times \mathbb{V}$, where $(\mathtt{read}, \mathsf{a}, \perp)$ is a read operation that returns the value stored at address a; and $(\mathtt{write}, \mathsf{a}, \mathsf{v})$ is a write operation that sets the value at address a to v. The RAM access algorithm $\mathsf{ptAccess}$ takes as input a RAM and an operation $\mathsf{op} \in \mathbb{O}$, and outputs a response ptResp together with a (possibly updated) RAM'. One could view a RAM as an array data type over address space $\mathbb{A} = \{1, \ldots, \#\mathbb{A}\}$ and value space $\mathbb{V}$, note that the majority of hierarchical oblivious RAM constructions consider this case. One could also view a RAM as a dictionary data type with fixed size and a static address space $\mathbb{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_N\}$ where $\mathsf{a}_i \in \{0,1\}^*$, for all $i \in \{1, \ldots, N\}$; and note that this is the case of the majority of tree-based ORAM constructions.

Basic cryptographic primitives. We make use of a symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and we say that SKE is random-ciphertext-secure against chosen-plaintext attacks (RCPA) if the ciphertexts it outputs are computationally indistinguishable from random even to an adversary that can adaptively query an encryption oracle. We also make use of pseudo-random permutation (PRP) and we say that a PRP is secure if it is indistinguishable from a random permutation by any probabilistic polynomial-time adversary. We refer the reader to [32] for more detailed security definitions of these primitives.

4 Definitions

4.1 Encrypted RAMs

We first provide the syntax for a distribution-aware RAM encryption scheme. In particular, a RAM encryption scheme supports two two-party protocols between a client and a server: a Setup protocol and an Access protocol, which we describe below. The main difference in the syntax below with respect to the standard definition is that we give a description of a distribution $\mathcal{D}$ to the setup protocol.

Definition 1 (Distribution-aware RAM encryption). *A distribution-aware RAM encryption scheme* $\Sigma_{\mathsf{RAM}} = (\mathsf{Setup}, \mathsf{Access})$ *consists of two two-party protocols that work as follows:*

- $(K, st; \mathsf{ERAM}) \leftarrow \mathsf{Setup}_{\mathbf{C},\mathbf{S}}(1^k, \mathcal{D}, \mathsf{RAM}; \perp)$*: is a two-party protocol executed between a client and a server where the client inputs a security parameter* 1^k*, a distribution* $\mathcal{D}$*, and a* RAM*. It outputs a secret key* K*, a state* st *and an encrypted memory* ERAM*.*
- $(st', r; \mathsf{ERAM}') \leftarrow \mathsf{Access}_{\mathbf{C},\mathbf{S}}(K, st, \mathsf{op}; \mathsf{ERAM})$*: is a two-party protocol executed between a client and a server where the client inputs a secret key* K*, a state* st*, an operation* op*, and the server inputs* ERAM*. The client receives as output a (possibly) updated state* st' *and a response* $r \in \mathbb{V} \cup \{\perp\}$ *while the server receives an updated encrypted memory* ERAM'*.*

Note that the syntax of a standard RAM encryption scheme can be recovered by removing the distribution, $\mathcal{D}$, from the input of the Setup protocol. Below, we use the standard adaptive security definition for structured encryption schemes [12, 9], formulated in the real/ideal paradigm, to define the security of distribution-aware RAM encryption schemes.[3] The key advantage of this security definition is that it is leakage-parameterized: it explicitly captures the information the adversary learns about the structure.

Definition 2 (Distribution-aware oblivious RAM). *Let* $\Sigma_{\mathsf{RAM}} = (\mathsf{Setup}, \mathsf{Access})$ *be a RAM encryption scheme and consider the following probabilistic experiments where* $\mathcal{C}$ *is a stateful challenger,* $\mathcal{A}$ *is a stateful adversary,* $\mathcal{S}$ *is a stateful simulator:*

- $\mathbf{Real}_{\Sigma_{\mathsf{RAM}},\mathcal{C},\mathcal{A}}(k,m)$*: given* 1^k*, the adversary* $\mathcal{A}$ *outputs a* RAM *and a distribution* $\mathcal{D}$*. The challenger and adversary execute* $(K, st; \mathsf{ERAM}) \leftarrow \mathsf{Setup}_{\mathcal{C},\mathcal{A}}(1^k, \mathcal{D}, \mathsf{RAM}; \perp)$*.* $\mathcal{A}$ *then adaptively chooses a polynomial-size sequence of operations* $(\mathsf{op}_1, \ldots, \mathsf{op}_m)$*. For all* $1 \leq i \leq m$*, the adversary and challenger execute* $\mathsf{Access}_{\mathcal{C},\mathcal{A}}$*, with* $\mathcal{A}$ *playing the role of the server and the challenger* $\mathcal{C}$ *playing the role of the client. Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*
- $\mathbf{Ideal}_{\Sigma_{\mathsf{RAM}},\mathcal{A},\mathcal{S}}(k,m)$*: given* 1^k*, the adversary* $\mathcal{A}$ *outputs a* RAM *and a distribution* $\mathcal{D}$*. Given* $\#\mathsf{RAM}, \mathcal{D}$*, the simulator executes* $\mathsf{Setup}_{\mathcal{S},\mathcal{A}}$ *with* $\mathcal{A}$ *who receives* ERAM*.* $\mathcal{A}$ *then adaptively chooses a polynomial-size sequence of operations* $(\mathsf{op}_1, \ldots, \mathsf{op}_m)$*. For each operation,* $\mathcal{A}$ *and* $\mathcal{S}$ *execute* $\mathsf{Access}_{\mathcal{S},\mathcal{A}}$ *with* $\mathcal{A}$ *playing the role of the server and* $\mathcal{S}$ *playing the role of the client. During these executions, the simulator* $\mathcal{S}$ *receives nothing. Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

We say that Σ_{RAM} *is a distribution-aware oblivious RAM if there exists a* PPT *simulator* $\mathcal{S}$ *such that for all* PPT *adversaries* $\mathcal{A}$*, for all* $m = \mathsf{poly}(k)$*,*

$$|\Pr[\mathbf{Real}_{\Sigma_{\mathsf{RAM}},\mathcal{C},\mathcal{A}}(k,m) = 1] - \Pr[\mathbf{Ideal}_{\Sigma_{\mathsf{RAM}},\mathcal{A},\mathcal{S}}(k,m) = 1]| \leq \mathsf{negl}(k).$$

The only change relative to the standard ORAM security definition is that the adversary must choose the distribution $\mathcal{D}$, which is then given to the simulator in the ideal experiment. This distribution can be viewed as additional leakage. As in standard oblivious RAM, the setup leakage also includes the size of the RAM, and our definition permits this leakage as well.

[3] Structured encryption [9] was introduced as a generalization of index-based searchable symmetric encryption [53, 12]. It is a primitive that encrypts a data structure so that the structure can still be queried, while both the data and the operations performed on it remain hidden up to a well defined leakage profile. We find it useful to equate the interaction between a CPU and untrusted memory with that between a client and an untrusted server; this correspondence allows us to view oblivious RAM through the lens of structured encryption. This conceptual equivalence was first observed by Kamara, Moataz, and Ohrimenko [30].

Address-space agnostic. Standard ORAM definitions assume the adversary $\mathcal{A}$ chooses addresses from the original RAM's address space, $\mathbb{A}$. If we allow the adversary to choose addresses outside of $\mathbb{A}$, we call the scheme an address-space agnostic oblivious RAM. The address-space agnostic property provides a slightly stronger security guarantee, as it prevents an adversary from distinguishing between accesses even when the requested address does not exist in the structure. For example, square-root ORAM [23] is not address-space agnostic, while Path ORAM [54] and most other tree-based ORAM constructions are.

Definition 3 (Correctness). *Let* $\Sigma_{\mathsf{RAM}} = (\mathsf{Setup}, \mathsf{Access})$ *be a RAM encryption scheme. We say that* Σ_{RAM} *is* correct *if for all* $k \in \mathbb{N}$*, for all* $m = \mathsf{poly}(k)$*, for all input* RAM_0*, for all* $\mathsf{op}_1, \ldots, \mathsf{op}_m \in \mathbb{O}$*, for all* $i \in [m]$*, it holds that:*

$$\Pr\left[\, r_i = \mathsf{ptResp}_i \,\right] \geq 1 - \mathsf{negl}(k),$$

where $(st', r_i; \mathsf{ERAM}_i) \leftarrow \mathsf{Access}_{\mathbf{C},\mathbf{S}}(K, st, \mathsf{op}_i; \mathsf{ERAM}_{i-1})$ *with* $(K, st; \mathsf{ERAM}_0) \leftarrow \mathsf{Setup}_{\mathbf{C},\mathbf{S}}(1^k, \mathsf{RAM}_0; \perp)$*, and* $(\mathsf{ptResp}_i, \mathsf{RAM}_i) \leftarrow \mathsf{ptAccess}(\mathsf{RAM}_{i-1}, \mathsf{op}_i)$*.*

In this paper we will also use *distribution-aware* correctness, where the correctness of RAM accesses only holds for addresses sampled from a fixed distribution $\mathcal{D}$.

Definition 4 (Distribution-aware correctness). *Let* $\Sigma_{\mathsf{RAM}} = (\mathsf{Setup}, \mathsf{Access})$ *be a RAM encryption scheme. We say that* Σ_{RAM} *is* distribution-aware *correct for a distribution* $\mathcal{D}$ *over* $\mathbb{A}$ *if for all* $k \in \mathbb{N}$*, for all* $m = \mathsf{poly}(k)$*, for all input* RAM_0*, for all* $\mathsf{op}_1, \ldots, \mathsf{op}_m \in \mathbb{O}$ *with* $(\mathsf{a}_1, \ldots, \mathsf{a}_m) \leftarrow \mathcal{D}^m$*, for all* $i \in [m]$ *it holds that:*

$$\Pr\left[\, r_i = \mathsf{ptResp}_i \,\right] \geq 1 - \mathsf{negl}(k),$$

where $(st', r_i; \mathsf{ERAM}_i) \leftarrow \mathsf{Access}_{\mathbf{C},\mathbf{S}}(K, st, \mathsf{op}_i; \mathsf{ERAM}_{i-1})$ *with* $(K, st; \mathsf{ERAM}_0) \leftarrow \mathsf{Setup}_{\mathbf{C},\mathbf{S}}(1^k, \mathcal{D}, \mathsf{RAM}_0; \perp)$*, and* $(\mathsf{ptResp}_i, \mathsf{RAM}_i) \leftarrow \mathsf{ptAccess}(\mathsf{RAM}_{i-1}, \mathsf{op}_i)$*.*

Finally we define distribution-aware correctness *with delay*, where the correct RAM accesses are performed with some known delay n when the addresses are sampled from a fixed distribution $\mathcal{D}$.

Definition 5 (Distribution-aware correctness with delay). *Let* $\Sigma_{\mathsf{RAM}} = (\mathsf{Setup}, \mathsf{Access})$ *be a RAM encryption scheme. We say that* Σ_{RAM} *is* distribution-aware *correct with* n*-length delay for a distribution* $\mathcal{D}$ *over* $\mathbb{A}$ *if for all* $k \in \mathbb{N}$*, for all* $m = \mathsf{poly}(k)$*, for all input* RAM_0*, there exists* $n \in \mathbb{N}$*, such that for all* $\mathsf{op}_1, \ldots, \mathsf{op}_m \in \mathbb{O}$ *with* $(\mathsf{a}_1, \ldots, \mathsf{a}_m) \leftarrow \mathcal{D}^m$*, for all* $i \in [m]$ *it holds that:*

$$\Pr\left[\, R_n \ni \mathsf{ptResp}_i \,\right] \geq 1 - \mathsf{negl}(k),$$

where $R_n = \{r_i, r_{i+1}, \ldots, r_\kappa\}$ *and* $\kappa = \min\{m, i+n-1\}$,[4] *with* $(st', r_i; \mathsf{ERAM}_i) \leftarrow \mathsf{Access}_{\mathbf{C},\mathbf{S}}(K, st, \mathsf{op}_i; \mathsf{ERAM}_{i-1})$*,* $(K, st; \mathsf{ERAM}_0) \leftarrow \mathsf{Setup}_{\mathbf{C},\mathbf{S}}(1^k, \mathcal{D}, \mathsf{RAM}_0; \perp)$*, and* $(\mathsf{ptResp}_i, \mathsf{RAM}_i) \leftarrow \mathsf{ptAccess}(\mathsf{RAM}_{i-1}, \mathsf{op}_i)$*.*

[4] We note that in R_n we use the notation r_j to denote both a single response as well as (possibly) a set of responses to the j^{th} query.

5 Distribution-Aware Square-root ORAM

In this section, we present our distribution-aware square-root ORAM construction DAS, which concretely improves, for skewed distributions, on the amortized communication complexity of the original square-root construction [23]. While asymptotically DAS has worse communication complexity than our tree-based ORAM constructions in Sections 6 and 7, it nonetheless serves as a good foundational first step, as its design encapsulates one of the core ideas we will reuse. For a high-level intuition, we refer the reader to Section 1.2. We recall the square-root ORAM construction below; readers already familiar with the construction may safely skip the next paragraph.

Overview of square-root ORAM. The scheme is parametrized by two variables: the number of accesses before reinitialization t and the RAM size N.

To set up the ORAM from the RAM, the client first appends t dummy blocks to the RAM and samples a pseudorandom permutation π. Using π, the client shuffles the real and dummy blocks, encrypts the entries of the RAM, initializes a t-sized cache (also known as the shelter), and sends the shuffled encrypted RAM and cache to the server. Note that once the blocks are shuffled, each block can be accessed exactly once without revealing anything about the underlying physical block being accessed.

To access an address a_i, the client first linearly scans the blocks in the cache. If the item is not found, the client requests $\pi(i)$ from the shuffled RAM, retrieves it, and adds it to the cache (modified, if the access is a write). If the item is found, the client instead performs a dummy access by requesting an unaccessed dummy block from the shuffled RAM.

After t accesses, the client re-initializes the structure using two oblivious sorts: one to remove the old dummy blocks associated with the prior permutation, and another to sort the data and new dummy blocks according to a freshly sampled permutation. Provided $t < N$, the cost of the oblivious sort is $O(N \log N)$, yielding an amortized cost of $O(\lambda + {}^{N \cdot \log N}/_{t})$ per access. In [23], t is set to $\sqrt{N}$, resulting in an amortized cost of $O(\sqrt{N} \log N)$ per access.

Overview of DAS. DAS is nearly identical to [23], except for a minor technical modification: we decouple the cache size λ from the epoch length t (in [23], $\lambda = t$). Due to space limitations, the pseudo-code of DAS is deferred to the full version. As highlighted in Section 1.2, the number of dummies and the cache size both depend on the access distribution and can be tuned as separate parameters. For skewed access distributions, a large number of accesses are made to a small number of high-frequency addresses; consequently, the cache size λ can be much smaller than the $\sqrt{N}$ used in prior work. Similarly, because most accesses target a small number of locations, the epoch length t can now be increased.

To preserve the security of DAS, we make one small modification to the protocol. If the cache is full before the t accesses of an epoch are complete, the client still performs a dummy access to the server. The original protocol does not have to consider this case, since it sets $\lambda = t$. This modification is required

to preserve obliviousness, because once the cache is full, the client can no longer verify whether the address has been accessed previously. With this modification, all accesses to the structure are unique, even if the access sequence is not sampled from the distribution $\mathcal{D}$ used to set up the ORAM.

5.1 Security and Correctness Analysis

We state below the security of DAS and show that it is distribution-aware correct as long as the cache has size $\lambda = \Theta(\mu + \log k)$, where μ represents the expected number of distinct addresses accessed during an epoch of length t, that is, $\mu = \sum_{i=1}^{N} (1 - (1 - p_i)^t)$. The correctness proof reduces to counting the number of distinct accesses, which can be viewed as the number of non-empty bins in the non-uniform balls-and-bins experiment. We state our security and correctness theorems below. Due to space limitations, all proofs are deferred to the full version.

Theorem 1. *If Σ_{SKE} is an* RCPA*-secure encryption scheme, then* DAS *is a distribution-aware oblivious RAM.*

Theorem 2. *Let $N \in \mathbb{N}$, $k, t \in \mathbb{N}_{>0}$, and $\mathcal{D} = \{p_1, \ldots, p_N\}$ be a discrete distribution over the support $\mathbb{A} = \{1, \ldots, N\}$. If $\lambda = \Theta(\mu + \log k)$, then* DAS *is distribution-aware correct, where $\mu = \sum_{i=1}^{N} (1 - (1 - p_i)^t)$.*

5.2 Efficiency Analysis

We first state the black-box asymptotics of DAS, followed by a concrete analysis for the case of the Zipf distribution.

Theorem 3. *Let $N \in \mathbb{N}$, $k, t \in \mathbb{N}_{>0}$, and let $\mathcal{D} = \{p_1, \ldots, p_N\}$ be a discrete distribution over the support $\mathbb{A} = \{1, \ldots, N\}$. Then* DAS *has communication complexity $\Theta\left(\mu + \log k + {}^{(N+t)\log(N+t)}/_{t}\right)$, storage complexity $O(N + t + \log k)$, and round complexity $O\left(\max\left\{\log(t + N)/t,\, 1\right\}\right)$, where $\mu = \sum_{i=1}^{N} (1 - (1 - p_i)^t)$.*

The proof follows directly from Theorem 2.

Concrete analysis. We now assume that the underlying distribution $\mathcal{D}$ is the Zipf distribution with parameter $s > 1$, which we denote by $\mathcal{Z}_s$. If X is Zipf-distributed with parameter $s > 1$, then $\Pr[\, X = i \,] = i^{-s}/H_{N,s}$, where $H_{N,s} = \sum_{i=1}^{N} i^{-s}$ is the generalized harmonic number. Note that the choice of the Zipf distribution is not arbitrarily: it is the most common distribution used to capture real-world query behavior [6, 36, 60]. As highlighted in the previous theorem, an important quantity we need to bound is μ, which counts the expected number of distinct addresses accessed in a t-long sequence. Below, we state a lemma that bounds μ for Zipf distributions with parameter $s > 1$.

Lemma 1. *Let $N \in \mathbb{N}$ and $t \in \mathbb{N}_{>0}$, if $\mathcal{D}$ is a Zipf distribution $\mathcal{Z}_s$ with $s > 1$, then*

$$\mu = \Theta\left(\min\left\{N,\, \left(\frac{t}{H_{N,s}}\right)^{1/s}\right\}\right).$$

With the above lemma, we can now state our main efficiency theorem for the Zipf distribution with parameter $s > 1$.

Theorem 4. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, and $s > 1$. If $\mathcal{D}$ is a Zipf distribution $\mathcal{Z}_s$ and $t = \Theta(N)$, then* DAS *has communication complexity $\Theta\left(\left({}^{N}/_{H_{N,s}}\right)^{1/s} + \log k\right)$, storage complexity $O(N + \log k)$, and round complexity $O(1)$.*

Concrete example. For $N = 2^{20}$, a Zipf distribution with parameter $s = 2$, and a probability of correctness failure of 2^{-80}, we have $\mu \approx 1{,}415$, yielding a communication complexity (in blocks) of $\approx 1{,}831$, compared to $\approx 2^{10} \cdot \log_2(2^{20}) = 20{,}480$ for the standard square-root ORAM. This is more than an order of magnitude reduction, due in part to the longer interval between costly rebuild operations: in DAS, the oblivious sort is amortized over 2^{20} accesses, whereas in the standard square-root ORAM it is amortized over only 2^{10} accesses.

6 Distribution-Aware Tree-based ORAM

In this section, we introduce our main construction DAT. For more intuition on the construction's design, as well as the proof's technical challenges, we refer the reader to Section 1.2. Below, we provide the details of the construction, followed by our security, correctness, and efficiency analysis.

6.1 Details

DAT makes black-box use of a symmetric encryption scheme $\Sigma_{\mathsf{SKE}} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and its pseudo-code, due to space limitation, is deferred to the full version. We provide high-level details below.

Setup. Given a security parameter k, a distribution $\mathcal{D}$, and a RAM RAM, the setup first builds a tree T of height $h+1$ with the following structure: the upper tree of height h is a full binary tree in which each node has capacity z_I blocks, and each node at level h has $\lceil N/2^h \rceil$ children, where each child has capacity z_L blocks. The height h is a function of the security parameter and the distribution $\mathcal{D}$, and is computed using the HeightComp subroutine (described further below) as $h \leftarrow \mathsf{HeightComp}(1^k, \mathcal{D})$. The client initializes an empty dictionary DX_{st}. This dictionary can be viewed as the position map of standard tree-based ORAMs, as it maps an address to its corresponding leaf. For each address $\mathsf{a} \in \mathbb{A}$, the client samples a leaf $\ell \xleftarrow{\$} [N]$ uniformly at random and, if the ℓth leaf still has empty space, adds the block $(\ell, \mathsf{a}, \mathsf{RAM}[\mathsf{a}])$ to that leaf node; otherwise, it adds the block to a list we refer to as the *leaf stash*, denoted lStash. This stash stores all blocks that could not be placed in their corresponding bins due to the bins reaching capacity z_L. The client also initializes an empty list uStash, which stores any block that cannot fit in the upper tree of height h; we refer to this as the *upper-tree stash*. This stash can be viewed as the stash used in standard tree-based ORAMs. The client further initializes a counter count $:= 0$, which keeps

track of the number of accesses in order to select the correct eviction path, as we detail later. The client then computes $K \leftarrow \Sigma_{\mathsf{SKE}}.\mathsf{Gen}(1^k)$ and encrypts every node node in the tree using the key K. The client outputs the key K and the state $st := (\mathsf{DX}_{st}, \mathsf{uStash}, \mathsf{lStash}, \mathsf{count})$, while the server outputs the encrypted RAM $\mathsf{ERAM} := \mathsf{T}$.

Access. Given a key K, a state st, and an operation op, the client first parses the operation op as $(\mathsf{type}, \mathsf{a}, \mathsf{v}_{\mathsf{new}})$. It also parses the state st as $(\mathsf{DX}_{st}, \mathsf{uStash}, \mathsf{lStash}, \mathsf{count})$ and $\mathsf{DX}_{st}[\mathsf{a}]$ as ℓ, where ℓ represents the path where the corresponding block resides. The client then samples a new leaf identifier $\ell_{\mathsf{new}} \xleftarrow{\$} [N]$ and updates the position map as $\mathsf{DX}_{st}[\mathsf{a}] := \ell_{\mathsf{new}}$. This new leaf represents the path where the block will reside at the end of this access operation.

The client first checks whether the block exists in either of the stashes (the leaf stash lStash or the upper-tree stash uStash). If the block does not exist there, the client requests the server to send back the path from the root to the ℓth leaf. Specifically, it runs the $\mathsf{ReadPath}$ subroutine, which we detail below and which outputs the correct block. Otherwise, if the block does exist in one of the stashes, the client still accesses the path but discards the output; running this subroutine in this case is important to maintain obliviousness.

Depending on the type of the operation, the client then proceeds as follows. If $\mathsf{type} = \texttt{read}$, the client outputs the block and adds the block $(\ell_{\mathsf{new}}, \mathsf{a}, \mathsf{v})$ to uStash, where v is the same as in the existing block. If $\mathsf{type} = \texttt{write}$, the client instead adds the block $(\ell_{\mathsf{new}}, \mathsf{a}, \mathsf{v}_{\mathsf{new}})$ to uStash. The client and server then run the eviction subroutine, which we detail below. Finally, the client outputs the updated state st, while the server outputs the updated encrypted RAM ERAM.

Subroutines. DAT makes use of the following subroutines:

- $\mathsf{HeightComp}$: this subroutine takes as input the security parameter k and the distribution $\mathcal{D} = \{p_{\mathsf{a}_1}, \ldots, p_{\mathsf{a}_N}\}$, and determines the height of the upper tree. It first computes μ, which represents the expected number of distinct addresses accessed within N accesses:

$$\mu = \sum_{i=1}^{N} \left(1 - (1 - p_{\mathsf{a}_i})^N\right),$$

  and then sets the height to $h = \lceil \log(\mu + \log k) \rceil$. Note that this height is not chosen arbitrarily but is the result of Theorem 6.
- $\mathsf{ReverseLex}$: this subroutine takes as input a counter count and the RAM size N. It flips the binary representation of count and outputs the corresponding decimal value. The main guarantee provided by this subroutine is that, as count is incremented, the sequence of outputs traverses a binary tree in an even, alternating manner: consecutive outputs are equally spaced.
- $\mathsf{ReadPath}$: this subroutine is a two-party protocol between the client and the server. The client inputs the key K, the leaf identifier ℓ, and the address a, while the server inputs the encrypted RAM ERAM. The client first fetches the

path $\mathcal{P}_\ell$, decrypts every node, and checks whether there is a block containing the address a. If such a block is found, the client outputs its corresponding value v, replaces the address content with a dummy element $\perp$, re-encrypts all nodes, and sends the new path back to the server.

– Eviction: this subroutine is a two-party protocol between the client and the server. The client inputs its key K and state st, while the server inputs the encrypted RAM ERAM. The eviction process pushes *real* blocks down the tree as far as possible, so that there is enough empty space in the upper part of the tree for future accesses. The upper-tree stash can be viewed as a data structure that stores the excess blocks that cannot (in certain cases) be pushed further down a specific path. The eviction underlying DAT is the deterministic, reverse-lexicographic eviction and is almost identical to the one used in Gentry et al. [20] and Ring ORAM [49], except for the additional handling of the leaf stash, as detailed below.

  The client first computes $\ell \leftarrow \mathsf{ReverseLex}(\mathsf{count}, N)$, which determines the path to be evicted in this round. It also initializes two sets S_ℓ and $S_{\bar{\ell}}$: the former will contain blocks that belong to the path to be evicted, $\mathcal{P}_\ell$, while the latter will contain blocks that belong to other paths (we discuss how they are populated and used below). The client then fetches the path $\mathcal{P}_\ell$ from the server. For each real block in the path of the form $(\ell', \mathsf{a}', \mathsf{v}')$, if $\ell' = \ell$, it adds the block to S_ℓ; otherwise, it adds the block to $S_{\bar{\ell}}$. Similarly, for each block in the upper-tree stash $(\ell', \mathsf{a}', \mathsf{v}')$, if $\ell' = \ell$, it adds the block to S_ℓ; otherwise, it adds the block to $S_{\bar{\ell}}$. It then reinitializes the upper-tree stash uStash. The client then adds S_ℓ to the leaf stash lStash and $S_{\bar{\ell}}$ to the upper-tree stash uStash.

  The client now begins rebuilding the new path. It first creates $h+2$ empty nodes, where the new leaf node node_{h+2} can hold up to z_L blocks and the other $h+1$ nodes can each hold up to z_I blocks. To populate the leaf node, the client traverses the leaf stash: for each block whose leaf identifier equals ℓ, if node_{h+2} still has empty space, the block is added to node_{h+2}; once the new leaf node reaches capacity, the procedure stops. If node_{h+2} still has empty slots after this traversal, the client fills them with dummies so that the leaf node always contains exactly z_L blocks.

  The client then moves on to the more standard part of the eviction protocol. Given the upper-tree stash and the $h+1$ remaining empty nodes, we use the standard Path ORAM eviction (also used in Ring ORAM), in which real blocks are pushed as far down the tree as possible; we refer to this as the standard eviction protocol $\mathsf{StEviction}$. Specifically, we start at the deepest internal node node_{h+1} and place into it all the real blocks that can reside on one of its children; if the node fills up and additional real blocks remain that could have been placed there, those blocks are placed in its parent, and so on. If, instead, all such blocks fit and the node still has capacity, we fill the remaining empty slots with dummies. We continue this process up to the root; any remaining real blocks are kept in the upper-tree stash uStash. Finally, the client encrypts all of the nodes and sends the new path to the server.

6.2 Security Analysis

In this section, we analyze the security of our distribution-aware construction DAT. While our security theorem (Theorem 5) holds for all adversarially-chosen access sequences, our correctness theorem (Theorem 6) only holds for access sequences from the distribution $\mathcal{D}$ chosen at setup time. The proof of Theorem 5 is deferred to the full version.

Theorem 5. *If Σ_{SKE} is an RCPA-secure symmetric encryption scheme, then* DAT *is a distribution-aware oblivious RAM.*

6.3 Correctness Analysis

Analyzing the correctness of DAT reduces to analyzing the overflow probability of its construction. Contrary to standard tree-based ORAMs, which have a single client stash, DAT has two client stashes: an upper-tree stash uStash and a leaf stash lStash. As part of the stash overflow analysis, we also need to determine the size of the internal nodes of the upper tree, z_I, and the size of the leaf nodes, z_L.

Our analysis proceeds in two steps. First, the challenge is to identify the height of the upper tree. This height is a function of the underlying access distribution and is determined by the number of distinct addresses that occupy this upper tree at any point in time. This result is central to our proof and is stated in Lemma 2. Once the height of the tree is determined, bounding the size of the upper-tree stash uStash and the internal node size follows from the Ring ORAM analysis, as we make use of reverse-lexicographic eviction. For the second part of the analysis, we show that the occupancy of the leaves can be viewed through the lens of the repeated balls-and-bins problem. In particular, we identify the appropriate bin-load truncation (i.e., the size of the leaf nodes) so that the excess routed to the leaf stash is bounded. Our main result is stated in Theorem 6, and the main building blocks of its proofs are detailed in Sections 6.3.1 and 6.3.2. The proof of the theorem itself is deferred to the full version due to space limitations.

Additional notation. We borrow the same notation used in [54]. We denote the sequence of triplets (a, x, y) as $\boldsymbol{s}$, where a is the address of the RAM being accessed, x denotes the current leaf assignment, and y the new leaf assignment. By $\mathsf{ORAM}_{h+1}^{z_I,z_L}[\boldsymbol{s}]$, we refer to the DAT ORAM tree structure with internal node size z_I, leaf node size z_L, and height $h+1$, after executing all access operations in $\boldsymbol{s}$. By $\mathsf{st_I}(\mathsf{ORAM}_{h+1}^{z_I,z_L}[\boldsymbol{s}])$ and $\mathsf{st_L}(\mathsf{ORAM}_{h+1}^{z_I,z_L}[\boldsymbol{s}])$, we refer to the sizes of the stashes uStash and lStash, respectively, and by $\mathsf{st}(\mathsf{ORAM}_{h+1}^{z_I,z_L}[\boldsymbol{s}])$, the combined size of both stashes.

Theorem 6. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, $m = \mathsf{poly}(k)$, and $\mathcal{D} = \{p_{\mathsf{a}_1}, \ldots, p_{\mathsf{a}_N}\}$ be a discrete distribution over the support $\mathbb{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_N\}$. For an internal node size $z_I = 4$, a leaf node size $z_L = \Theta(\log k / \log\log k)$ with $z_L \geq 2e$, upper-tree height $h = \Theta(\log(\mu + \log k))$, upper-tree stash size $R_I = \omega(\log k)$, and leaf stash size $R_L = \Theta(\log k)$,* DAT *is distribution-aware correct, where $\mu = \sum_{i=1}^{N} \left(1 - (1 - p_{\mathsf{a}_i})^N\right)$.*

6.3.1 Internal Nodes

In the following, we determine the size of the internal nodes and bound the upper-tree stash size. The core of our proof is Lemma 2, in which we identify the smallest height h of the upper tree such that the maximum number of distinct addresses, at any point in time, is at most the number of leaves of this upper tree, i.e., 2^h. The key observation is that, given a distribution $\mathcal{D}$, we can bound the number of distinct addresses that the client accesses within any interval of time. In particular, we leverage the deterministic eviction to show that, in any interval of length N (where N represents the number of distinct addresses in the RAM), the number of addresses accessed within this interval is bounded; and by an inductive argument, we show that this property holds throughout all access operations. Once we identify the minimum height h, the upper tree of DAT can be viewed exactly as an instance of the standard Ring ORAM tree, where, instead of operating on N distinct addresses, we now operate on 2^h. As such, the stash analysis is exactly the same (modulo some small technicalities that we detail later), and we can therefore borrow the existing results to bound the size of the internal nodes z_I as well as the size of the upper-tree ORAM stash uStash. As shown in the following lemma, h is a function of the access distribution, and it can be small for skewed distributions, as we demonstrate in Section 6.4.

Lemma 2. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, $m = \mathsf{poly}(k)$, and $\mathcal{D} = \{p_{\mathsf{a}_1}, \ldots, p_{\mathsf{a}_N}\}$ be a discrete distribution over the support $\mathbb{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_N\}$, the upper tree of height h contains at most 2^h distinct addresses with all but negligible probability in k, where*

$$h = \Theta\left(\log\left(\mu + \log k\right)\right)$$

and $\mu = \sum_{i=1}^{N} \left(1 - (1 - p_{\mathsf{a}_i})^N\right)$.

The proof of Lemma 2 is in the full version. We are now ready to state the result bounding the size of the upper-tree stash uStash and determining the internal node size z_I.

Lemma 3. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, $m = \mathsf{poly}(k)$, and $\mathcal{D} = \{p_{\mathsf{a}_1}, \ldots, p_{\mathsf{a}_N}\}$ be a discrete distribution over the support $\mathbb{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_N\}$. For a node size $z_I = 4$, tree height $h = \Theta(\log(\mu + \log k))$, and upper-tree stash size R_I, the probability that DAT's upper-tree stash overflows after a sequence of read/write operations over $\boldsymbol{a}$ is at most*

$$\Pr\left[\mathsf{st}_{\mathsf{I}}(\mathsf{ORAM}_{h+1}^{4,z_L}[\boldsymbol{s}]) > R_I\right] \leq 1.46\, m \cdot (0.25)^{R_I} + k^{-\omega(1)},$$

where the probability is over the randomness that determines $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\boldsymbol{s} = (\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{y})$, and $\boldsymbol{a}$ is drawn from $\mathcal{D}$.

The proof of Lemma 3 is in the full version. The following corollary follows directly from the lemma above.

Corollary 1. *If $R_I = \omega(\log k)$, then*

$$\Pr\left[\mathsf{st}_{\mathsf{I}}(\mathsf{ORAM}_{h+1}^{4,z_L}[\boldsymbol{s}]) > R_I\right] \leq \mathsf{negl}(k).$$

6.3.2 Leaf Nodes

Contrary to the analysis of the internal nodes, the leaf nodes operate on all addresses. In particular, at setup, we load all the (distinct) addresses once, independently of the underlying distribution over these addresses. We therefore take a different approach for the leaf node size and leaf stash analysis, viewing the randomized process underlying the setup and the subsequent read/write access operations as an instance of the *repeated* balls-and-bins experiment. For the setup, recall that we assign each address to a leaf uniformly at random, which is exactly an N-balls-into-N-bins problem. An access is then equivalent to picking a ball from a bin and re-throwing it uniformly at random into the N bins. Note that in our design, it may take many accesses before an address reaches the leaf (as part of the deterministic eviction of the DAT scheme), which means that it may take some time for the ball to land in its corresponding bin after being re-thrown. Abstracting away this delay, it is sufficient to analyze the case in which the address is evicted immediately, so in our analogous experiment the ball lands in the corresponding bin right after being thrown; this captures the worst case, where the internal nodes do not exist and there is no delay between throwing the ball and its landing in the bin. Below, we state our main Lemma 4 (whose proof is deferred to the full version), which bounds the size of the leaf stash and determines the size of the leaf nodes.

Lemma 4. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, $m = \mathsf{poly}(k)$, and $\mathcal{D} = \{p_{\mathsf{a}_1}, \ldots, p_{\mathsf{a}_N}\}$ be a discrete distribution over the support $\mathbb{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_N\}$. For a leaf stash of size R_L and a leaf size $z_L \geq 2e$, the probability that DAT's leaf stash overflows after a sequence of read/write operations over $\boldsymbol{a}$ is at most*

$$\Pr\left[\,\mathsf{st}_{\mathsf{L}}(\mathsf{ORAM}_{h+1}^{4,z_I}[\boldsymbol{s}]) > R_L\,\right] \leq m \cdot \left(\frac{e \cdot N}{R_L} \cdot \left(\frac{e}{z_L}\right)^{z_L - 1}\right)^{R_L},$$

where the probability is over the randomness that determines $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\boldsymbol{s} = (\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{y})$, and $\boldsymbol{a}$ is drawn from $\mathcal{D}$.

Corollary 2. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, $m = \mathsf{poly}(k)$, if $R_L = \Theta(\log k)$ and $z_L = \Theta(\log k / \log\log k)$, then*

$$\Pr\left[\,\mathsf{st}_{\mathsf{L}}(\mathsf{ORAM}_{h+1}^{z_I,z_L}[\boldsymbol{s}]) > R_L\,\right] \leq \mathsf{negl}(k).$$

6.4 Asymptotic Analysis

We first start by stating the asymptotics of DAT for any discrete access distribution then follow up with a concrete analysis when the access distribution is the Zipf distribution.

Theorem 7. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, $m = \mathsf{poly}(k)$, and $\mathcal{D} = \{p_{\mathsf{a}_1}, \ldots, p_{\mathsf{a}_N}\}$ be a discrete distribution over the support $\mathbb{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_N\}$. The communication complexity of DAT is $\Theta(\log(\mu + \log k) + {\log k}/{\log\log k})$, a storage complexity of $O(N \cdot {\log k}/{\log\log k})$, and a round complexity equal to $O(1)$, where $\mu = \sum_{i=1}^{N}(1 - (1 - p_{\mathsf{a}_i})^N)$.*

The proof of this theorem follows trivially from Theorem 6.

The Zipf case. In the following, we assume that the access distribution is a Zipf distribution with parameter $s > 1$. This theorem follows directly from Theorem 7 and Lemma 1.

Theorem 8. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, $m = \mathsf{poly}(k)$ and $s > 1$. If $\mathcal{D}$ is a Zipf distribution $\mathcal{Z}_s$, then* DAT *has communication complexity*

$$\Theta\left(\log\left(\left({}^{N}/_{H_{N,s}}\right)^{1/s} + \log k\right) + \frac{\log k}{\log\log k}\right),$$

storage complexity $O\left(N \cdot {}^{\log k}/_{\log\log k}\right)$, and round complexity $O(1)$.

6.5 Recursive DAT

So far, we have assumed that the client stores the entire position map DX_{st} client-side, but this entails a client storage complexity that is linear in the size of the RAM.[5] A standard approach is to store the position map in multiple trees whose sizes grow geometrically as a function of the RAM size, where each tree is an instantiation of a tree-based ORAM. The number of such trees is $O(\log N)$. In this work, we simply use Path ORAM to instantiate these trees. As shown in [54, Lemma 6], the size of the stashes introduced by this recursion does not affect the analysis of the main ORAM, and the same analysis applies to DAT.

From an efficiency standpoint, the recursion has the following asymptotic impacts:

- *Client state*: the client storage complexity becomes $\omega(\log k)$ instead of $O(N)+\omega(\log k)$.
- *Communication complexity*: the communication complexity becomes

$$\Theta\left(\log(\mu + \log k) + \frac{\log k}{\log\log k} + \frac{\log^3 N}{B}\right),$$

  where B is the block size in bits. Note that for $B = \Omega(\log^3 N)$, the asymptotics reduce to those of the non-recursive DAT.
- *Round complexity*: the round complexity becomes $O(\log N)$ instead of $O(1)$, due to the logarithmic number of trees.
- *Storage complexity*: adding recursion does not asymptotically increase the storage complexity.

To obtain the concrete asymptotics for the Zipf case, we simply substitute μ with the value obtained in Lemma 1.

[5] In practice, the size of the position map in bits is small for a wide range of N.

7 Distribution-Aware Black-Box Construction

In this section, we describe our black-box *distribution-aware* construction DAB. Contrary to DAS and DAT, which are two concrete instantiations, DAB builds on an underlying ORAM in a black-box fashion. For more intuition on the construction's design, as well as the proof's technical challenges, we refer the reader to Section 1.2. Below, we provide the details of the construction, followed by our security, correctness, and efficiency analysis. Due to space limitations, the pseudocode of DAB is deferred to the full version, along with all proofs in this section.

Setup. During setup, the client partitions the input address space $\mathbb{A}$ into a high-probability subset $\mathbb{A}^h$ and a low-probability subset $\mathbb{A}^\ell$ using the $\mathsf{Partition}$ algorithm, which takes the access distribution $\mathcal{D}$ as input (we discuss our instantiation of this algorithm further below). The client then initializes empty RAMs RAM^h and RAM^ℓ to hold the high- and low-probability addresses, respectively. To populate RAM^h, the client sets each high-probability address to its corresponding data value in the input RAM. Similarly, it populates RAM^ℓ by setting each address in $\mathbb{A}^\ell$ to its corresponding data value in RAM. Next, the client encrypts RAM^h and RAM^ℓ using Σ_{RAM}, obtaining the encrypted RAMs ERAM^h and ERAM^ℓ. It also initializes an empty queue Q and a counter $\mathsf{count} := 0$. At the end of Setup, the client outputs the keys (K^h, K^ℓ) and the corresponding states (st^h, st^ℓ) produced by Σ_{RAM}, together with the queue Q and the counter count. The server outputs the combined encrypted RAM $\mathsf{ERAM} = (\mathsf{ERAM}^h, \mathsf{ERAM}^\ell)$.

Access. Given an operation op, the client first increments the counter count, then runs the access protocol of Σ_{RAM} on ERAM^h and obtains the response r. If the response is $\perp$ (i.e., the address was not found in RAM^h), the operation op is added to the queue Q. Whenever the counter count reaches n, the client executes λ accesses on ERAM^ℓ: for each access, it dequeues an operation op from Q and executes it on ERAM^ℓ. If the queue empties before all λ accesses are complete, the client queries $\perp$ for the remaining accesses. After the λ accesses, the client resets the counter count to 0.

Partitioning. The partitioning scheme determines which addresses are considered high- or low-probability according to the distribution $\mathcal{D}$. In DAB, we use the partitioning algorithm $\mathsf{Partition}$ described in Figure 2. The goal is to choose a partition point in the address space $\mathbb{A}$ of the input RAM that minimizes the amortized communication complexity of DAB. Specifically, if the permutation π orders the addresses by decreasing access probability, the partition point is the index $x \in \{1, \ldots, N\}$ such that the addresses $\mathsf{a}_{\pi(1)}, \ldots, \mathsf{a}_{\pi(x)}$ are assigned to $\mathbb{A}^h$ and the addresses $\mathsf{a}_{\pi(x+1)}, \ldots, \mathsf{a}_{\pi(N)}$ are assigned to $\mathbb{A}^\ell$. For such a partition point x, RAM^h contains x addresses and RAM^ℓ contains $N - x$ addresses.

Recall that the total communication for n accesses to DAB consists of n accesses to ERAM^h and λ accesses to ERAM^ℓ. The amortized communication of

– $\mathsf{Partition}(\mathcal{D})$

1. parse $\mathcal{D}$ as $(p_1, \ldots, p_N)$;
2. let π be the permutation that orders $(p_1, \ldots, p_N)$ in a decreasing order such that
$$p_{\pi(1)} \geq p_{\pi(2)} \cdots \geq p_{\pi(N)}$$
and set $\alpha(x, N) = \sum_{i=x+1}^{N} p_{\pi(i)}$.
3. define the function f for all $x \in \{1, \ldots, N\}$, such that
$$f(x) = \mathsf{Com}_{\Sigma}[x] + 3\alpha(x, N) \cdot \mathsf{Com}_{\Sigma}[N - x]$$
4. compute the global minimizer of f such that
$$\gamma := \arg\min_{x \in S} f(x)$$
where,
$$S := \left\{ x : f(x+1) - f(x) \leq 0 \text{ and } f(x+2) - f(x+1) > 0 \right\}$$
5. set $\mathbb{A}^h := \{\mathsf{a}_{\pi(1)}, \ldots, \mathsf{a}_{\pi(\gamma)}\}$ and $\mathbb{A}^\ell := \{\mathsf{a}_{\pi(\gamma+1)}, \ldots, \mathsf{a}_{\pi(N)}\}$;
6. output $\mathbb{A}^h$ and $\mathbb{A}^\ell$.

Fig. 2: Distribution-aware partitioning scheme.

DAB can therefore be written as

$$f(x) = \mathsf{Com}_{\Sigma}[x] + \frac{\lambda}{n} \cdot \mathsf{Com}_{\Sigma}[N - x],$$

where $\mathsf{Com}_{\Sigma}[N]$ denotes the communication complexity of Σ_{RAM} for a RAM of size N. By Theorem 10, the ratio λ/n can be set to $3\alpha(x, N)$, where $\alpha(x, N)$ is the total access probability of the low-probability addresses $\mathsf{a}_{\pi(x+1)}, \ldots, \mathsf{a}_{\pi(N)}$. Therefore, the index γ that minimizes f is the optimal partition point for minimizing the amortized communication complexity of DAB. Finally, the partitioning scheme outputs $\mathbb{A}^h$ and $\mathbb{A}^\ell$.

7.1 Security and Correctness Analysis

In this section, we analyze the security and correctness of our distribution-aware construction DAB. While our security theorem (Theorem 9) holds for all adversarially chosen access sequences, our correctness theorem (Theorem 10) holds only for access sequences drawn from the distribution $\mathcal{D}$ chosen at setup time.

Theorem 9. *If Σ_{RAM} is an address-space agnostic oblivious RAM, then* DAB *is a distribution-aware oblivious RAM, for all $\lambda, n \in \mathbb{N}_{>0}$.*

We now state our correctness theorem.

Theorem 10. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, $m = \mathsf{poly}(k)$, and let $\mathcal{D} = (p_{\mathsf{a}_1}, \ldots, p_{\mathsf{a}_N})$ be a discrete distribution over the support $\mathbb{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_N\}$. If Σ_{RAM} is a correct oblivious RAM, $n = \omega(\log k / \alpha(\gamma, N))$, and $\lambda = \omega(\log k)$, then DAB is distribution-aware correct with n-length delay over $\mathcal{D}$, where $(\mathbb{A}^h, \mathbb{A}^\ell) \leftarrow \mathsf{Partition}(\mathcal{D})$, $\gamma = \#\mathbb{A}^h$, and $\alpha(\gamma, N) = \sum_{\mathsf{a} \in \mathbb{A}^\ell} p_{\mathsf{a}}$.*

7.2 Asymptotic Analysis

We first describe the black-box asymptotic complexity of DAB, followed by a concrete analysis.

Black-box analysis. We denote by $\mathsf{State}_\Sigma[N]$, $\mathsf{Size}_\Sigma[N]$, and $\mathsf{Com}_\Sigma[N]$ the client state size, the server storage complexity, and the communication complexity of an address-space agnostic oblivious RAM Σ on an N-sized RAM, respectively, for all $N \in \mathbb{N}$. For a fixed access distribution $\mathcal{D}$, DAB has client storage

$$O\left(\mathsf{State}_\Sigma[\gamma] + \mathsf{State}_\Sigma[N - \gamma] + \lambda\right),$$

where $(\mathbb{A}^h, \mathbb{A}^\ell) \leftarrow \mathsf{Partition}(\mathcal{D})$ and $\gamma = \#\mathbb{A}^h$. The server storage complexity is

$$O\left(\mathsf{Size}_\Sigma[\gamma] + \mathsf{Size}_\Sigma[N - \gamma]\right),$$

and the (amortized) communication complexity is

$$O\left(\mathsf{Com}_\Sigma[\gamma] + \frac{\lambda}{n} \cdot \mathsf{Com}_\Sigma[N - \gamma]\right),$$

where λ is the maximum queue size and n is the epoch length.

Concrete analysis. We now consider the case where Σ_{RAM} is instantiated using the recursive variant of Path ORAM [54]. Recall that Path ORAM has client storage complexity $\omega(\log k)$, communication complexity $O(\log^2 N)$, and server storage complexity $O(N)$.

Theorem 11. *Let $N \in \mathbb{N}$, $k \in \mathbb{N}_{>0}$, $m = \mathsf{poly}(k)$, and $s > 1$. If $\mathcal{D}$ is a Zipf distribution $\mathcal{Z}_s$ and Σ_{RAM} is instantiated using the recursive variant of Path ORAM, then DAB has client storage complexity $\omega(\log k)$, server storage complexity $O(N)$, communication complexity $O\left(\log^2\left(\log^2 N / H_{N,s}\right)\right)$, and delay $\omega\left(\log k \cdot \log^2 N / \log\log N\right)$.*

8 Evaluation

To demonstrate the practical improvement of our schemes, we implemented a simulation of our tree-based protocols as well as Path ORAM [54] in C++. Throughout our experiments, we utilize the Zipf distribution with the specific parameter s over the address space $\{1, \ldots, N\}$.

8.1 Computation and Communication

Table 1 and Table 2 show the (amortized) per-access computation and communication, as well as the total communication of our schemes when compared to the Path ORAM baseline on $N = 2^{17}$ elements, a block size $B = 4$KB, for various Zipf distributions with parameter $s \in \{1.5, 2, 3\}$.

Computation. In the non-recursive setting, DAT achieves a 16–25× speedup over Path ORAM. This computation gain stems from the fact that each operation scans through less memory due to the smaller height of the tree. Interestingly, this gain is further improved because most of the tree can fit inside of the L3 cache. Unlike Path ORAM, the number of cache misses significantly decreases, thus improving the construction not captured by the protocol directly. Similarly, DAB instantiated with Path ORAM achieves a 3.7–8.8× speedup, also improving with higher skew. This is because the cost of full accesses is amortized across many accesses to the smaller ORAM. In the recursive setting, recursive-DAT achieves a 4.4–4.7× speedup over recursive Path ORAM, while recursive-DAB instantiated with Path ORAM achieves a 4.7–11× speedup.

Communication. Path ORAM transfers a fixed 720 KB per access independent of the distribution. DAT achieves better access communication but comparable or lower total communication (when accounting for the deterministic eviction), with the improvement growing as the distribution's concentration reduces the effective tree height. DAB delivers up to 4.3× improvement over Path ORAM. In the recursive setting, DAB with the recursive variant of Path ORAM reduces bandwidth up to 6.1×, while the recursive variant of DAT has comparable communication to recursive Path ORAM. This is because DAT requires using standard Path ORAM for each of the recursive layers, since the uniform mapping no longer has the same skew of the original distribution.

A note on delays. While DAB has better computation time and communication than DAT and Path ORAM, it has worse latency. This metric depends on the temporal distribution of accesses and can vary substantially across workloads: in some cases, inter-arrival times can range from milliseconds to seconds. DAB is therefore best suited to applications that can tolerate response delay.

Table 1: Non-Recursive ORAM Schemes Performance

Scheme	Zipf $s = 1.5$			Zipf $s = 2$			Zipf $s = 3$		
	Comp. (μs)	Comm. (KB)	Total Comm.	Comp. (μs)	Comm. (KB)	Total Comm.	Comp. (μs)	Comm. (KB)	Total Comm.
Path ORAM	53.05	360	720	52.43	360	720	52.10	360	720
DAT	3.18	244	732	2.48	212	636	2.04	164	492
DAB	14.49	180	367	8.59	120	247	5.89	80	167

Table 2: Recursive ORAM Schemes Performance

Scheme	Zipf $s = 1.5$			Zipf $s = 2$			Zipf $s = 3$		
	Comp. (μs)	Comm. (KB)	Total (KB)	Comp. (μs)	Comm. (KB)	Total (KB)	Comp. (μs)	Comm. (KB)	Total (KB)
Path ORAM	69.25	520	1040	67.97	520	1040	67.89	520	1040
DAT	15.70	488	1052	14.77	424	956	14.39	328	812
DAB	14.77	180	370	8.92	120	250	6.15	80	170

8.2 Node Load Distribution

Figure 3 shows the average node load per level in DAT for various internal node sizes, over $N = 2^{17}$ accesses. Notably, the average capacity of the internal nodes is lower than in the corresponding experiment for Path ORAM [54, Figure 6], especially around levels 6 through 11, where the average node load is extremely low. This suggests that the height of the upper tree can be set shorter than its current value; in particular, we believe that the computation of h in Theorem 6 can potentially be tightened.

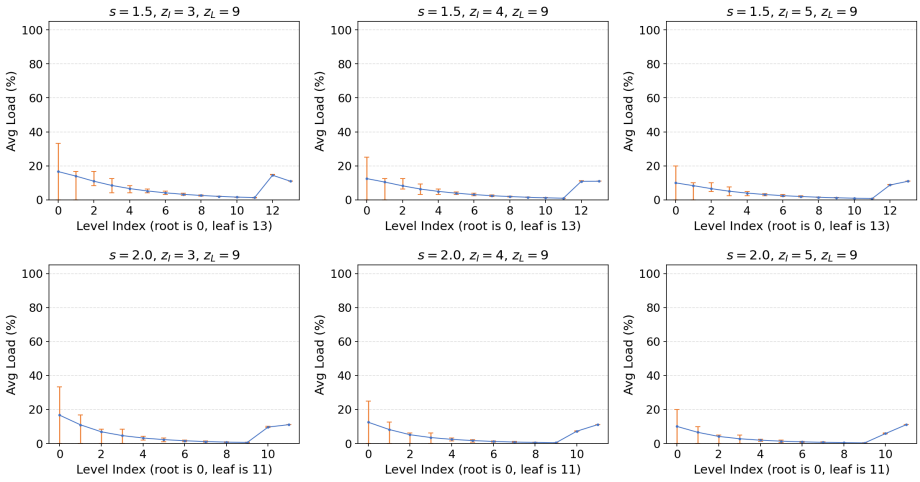


Fig. 3: Average node load of each level for different internal node size with middle quartile error bars for domain size $N = 2^{17}$ and Zipf parameter $s \in \{1.5, 2\}$.

AI Tool Use Disclosure. The authors used generative AI tools to assist in drawing Figure 1 and coding for the tables and the figures in the evaluation.

References

1. Arpaci, O.E., Kerschbaum, F., Maiyya, S.: Cloak: Heuristic oram optimization through fixed temporal distribution. arXiv preprint arXiv:2605.27565 (2026)
2. Asharov, G., Komargodski, I., Lin, W.K., Nayak, K., Peserico, E., Shi, E.: Optorama: optimal oblivious ram. In: Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30. pp. 403–432. Springer (2020)
3. Asharov, G., Komargodski, I., Michelson, Y.: Futorama: A concretely efficient hierarchical oblivious ram. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 3313–3327 (2023)
4. Bahi, A., Minaud, B., Moataz, T.: Resizable oblivious ram. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 273–302. Springer (2026)
5. Boyle, E., Chung, K.M., Pass, R.: Oblivious parallel ram and applications. In: Theory of Cryptography Conference. pp. 175–204. Springer (2015)
6. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: Evidence and implications. In: IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320). vol. 1, pp. 126–134. IEEE (1999)
7. Cash, D., Drucker, A., Hoover, A.: A lower bound for one-round oblivious ram. In: Theory of Cryptography Conference. pp. 457–485. Springer (2020)
8. Cash, D., Liu, F.H., O'Neill, A., Zhandry, M., Zhang, C.: Parameter-hiding order revealing encryption. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 181–210. Springer (2018)
9. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16. pp. 577–594. Springer (2010)
10. Chen, S., Ailamaki, A., Athanassoulis, M., Gibbons, P.B., Johnson, R., Pandis, I., Stoica, R.: Tpc-e vs. tpc-c: Characterizing the new tpc-e benchmark via an i/o comparison study. ACM Sigmod Record **39**(3), 5–10 (2011)
11. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154 (2010)
12. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM conference on Computer and communications security. pp. 79–88 (2006)
13. Demaine, E.D., Harmon, D., Iacono, J., Kane, D., Pătraşcu, M.: The geometry of binary search trees. In: Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09). pp. 496–505. Society for Industrial and Applied Mathematics (2009)
14. Demaine, E.D., Harmon, D., Iacono, J., Pătraşcu, M.: Dynamic optimality—almost. SIAM Journal on Computing **37**(3), 799–813 (2007). https://doi.org/10.1137/S0097539705447347, https://doi.org/10.1137/S0097539705447347
15. Eskandarian, S., Zaharia, M.: Oblidb: Oblivious query processing for secure databases. Proceedings of the VLDB Endowment **13**(2) (2019)

16. Espiritu, Z., Kamara, S., Moataz, T., Park, A.: {PolySys}: an algebraic leakage attack engine. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 3357–3376 (2025)
17. Fletcher, C.W., Dijk, M.v., Devadas, S.: A secure processor architecture for encrypted computation on untrusted programs. In: Proceedings of the seventh ACM workshop on Scalable trusted computing. pp. 3–8 (2012)
18. Fletcher, C.W., Ren, L., Kwon, A., Van Dijk, M., Devadas, S.: Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 103–116 (2015)
19. Garg, S., Mohassel, P., Papamanthou, C.: TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption, pp. 563–592. Springer Berlin Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_20, http://dx.doi.org/10.1007/978-3-662-53015-3_20
20. Gentry, C., Halevi, S., Jutla, C., Raykova, M.: Private database access with he-over-oram architecture. In: Applied Cryptography and Network Security: 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers 13. pp. 172–191. Springer (2015)
21. George, M., Kamara, S., Moataz, T.: Structured encryption and dynamic leakage suppression. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 370–396. Springer (2021)
22. George, M., Kamara, S., Moataz, T., Espiritu, Z.: Structured encryption and distribution-aware leakage suppression. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 203–237. Springer (2025)
23. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) **43**(3), 431–473 (1996)
24. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 513–524 (2012)
25. Grubbs, P., Khandelwal, A., Lacharité, M.S., Brown, L., Li, L., Agarwal, R., Ristenpart, T.: Pancake: Frequency smoothing for encrypted data stores. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2451–2468 (2020)
26. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proceedings of the IRE **40**(9), 1098–1101 (1952)
27. Kamara, S., Kati, A., Moataz, T., Schneider, T., Treiber, A., Yonli, M.: Sok: Cryptanalysis of encrypted search with leaker–a framework for leakage attack evaluation on real-world data. In: 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P). pp. 90–108. IEEE (2022)
28. Kamara, S., Moataz, T.: Sql on structurally-encrypted databases. In: Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part I 24. pp. 149–180. Springer (2018)
29. Kamara, S., Moataz, T.: Computationally volume-hiding structured encryption. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 183–213. Springer (2019)
30. Kamara, S., Moataz, T., Ohrimenko, O.: Structured encryption and leakage suppression. In: Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38. pp. 339–370. Springer (2018)

31. Kato, F., Cao, Y., Yoshikawa, M.: Olive: Oblivious federated learning on trusted execution environment against the risk of sparsification. Proc. VLDB Endow. **16**(10), 2404–2417 (2023). https://doi.org/10.14778/3603581.3603583, https://www.vldb.org/pvldb/vol16/p2404-kato.pdf
32. Katz, J., Lindell, Y.: Introduction to modern cryptography: principles and protocols. Chapman and hall/CRC (2007)
33. Kellaris, G., Kollios, G., Nissim, K., O'neill, A.: Generic attacks on secure outsourced databases. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1329–1340 (2016)
34. Keller, M., Scholl, P.: Efficient, oblivious data structures for mpc. In: Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20. pp. 506–525. Springer (2014)
35. Knuth, D.E.: Optimum binary search trees. Acta Informatica **1**(1), 14–25 (1971). https://doi.org/10.1007/BF00264289
36. Krashakov, S.A., Teslyuk, A.B., Shchur, L.N.: On the universality of rank distributions of website popularity. Computer Networks, **50**, 1769–1780 (2006), https://arxiv.org/pdf/cs/0404010.pdf
37. Lacharité, M.S., Minaud, B., Paterson, K.G.: Improved reconstruction attacks on encrypted data using range query leakage. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 297–314. IEEE (2018)
38. Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious ram lower bound! In: Annual International Cryptology Conference. pp. 523–542. Springer (2018)
39. Larsen, K.G., Simkin, M., Yeo, K.: Lower bounds for multi-server oblivious rams. In: Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18. pp. 486–503. Springer (2020)
40. Lehmkuhl, R., Henzinger, A., Corrigan-Gibbs, H.: Distributional private information retrieval. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 3377–3396 (2025)
41. Leutenegger, S.T., Dias, D.: A modeling study of the tpc-c benchmark. ACM Sigmod Record **22**(2), 22–31 (1993)
42. Li, W.: Random texts exhibit zipf's-law-like word frequency distribution. IEEE Transactions on information theory **38**(6), 1842–1845 (1992)
43. Mehlhorn, K.: Nearly optimal binary search trees. Acta Inf. **5**(4), 287–295 (Dec 1975). https://doi.org/10.1007/BF00264563, https://doi.org/10.1007/BF00264563
44. Moataz, T., Mayberry, T., Blass, E.O., Chan, A.H.: Resizable tree-based oblivious ram. In: Financial Cryptography and Data Security: 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers 19. pp. 147–167. Springer (2015)
45. Patel, S., Persiano, G., Raykova, M., Yeo, K.: Panorama: Oblivious RAM with logarithmic overhead. In: Thorup, M. (ed.) 59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018. pp. 871–882. IEEE Computer Society (2018). https://doi.org/10.1109/FOCS.2018.00087, https://doi.org/10.1109/FOCS.2018.00087
46. Patel, S., Persiano, G., Yeo, K., Yung, M.: Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. pp. 79–93 (2019)

47. Persiano, G., Yeo, K.: Lower bounds for differentially private rams. In: Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38. pp. 404–434. Springer (2019)
48. Persiano, G., Yeo, K.: Limits of breach-resistant and snapshot-oblivious rams. In: Annual International Cryptology Conference. pp. 161–196. Springer (2023)
49. Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., Van Dijk, M., Devadas, S.: Constants count: Practical improvements to oblivious {RAM}. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 415–430 (2015)
50. Sasy, S., Gorbunov, S., Fletcher, C.W.: Zerotrace : Oblivious memory primitives from intel SGX. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society (2018), https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_02B-4_Sasy_paper.pdf
51. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious ram with o ((log n) 3) worst-case cost. In: International Conference on The Theory and Application of Cryptology and Information Security. pp. 197–214. Springer (2011)
52. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the ACM (JACM) **32**(3), 652–686 (1985)
53. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceeding 2000 IEEE symposium on security and privacy. S&P 2000. pp. 44–55. IEEE (2000)
54. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 299–310 (2013)
55. Stefanov, E., Dijk, M.v., Shi, E., Chan, T.H.H., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. Journal of the ACM (JACM) **65**(4), 1–26 (2018)
56. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS (2014)
57. Stefanov, E., Shi, E., Song, D.: Towards practical oblivious ram. In: Proc. Netw. Distrib. Syst. Security,. pp. 1–19 (2012)
58. Storage Performance Council: SPC benchmark-1 (SPC-1) specification. http://www.storageperformance.org/specs/ (2020)
59. Wang, X.S., Huang, Y., Chan, T.H., Shelat, A., Shi, E.: Scoram: oblivious ram for secure computation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 191–202 (2014)
60. Yang, L., Mei, Q., Zheng, K., Hanauer, D.A.: Query log analysis of an electronic health record search engine. In: AMIA annual symposium proceedings. vol. 2011, p. 915 (2011)
61. Zheng, L., Zhang, Z., Dong, W., Zhang, Y., Wu, Y., Wang, C.: H2O2RAM: A high-performance hierarchical doubly oblivious RAM. In: Bauer, L., Pellegrino, G. (eds.) 34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025. pp. 8501–8520. USENIX Association (2025), https://www.usenix.org/conference/usenixsecurity25/presentation/zheng