

Optimal Statistical Oblivious RAM and Search Tree in the External-Memory Model

Andrew Krapivin
Carnegie Mellon University
andrew@krapivin.net

Elaine Shi
Carnegie Mellon University
elaineshi@cmu.edu

Abstract

Oblivious RAM (ORAM) allows a trusted client to access (encrypted) data stored in untrusted memory, such that the access patterns reveal nothing about the data or the queries. It is well-known that any ORAM scheme must suffer from a logarithmic overhead. More precisely stated, each request must incur $\Omega(\log_B N)$ block transfers on average, where N is the size of the logical memory and B is the block size of the oblivious implementation. Moreover, the same lower bound extends to oblivious Search Trees (ST).

For upper bounds, however, we have been long stuck with a $(\log_B N)^2$ curse on two fronts:

- Log-squared curse for statistically secure ORAMs: although optimality is known for *computationally* secure ORAMs for $B = \Theta(1)$ or $B \geq \log^{1+\epsilon} N$ (where $\epsilon > 0$ is an arbitrarily small constant), progress has been stagnant when it comes to *statistical* security.
- Log-squared curse for oblivious search trees: for search trees, we do not know how to overcome the log-squared curse *even when allowing computational assumptions*.

In this paper, we break this long-standing log-squared curse for both oblivious search trees and statistically secure ORAM on broad parameter regimes:

- Our first result gives an optimal, statistically secure ORAM with $O(\log_B N)$ cost as long as $B \geq \log^{1+\epsilon} N$.
- Our second result gives an optimal, statistically secure oblivious search tree with $O(\log_B N)$ cost when $B \geq \log^{2+\epsilon} N$, matching the performance of the best known non-oblivious search trees in this parameter region. Notably, this means that for search trees, we can get obliviousness *for free*!

The assumptions on B required to achieve optimality are standard and mild. They not only coincide with the standard *wide cache-line* assumption prevalent in the classical algorithms literature, but also align well with the hardware-enclave architectures that employ oblivious algorithms in practice, such as the implementations of Signal and Meta.

Contents

1	Introduction	1
1.1	Our Results and Contributions	1
1.2	Additional Related Work	3
2	Technical Overview	4
2.1	Statistical ORAM	5
2.2	Statistical Oblivious Search Tree	8
3	Preliminaries	12
3.1	External-Memory Random-Access Machine	12
3.2	Oblivious Simulation	13
3.3	Existing Building Blocks	14
4	Semi-Functional ORAM	15
4.1	Non-Oblivious Meta-Algorithm	15
4.2	Analysis of the Semi-Functional ORAM's Meta-Algorithm	18
4.3	Oblivious Implementation of the Meta-Algorithm	21
4.4	Efficiency Analysis	24
4.5	Proofs for the Semi-Functional ORAM	25
5	Full Oblivious RAM Construction	27
5.1	Modifications to the Semi-Functional ORAM	27
5.2	Full ORAM Construction	30
6	Proofs for the Full ORAM	31
6.1	A Limited-Independence Occupancy Bound	31
6.2	Setup for the ORAM Analysis	32
6.3	Failure and Stash Bounds	35
6.4	Rapid Decrease on Every Time Interval	37
6.5	Correctness, Obliviousness, and Efficiency	39
7	Oblivious Search Tree	40
7.1	Parameters and Notation	40
7.2	Data Structure	41
7.3	Non-Oblivious Meta-Algorithm	42
7.4	Correctness of the Meta-Algorithm	44
7.5	Oblivious Implementation of the Meta-Algorithm	47
7.6	Efficiency Analysis	51
7.7	Putting It Together	52
A	Achieving Linear Space	57
A.1	Blueprint	57
A.2	Externally Indexed Bucket ORAM for Super-Words	58
A.3	Block Layout for the Externally Indexed Bucket ORAM	59

B	Extension to Unordered Dictionary	60
B.1	Construction	60
B.2	Proofs	61
C	Extension to General Private Cache Size M	63
C.1	Oblivious RAM and Dictionary	63
C.2	Oblivious Search Tree	64

1 Introduction

Oblivious algorithms, first proposed by Goldreich and Ostrovsky [GO96, Gol87], allow a trusted client (e.g., a hardware enclave or a user’s mobile phone) to access (encrypted) data stored on untrusted storage (e.g., memory or disk, or cloud server), such that the access patterns reveal nothing about the secret data or the queries. Oblivious algorithms have been deployed in large-scale real-world applications such as in Signal’s private contact discovery [Con].

Oblivious RAM (ORAM) [GO96, Gol87] obviously realizes a simple memory abstraction, allowing the client to read and write a memory array in units of *words*, indexed by their addresses. It is well-known that an ORAM scheme must incur at least logarithmic slowdown [GO96], and further, this lower bound holds even when we allow cryptographic hardness assumptions [LN18]. A generalized form of this lower bound [KL21] states that, when the ORAM is implemented on an external-memory machine where memory reads and writes must occur in atomic units of *blocks*, each request must, on average, probe $\Omega(\log_B \frac{N}{M})$ blocks. Throughout the paper, we use B to denote the block size, and use $M \geq B$ to denote the size of the client’s private cache (both measured in words). Relying on standard cryptographic hardness assumptions, optimal ORAM constructions are known for most parameter regimes, tightly matching the above lower bound. Specifically, the OptORAMa work [AKL⁺20] gives an ORAM with $O(\log \frac{N}{M})$ overhead for $M \geq B = \Theta(1)$; and the subsequent work MegaBlocks [AEKL25] showed how to achieve $O(\log_B \frac{N}{M})$ overhead as long as $M \geq B \geq \log^{1+\epsilon} N$ where ϵ is an arbitrarily small positive constant. Both of these results rely on the existence of one-way functions.

A long-standing open question in the ORAM literature is whether we can achieve optimality without relying on any hardness assumptions — such an ORAM is often called *statistically secure*. Previously, the best known statistically secure ORAM, based on Circuit ORAM [WCS15] and the standard van Emde Boas layout [vEBKZ76], achieves $O((\log_B \frac{N}{M})^2)$ overhead for all choices of B and $M \geq B$. Unfortunately, this is suboptimal for nearly all parameter regimes of practical interest, except for the unrealistic setting when $B \geq N^\delta$ for some constant $\delta \in (0, 1)$.

Besides ORAM, oblivious search tree (ST) is another fundamental primitive for privacy-preserving database applications. Besides lookup, insertion, and deletion, an ST additionally supports predecessor (or successor) queries. Oblivious ST inherits the same $\Omega(\log_B \frac{N}{M})$ lower bound for ORAM. Previously, the best known oblivious ST, based on Wang et al. [WNL⁺14] and the standard van Emde Boas layout [vEBKZ76], achieves $O((\log_B \frac{N}{M})^2)$ overhead with statistical security¹. It remains open how to improve upon this bound, even when allowing computational hardness assumptions. Note that without the obliviousness requirement, STs can be realized with $O(\log_B \frac{N}{M})$ cost per query, e.g., using a B-tree [BM70].

In summary, we ask the following natural and long-standing questions:

- *Can we construct a statistically secure ORAM with optimal performance?*
- *Can we construct an oblivious ST with optimal performance?*

1.1 Our Results and Contributions

Main results. We answer the above questions affirmatively for a large parameter regime of practical interest. We now state our main theorems. Since we care about the asymptotic behavior of the algorithms for large N , we may assume without loss of generality that N is larger than

¹To get the $O((\log_B \frac{N}{M})^2)$ bound, we need to change the logical tree in Wang et al. [WNL⁺14] from an AVL tree to a B-tree. If an AVL tree is used as in their original paper, the bound would become $O(\log_B \frac{N}{M} \cdot \log \frac{N}{M})$.

the security parameter, and our theorems below hold with a statistical failure probability that is negligibly small in N .

Theorem 1.1 (Optimal statistically secure ORAM). *Suppose $B \geq \log^{1+\epsilon} N$ where ϵ is an arbitrarily small positive constant. There exists a statistically secure ORAM that supports obliviously reading and writing the contents at any given address, requiring $O(1 + \log_B \frac{N}{M})$ amortized cost per operation.*

Theorem 1.1 can be viewed as a strict improvement over MegaBlocks [AEKL25], in which the same bound was shown with the same assumption on B , but relying on pseudorandom functions (PRFs).

To obtain Theorem 1.1, we develop new algorithmic techniques that depart substantially from prior ORAMs. Previous hierarchical ORAMs [GO96, Gol87] use a pseudorandom function (PRF) for three purposes: locating items, hiding their locations, and balancing the load across buckets. We separate these roles, using a k -wise independent hash for load balancing and true randomness for security. Because these random choices cannot themselves be recomputed to locate an item, we introduce a new indexing structure that supports efficient lookups.

Theorem 1.2 (Optimal statistically secure oblivious ST). *Suppose $B \geq \log^{2+\epsilon} N$ where ϵ is an arbitrarily small positive constant. There exists a statistically secure oblivious search tree that supports lookup, predecessor, successor queries, as well as insertions and deletions of key-value pairs, requiring $O(1 + \log_B \frac{N}{M})$ amortized cost per operation.*

By contrast, no optimal oblivious ST was previously known, even under computational hardness assumptions. Our construction shows that, when $B \geq \log^{2+\epsilon} N$, obliviousness for search trees comes essentially *for free*: its performance matches that of the best known non-oblivious ST [BM70]. Previously, comparable free-lunch results were known only for less powerful data structures, such as priority queues [JLS21, MS26], which do not imply ORAM. An oblivious ST, by contrast, extends the ORAM’s interface: it supports lookup, insertion, and deletion by keys from a universe much larger than the address space, together with predecessor and successor queries. Thus, Theorem 1.2 shows that optimal performance remains possible with this richer functionality and without computational assumptions.

In both of the above theorems, we make no assumption on M (except that $M \geq \Theta(B)$), and our assumptions on the block size B are very mild: they not only coincide with the standard *wide cache-line* assumption prevalent in the external-memory algorithms literature [AV88, Dem02], but also fit real-life architectures where oblivious algorithms are deployed. For example, Signal and Meta’s implementations [Con, met] employ hardware enclaves, where the block corresponds to a page, typically 4KB in size.

Our work is motivated primarily by the theoretical question of asymptotic optimality for ORAM and oblivious algorithms; we do not claim that our schemes are practical in their current forms. Recent works [AKM23, ZZD⁺25] have translated ideas from asymptotically optimal computationally secure ORAMs [PPRY18, AKL⁺20] into concretely efficient constructions, demonstrating that theoretical advances can ultimately yield concrete improvements. A natural direction for future work is to investigate whether the techniques introduced here can undergo a similar development.

Additional results. In Section B, we additionally extend our ORAM to obtain an oblivious unordered dictionary with the same asymptotic costs, as stated in the following corollary. Unlike ORAM which has a predetermined contiguous address space, a dictionary may store keys that are discontinuous and drawn from a significantly larger universe.

Corollary 1.3 (Optimal statistically secure unordered dictionary). *There exists a statistically oblivious unordered dictionary that achieves $O(1 + \log_B \frac{N}{M})$ amortized cost for each lookup, insertion, and deletion query, as long as $B \geq \log^{1+\epsilon} N$ where ϵ is an arbitrarily small positive constant.*

1.2 Additional Related Work

Application of oblivious algorithms and metrics. We formulate oblivious algorithms in the external-memory model, with block size B and cache size M , both measured in words. Prior work has considered three broad application settings, each of which motivates different choices of M and B and a different primary cost metric:

1. *Multi-party computation* [GKK⁺12, WCS15, ZE13, LWN⁺15], which commonly uses the word-RAM setting $M = B = \Theta(1)$. Here each instruction operates on $O(1)$ words and each memory probe transfers $O(1)$ words. Thus running time, block-probe complexity, and the number of words transferred (or *bandwidth*) are all united. This alignment is especially appropriate for MPC, which realizes instructions and memory accesses through cryptographic computation whose cost grows with every bit of computation and communication.
2. *Privacy-preserving storage outsourcing* [SSS12, GMOT12, SS13, SZE⁺16, RFK⁺15], where $M \geq B = \Theta(1)$. Bandwidth is typically the bottleneck in this setting [SvDS⁺13, SS13]. Taking $B = \Theta(1)$ makes the number of block probes coincide asymptotically with the number of words transferred, which is the quantity these schemes seek to minimize. The client, such as a laptop or smartphone, may have a moderate amount of local storage, motivating the more general choice $M \geq B$. Prior work [SvDS⁺13] further observes that, when application payloads are large, the ORAM components handling payloads and metadata may use machines with different word sizes.
3. *Hardware enclaves* [TGS23, Con, met], which are naturally captured by the external-memory model. In this setting, the standard wide-cache-line assumption $B \geq \log^{c_1} N$ and tall-cache assumption $M \geq B^{c_2}$ are appropriate for suitable constants $c_1, c_2 \geq 1$ [AV88]. Some empirical studies [TGS23] have demonstrated that the number of block probes (also called the page swap overhead) is a dominating cost in this setting.

Known lower-bound techniques work only in a so-called cell-probe model [LN18, KL21], and establish bounds only on block probes (and on bandwidth when these metrics coincide). Since our work is motivated primarily by the theoretical exploration of optimality, we focus on optimizing the number of block probes for which lower bounds are well-understood. We do not claim optimality in bandwidth or running time. Thus, our metric and parameter regime primarily target hardware enclaves, including the settings of the Signal and Meta implementations [Con, met], rather than MPC or storage-outsourcing applications whose principal cost metrics differ.

Our schemes achieve optimality regardless of the word size w (measured in bits), as long as the standard condition $w \geq \log N$ holds, that is, a word should be large enough to store its own address. By contrast, some earlier works on oblivious storage of large files or records [SS13, RFK⁺15] additionally assume a moderately large word size w . More precisely, for general word size w , the previously best known statistical ORAM achieves $O(\log_B \frac{N}{M} \cdot \log_{B'} \frac{N}{M'})$ cost per query [WCS15, vEBKZ76] where $B' = B \cdot \frac{w}{\log N}$ and $M' = M \cdot \frac{w}{\log N}$. On the other hand, the best known oblivious ST does not similarly benefit from the large word size and its per-query cost is simply $O((\log_B \frac{N}{M})^2)$ [WNL⁺14, vEBKZ76].

Brief history of oblivious algorithms. Goldreich and Ostrovsky [GO96, Gol87] constructed the first ORAM scheme with polylogarithmic overhead, assuming the existence of PRFs. Moreover, they proposed the hierarchical framework, which was in turn inspired by Bentley and Saxe’s technique for making a static data structure dynamic [BS80]. The hierarchical paradigm was later adopted in various subsequent works [GMOT12, GM11, AKL⁺20, AEKL25] for obtaining computationally secure ORAM with better performance bounds. The first statistically secure ORAMs were proposed by Ajtai [Ajt10] and Damgård et al. [DMN11]. Later, Shi et al. [SCSL11] introduced a new binary-tree paradigm for constructing ORAMs. This framework was adopted by subsequent works [SvDS⁺13, WCS15, CS17] which eventually improved the overhead of statistically secure ORAMs to $O((\log_B \frac{N}{M})^2)$ [WCS15].

Goldreich and Ostrovsky [GO96] proved a logarithmic lower bound on ORAM. Specifically, they showed that for $B = 1$, any ORAM scheme must incur at least $\Omega(\log_M N)$ cost per request. However, Goldreich and Ostrovsky’s lower bound applies only to statistically secure ORAMs in the balls-and-bins model. Subsequently, Larsen and Nielsen [LN18] proved a lower bound of $\Omega(\frac{1}{B} \log \frac{N}{M})$ which works even for computationally secure ORAMs. However, their lower bound is only tight for $B = \Theta(1)$. Finally, Komargodski and Lin [KL21] proved a lower bound of $\Omega(\log_B \frac{N}{M})$, which strictly improves over the bound of Larsen and Nielsen for $B = \omega(1)$.

We stress that the aforementioned ORAM lower bounds [GO96, LN18, KL21] apply only when one insists on having a generic compiler that transforms *any* non-oblivious algorithm to an oblivious counterpart. For a concrete computational task, however, blindly compiling the best known non-oblivious algorithm via ORAM may not yield the most efficient oblivious solution. Task-specific oblivious algorithmic design often leads to strictly better performance. Our oblivious ST is a great example showing that obliviousness can essentially come for free when $B \geq \log^{2+\epsilon} N$. For this reason, various prior works also considered oblivious algorithms tailored for common computational tasks including sorting and shuffling [Goo14, LSX19, RS21], common data structures [ZE13, WNL⁺14, JLS21, KS14] such as queues, stacks, and priority queues, and graph algorithms [GOT12, EGT10, BSA13, RS21, NWI⁺15]. Several works [GM11, NWI⁺15, AHR24] also considered computational models that are “obliviousness-friendly”, allowing us to capture broad classes of algorithms that are amenable to efficient oblivious simulation without going through the generic ORAM transformation.

2 Technical Overview

In this section, we briefly describe the techniques we use to build our oblivious RAM and search trees. Both of these data structures employ the hierarchical design pattern, which, at a high level, works as we now describe. For simplicity, we shall assume for now that $M = \Theta(B)$, since it is easy to generalize the result to arbitrary $M \geq B$ later.

Hierarchical ORAM. We begin with a brief overview of the hierarchical ORAM paradigm [GO96, Gol87], and why previous constructions are incompatible with statistical security.

First, the structure is composed of levels numbered $0, 1, 2, \dots, L - 1$ of geometrically increasing size and organized as buckets. For example, level 0 may have size 2 with one bucket, level 1 size 4 with 2 buckets, level 2 size 8 with four buckets, and so on. Each update of some (k, v) pair is first placed in level 0. Once level 0 fills up, level 0 is merged into level 1, assuming level 1 has space, and then every (key, value) pair is obliviously shuffled into a new “random” bucket in level 1. If level 1 does not have sufficient space for all items in level 0 and those previously in level 1, then levels 0 and 1 are merged into level 2, followed by a shuffle into random buckets in level 2. If level

2 is full, levels 0, 1, 2 are merged into level 3, followed by a shuffle, and so on. Thus, every key lies in a “random” bucket in *some* level.

A lookup of k simply checks the “random” bucket k would have gone in level 0, then the random bucket k would have gone in level 1, and so on. If at any point k is found, then all accesses to subsequent levels are to completely random locations independent of where the key k would have gone. Loosely speaking, this scheme is secure because every lookup touches exactly one bucket in every level, and every bucket it touches is a uniformly random location.

Challenges. The key difficulty in our case is identifying this “random” bucket k should go into. Computationally, this is trivial: the assignment is given by a PRF, so the client stores only its short secret key and recomputes the bucket index for any key on demand. Statistically, no such succinct description exists — writing down a truly random assignment of N keys to buckets takes $\Omega(N)$ bits, far more than the $O(B)$ words of client memory, and this even applies to the first level! At first sight, this makes the hierarchical paradigm fundamentally incompatible with statistical security. As we will see, however, there is a way out, by isolating the three purposes that the PRF serves in previous hierarchical ORAMs [GO96, Gol87], and handling them separately: we will use k -wise independent hashing for load balancing, and use random coins for security. Because the random choices cannot be efficiently recomputed on the fly to locate items during lookup, we will instead build a carefully designed index structure to locate items.

2.1 Statistical ORAM

We now describe our statistical ORAM construction. Existing hierarchical ORAMs [GO96, Gol87, KLO12, GMOT11, PPRY18, AKL⁺20, AEKL25] rely on a hash function for security, and this is why the hash function needs to be realized with a pseudorandom function (PRF). We will also employ a hash function, but we use it only for load balancing among groups, and thus the hash function can be instantiated with a $\log^{1+\epsilon} N$ -wise independent hash [CW77, SSS95, BR94].

Logical tree. As shown in Figure 1 (top), we will build a q -ary logical tree of depth $L = \lceil \log_q N \rceil + 1$, where $q = B^{\epsilon/6}$ and the root is assumed to be at level 0. Each level $\ell \in \{0, 1, \dots, L-1\}$ has q^ℓ logical tree nodes. If an item by the address `addr` resides in level $\ell > 0$ of the tree, then it must belong to a node determined by the hash of its address. Let $H : \{0, 1, \dots, N-1\} \rightarrow \{0, 1, \dots, q-1\}^{L-1}$ be a $\log^{1+\epsilon} N$ -wise independent hash function that outputs an $(L-1)$ -digit number denoted $k_1 \dots k_{L-1}$ where each digit $k_i \in \{0, 1, \dots, q-1\}$. Given `addr`, let $k_1 \dots k_{L-1} \leftarrow H(\text{addr})$. Then, `addr` must reside in the node indexed $k_1 \dots k_\ell$ if it resides in some level $\ell > 0$. In our logical tree, the level- ℓ node $k_1 \dots k_\ell$ is a direct child of the level- $(\ell-1)$ node indexed $k_1 \dots k_{\ell-1}$.

Hierarchical data structure. As shown in Figure 1 (bottom), the logical tree nodes are stored in a corresponding hierarchical data structure with levels geometrically growing in size. Level $\ell \in \{0, 1, \dots, L-1\}$ in the hierarchical data structure is allocated $\Theta(q^{\ell+3})$ space, organized into $\lceil q^\ell / Z \rceil$ buckets each of capacity $\Theta(q^3 \cdot Z)$, where $Z = \Theta(B/q^3)$. All level- ℓ tree nodes are stored in level ℓ of the hierarchical data structure. Each level ℓ of the hierarchical data structure can be viewed as an oblivious hash table that allows looking up each distinct logical tree node up to q times, before the level is rebuilt again. To achieve this, we make q clones of each node, and have every clone store the pointers to all q clones of the node’s children. Whenever a level is rebuilt, every node clone is tossed into a fresh random bucket in the corresponding level. In expectation, $qZ = \Theta(B/q^2)$ node clones land in each bucket. Each node stores q^2 pointers and $O(1)$ items in expectation. As

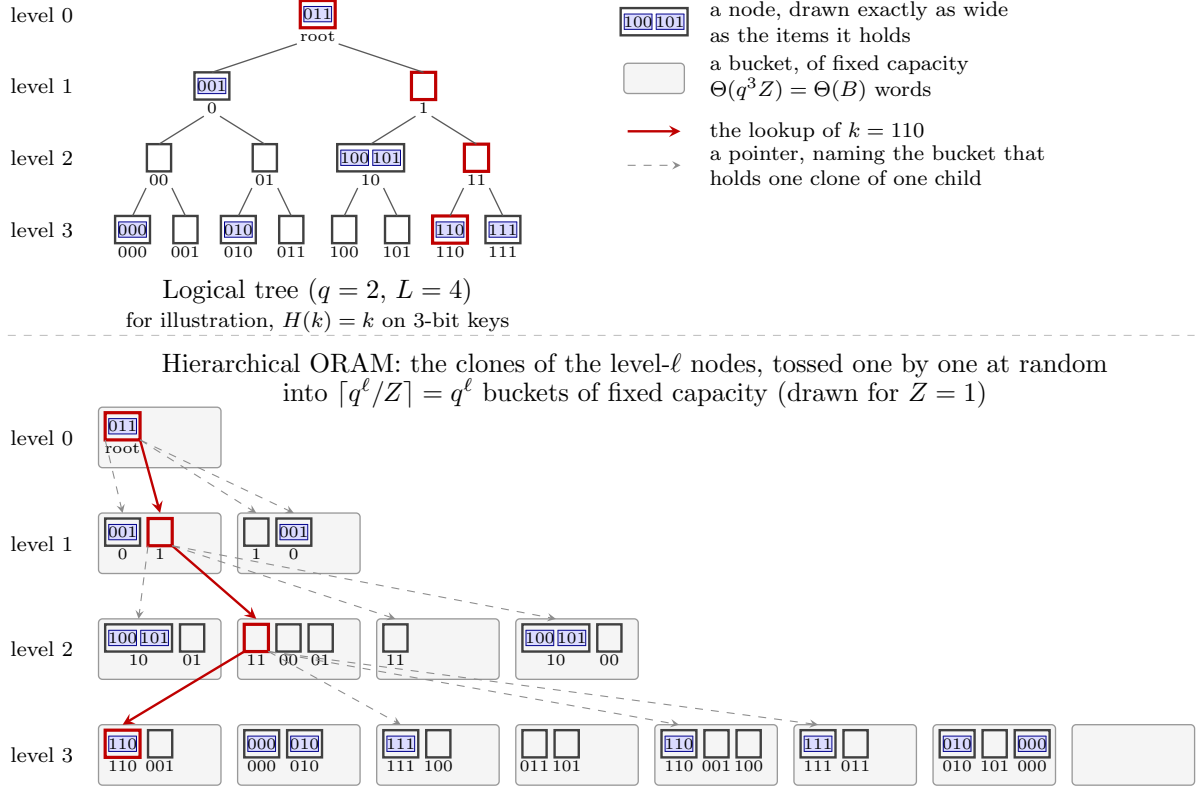


Figure 1: Our statistically secure semi-functional ORAM, drawn for $q = 2, L = 4$ and $Z = 1$, with H the identity on 3-bit keys. Nodes are not fixed-size objects: a (clone of a) node is drawn as a frame around exactly the items it holds. An item with key k stored in level ℓ sits in the node indexed by the length- ℓ prefix of $H(k)$. In our scheme, a pointer actually points to the bucket holding a clone (but we draw it as pointing to the clone itself for clarity).

long as $B \geq \log^{1+\epsilon} N$, each bucket will not exceed its storage capacity $O(q^3 \cdot Z) = O(B)$ except with negligible probability.

Look up by address. To look up an item by its address **addr**, we will probe one bucket per level. In the root level, we simply look up the single bucket. Afterwards, when we search in level ℓ , we can discover which bucket to visit in the next level $\ell + 1$. Specifically, as we scan the corresponding bucket in level ℓ , we look for an unconsumed clone of the node indexed $H(\mathbf{addr})[\ell]$, i.e., the length- ℓ prefix of $H(\mathbf{addr})$. There are three cases:

1. The desired **addr** is found in this node clone. In this case, we simply fake-read a random bucket in all future levels.
2. From the pointers stored in this clone, we discover the location (by the bucket) of an unconsumed clone of the child node $H(\mathbf{addr})[\ell + 1]$ — this is the bucket we should read in the next level.
3. We have exhausted the clones of the child we want to visit. In this case, we visit a random bucket in all future levels, and the algorithm returns “**error**”. Because our base ORAM scheme occasionally returns “**error**”, we call it a *semi-functional* ORAM. Later, we will show how to upgrade the semi-functional ORAM to a full ORAM with $1 - \text{negl}(N)$ correctness.

One question remains to be answered: as we traverse the levels, *how do we decide which pointer to a child clone is unconsumed?* We will answer this question after we introduce the rebuild procedure.

Periodic rebuild. After looking up some item, we immediately add it back to the root level. Every q^ℓ requests, we rebuild levels $0, 1, \dots, \ell + 1$. During the rebuild, all items stored in levels $0, 1, \dots, \ell + 1$ are placed into the correct node in level $\ell + 1$, and the nodes in smaller levels are emptied. Further, we re-toss the node clones in each level $0, 1, \dots, \ell + 1$ into random buckets using fresh coins. The rebuild also needs to maintain a correctness guarantee: all node clones in levels $0, 1, \dots, \ell$ need to modify their pointers accordingly to reflect the new buckets assigned to their children’s clones. Later in Section 4, we show how to accomplish the rebuild obliviously through $O(1)$ oblivious sorts per level being rebuilt.

Identify the next unconsumed clone. Given the above rebuild procedure, we can now answer the question of how to identify the next unconsumed child clone during lookup. Fix some level ℓ . As we traverse clones of nodes in the logical tree during lookup, if we still have not found the desired address addr in levels $0, \dots, \ell - 1$, we will need to consume a clone of the node indexed $H(\text{addr})[: \ell]$ in level ℓ . To discover how many clones of this node might have been consumed, we want to know since the last time level ℓ was rebuilt, how many distinct addresses we have looked up whose hashes share the same length- ℓ prefix as $H(\text{addr})$. Fortunately, all such items must reside in levels $0, 1, \dots, \ell - 1$, and along the path traversed when we look up addr . Therefore, we can tally this count once we have traversed levels 0 through $\ell - 1$. By reading the corresponding pointer in the corresponding node clone in level $\ell - 1$, we can then discover which bucket and which child clone to read in the next level ℓ .

Upgrade to full ORAM. In our semi-functional ORAM, an error is returned whenever we run out of clones at some level. The probability that this bad event happens is roughly $\exp(-\log^{\Omega(\epsilon)} N)$. We can boost the correctness probability by introducing a total of $\gamma = O(1/\epsilon)$ semi-functional ORAMs, where all instances adopt independent hash functions $H(\cdot)$. The i -th semi-functional ORAM serves as a fallback for catching any “error” exception raised by the previous $i - 1$ ORAMs. Specifically, if all previous ORAMs failed to produce a correct result, the request is then forwarded to the next fallback ORAM; else if we have already obtained a correct result, then we issue a fake request to all remaining ORAMs. Upon receiving a fake request, the semi-functional ORAM simply fake-reads a random bucket per level. Every time we fetch an item, we add the item (possibly with a newly written value) to all γ semi-functional ORAMs, so all ORAMs store up-to-date items, and they all rebuild in sync.

The most important observation is that a fake request does not consume any node clones. Later in Section 6, we prove that the number of real requests received by the semi-functional ORAMs decreases at doubly-exponential speed. More specifically, the i -th semi-functional ORAM receives $\exp\left(-\left(\log^{\Omega(\epsilon)} N\right)^{i-1}\right)$ fraction of the real requests, and the correctness failure probability at the i -th ORAM is at most $\exp\left(-\left(\log^{\Omega(\epsilon)} N\right)^i\right)$. Therefore, $\gamma = O(1/\epsilon)$ semi-functional ORAMs suffice to bring the error probability down to negligibly small. The probabilistic analysis here is similar to that of multi-tier hashing [ACMR95, CGLS17], except that we prove it using $\log^{1+\epsilon} N$ -wise independence rather than full randomness.

Finally, we note that there is an additional technicality in proving the desired probabilistic bounds. Specifically, the doubly exponential decrease in the number of real requests can be sustained only while the number of real requests remains above a certain threshold. To address this

issue, our final scheme, described in Section 5, augments each semi-functional ORAM with an additional per-level stash occupying $O(1)$ blocks. Once the doubly exponential decrease can no longer be maintained, this stash is used to capture all nodes that are about to exhaust their clones.

Efficiency. We first analyze the efficiency of a single semi-functional ORAM. The read phase visits one bucket per level, and since each bucket fits in one block, the number of block transfers is bounded by $O(\frac{1}{\epsilon} \log_B N)$. The rebuild phase is where we exploit the locality. Specifically, level ℓ is involved in a rebuild every $q^{\ell-1}$ steps, and the cost of the rebuild is upper bounded by the cost of obviously sorting an array of size $q^{\ell+3}$ which incurs $O(\frac{1}{B} \cdot q^{\ell+3} \log(q^{\ell+3}))$ block transfers. Specifically, the $\frac{1}{B}$ term embodies how the underlying oblivious sort benefits from locality, and as shown below, the $\frac{1}{B}$ term also effectively cancels out the extra $q^{O(1)} \cdot \log N$ factors in our final bound. In summary, the amortized cost of the rebuild phase is bounded by

$$\sum_{\ell=1}^{L-1} O(1) \cdot \frac{1}{q^{\ell-1}} \cdot \frac{1}{B} \cdot q^{\ell+3} \log(q^{\ell+3}) \leq O(1) \cdot \sum_{\ell=1}^{L-1} \frac{q^4}{B} \log N = O\left(\frac{1}{\epsilon} \cdot \log_B N \cdot \frac{\log N}{B^{1-2\epsilon/3}}\right)$$

For $B \geq \log^{1+\epsilon} N$ and any $\epsilon < 1/2$, the above is upper bounded by $O(\frac{1}{\epsilon} \cdot \log_B N)$.

Finally, our full ORAM is composed of $O(1/\epsilon)$ semi-functional ORAMs. Therefore, the amortized cost per query is $O(\frac{1}{\epsilon^2} \cdot \log_B N)$. The above analysis assumes that $M = \Theta(B)$. In Section C, we describe how to generalize the analysis for an arbitrary choice of $M \geq B$ while still retaining optimality.

Adaptive security. We stress that our ORAM construction satisfies adaptive security, i.e., it is secure even if the adversary can adaptively choose the queries after having observed answers to previous queries as well as the access patterns emitted so far. Intuitively, in our algorithm, except with negligibly small probability, the access patterns and the answers to the queries reveal no information about the load-balancing hash function $H(\cdot)$ — equivalently, $H(\cdot)$ is independent from the adversary’s random coins. Therefore, even an adaptive adversary cannot craft the requests to cause the load balancing properties to fail (except with negligible probability).

2.2 Statistical Oblivious Search Tree

In a search tree (ST), we can no longer employ the same load-balancing strategy as in our ORAM. Directly hashing each key to a random bucket would destroy ordering among keys. Instead, we build a logical search tree and lay it out directly in the hierarchy described above. A key issue to overcome here is that there is no longer any bound on how often a node can be accessed: as the keys are not hashed, the adversary can deliberately access a subtree as often as it wishes. Rather than maintaining clones of each node, we reinsert an entire node (including all keys, pointers, and metadata for that node) into the root if it is ever accessed, so that no node is ever accessed twice in the same location, and it no longer matters how often the adversary revisits a subtree. Moreover, a lookup of a single key will reinsert into the root all the keys involved on a path from root to leaf. While this may seem extremely wasteful, with a sufficiently large block size, the cost of these reinsertions will amortize down to less than the cost of a query. The main technical challenge of our construction will rather be to make sure that the tree is still valid after every reinsertion and rebuild.

Logical tree. All nodes have fanout anywhere from 0 to $2q$, where $q = B^{\epsilon/(3(2+\epsilon))}$. Specifically, a logical tree node has the following form:

$$\text{item}_1, ptr_1, \text{item}_2, ptr_2, \dots, \text{item}_{2q-1}, ptr_{2q-1}, \text{item}_{2q}, ptr_{2q},$$

where each ptr_i points to a subtree if it exists (and otherwise is set to null), and each item_i is either a dummy element or a key-value pair.² Henceforth, we use $\text{item}_i.k$ to refer to the item's key field. We require the standard ordering invariant to be respected just like in a standard search tree:

- $\text{item}_1.k < \text{item}_2.k < \dots < \text{item}_{2q}.k$; and
- all items in the subtree rooted at ptr_i must have keys in the half-open interval $[\text{item}_i.k, \text{item}_{i+1}.k)$ — we may assume that $\text{item}_{2q+1}.k = \infty$.

Finally, in our design we ensure one additional invariant: if a node

$$\text{item}_1^p, ptr_1^p, \text{item}_2^p, ptr_2^p, \dots, \text{item}_{2q-1}^p, ptr_{2q-1}^p, \text{item}_{2q}^p, ptr_{2q}^p$$

is a parent of

$$\text{item}_1^c, ptr_1^c, \text{item}_2^c, ptr_2^c, \dots, \text{item}_{2q-1}^c, ptr_{2q-1}^c, \text{item}_{2q}^c, ptr_{2q}^c$$

through the pointer ptr_k^p adjacent to item_k^p , then item_k^p is a *duplicate* of item_1^c — the two items are copies of the same key-value pair. Every item carrying a non-null pointer is therefore a copy of the first item of the node it points to, which is exactly what makes the interval above half-open on the left: the key $\text{item}_i.k$ itself lives in the subtree hanging off ptr_i . This is similar to the duplication that supports fractional cascading in the lookahead array [BFCF⁺07]. See the top of Figure 2 for an example logical tree.

Hierarchical representation. We store the logical tree in a hierarchical data structure where level ℓ is provisioned with $O(q^{\ell+1})$ space, so it can store q^ℓ logical tree nodes, as seen in Figure 2. Perhaps counterintuitively, the levels in the hierarchy do not directly correspond to levels of the logical tree. Instead, we only have the following (looser) guarantee: a child of a node in the logical tree resides in a level strictly below the level the parent resides in. This property is crucial to supporting queries (and insertions, deletions), which consist of the two steps:

- *Lookup*: first, a query logically traverses a path in the logical tree, stored in random locations in a subset of the levels. This corresponds to fetching exactly one node in each level, doing a real or dummy lookup depending on whether the level stores a node on a path.
- *Path reinsertion*: after the lookup, all the keys and pointers on the path are merged into the first few levels of the logical tree. Thus, nodes *not* on the fetched path but pointed to by a node on the fetched path now also belong to one of the first few levels of the logical tree. However, these nodes remain where they were in the hierarchy — for example, in Figure 2, the node (17, 18, 19, 20) got promoted in the logical tree after the rebuild but its location in the hierarchy is unchanged. This is exactly where the above property comes in: children in the logical tree are below their parent in the hierarchical representation but potentially not immediately below.

²Note that the number of items is equal to the number of pointers in a node, which deviates from the typical arrangement where there is one more pointer than item in a node. Such a layout should absolutely be possible, but this node format leads to a slightly simpler design.

Periodic rebuilds. To support a maximum of N keys, the hierarchical representation needs $L = \lceil \log_q N \rceil$ levels. Thus, each operation accesses and then reinserts a path with $2Lq$ entries (some real, some dummies) into the root. Insertions also add an additional item, so, for simplicity, we ensure that every operation reinserts $2Lq + 1$ entries, padding with dummies if necessary. We treat this reinsertion as atomic, meaning we merge all $2Lq + 1$ items into the tree as a unit. To facilitate this, levels 0 through $L^* = \lceil \log_q(10Lq) \rceil = O(1)$ are rebuilt on every operation; the constant slack in L^* also absorbs the duplicate items that a rebuilt level pushes up into the level above it. In general, for $\ell \geq 0$, level $L^* + \ell$ is rebuilt every q^ℓ operations.

A rebuild of level ℓ works as follows: first, all $(\text{item}, \text{ptr})$ pairs in the reinserted path and levels 0 through ℓ are concatenated, tagged by the level they come from — keys in the reinserted path are given *tags* according to the level they came from and -1 for a newly inserted key-value pair. The entries are then obviously sorted by the key and the tag of the item. Following this, we deduplicate all keys, preserving the value of the duplicate with the smallest tag and the pointer of the duplicate with the largest tag. We split this sorted and deduplicated array into nodes with exactly $2q$ $(\text{item}, \text{ptr})$ pairs each, except possibly for the last node. Then, we duplicate the smallest item from each node to rebuild level $\ell - 1$ (obtaining a pointer to the node each item came from). This process is repeated: we repeatedly split the current level being rebuilt into nodes, duplicating the smallest item from each node to rebuild the next level, till the entire level fits within a single node. Each level is shuffled after the level above is built and correctly stores pointers to the shuffled children node locations. Figure 2 shows a simplified rebuild following a lookup.

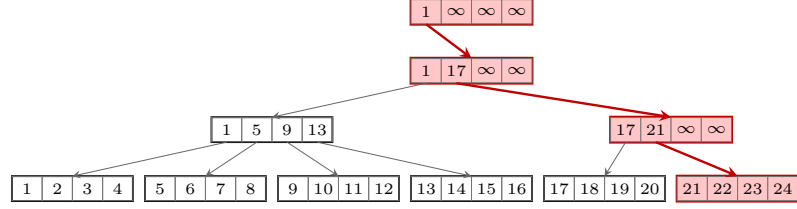
Queries, insertions, deletions. Every operation begins with a search from the root. For exact **Lookup**(k), strict **Successor**(k), insertion, and deletion, we follow the pointer paired with the largest key at most k , stopping if that pointer is null or no such key exists. We continue even upon finding an exact match: by the duplication invariant, all copies of k lie on this path. For strict **Predecessor**(k), we instead follow the pointer paired with the largest key strictly below k , so an exact separator match sends us into the preceding interval. Along the path, **Lookup** records an exact match, **Successor** records the smallest key strictly greater than k , and **Predecessor** records the largest key strictly smaller than k . Each returns the recorded item’s value, or $\perp$ if no candidate exists. Equal-key ties keep the first copy encountered, whose value is authoritative. Skipped levels and levels after the search ends receive dummy accesses, so every operation fetches exactly one node per level, residing at a random unvisited location.

We then consume the fetched nodes at their old locations and reinsert their entries through the scheduled rebuild. For **Insert**(k, v), we append (k, v) with a null pointer and tag -1 to the reinserted path; deduplication gives this new value priority over every old copy of k . For **Delete**(k), we instead remove every entry with key k from the fetched path before reinsertion. All copies of k have been fetched, and any node reached by one of their pointers has also been fetched, so its remaining entries are preserved by reinsertion. Deleting an absent key leaves the represented set unchanged. In every case, we pad the reinsertion to $2Lq + 1$ entries and use the same public rebuild schedule.

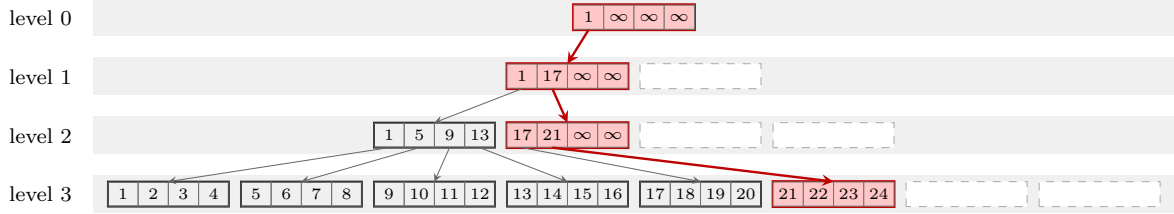
Efficiency analysis. The read phase costs $O(L) = O(\frac{1}{\epsilon} \log_B N)$ block transfers: it reads one node per level, and each node occupies $O(q)$ words and hence $O(1)$ blocks. For the rebuild phase, we amortize directly over operations, each of which atomically reinserts $2Lq + 1$ entries. Let $s = \min\{L^*, L - 1\}$ be the last level in the prefix rebuilt on every operation. A rebuild whose largest level is $k \geq s$ occurs at most once every q^{k-s} operations.

Rebuilding levels 0 through k processes $O(q^{k+1})$ entries. Sorting and deduplicating these entries,

logical tree

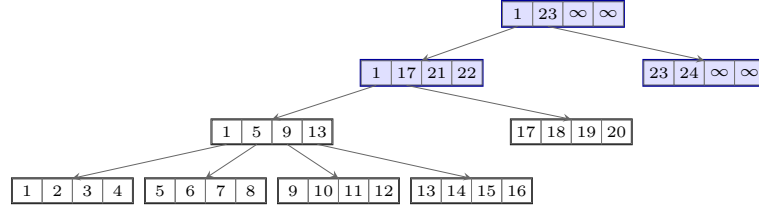


levels of the hierarchy

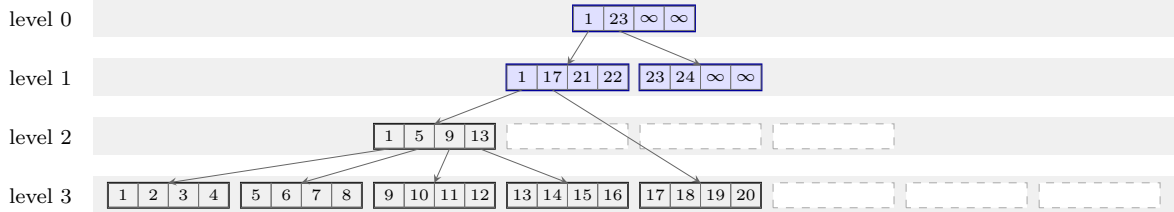


- (a) Before the lookup. Dashed boxes are unoccupied node slots. Every item carrying a pointer duplicates the first item of the node below it. **Lookup**(22) visits the **red** path.

logical tree



levels of the hierarchy



- (b) After the rebuild. Levels 0 and 1 have been rebuilt (**blue**) from the reinserted path together with the old levels 0 and 1. The node (17, 18, 19, 20) gets promoted in the logical tree but remains in the same location in the physical hierarchy. For the levels that were just rebuilt, the logical subtree and the levels in the hierarchy correspond exactly.

Figure 2: The effect of a lookup and the rebuild that follows it (for $q = 2$, $L = 4$). For clarity, we do not depict the levels as shuffled. In our actual algorithm, a rebuild must always involve at least $L^* = \lceil \log_q(10Lq) \rceil > 2$ levels, but for simplicity we pretend that only levels 0 and 1 are involved in this particular rebuild.

constructing the smaller levels by duplicating node minima, and shuffling the nodes require a constant number of oblivious sorts, permutations, and scans per rebuilt level. Since the level sizes decrease geometrically, their total cost is $O(L + \frac{q^{k+1}}{B} \log N)$ block transfers. The additive $O(L)$ accounts for rounding up transfers at levels smaller than a block. Consequently, the amortized rebuild cost per operation is at most

$$\begin{aligned} \sum_{k=s}^{L-1} \frac{1}{q^{k-s}} \cdot O\left(L + \frac{q^{k+1}}{B} \log N\right) &= O\left(L + \frac{Lq^{s+1}}{B} \log N\right) \\ &= O\left(L + \frac{L^2 q^3}{B} \log N\right) \\ &= O\left(\frac{1}{\epsilon} \log_B N + \frac{1}{\epsilon^2} \cdot \frac{(\log_B N)^2 \log N}{B^{2/(2+\epsilon)}}\right). \end{aligned}$$

Here we used $q^{s+1} \leq q^{L^*+1} \leq 10Lq^3$, including the rounding in $L^* = \lceil \log_q(10Lq) \rceil$. Our choice of q gives $B/q^3 = B^{2/(2+\epsilon)}$. Thus, when $B \geq \log^{2+\epsilon} N$, we have $B/q^3 \geq (\log N)^2 \geq \log N \cdot \log_B N$. The rebuild cost, and hence the total amortized cost per operation, is $O(\frac{1}{\epsilon^2} \log_B N) = O(\log_B N)$ for a fixed constant ϵ .

3 Preliminaries

3.1 External-Memory Random-Access Machine

External-memory model. We consider a random-access machine (RAM) with an external memory [AV88]. We make the standard assumption that each memory *word* can at least store an arbitrary address, i.e., the word size is at least $\log_2 N$ bits where N is the space of the RAM (in terms of words). The external-memory model was introduced to capture real-world architectures in which data stored on disks (also called external memory) is accessed in atomic units of *blocks* — typically ranging from 512B to 4KB — rather than individual words. We use the following standard notations for the external-memory model:

- let M denote the CPU’s private cache size, measured in terms of words;
- let B denote the block size, also measured in terms of words.

Performance metric. The total number of *blocks probed* during an algorithm’s execution, often referred to as the *I/O overhead*, is therefore a key performance metric in this setting, and also the main complexity measure adopted throughout this paper. Intuitively, algorithms that effectively exploit locality can achieve smaller I/O overhead.

Fit for hardware-enclave architectures. Several recent works [RS21, TGS23, GWT⁺25] observed that the external-memory model is a natural fit for hardware enclaves such as Intel SGX. In particular, because a hardware enclave has only limited protected memory, data is typically stored in encrypted form in unprotected external memory or on disk. When the enclave needs to access encrypted data residing outside its protected region, it issues requests for data in atomic units of pages (typically 4KB), and the operating system cooperates by swapping these encrypted pages in and out. The page swap is a heavy-weight operation that requires encryption, decryption, and context switches. Therefore, oblivious algorithms designed for hardware enclaves should aim to minimize the page swap overhead.

3.2 Oblivious Simulation

In this section, we introduce the notion of oblivious simulation for both reactive data structures and non-reactive algorithms. Since our focus is on statistical security, all definitions presented below provide statistical security guarantees.

Observable access patterns. Like in the standard literature on oblivious algorithms [GO96, Gol87, LSX19, AEKL25], we assume that during the execution of some data structure or algorithm, the adversary can observe the *access patterns* of the program, but cannot observe the memory contents — one can imagine that the memory contents are encrypted. In this paper, we assume a strong adversary who can observe not only which block is accessed, but also which exact memory word is accessed in each time step. Moreover, the adversary can observe whether the physical access is a read or a write operation. Security against such a strong adversary is commonly referred to as *strongly oblivious* or *doubly oblivious* in prior literature [LSX19, MPC⁺18, TGS23].

In real-life deployments such as Signal’s case, the oblivious algorithm is typically running inside a hardware enclave (e.g., Intel’s SGX). In this setting, block-level accesses correspond to page swaps performed by the enclave to read or write encrypted blocks stored in unprotected external memory or disk. These page-level access patterns are directly observable by the untrusted operating system (OS), which services the page faults. In addition, fine-grained memory accesses within the enclave may be leaked to an untrusted user application or to the OS through side channels such as cache-timing attacks [RTSS09, DMWS12]. For this reason, real-life deployments such as Signal [pri] desire strong obliviousness to achieve high assurance.

Oblivious simulation of a reactive functionality. We formally define oblivious simulation of a reactive functionality $\mathcal{F}$. A reactive functionality maintains an internal data structure while interacting with a caller. It answers the caller’s queries and updates the internal data structure accordingly. In this paper, we consider oblivious RAM (ORAM) and oblivious search tree (OST). They obviously realize the following reactive functionalities, respectively:

- $\mathcal{F}_{\text{mem}}$: The functionality implements a memory abstraction. Specifically, consider a memory array denoted **Mem** of size N , initialized to all 0s. Upon receiving a **Read(addr)** query, the functionality returns **Mem[addr]**; upon a **Write(addr, data)** query, the functionality updates **Mem[addr]** $\leftarrow$ **data** and returns $\perp$. The queried address **addr** is always guaranteed to be in the range $[N]$.
- $\mathcal{F}_{\text{st}}$: The functionality implements a search tree abstraction. Initially, the data structure is empty. Upon receiving an **Insert**(k, v) query, the functionality records (k, v) and returns $\perp$ — for each unique key k , the functionality only needs to record the value the last time k was inserted. Upon receiving a **Delete**(k) query, the functionality removes k from the tree if k is present and then returns $\perp$. Upon receiving a **Lookup**(k) query, the functionality returns the current value associated with k , or $\perp$ if k is absent. **Successor**(k) returns the current value associated with the smallest stored key strictly greater than k , and **Predecessor**(k) returns the current value associated with the largest stored key strictly smaller than k ; either returns $\perp$ if no such key exists. All three queries leave the stored items unchanged.

Definition 1 (Oblivious simulation of a reactive functionality). We say that a data structure implementation, parametrized by the security parameter 1^λ , obviously realizes some reactive functionality $\mathcal{F}$, iff there is some stateful simulator algorithm **Sim** such that the following holds: for any polynomial function $T(\cdot)$, there exists a suitable negligible function $\text{negl}(\cdot)$, such that for any

(even unbounded) adversary $\mathcal{A}$, for any $\lambda \in \mathbb{N}$, $\mathcal{A}$'s views in the experiments below have statistical distance at most $\text{negl}(\lambda)$:

- *Real*: Initialize the data structure with 1^λ , and for $i = 1, 2, \dots, T(\lambda)$:
 - $\mathcal{A}$ outputs a next query;
 - $\mathcal{A}$ receives the answer returned by the data structure implementation, as well as the access patterns incurred by the query.
- *Ideal*: Initialize the ideal functionality $\mathcal{F}$, and for $i = 1, 2, \dots, T(\lambda)$:
 - $\mathcal{A}$ outputs a next query;
 - $\mathcal{A}$ receives the answer returned by $\mathcal{F}$, as well as the simulated access patterns output by $\text{Sim}(1^\lambda, T(\lambda), i)$.

Oblivious simulation of a non-reactive functionality. We now define oblivious simulation for a non-reactive algorithm (e.g., sorting, deduplication, random permutation). Our definition captures oblivious simulation of randomized functionalities too.

Definition 2 (Oblivious simulation of a non-reactive functionality). Given a possibly randomized non-reactive functionality $\mathcal{F}$, we say that some algorithm Alg obviously realizes $\mathcal{F}$, iff there is some simulator algorithm Sim such that the following holds: for any polynomial function $n(\cdot)$, there exists a negligible function $\text{negl}(\cdot)$ such that for any $\lambda \in \mathbb{N}$, for any input Inp of length $n(\lambda)$, the following distributions have statistical distance at most $\text{negl}(\lambda)$:

- *Real*: run $\text{Alg}(1^\lambda, \text{Inp})$, output the outcome as well as the access patterns.
- *Ideal*: output the correct outcome $\mathcal{F}(\text{Inp})$ and simulated access patterns $\text{Sim}(1^\lambda, n(\lambda))$.

Remark 1 (A note on the security parameter). We can without loss of generality assume that $N \geq \lambda$, since we care about bounding the asymptotic performance for large N . Therefore, it suffices to achieve $\text{negl}(N) \leq \text{negl}(\lambda)$ failure probability.

3.3 Existing Building Blocks

Oblivious sort. Earlier works [ACN⁺20, RS21] have shown that for $B \geq \log^{1+\epsilon} \lambda$, we can obliviously sort n items with all but $\text{negl}(\lambda)$ probability, paying at most $O(\frac{n}{B} \log \frac{M}{B} \frac{n}{B})$ block transfers. For the special case where $M = \Theta(B)$, the bound simplifies to $O(\frac{n}{B} \log \frac{n}{B})$.

Oblivious deduplication. Oblivious deduplication realizes the following abstraction. Given an input array containing n items, where each item is either a key-value pair or a filler denoted $\perp$, output a deduplicated array where each distinct key k in the input appears exactly once. Moreover, if multiple items with the key k appear in the input, the output should choose one of the values of key k based on some publicly known preference function. The output is padded with fillers to an exact length of n . Prior works [GO96, Gol87] showed how to construct oblivious deduplication consuming $O(1)$ oblivious sorts and $O(1)$ scans over the array.

Oblivious bin placement. Bin placement realizes the following abstraction. Let n denote the input length, let β denote each bin’s capacity, and let m denote the number of bins. Given an input array containing n (possibly filler) items, where each real item is tagged with an integer from $[m]$ indicating which bin it wants to go to, output a list of m bins each of size β , such that all items land in their destined bins, and each bin is padded with fillers to a fixed size β . If some bin receives more than β items, the algorithm outputs “error”. Prior work [CS17] has shown how to realize oblivious bin placement consuming $O(1)$ oblivious sorts and $O(1)$ scans of arrays of size $O(n + m\beta)$.

Oblivious routing. Oblivious routing realizes the following abstraction. Let S be a source array containing n items, where each item is either a filler or a key-value pair. It is promised that all keys in the input are distinct. Let T be a destination array of size m where each entry (also called a *receiver*) either declares some desired key k or is a filler $\perp$. We want each non- $\perp$ receiver in the destination array T to receive the value associated with its desired key in S . Earlier works [CS17, RS21] showed how to accomplish oblivious routing using $O(1)$ oblivious sorts and $O(1)$ scans of arrays of size $O(n + m)$.

Oblivious random permutation. We want to obliviously realize the following randomized ideal functionality: upon receiving an array with n (possibly filler) items, choose a random permutation and output a permuted copy of the input array. Our earlier definition of oblivious simulation (Definition 2) also works for randomized functionalities. When applied to the above ideal functionality, Definition 2 implies that the algorithm’s access patterns should reveal nothing about the permutation that has been applied (up to a negligibly small statistical distance). We will rely on the oblivious random permutation algorithm of Asharov et al. [ACN⁺20], which requires $O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ block transfers, just like oblivious sorting.

4 Semi-Functional ORAM

We next present our semi-functional ORAM construction, which maintains statistical security but may fail to correctly serve a small fraction of requests. To simplify notation, we *rescale* ϵ by a $1/6$ factor relative to the technical overview (Section 2). As ϵ is arbitrarily small, this does not change any of our results.

Notations. Suppose, for some sufficiently small ϵ , $B \geq \log^{1+6\epsilon} N$. Let $q = B^\epsilon$ where ϵ is any sufficiently small positive constant, and without loss of generality, assume that q is an integer. Let $L = O(\frac{1}{\epsilon} \log_B N)$ be the smallest integer such that $q^{L-1} \geq N$. Let $H : \{0, 1, \dots, N-1\} \rightarrow \{0, 1, \dots, q-1\}^{L-1}$ be a $\log^{1+6\epsilon} N$ -wise independent uniform hash function needed for load balancing. Such a hash function can be stored in $O(1)$ blocks of memory using known constructions [HS24]. Thus, given an address $\text{addr} \in \{0, 1, \dots, N-1\}$, $H(\text{addr})$ outputs an $(L-1)$ -digit number denoted $k_1 \dots k_{L-1}$, where each digit is from the range $\{0, 1, \dots, q-1\}$.

4.1 Non-Oblivious Meta-Algorithm

To aid intuition, we begin by describing a meta-algorithm that captures the desired operational semantics, without yet addressing obliviousness. Later, in Section 4.3, we present an oblivious construction that faithfully emulates these semantics while preserving obliviousness.

Logical tree. We can imagine a q -ary logical tree with L levels numbered $0, 1, \dots, L-1$. Level 0 contains a single root node, and every other level $\ell \in [L-1]$ contains q^ℓ nodes. Each node in level $\ell \geq 1$ can be indexed by an ℓ -digit string $k_1 \dots k_\ell$. Further, the level- $(\ell-1)$ node indexed $k_1 \dots k_{\ell-1}$ is the parent of the level- ℓ node indexed $k_1 \dots k_\ell$.

Given an addr , let $H(\text{addr}) = k_1 \dots k_{L-1}$, and let $H(\text{addr})[: \ell] = k_1 \dots k_\ell$ denote the length- ℓ prefix of its hash. If some addr is in level $\ell \geq 1$, then it must reside in the node indexed by $H(\text{addr})[: \ell] = k_1 \dots k_\ell$. We define $H(\text{addr})[: 0] = 0$ since there is only a single node in the root level.

Data structure. We will rely on a hierarchical data structure containing levels $0, 1, \dots, L-1$, where level $\ell \in \{0, 1, \dots, L-1\}$ contains

1. $\lceil q^\ell / Z \rceil$ buckets where $Z = \Theta(B/q^3)$. In the meta-algorithm, we impose no constraints on the bucket size but we will show that each bucket consumes at most $O(q^3 \cdot Z) = O(B)$ words of space except with negligible probability.
2. a *stash* whose size is a single block. Henceforth, we use Stash_ℓ to denote level ℓ 's stash.

Each level $\ell \in \{0, 1, \dots, L-1\}$ stores the logical tree nodes at level ℓ . Except for the root node which has only one copy, every non-root logical tree node is forked into q clones indexed $0, 1, \dots, q-1$, and each clone is placed in an independent random bucket when the level is being rebuilt. Recall that each node in level ℓ has q children. When level ℓ is rebuilt, we make sure that each node in level ℓ stores the correct pointers to all q clones of each of its q children. The pointers will be at a bucket-granularity, i.e., each pointer records which bucket to read in the next level. In other words, each node stored in the hierarchical structure has the following format:

$$\text{Node} := (\text{list of } (\text{addr}, \text{data}) \text{ pairs}, \{ptr_{i,j}\}_{i,j \in \{0, \dots, q-1\}})$$

where $ptr_{i,j}$ points to the bucket in the next level that contains the j -th clone of the i -th child node. Further, for all items $(\text{addr}, \text{data})$ contained in the node, the prefix of $H(\text{addr})$ must agree with the index of the node. Initially, every addr is stored with value 0 in level $L-1$, and all levels are built as in **Rebuild** $(L-1)$ below.

Algorithms. The intuition behind our algorithm was explained earlier in Section 2.1. Below, we provide a detailed description of the meta-algorithm for our semi-functional ORAM.

Rebuild(ℓ^*)

- **Prepare.** Prepare a list of items denoted **Items** as follows: for each distinct addr that exists in levels $0, 1, \dots, \ell^*$, add the freshest copy $(\text{addr}, \text{data})$ to **Items**. If an addr exists in multiple levels, the one in a smaller level is considered fresher.

Empty the stashes of all levels $0, 1, \dots, \ell^*$.

- **Rebuild level ℓ^* .**

- For each length- ℓ^* prefix $k_1 \dots k_{\ell^*}$: remove all duplicated clones and preserve only a single copy of the level- ℓ^* node indexed $k_1 \dots k_{\ell^*}$; empty the list of items in the node but preserve its q^2 children pointers $\{ptr_{i,j}\}_{i,j \in \{0, \dots, q-1\}}$.
- For any item $(\text{addr}, -) \in \text{Items}$, place it in the level- ℓ^* node indexed $H(\text{addr})[: \ell^*]$.
- If $\ell^* > 0$: create q clones of each tree node, and toss each clone into an independent

random bucket among the $\lceil q^{\ell^*}/Z \rceil$ buckets. Else, all nodes (without cloning) are directly placed in the single bucket of root level $\ell^* = 0$.

- **Rebuild smaller levels.** For levels $\ell = \ell^* - 1$ down to 0 do:
 - For each length- ℓ prefix $k_1 \dots k_\ell$: remove all duplicates and preserve only one copy of the node $k_1 \dots k_\ell$; empty the list of items in the node, and modify its q^2 pointers $\{ptr_{i,j}\}_{i,j \in \{0, \dots, q-1\}}$ such that $ptr_{i,j}$ records the location of the bucket that contains the j -th clone of its i -th child in level $\ell + 1$.
 - If $\ell > 0$: create q clones of all nodes in level ℓ , and toss each clone into an independent random bucket in level ℓ . Else, place the nodes (without cloning) into the single root bucket.

Read(addr) or Write(addr, data)

Read phase: ReadRm(addr). Suppose $H(\text{addr}) = k_1 \dots k_{L-1} \in \{0, 1, \dots, q-1\}^{L-1}$. The algorithm **ReadRm(addr)** works as follows where col maintains the set of addresses encountered so far that share the same hash prefix as addr for prefixes incrementing in length.

1. Initially, let Bkt be the single bucket in level 0, let $\text{found} \leftarrow \text{False}$, and let $\text{col} \leftarrow \emptyset$.
2. For level $\ell = 0$ to $L - 1$ do:
 - Search both in Bkt and Stash_ℓ to find a clone of the node indexed $H(\text{addr})[: \ell]$, henceforth denoted **Node**. If not found, then let $\text{data}^* \leftarrow \mathcal{O}(\text{addr})$ where $\mathcal{O}$ denotes some external oracle that always returns the correct data associated with the queried address, and break.
 - If $|\text{col}| = q - 1$ and there is enough empty space in Stash_ℓ , then let $\text{Stash}_\ell \leftarrow \text{Stash}_\ell \cup \{\text{Node}\}$.
 - If an item $(\text{addr}, \text{data}^*)$ matching the requested address addr is found in **Node**, record data^* , let $\text{found} \leftarrow \text{True}$, and break.
 - If $\ell = L - 1$, let $\text{data}^* = \perp$ indicating not found and break.
 - Let col be the set of addresses in $\text{col} \cup \{\text{addresses in Node}\}$ that hash to the length- $(\ell + 1)$ prefix $k_1 \dots k_{\ell+1}$.
 - If $|\text{col}| \geq q$, then let Bkt be a random level- $(\ell + 1)$ bucket. // *Bad event: depleted clone*
 - Else, let Bkt be the level- $(\ell + 1)$ bucket represented by the pointer $ptr_{k_{\ell+1}, |\text{col}|}$ contained in **Node**. We say that the $|\text{col}|$ -th clone of the level- $(\ell + 1)$ node indexed $H(\text{addr})[: \ell + 1]$ is *pointer-selected*.
3. Output data^* .

Rebuild phase: Add(addr, data). // If request is of type **Read(addr)**, let $\text{data} \leftarrow \text{data}^*$.

4. Add $(\text{addr}, \text{data})$ to the root node.

5. Determine the largest level ℓ^* that should be rebuilt as follows, and call **Rebuild**(ℓ^*).
Specifically, at the T -th request, let q^ℓ be the largest power of q such that T is a multiple of q^ℓ . Then, $\min(\ell + 1, L - 1)$ is the maximum level to rebuild.

4.2 Analysis of the Semi-Functional ORAM's Meta-Algorithm

We first quickly justify that our meta-algorithm is correct, in that it always returns the last written value for any queried address.

Fact 4.1. *Suppose that we do not place any restrictions on the size of the buckets or the col set, and that the external oracle $\mathcal{O}$ always returns the last written value for any queried address. Then, the above meta-algorithm is always correct.*

Proof. Until the algorithm gives up and falls back to the external oracle $\mathcal{O}$, it is always able to retrieve the correct node identified by $H(\text{addr})[:\ell]$ in every level ℓ of the hierarchical data structure, where addr is the requested address. Further, due to the definition of the rebuild algorithm, the first time a matching address is found in a node whose index agrees with some prefix of $H(\text{addr})$, the corresponding value must be the last written. $\square$

The following results will be relevant for the oblivious implementation of our meta-algorithm. As they also characterize the meta-algorithm, it is simpler to reason about them here. In our oblivious implementation, we allocate B words (one block) to each bucket and to the col set. The next three lemmas provide the tail and space bounds showing that, for a polynomial-length request sequence independent of $H(\cdot)$, these capacities suffice except with negligible probability. Fact 4.5 then summarizes the resulting agreement with the unrestricted meta-algorithm and correctness under these size restrictions.

Finally, we establish two facts used in the security analysis. Fact 4.6 states that, between rebuilds of a given level ℓ , any particular addr is looked up at most once in level ℓ , so repeated queries for the same address do not exhaust clones in that level. Fact 4.7 then shows that each non-root node clone is pointer-selected at most once between rebuilds of its level, even across queries for different addresses. This property is used in the later obliviousness proof.

To show that no bucket overflows, we will make use of Lemma 2.3 from [BR94]:

Lemma 4.2 (Tail bound under k -wise hash [BR94]). *Let $k \geq 4$ be an even integer, and suppose that $X_1, \dots, X_n$ are k -wise independent Bernoulli random variables, and let $X = \sum_{i=1}^n X_i$. Letting $\mu = \mathbb{E}[X]$, we have that, for any $A > 0$,*

$$\Pr[X - \mu \geq A] \leq 8 \left(\frac{k\mu + k^2}{A^2} \right)^{k/2}.$$

Below, in Lemma 4.3 and Lemma 4.4, we may assume that the random coins consumed by the rebuild phase for tossing clones into buckets are chosen freshly at random on the fly. Therefore, these coins must be independent of the requests so far.

Lemma 4.3 (Bound on bucket size). *Suppose $0 < \epsilon < \frac{1}{12}$ and $B \geq \log^{1+6\epsilon} N$. Suppose that the requests are chosen independently from the hash function $H(\cdot)$. Then, except with $\text{negl}(N)$ probability, the space of every bucket is bounded by B .*

Proof. We first show that the space used for node pointers in each bucket will be bounded by $B/2$, and then we show that the space used for $(\text{addr}, \text{data})$ pairs will be bounded by $B/2$, both with $\text{negl}(N)$ failure probability. The lemma trivially holds for level 0 since level 0 does not have clones in the meta-algorithm. We will do the analysis separately for each level $\ell \geq 0$; the full lemma then follows from a union bound.

Note that, for each node, the list of $\{ptr_{i,j}\}$ in each node is fixed in length, having length $O(q^2)$. In total, there are $q^{\ell+1}$ node clones, as there are q^ℓ nodes and q clones of each. As each node clone independently and randomly chooses a bucket, the expected number of clones that fall into a bucket is $q^{\ell+1}/[q^\ell/Z] \leq qZ$. Additionally, by a simple Chernoff bound, we get that the probability that more than $2qZ$ nodes fall into a bucket is at most

$$\frac{1}{(e^2/4)^Z} \leq \text{negl}(N).$$

The space used by $2qZ$ nodes is $O(q^3Z) = O(B)$; by adjusting the constant in the definition of Z , we ensure that the space used by the node pointers is at most $B/2$.

Now, we bound the space used by $(\text{addr}, \text{data})$ pairs in each bucket. To see this, we will fix an allocation of node clones to buckets and assume that every bucket gets at most $2qZ$ clones, which happens with all but negligible probability as demonstrated above. Let X be the number of $(\text{addr}, \text{data})$ pairs that end up in a given bucket T , and let X' be the number of *unique* $(\text{addr}, \text{data})$ pairs in T , i.e., suppressing the duplicate pairs that arise from node cloning. As every node has q clones, we deterministically have that $X \leq qX'$. As each pair uses $O(1)$ words of space, it suffices to show that $X' \leq 4qZ$ with all but negligible probability.

Let S be the set of nodes in T , removing duplicates, and recall we assume $|S| \leq 2qZ$. Further, let A be the set of addresses inserted into level ℓ in the last rebuild, and for each $a \in A$, let Y_a denote whether $H(a)[\ell] \in S$. We then have that

$$X' = \sum_{a \in A} Y_a.$$

As the hash function H is $\log^{1+6\epsilon} N$ -wise (and thus $\log^{1+\epsilon} N$ -wise) independent (and independent of the assignment of nodes to buckets), we have that the Y_a variables are $\log^{1+\epsilon} N$ -wise independent, and $\Pr[Y_a = 1] = |S|/q^\ell \leq 2qZ/q^\ell$. Therefore,

$$\mathbb{E}[X'] = \sum_{a \in A} \Pr[Y_a = 1] = |A| \cdot \frac{|S|}{q^\ell} \leq q^\ell \cdot \frac{|S|}{q^\ell} \leq 2qZ.$$

By Lemma 4.2 (applied with k equal to $\log^{1+\epsilon} N$ rounded down to an even integer; we suppress this rounding, which does not affect the bound), we have that

$$\Pr[X' \geq 4qZ] \leq 8 \left(\frac{(2qZ) \cdot (\log^{1+\epsilon} N) + (\log^{1+\epsilon} N)^2}{(2qZ)^2} \right)^{0.5 \log^{1+\epsilon} N} \leq 8 \left(\frac{1}{2} \right)^{0.5 \log^{1+\epsilon} N} \leq \text{negl}(N).$$

This concludes the proof. $\square$

Lemma 4.4 (Bound on the size of the collision set). *Suppose $B \geq \log^{1+6\epsilon} N$, and that the requests are chosen independently of the hash function $H(\cdot)$ and form a polynomial-length sequence. Then, with $1 - \text{negl}(N)$ probability, the following holds for all ℓ : at the beginning of the iteration for level ℓ , the size of col does not exceed B .*

Proof. Recall that every level ℓ but the last one is emptied into the next level every q^ℓ steps. At the beginning of the iteration for every level ℓ , the number of elements in col (henceforth denoted $|\text{col}|$) is equal to the number of distinct elements (denoted A below) looked up in the past $\min(q^\ell, N)$ requests that share the same length- ℓ prefix as the current requested address. Note that A is independent of H . Thus, for

$$X = \sum_{\text{addr}' \in A \setminus \{\text{addr}\}} \mathbf{1}[H(\text{addr}')[:\ell] = H(\text{addr})[:\ell]],$$

we have $|\text{col}| \leq 1 + X$ and $\mu := \mathbb{E}[X] \leq |A|/q^\ell \leq 1$.

Let

$$T = \lceil \log^{1+6\epsilon} N \rceil \quad \text{and} \quad k = 2 \left\lfloor \frac{T-1}{4} \right\rfloor.$$

For any $r \leq k$ distinct addresses $\text{addr}_1, \dots, \text{addr}_r \in A \setminus \{\text{addr}\}$, the collision indicators for these addresses depend on the $r+1 \leq k+1 \leq T$ hash values $H(\text{addr}), H(\text{addr}_1), \dots, H(\text{addr}_r)$. Since H is T -wise independent, these indicators have the same joint distribution as under a fully random hash function and hence are k -wise independent Bernoulli random variables. Since $B \geq T$, the limited-independence tail bound stated in Lemma 4.2 gives

$$\begin{aligned} \Pr[|\text{col}| > B] &\leq \Pr[|\text{col}| > T] \leq \Pr[X \geq T] \\ &\leq 8 \left(\frac{k\mu + k^2}{(T-\mu)^2} \right)^{k/2} \leq 8 \left(\frac{1}{2} \right)^{k/2} \leq \text{negl}(N), \end{aligned}$$

where the penultimate inequality holds for all sufficiently large N . Finally, union bound over all levels and all requests in the polynomial-length sequence. $\square$

Fact 4.5. *Suppose the above parameter assumptions hold, the request sequence has polynomial length and is independent of $H(\cdot)$, and the external oracle $\mathcal{O}$ is correct. Then, with probability $1 - \text{negl}(N)$, every bucket and the col set fit in one block throughout the unrestricted meta-algorithm's execution. On this event, restricting each to one block leaves the execution unchanged when using the same random coins, and the restricted meta-algorithm is correct.*

Proof. By Lemmas 4.3 and 4.4 and a union bound over all rebuilds, no bucket or col set overflows except with $\text{negl}(N)$ probability. Until an overflow occurs, the restricted algorithm behaves identically to the unrestricted meta-algorithm using the same random coins, so correctness follows from Fact 4.1. $\square$

Fact 4.6. *Fix any arbitrary level ℓ . In between two consecutive rebuilds of level ℓ , we look for any distinct address addr at most once in level ℓ .*

Proof. The fact trivially holds for level 0 because it is rebuilt with every request. Now, consider a level $\ell > 0$. Every time we request some addr , we add addr back to the root node and it gets flushed down into bigger levels over time. Before level ℓ is rebuilt again, an item with addr will exist in some level smaller than ℓ . This means that we will have found addr even before we reach level ℓ (possibly from the external oracle). So by the definition of the meta-algorithm, we will not look for addr again in level ℓ . $\square$

Fact 4.7. *In the above meta-algorithm, each node clone in a non-root level is pointer-selected at most once till the level is next rebuilt.*

Proof. Fix a nonroot level ℓ and a logical node v in that level. Consider the interval between two consecutive rebuilds of level ℓ . List, in chronological order, the requests that pointer-select a clone of v during this interval. For the j -th such request, let a_j be its requested address, let col_j be the collision set immediately before the pointer is followed into level ℓ , and let $c_j = |\text{col}_j|$. By the definition of pointer selection, the j -th request pointer-selects clone c_j of v . A clone is pointer-selected only when $c_j < q$, so every pointer-selected clone index belongs to $\{0, \dots, q-1\}$. We claim that

$$c_1 < c_2 < \dots$$

Consider two consecutive requests in this list. Every address in col_j remains represented in some level strictly above ℓ until level ℓ is rebuilt. Rebuilds of smaller levels may move or deduplicate its copies, but they preserve a freshest copy in a level strictly above ℓ . Because all addresses in col_j have the same length- ℓ hash prefix as v , the next request that reaches the transition into v encounters these addresses along the same ancestor path. Hence $\text{col}_j \subseteq \text{col}_{j+1}$. Furthermore, after the request for a_j , the rebuild phase inserts a fresh copy of a_j into the root. Until level ℓ is rebuilt, this copy remains in some level strictly above ℓ . The address a_j is not already in col_j : otherwise a fresher copy of a_j would have been found before the request reached level ℓ , contradicting that it pointer-selected a clone of v . Since $H(a_j)[\ell] = v$, the next request that reaches the transition into v encounters a_j and includes it in its collision set. Therefore, $\text{col}_j \cup \{a_j\} \subseteq \text{col}_{j+1}$, and consequently $c_{j+1} \geq c_j + 1$.

Thus the indices $c_1, c_2, \dots$ of the pointer-selected clones of v are strictly increasing. Since every such index is smaller than q , no clone index can be pointer-selected twice.

If the collision-set size is at least q , the algorithm follows the random-bucket branch and does not pointer-select any clone. Repeated random-bucket scans or stash scans therefore do not contradict the claim. Since v was arbitrary, the result holds for every logical node in every nonroot level. $\square$

4.3 Oblivious Implementation of the Meta-Algorithm

We next describe how to obliviously realize the meta-algorithm.

Data structure. Just like in the meta-algorithm, we will have L levels, where level $\ell \in \{0, \dots, L-1\}$ has $\lceil q^\ell/Z \rceil$ buckets each provisioned with $C \cdot q^3 \cdot Z$ words of space for some suitably large constant C . Each level ℓ also has a stash denoted Stash_ℓ whose size is one block. Initialization is as in the meta-algorithm: every address starts in level $L-1$, and all levels are built by **Rebuild**($L-1$) below.

Since nodes may not be uniform in size, instead of directly storing nodes in the buckets, we break each node into raw items and pointers, and store the raw items and pointers in the bucket. We assume that each item and pointer carries sufficient metadata to recover the level, node, and clone with which it is associated. More specifically, consider a node of the form

$$\text{Node} := (\text{list of } (\text{addr}, \text{data}) \text{ pairs}, \{ptr_{i,j}\}_{i,j \in \{0, \dots, q-1\}}).$$

The node is decomposed into the following entries, each fitting in $O(1)$ words:

- A list of items, where each item is of the form $(\text{addr}, \text{data}) : \{\ell, \text{cid}\}$ where cid denotes the clone index. The number of items is a random variable.
- A list of q^2 pointers where each pointer is of the form $ptr_{i,j} : \{i, j, \ell, \text{nid}, \text{cid}\}$ where $i, j \in \{0, \dots, q-1\}$, and nid and cid denote which node and which clone the pointer belongs to. For simplicity, we often write $ptr_{i,j} : \{\ell, \text{nid}, \text{cid}\}$ since we may assume that the $ptr_{i,j}$ part already carries the indices (i, j) .

In the above item and pointer format, one can view the part in $\{\}$ as the *header* field that ties the entry to the corresponding level, node, and clone. For an item, because we can compute the node index from the stored **addr**, the header part need not store the node index redundantly.

Algorithm description. The intuition behind the algorithm was explained earlier in Section 2.1. Below, we describe the oblivious implementation of our semi-functional ORAM in detail.

Rebuild(ℓ^*)

1. **Prepare.** Copy all entries of levels 0 through ℓ^* into an array denoted **Items**. In a single linear scan of **Items**, preserve only the items and replace all pointers with fillers. Using oblivious deduplication, output a deduplicated array **Items** of length q^{ℓ^*} that contains a single, freshest copy of each distinct **addr**, possibly padded with fillers.
2. **Rebuild level ℓ^* .**
 - (a) If $\ell^* = L - 1$, then this level does not need pointers and we let **Ptrs** = $\emptyset$; else do the following. In a single scan of level ℓ^* , preserve only the pointers and replace all items with fillers. Using oblivious deduplication, obtain an array **Ptrs** of length q^{ℓ^*+2} that preserves exactly one copy of the pointer $ptr_{i,j} : \{\ell^*, k_1 \dots k_{\ell^*}, -\}$ for each node $k_1 \dots k_{\ell^*} \in \{0, \dots, q-1\}^{\ell^*}$ and each $i, j \in \{0, \dots, q-1\}$.
 - (b) For each node $k_1 \dots k_{\ell^*} \in \{0, \dots, q-1\}^{\ell^*}$, for each clone index $j \in \{0, \dots, q-1\}$, choose a random bucket for the j -th clone of the node $k_1 \dots k_{\ell^*}$. Write down an array **BktIdx $_{\ell^*}$** containing the random bucket indices chosen.
 - (c) Make q clones for each item in **Items** and each pointer in **Ptrs** and assign them the clone indices 0 to $q-1$, respectively. All clones are associated with the level ℓ^* .
 - (d) Using oblivious routing where **BktIdx $_{\ell^*}$** acts as the source, all clones (of items and pointers) receive their destined bucket number depending on which node and which clone they belong to.
 - (e) Using oblivious bin placement, toss all clones of items and pointers to their destined buckets. The new level ℓ^* consists of the resulting buckets and an empty stash.
3. **Rebuild smaller levels.** For level $\ell = \ell^* - 1$ downto 0 do:
 - (a) Relying on the array **BktIdx $_{\ell+1}$** , prepare a pointer array **Ptrs** containing the following: for each level- ℓ node $k_1 \dots k_{\ell}$, write down a list of pointers $\{ptr_{i,j} : \ell, k_1 \dots k_{\ell}, -\}_{i,j \in \{0, \dots, q-1\}}$ that correctly record the locations of the buckets for all clones of the node's children.
 - (b) For each node $k_1 \dots k_{\ell} \in \{0, \dots, q-1\}^{\ell}$, for each clone index $j \in \{0, \dots, q-1\}$, choose a random bucket for the j -th clone of the node $k_1 \dots k_{\ell}$. Write down an array **BktIdx $_{\ell}$** containing the random bucket indices chosen.
 - (c) Make q clones of each pointer in **Ptrs** and assign them distinct clone indices. Each pointer clone reads the appropriate location in **BktIdx $_{\ell}$** to learn its destined bucket.
 - (d) Using oblivious bin placement, toss all pointer clones into their destined buckets. The new level ℓ consists of the resulting buckets and an empty stash.

Read(addr) or Write(addr, data)

/ The sets col and each level's stash are stored as a list with capacity B. Insertions into col or the stashes are performed as in Remark 3. */*

Read phase: ReadRm(addr).

1. Initially, let **Bkt** be the single bucket in level 0, let **found** $\leftarrow$ **False**, and let **col** $\leftarrow$ $\emptyset$.
2. For level $\ell = 0$ to $L - 1$:

If **found** = **True**, let **Bkt** be a random bucket at level ℓ , emulate the access patterns of the **else** branch below without registering any side effect.

Else, let $\text{cid} \leftarrow |\text{col}|$; if $\ell < L - 1$, $\text{col} \leftarrow \{\text{addr}' : \text{addr}' \in \text{col} \wedge H(\text{addr}')[\ell + 1] = H(\text{addr})[\ell + 1]\}$; scan **Bkt**||**Stash** $_{\ell}$ twice and do the following:

 - (a) (*1st scan*): If an item (**addr**, **data**^{*}) matching the requested **addr** is found, then record **data**^{*}, and let **found** $\leftarrow$ **True**.
 - (b) (*1st scan*): Whenever we encounter any item of the form $(\text{addr}', -) : \{\ell, \text{cid}'\}$, if $H(\text{addr}')[\ell] = H(\text{addr})[\ell]$ and $\text{cid}' = \text{cid} = q - 1$, then add the item to **Stash** $_{\ell}$.
Whenever we encounter any pointer of the form $\text{ptr}_{i,j} : \{\ell, \text{nid}, \text{cid}'\}$, if $\text{nid} = H(\text{addr})[\ell]$ and $\text{cid}' = \text{cid} = q - 1$, then add the pointer to **Stash** $_{\ell}$.
Addition to the stash is atomic for all items and pointers belonging to the same node clone — see Remark 3. *// Add a node to Stash $_{\ell}$ when it is about to deplete clones*
 - (c) (*1st scan*): If no pointer with $\text{nid} = H(\text{addr})[\ell]$ is found and no matching item was found in step (a), then let **data**^{*} $\leftarrow$ $\mathcal{O}(\text{addr})$, and let **found** $\leftarrow$ **True**. *// Desired node not found in Bkt or Stash $_{\ell}$, resort to $\mathcal{O}$*
 - (d) (*1st scan*): If $\ell < L - 1$: whenever we encounter any item of the form $(\text{addr}', -) : \{\ell, -\}$, if $H(\text{addr}')[\ell + 1] = H(\text{addr})[\ell + 1]$, let $\text{col} \leftarrow \text{col} \cup \{\text{addr}'\}$. If $|\text{col}| \geq q$, i.e., a “deplete clone” event has occurred, then let **Bkt** be a random level- $(\ell + 1)$ bucket.
 - (e) (*2nd scan*): If $\ell < L - 1$ and $|\text{col}| < q$: let $k_1 \dots k_{L-1} \leftarrow H(\text{addr})$, search for a pointer of the form $\text{ptr}_{k_{\ell+1}, |\text{col}|} : \{\ell, k_1 \dots k_{\ell}, -\}$, and update **Bkt** to be the level- $(\ell + 1)$ bucket that it points to.
3. Output **data**^{*}.

Rebuild phase: Add(addr, data). *// If request is of type Read(addr), let data $\leftarrow$ data^{*}.*

Continue just like the **Add(addr, data)** subroutine in the meta-algorithm, except that we now call the new oblivious implementation of the **Rebuild** subroutine.

Fake read phase: ReadRm($\perp$). *// For the full ORAM in Section 5 only.*

Emulate the access patterns of a real **ReadRm** request with no side effect, where **Bkt** is set to be a random bucket at every level.

Remark 2 (Convention for writing oblivious algorithms). Throughout, whenever the oblivious algorithm has an **if**-branch conditioned on secret variables, we assume that both branches are padded with fake instructions that register no side effect, such that both branches exhibit identical

memory access patterns (up to the word-level). We omit explicitly writing this padding for clarity. For example, the conditional statement `if (s) then A, B; else C` is actually implemented as `(s) then A, B, fake-C; else fake-A, fake-B, C`.

Remark 3 (Operations on the stash and the collision set). Step 2(b) above is implemented as follows. As we iterate through each item and pointer contained in $\text{Bkt}|\text{Stash}_\ell$: make a scan through the stash which is a single block in size, and insert the item or pointer if it belongs to relevant node clone and there is space in the stash. At the end of this, make another scan over stash: if any insertion attempt failed earlier, then remove any already added item or pointer belonging to the relevant node clone during the process. This ensures that the items and pointers belonging to a node clone are inserted into the stash atomically, just like in the meta-algorithm.

Downselecting elements in col is accomplished by making a linear scan of col . Step 2(d) is implemented as follows. As we iterate through each item and pointer contained in $\text{Bkt}|\text{Stash}_\ell$: make a scan through col , inserting an element in the process if desired and there is available space. An insertion performs nothing if the col is already full.

Lemma 4.8 (Obliviousness of our semi-functional ORAM). *Suppose that the calls to the external oracle $\mathcal{O}$ do not emit any observable access patterns. Then, the semi-functional ORAM described in Section 4.3 satisfies obliviousness.*

Proof. The detailed proof of this lemma is provided in Section 4.5. $\square$

4.4 Efficiency Analysis

Cost of the read phase. Suppose that the collision set col which fits in one block is stored in the cache. Then, during the read phase, for each level, we fetch one bucket and the stash into the cache. We then perform $O(1)$ scans over the bucket and the stash while updating the col set and the stash — these scans consume only work but do not incur extra block transfers. Therefore, the read phase incurs $O(\frac{1}{\epsilon} \log_B N)$ block transfers.

Cost of the rebuild phase. Level 0 which has only 1 block is rebuilt every request. Each level $\ell \in [L-1]$ is rebuilt every $q^{\ell-1}$ steps. Since level ℓ contains $\lceil q^\ell/Z \rceil$ buckets, each provisioned with $\Theta(q^3 Z)$ words, its physical size is

$$V_\ell = \Theta\left(q^3 Z \left\lceil \frac{q^\ell}{Z} \right\rceil\right) = O(q^{\ell+3} + q^3 Z) = O(q^{\ell+3} + B),$$

where the last equality uses $Z = \Theta(B/q^3)$. If $q^\ell < Z$, the level consists of one $O(B)$ -word bucket and can be transferred and processed in cache using $O(1)$ block transfers. Otherwise, $V_\ell = \Theta(q^{\ell+3})$, and the rebuild is dominated by obviously sorting this many words. Thus, uniformly over all levels, a level- ℓ rebuild costs at most

$$O\left(1 + \frac{q^{\ell+3}}{B} \log(q^{\ell+3})\right)$$

block transfers. Therefore, the amortized cost (in terms of block transfers) for the rebuild phase is upper bounded by

$$\begin{aligned} & O\left(\sum_{\ell=1}^{L-1} \frac{1}{q^{\ell-1}}\right) + O\left(\sum_{\ell=1}^{L-1} \frac{q^4}{B} \log(q^{\ell+3})\right) \\ & \leq O(1) + O\left(\frac{1}{\epsilon} \cdot \frac{1}{B^{1-4\epsilon}} \cdot \log N \cdot \log_B N\right), \end{aligned}$$

where the first sum is geometric. Observe that for $B > \log^{1+6\epsilon} N$ and $0 < \epsilon < 1/48$, we have $B^{1-4\epsilon} > \log N$ and the amortized rebuild cost is upper bounded by $O(1 + \frac{1}{\epsilon} \log_B N) = O(\frac{1}{\epsilon} \log_B N)$ in the nontrivial regime $B \leq N$.

Space. At level ℓ , the buckets occupy

$$\underbrace{O\left(\left\lceil q^\ell/Z \right\rceil\right)}_{\# \text{ buckets}} \cdot \underbrace{O(q^3 Z)}_{\text{words per bucket}}$$

words. Hence the space of all levels is

$$O\left(\sum_{\substack{0 \leq \ell < L \\ q^\ell < Z}} B + \sum_{\substack{0 \leq \ell < L \\ q^\ell \geq Z}} q^{\ell+3}\right) = O(B(1 + \log_q Z) + q^{L+2}) = O\left(\frac{B}{\epsilon} + q^{L+2}\right).$$

Indeed, if $q^\ell < Z$, level ℓ has one $O(B)$ -word bucket, and there are at most $1 + \log_q Z = O(1/\epsilon)$ such levels. If $q^\ell \geq Z$, then $\lceil q^\ell/Z \rceil \leq 2q^\ell/Z$, so that level occupies $O(q^{\ell+3})$ words and these terms form a geometric sum. Finally, the minimality of L gives $q^{L-1} < qN$. Consequently, for every fixed $\epsilon > 0$ and all sufficiently large N , the total space is $O(\frac{B}{\epsilon} + q^4 N) = O(B^{4\epsilon} N)$. One of the q factors comes from the fact that q^{L-1} may be up to a q factor larger than N . This q factor can easily be removed by allowing a different fanout in the last level. Specifically, we may assume without loss of generality that N is a multiple of q^{L-2} , that is, $N = q' \cdot q^{L-2}$ for some integer $q' \leq q$ — if not we can always round N to the next multiple of q^{L-2} incurring only constant blowup. Now, the number of buckets in the last level can be set to $\lceil q' \cdot q^{L-2}/Z \rceil$ instead of $\lceil q^{L-1}/Z \rceil$. With this optimization, the last level occupies

$$O(\lceil N/Z \rceil) \cdot O(q^3 Z) = O(q^3 N + q^3 Z) = O(B^{3\epsilon} N + B) = O(B^{3\epsilon} N),$$

while the sum through level $L-2$ is $O(q^{L+1} + B/\epsilon) = O(q^3 N + B/\epsilon)$. Consequently, for every fixed $\epsilon > 0$ and all sufficiently large N , the total space is bounded by $O(q^3 N + \frac{B}{\epsilon}) = O(B^{3\epsilon} N)$.

In Section A, we describe a modification to this basic scheme such that we can improve the space consumption to $O(N)$.

4.5 Proofs for the Semi-Functional ORAM

Hybrid algorithm and hybrid experiment. We consider a hybrid algorithm that is otherwise the same as the oblivious implementation except that we now replace the building blocks including oblivious deduplication and oblivious routing with ideal functionalities that always output correct answers. For oblivious bin placement, its ideal functionality reports error if any bucket overflows and else outputs the correct answer.

We consider a hybrid experiment similar to the real experiment in Definition 1, except that now the adversary is interacting with the hybrid algorithm; moreover, the access patterns it observes from the oblivious deduplication, oblivious routing, and oblivious bin placement primitives are replaced with simulated access patterns output by the simulators of the corresponding primitives. Due to the obliviousness of the underlying primitives, the adversary's view in the hybrid experiment (including the outputs and the access patterns for all queries) is statistically indistinguishable from its view in the real experiment.

Fact 4.9 (Correctness). *As long as no bucket overflows and no collision set col ever overflows, the hybrid algorithm faithfully emulates the semantic behavior of the meta-algorithm and always returns correct answers. Moreover, Fact 4.6 and Fact 4.7 still hold for the hybrid algorithm.*

Proof. Unless some bucket is assigned more than its capacity, the ideal functionality of the oblivious bin placement always correctly places items into buckets. Now, if the col set in the hybrid algorithm is provisioned with ample space such that it never overflows, then the main semantic difference between the meta-algorithm and the hybrid algorithm is that the latter breaks up a node into individual items and pointers, and stores the raw items and pointers in the bucket. Moreover, every item or pointer carries enough information for the algorithm to recover which node and which clone they belong to. This way, whenever the meta-algorithm scans some node clone during the read phase, the hybrid algorithm scans the corresponding bucket and looks for items and pointers that belong to this particular node clone. Therefore, the hybrid algorithm exactly emulates the semantic behavior of the meta-algorithm. Correctness then follows from Fact 4.1. $\square$

We next consider a variant of the hybrid experiment, henceforth called the ∞ -hybrid experiment, which is otherwise identical to the above hybrid experiment except that we now provision the buckets and the col set infinite space (i.e., sufficiently large w.r.t. N), such that no overflow event occurs. In the following Lemma 4.10, we may imagine that scanning an infinite-capacity bucket or col introduces a nominal trace that records either the level and bucket numbers, or the identifier “ col ”.

Lemma 4.10. *Suppose that calls to the external oracle $\mathcal{O}$ do not emit any observable access patterns. Then, there exists a simulator Sim such that even with an unbounded, adaptive adversary, its view in the ∞ -hybrid experiment is identically distributed as always seeing the correct answers and simulated access patterns output by $\text{Sim}(N, T)$ which is provided only with the space N and the number of requests T .*

Proof. By Fact 4.9, the ∞ -hybrid experiment guarantees correctness. We define the simulator Sim as follows:

- For the rebuild phase, all access patterns (except the parts output by the simulators of the underlying oblivious primitives) are predetermined and data independent. Therefore, the access patterns of the rebuild phase can be perfectly simulated by calling the simulators of the underlying oblivious primitives, and simulating the remaining predetermined access patterns in a trivial manner.
- The read phase’s access patterns can be easily simulated by selecting a random Bkt per level, and then emulating the access patterns of a real **ReadRm** incurred on $\text{Bkt} \parallel \text{Stash}_\ell$. Observe that once a Bkt is fixed for every level, then the access patterns of a real **ReadRm** are fixed and data-independent.

Due to Fact 4.9 and Fact 4.7, in the hybrid experiment, with infinite bucket sizes, in each level, we either visit the bucket associated with a previously unconsumed node clone or visit a random bucket. Recall that each node clone is assigned to an independent random bucket during rebuild. Moreover, the rebuild phase’s access patterns are independent of the data or queries, and reveal nothing about the random assignment of clones to buckets. Therefore, the read phase’s access patterns in the ∞ -hybrid experiment are identically distributed as the above simulated distribution. $\square$

Lemma 4.11. *Suppose that the calls to the external oracle $\mathcal{O}$ do not emit any observable access patterns. The hybrid algorithm satisfies obliviousness.*

Proof. Henceforth, we assume that the access trace introduced by scanning a finite-capacity bucket or col set is nominal, i.e., it records either the level and bucket numbers or the identifier “col”. If obliviousness holds with the nominal trace, it also holds when each nominal trace is replaced with an actual scan of the corresponding data structure. More specifically, we can always modify the simulator for the nominal to expand each token in the nominal trace with the actual scan of the corresponding data structure.

The proof of Lemma 4.10 implies that the adversary’s view in the ∞ -hybrid experiment is identically distributed as always seeing correct answers and the simulator’s access patterns. Further, observe that the simulator’s random coins are independent of the hash function $H(\cdot)$. Therefore, the adversary learns nothing about $H(\cdot)$ in the ∞ -hybrid experiment. In other words, the resulting adaptive request sequence is distributed independently of the hash. Through the emulation in Fact 4.9, Fact 4.5 therefore implies that, over polynomially many requests, the probability that any bucket or the col set in the ∞ -hybrid experiment exceeds B words is at most $\text{negl}(N)$. Couple the hybrid and ∞ -hybrid experiments using the same random coins. They agree until the first such overflow, so their views are statistically indistinguishable. Together with Lemma 4.10, this proves the lemma. $\square$

Lemma 4.12 (Restatement of Lemma 4.8). *Suppose that the calls to the external oracle $\mathcal{O}$ do not emit any observable access patterns. Then, the semi-functional ORAM described in Section 4.3 satisfies obliviousness.*

Proof. Recall that any adaptive adversary’s view when interacting with real algorithm is statistically indistinguishable from its view when interacting with the hybrid algorithm. Therefore, the lemma follows directly from Lemma 4.11. $\square$

5 Full Oblivious RAM Construction

5.1 Modifications to the Semi-Functional ORAM

In Section 4, we only showed that node clones are never reused in a semi-functional ORAM. However, jumping ahead, in our full ORAM, we want to do the clone accounting more tightly. First, a fake **ReadRm** request should not contribute to the collision set for clones. Second, if a real **ReadRm** request fails at level ℓ , it does not contribute to the collision set at level ℓ ; at level $\ell + 1$ and deeper, it performs only fake accesses and likewise does not contribute. Third, it would make the analysis easier if we ensure that clones are consumed one by one without any skipping. By contrast, in Section 4, after a lookup failure, the collision set could conservatively overcount earlier visits and skip unused clones.

To achieve these goals, we attach a *depth marker* denoted **dep** to each inserted item, where **dep** is the deepest level at which that read obtained a valid node (and **dep** = -1 for a fake read). When selecting a clone at level ℓ , we count only markers with depth at least ℓ . The counter then records exactly the earlier visits to that node since its last rebuild. In particular, the first q visits use its q clones in order, and the last clone supplies the copy to be stashed. Rebuilding retains the freshest data value and maximum marker independently. We now elaborate on how to augment the semi-functional ORAM with the depth marker, and we use this modified version in the remainder of this section.

Depth marker. The data structure is the same as in Section 4.3, except that each item decomposed from a node clone now carries a depth marker:

$$(\text{addr}, \text{data}, \text{dep}) : \{\ell, \text{cid}\}.$$

Here ℓ and cid identify the level and clone, as before. On insertion, dep is the deepest level at which the preceding real **ReadRm** obtained a valid node, or -1 if that call was fake. Pointers are unchanged; otherwise the data structure is the same as before.

Initially, all items have marker -1 . Rebuilding retains the maximum marker for each address independently of its freshest data value. Thus an item contributes to the collision set at level ℓ only when $\text{dep} \geq \ell$.

Construction. Use the same parameters, buckets, node clones, pointers, and one-block stashes as in Section 4.3. Recall that except for the root node which has only one copy, every non-root logical tree node is forked into q clones. Initialize as in Section 4.3, with the initial data at the last level, and leave all stashes empty.

Read phase: ReadRm(addr).

1. Initially, let **Bkt** be the single bucket in level 0, let $\text{stop} \leftarrow \text{False}$, and let $\text{col} \leftarrow \emptyset$. Also let $\text{dep} \leftarrow -1$ and $\text{data}^* \leftarrow \perp$, where $\perp$ is a distinguished failure result outside the data domain. The set col now stores address–depth pairs, with at most one pair per address.

2. For level $\ell = 0$ to $L - 1$:

If $\text{stop} = \text{True}$, let **Bkt** be a random bucket at level ℓ , emulate the access patterns of the **else** branch below without registering any side effect.

Else let $\text{cid} \leftarrow |\text{col}|$, if $\ell < L - 1$, retain in col only pairs $(\text{addr}', \text{dep}')$ satisfying

$$H(\text{addr}')[:\ell + 1] = H(\text{addr})[:\ell + 1], \quad \text{dep}' \geq \ell + 1,$$

scan **Bkt** || **Stash_ℓ** twice and do the following:

- (a) (*1st scan*): If an item $(\text{addr}, \text{data}^*, -)$ matching the requested addr is found, then record data^* , and let $\text{stop} \leftarrow \text{True}$.
- (b) (*1st scan*): Whenever we encounter any item of the form $(\text{addr}', -, \text{dep}') : \{\ell, \text{cid}'\}$, if $H(\text{addr}')[:\ell] = H(\text{addr})[:\ell]$ and $\text{cid}' = \text{cid} = q - 1$, then add the item to **Stash_ℓ**.
Whenever we encounter any pointer of the form $\text{ptr}_{i,j} : \{\ell, \text{nid}, \text{cid}'\}$, if $\text{nid} = H(\text{addr})[:\ell]$ and $\text{cid}' = \text{cid} = q - 1$, then add the pointer to **Stash_ℓ**.
Addition to the stash is atomic for all items and pointers belonging to the same node clone—see Remark 3. // Add a node when it is about to deplete clones
- (c) (*1st scan*): If no pointer with $\text{nid} = H(\text{addr})[:\ell]$ is found, and no matching item was found in step (a), then let $\text{stop} \leftarrow \text{True}$. Otherwise set $\text{dep} \leftarrow \ell$.
- (d) (*1st scan*): If $\ell < L - 1$, do the following: whenever we encounter any item of the form $(\text{addr}', -, \text{dep}') : \{\ell, -\}$, if $H(\text{addr}')[:\ell + 1] = H(\text{addr})[:\ell + 1]$ and $\text{dep}' \geq \ell + 1$, let $\text{col} \leftarrow \text{col} \cup \{(\text{addr}', \text{dep}')\}$. If addr' already occurs in col , keep its maximum depth. If $|\text{col}| \geq q$, i.e., a “deplete clone” event has occurred, then let **Bkt** be a random level- $(\ell + 1)$ bucket.

- (e) (*2nd scan*): If $\ell < L - 1$ and $|\text{col}| < q$: let $k_1 \dots k_{L-1} \leftarrow H(\text{addr})$, search for a pointer of the form $\text{ptr}_{k_{\ell+1}, |\text{col}|} : \{\ell, k_1 \dots k_{\ell}, -\}$, and update **Bkt** to be the level- $(\ell + 1)$ bucket that it points to.

3. Output data^* , with $\perp$ indicating failure. Use depth dep for the following **Add**.

Rebuild phase: **Add**(addr , data , dep). // If request is of type **Read**(addr), let $\text{data} \leftarrow \text{data}^*$.

4. Add (addr , data , dep) to the root node. Here dep is the depth saved by the preceding read, including $\text{dep} = -1$ for a fake read. This insertion is real even when the preceding read was fake.
5. Determine the largest level ℓ^* that should be rebuilt like in Section 4.1, and call the oblivious implementation of **Rebuild**(ℓ^*) from Section 4.3. In the preparation step of **Rebuild**, deduplicate the collected items by address: for each address, retain the **freshest data value** and **the maximum depth** . The freshest value and maximum depth may come from different copies. Carry the retained depth with the item through cloning and placement.

Fake read phase: **ReadRm**($\perp$).

Emulate the access patterns of a real **ReadRm** request with no side effect, where **Bkt** is set to be a random bucket at every level. Use depth $\text{dep} = -1$ for the following **Add**.

As in Remark 3, a failed stash insertion is rolled back by removing all entries of the attempted clone. Thus every retained stash copy is complete, without changing the format of items or pointers. Each entry insertion and collision-set update scans a one-block list, and the rollback uses one further stash scan. All scans and secret branches are padded as in Remark 2.

What the markers count. In the full ORAM, each semi-functional ORAM replica may receive a fake **ReadRm** followed by a real **Add**. In fact, every **ReadRm** request, no matter real or fake, will always be followed by a real **Add** request. The following Fact 5.1 holds in this setting.

For each level $\ell \geq 1$, a *reset* is a rebuild that reaches ℓ . Treat initialization as a reset at time zero. An *epoch* is the interval of reads between two successive resets. A reset occurs after the insertion ending the preceding epoch.

Intuitively, when a read reaches a node at level ℓ , its collision set counts exactly the earlier reads that successfully reached that same node during the current epoch. Fake reads and reads that stopped above ℓ do not contribute. These visits have distinct addresses: after a request for an address, its real **Add** leaves a fresh copy above ℓ , so a repeated request finds that copy or fails before reaching ℓ . Consequently, the first q visits access the node's q clone buckets in order. A read can fail at this level only after those q visits have occurred. This observation is formally proven in the fact below.

Fact 5.1 (Failures require prior visits). *Suppose buckets and collision sets do not overflow and the oblivious primitives used in rebuilds do not fail. If a lookup fails at a node at level $\ell \geq 1$, then at least q earlier lookups for distinct addresses obtained a valid copy of that node during the same epoch. In each epoch, the first q valid visits to any non-root node access clones $0, \dots, q-1$ in order, with a stash insertion attempted only on the q th visit.*

Proof. Fix a replica with grouping hash H , a level $\ell \geq 1$, and one of its epochs. Let addr_t be the address of the current logical request, and let $A_\ell(t)$ be the set of distinct addresses of earlier calls

in this epoch that obtained a valid level- ℓ node. We first show that whenever the current search reaches level ℓ , before trying to obtain its node its counter satisfies

$$|\text{col}| = |\{\text{addr}' \in A_\ell(t) : H(\text{addr}')[:\ell] = H(\text{addr}_t)[:\ell]\}|. \quad (1)$$

Immediately after the reset, all record lists above ℓ are empty. Until the next reset, the set of addresses represented there by markers of depth at least ℓ is exactly $A_\ell(t)$. Indeed, obtaining a valid node at ℓ generates such a marker on insertion; other calls generate none. Rebuilds confined to levels above ℓ preserve this set by maximum-depth deduplication. A rebuild reaching ℓ ends the epoch and moves these markers out of the upper levels.

All clones of a node have identical items, markers, and child pointers throughout its epoch, and retained stash copies are complete by atomic insertion. The address and prefix tests in the scans exclude other nodes; accepting several copies of the same marker does not affect the set. Obtaining a node from a randomly scanned bucket therefore supplies exactly the same information as obtaining it from its designated bucket or stash. Such a visit is included in $A_\ell(t)$ and sets the depth marker in the same way. At a leaf, finding the requested item is a valid visit; if a search has obtained every preceding node, the freshest-data invariant ensures that this item is present in its leaf.

Every qualifying marker sharing the target's length- ℓ prefix lies in an ancestor node of the target. A search reaching ℓ has obtained all those ancestors, and hence encountered the marker. Conversely, the depth and prefix filters discard everything else. This proves (1).

After any earlier operation on the same address, even a fake read in this replica, its correct fresh data would remain above ℓ until the next reset. The current search would therefore find its data or fail above ℓ . It could not reach ℓ . Thus each valid visit adds one new address to A_ℓ , and the node's counter increases by exactly one between successive valid visits. The first q visits access the designated buckets, which contain the corresponding clones. On the q th visit the stash is attempted. Later visits scan a fresh random bucket and the stash, and can succeed using either. A failure therefore requires that the stash attempt failed and that the random bucket supplied no usable copy. A failed stash insertion does not invalidate the q th valid visit. The assertions follow. $\square$

5.2 Full ORAM Construction

Set $\gamma = \lceil 2/\epsilon \rceil + 2$ and create γ modified semi-functional ORAM replicas $\text{ORAM}_1, \dots, \text{ORAM}_\gamma$, each with independent grouping hash and independent clone placements. They store the same initial logical data and follow the same public rebuild schedule. Replace the external oracle of Section 4 by the private failure result $\perp$.

For a logical **Read**(addr) or **Write**(addr, data), proceed as follows.

1. Set $\text{found} = \text{False}$. In order $i = 1, \dots, \gamma$, issue $\text{ORAM}_i.\text{ReadRm}(\text{addr})$ if found is false, and issue $\text{ORAM}_i.\text{ReadRm}(\perp)$ otherwise. On the first successful real call, save its returned value $\text{data}^* \neq \perp$ and set $\text{found} = \text{True}$. Retain each replica's own depth dep_i for its subsequent insertion.
2. For a read, return data^* and set $\text{data} = \text{data}^*$. For a write, use the requested new value data . Call $\text{ORAM}_i.\text{Add}(\text{addr}, \text{data}, \text{dep}_i)$ for every replica and perform the scheduled rebuilds.

If every replica fails, the full ORAM reports failure. We prove below that this event has negligible probability over any polynomially bounded execution. Success flags and depths are internal states of the execution and thus not revealed to the adversary by our security definition.

6 Proofs for the Full ORAM

6.1 A Limited-Independence Occupancy Bound

We first prove a generic hashing lemma that is independent of the ORAM parameter choices introduced below.

Lemma 6.1 (Selected late arrivals). *Fix a sequence A of at most m distinct keys and a subset $D \subseteq A$. For integers $r \geq 4$ and $s \geq 1$, hash the keys uniformly into m bins using $(r+1)s$ -wise independence. Call a key late if at least r keys preceding it in A hash to its bin. Except with probability $2^{-\Omega(s)}$, the number of late keys in D is at most*

$$C_0 \left[s + |D| \left(\frac{e|A|}{rm} \right)^{r/2} \right], \quad (2)$$

where C_0 is an absolute constant.

Proof. We prove a more general tail bound, replacing $|A| \leq m$ by the weaker assumption $e|A| \leq rm$. Let $\text{Late}(D)$ count the late keys in D , and write

$$\eta(v, m) = \left(\frac{ev}{rm} \right)^{r/2}, \quad \eta = \eta(|A|, m).$$

We will show that, for every integer $z \geq 2s$,

$$\Pr[\text{Late}(D) \geq z] \leq \left(\frac{4e^2|D|(s/m + \eta)}{z} \right)^s. \quad (3)$$

If $|A| < r$ or $|D| < s$, the claimed bounds are immediate. Otherwise, we bound $\mathbb{E} \binom{\text{Late}(D)}{s}$: this is the expected number of ways to select s late keys from D . A large value of $\text{Late}(D)$ would produce many such selections, so a bound on this expectation will give a tail bound for $\text{Late}(D)$.

We separate these selections according to how many bins they occupy. For $1 \leq j \leq s$, define

$$T_j := \sum_{\substack{S \subseteq D \\ |S|=s}} \Pr[\text{all keys of } S \text{ are late and } S \text{ occupies exactly } j \text{ bins}].$$

By linearity of expectation, $\mathbb{E} \binom{\text{Late}(D)}{s}$ is the sum, over all s -element subsets $S \subseteq D$, of the probability that all keys of S are late. Partitioning each event by the number of bins occupied by S gives $\mathbb{E} \binom{\text{Late}(D)}{s} = \sum_{j=1}^s T_j$. We call the keys in S the *selected keys*; they are the keys whose lateness we are counting.

Fix such a selection occupying j bins. In each occupied bin, consider the selected key that occurs first in the order on A . Since this key is late, at least r keys of A occur before it in the same bin. Choose any r of them. We call these predecessors *witnesses*: their presence demonstrates that the earliest selected key, and hence every selected key in this bin, is late. Witnesses are only a device for the proof; the algorithm does not find or store them.

None of these witnesses is a selected key, because it precedes the earliest selected key in its bin. Nor can two different bins share a witness, since a key hashes to only one bin. The s selected keys and the rj witnesses are therefore all distinct. Several selected keys in one bin can share the same r witnesses. This is why we count witnesses per occupied bin, rather than assume that the selected keys' lateness events are independent.

For an upper bound, there are $\binom{m}{j}$ choices of occupied bins, $\binom{|D|}{s}$ choices of selected keys, at most j^s assignments of those keys to the bins, and at most $\binom{|A|}{r}^j$ choices of predecessor sets. Some of these choices may be invalid, e.g., with empty bins or overlapping witnesses. However, we may count these invalid choices for the purpose of obtaining an upper bound. For each valid choice, all $s + rj$ distinct keys must hit specified bins. Its probability is exactly $m^{-(s+rj)}$, because $s + rj \leq (r+1)s$. Every selection counted by T_j has at least one such witness choice. A union bound therefore gives

$$\begin{aligned} T_j &\leq \binom{m}{j} \binom{|D|}{s} j^s \binom{|A|}{r}^j m^{-(s+rj)} \\ &\leq \left(\frac{e|D|j}{sm} \right)^s \left(\frac{em}{j} \eta^2 \right)^j, \end{aligned}$$

where the last inequality uses $\binom{a}{b} \leq (ea/b)^b$ and $\eta^2 = (e|A|/(rm))^r$.

It remains to bound this expression uniformly over j . Write $\theta = j/s \in (0, 1]$. Taking s th roots and rearranging yields

$$T_j^{1/s} \leq e^{1+\theta} \left(\frac{|D|\theta}{m} \right)^{1-\theta} \left(\frac{|D|\eta^2}{s} \right)^\theta \leq e^2 \max \left\{ \frac{|D|}{m}, \frac{|D|\eta}{s} \right\}.$$

Indeed, $\theta \leq 1$ and $\eta \leq 1$, and a weighted geometric mean is at most the larger of its two factors. Summing over the at most $s \leq 2^s$ possible values of j gives

$$\mathbb{E} \binom{\text{Late}(D)}{s} \leq \left(2e^2 \max \left\{ \frac{|D|}{m}, \frac{|D|\eta}{s} \right\} \right)^s.$$

Finally, if $\text{Late}(D) \geq z$, then $\binom{\text{Late}(D)}{s} \geq \binom{z}{s}$. Since $z \geq 2s$ implies $\binom{z}{s} \geq (z/(2s))^s$, Markov's inequality gives

$$\Pr[\text{Late}(D) \geq z] \leq \frac{\mathbb{E} \binom{\text{Late}(D)}{s}}{\binom{z}{s}} \leq \left(\frac{4e^2 \max\{|D|s/m, |D|\eta\}}{z} \right)^s.$$

Bounding the maximum by the sum proves (3). When $|A| \leq m$, the condition $e|A| \leq rm$ holds because $r \geq 4$. We also have $|D| \leq m$, so the numerator is at most $4e^2(s + |D|\eta)$. Taking z to be a sufficiently large constant multiple of $s + |D|\eta$ makes the probability $2^{-\Omega(s)}$, proving the lemma's bound (2); integer rounding is absorbed in the constant. If a witness would require more keys than the hash domain contains, that choice is impossible and contributes nothing to the count. $\square$

6.2 Setup for the ORAM Analysis

We analyze polynomially many requests using $M = \Theta(B)$ client memory; Section C handles larger client memory.

∞ -hybrid. As with our analysis of a single semi-functional ORAM, we analyze an ∞ -hybrid of the full ORAM construction from Section 5.2, in which

- (i) Buckets and collision sets have unbounded capacity (stashs retain their original finite capacity).
- (ii) Oblivious primitives, such as sorting, are assumed to never fail.

(iii) If all replicas fail, an (oblivious) oracle will supply the correct answer.

It is straightforward, given Lemma 4.10 from our single semi-functional ORAM analysis, to see that this hybrid is both correct and oblivious (the simulation is spelled out below). It remains to show that the real algorithm behaves identically to this hybrid, except with negligible probability. (i) and (ii) have a negligible impact on execution, just as we saw with the single semi-functional ORAM analysis. Therefore, our main focus will be on point (iii): except with negligible probability, every request is answered correctly by some replica.

Technical challenges. A logical request is sent as a real read to the first semi-functional ORAM and remains real in successive replicas until one succeeds. In other words, the real calls reaching replica $i + 1$ are exactly the calls that fail in replica i . The key property this enables (as in other multi-tier hashing schemes [ACMR95, CGLS17]) is that, as fewer and fewer real calls are made into each successive replica, the chance that enough hashes collide to exhaust any given group plummets, and thus not just the number but the fraction of failures in each successive replica plummets. This leads to a doubly exponentially decreasing number of real calls made into each successive ORAM. The main technical hurdles we need to handle (relative to other multi-tier hashing schemes) are, first, the limited independence of our hash functions, handled in Lemma 6.1, and, second, interactions between the different levels in each semi-functional ORAM—failures at any level send requests to the next replica, where they may reach different levels—which the majority of the analysis is devoted to handling.

This interaction is also why the first two lemmas are phrased in a somewhat strange way. In Lemma 6.1, we hash a whole sequence of keys into bins, but count the overflows only among a subset of that sequence. Likewise, in Lemma 6.2, we count the failures occurring in an interval J of a level’s epoch, while measuring lateness against all of the real calls made since that epoch’s reset. In both cases the reason is the same: a failure inside J can be caused by calls made earlier in the epoch, but failures outside of J must not be charged to J .

The reason we need such intervals is the interaction between levels. Every real access to a level ℓ of any replica after the first is caused by a call that failed at some level ℓ' of the previous replica, and there is no reason for ℓ' to equal ℓ . Thus, to bound the number of real accesses to level ℓ during one of its epochs, we must bound the number of failures at *every* level ℓ' of the previous replica during that same epoch. Since level ℓ' has epoch length $q^{\ell'-1}$, the epoch for ℓ sits differently inside level ℓ' ’s epoch structure depending on whether ℓ' is larger or smaller than ℓ , and this is what forces us to prove two separate bounds. For a level $\ell' > \ell$, the epoch for ℓ is contained in the (longer) epoch for ℓ' , so what we need is the number of failures at level ℓ' during a subinterval of its own epoch; this is exactly what the fragment bound (6) provides. For a level $\ell' < \ell$, by contrast, the epoch for ℓ contains many complete epochs for ℓ' , so we must sum the failures over all of them. The fragment bound applies here too, but its additive “fudge factor” ($+s$) would then be paid once per epoch, which we cannot afford; the whole-epoch bound (7) removes the additive term, at the cost of only applying to intervals that begin at a reset. Finally, Lemma 6.2 has a third bound for the case that the level is lightly loaded, that is, so few real calls have been made to it since the reset that everything requiring stashing fits in the stash and there are no failures at all.

Proof roadmap. Lemma 6.1 is a somewhat specialized balls-and-bins bound, counting the number of overflows in bins with fixed capacity under limited independence (in a subset of a sequence of insertions). Lemma 6.2 bounds failures within arbitrary epoch subintervals and gives a stronger bound, without an additive term, for intervals starting at the reset. Lemma 6.3 combines these estimates across levels and epochs to establish a rapid-decrease bound for real calls across replicas

on every time interval. Finally, Lemma 6.4 applies the sparse case to show that the last replica never fails, except with negligible probability.

Simulation. The proof of Lemma 4.10 shows that the adversary's view of one semi-functional ORAM in the ∞ -hybrid is identically distributed as always seeing correct answers and the simulator's access patterns. The same simulation applies independently to every replica of the full ORAM: every request accesses all replicas on a fixed schedule, real and fake calls have identical access-pattern distributions, rebuild times and array lengths are public, ideal subroutines reveal only their simulated traces, and padded scans hide pointer values, stash decisions, and which source supplied a node. We sample all simulator coins independently of the grouping hashes $H_1, \dots, H_\gamma$.

For completeness, this is a joint simulation of the whole read sequence, not merely the observation that each individual bucket is marginally uniform. Expose a clone's bucket location only when a pointer first selects that clone. By Fact 5.1, a read whose counter is below q selects the next unused clone, whose location is still independent and uniform. A read whose counter is at least q scans a fresh uniform bucket; all locations of the requested node have already been exposed, so a rescue from that bucket does not condition any unused clone. Assigning each newly selected clone the bucket sampled by the simulator gives exactly the same joint distribution as choosing all clone locations independently during the rebuild.

Conditioning. By the simulation, fixing the adversary's and simulators' coins fixes the logical request sequence independently of the grouping hashes. When analyzing replica i , we also fix $H_1, \dots, H_{i-1}$. Together with the fixed coins, which determine every exposed clone location and random bucket, these fix the executions of replicas $1, \dots, i-1$ and hence the real/fake sequence entering replica i , while H_i keeps its original distribution. So the hashing bounds apply to that fixed sequence.

Proof parameters and notation. For the tail bounds, write

$$r = \lceil (\log N)^\epsilon \rceil, \quad s = \lceil (\log N)^{1+2\epsilon} \rceil.$$

Both are analysis parameters. The scheme still has q clones per node, and an actual failure requires q earlier candidate visitors to the same node. Witnessing s such failures directly could therefore involve qs hash values. Instead, we choose $r \leq q$ and count every node with only r candidate visitors as potentially problematic. This overcounting includes every node that can actually fail, while reducing the witness size to $O(rs)$ and hence the required hash independence to $O(rs)$. Let $k = \lceil (\log N)^{1+6\epsilon} \rceil$ denote the independence of each grouping hash. For all sufficiently large N ,

$$4 \leq r < q, \quad rs = \Theta((\log N)^{1+3\epsilon}) = o(k), \quad k = O(B).$$

Here $r < q$ follows from $q \geq \frac{1}{2}(\log N)^{\epsilon(1+6\epsilon)}$. Every fixed-length prefix of H is uniform and k -wise independent. We also have

$$q^2 s = o(B), \quad r \log r = \omega(\log L), \quad \frac{q^4 \log N}{B} = o(1). \quad (4)$$

Indeed, at the smallest allowed B , the exponent of $q^2 s$ is $1+4\epsilon+12\epsilon^2 < 1+6\epsilon$, and $(1+6\epsilon)(1-4\epsilon) > 1$; larger B improves both ratios. Finally, $L = O(\epsilon^{-1}(1 + \log_B N))$, so $\log L = O(\log \log N)$. These estimates also verify that the hash description fits in one $O(B)$ -word block.

We now define some fixed set notations. Fix a level $\ell \geq 1$ and a whole clone-reset epoch, or a prefix E starting at its reset. Define $A_i(E)$ to be the distinct addresses of real calls in E , ordered

by their first real occurrence. By the conditioning order above, $A_i(E)$ is fixed independently of H_i . The set of addresses that actually visit level ℓ may depend on H_i , but every such visitor belongs to $A_i(E)$. Moreover, an address that first fails at level ℓ has at least q earlier addresses of $A_i(E)$ in its bin. We therefore count collisions in the fixed superset $A_i(E)$; we never claim that the actual visitor set is independent of H_i .

A second fixed set is needed to bound the complete contents of nodes that may need stashing, because fake reads are also followed by real insertions. Let W_E contain the addresses supporting all items at level ℓ immediately after the reset beginning the containing epoch. This set depends only on the reset, so the same W_E works for every prefix of that epoch and is fixed independently of H_i . For a nonlast level, take all addresses from the at most q^ℓ operations since its last emptying into the next level, up to this reset; for the last level, take $[N]$. Thus

$$|W_E| \leq q^\ell. \quad (5)$$

Each clone contains at most one item, with its attached marker, per address. The retained marker depths can depend on H_i , but their supporting addresses are contained in this fixed set. A node's complete size is at most $C(q^2 + |\{\text{addr} \in W_E : H_i(\text{addr})[\ell] = u\}|)$ words for an absolute C .

6.3 Failure and Stash Bounds

We first carry out the argument in the ∞ -hybrid. The conditioning order above justifies, replica by replica, why the candidate sequence can be fixed independently of the current replica's hash. At the end, bucket and collision-set bounds transfer the conclusion to the real finite-capacity algorithm.

Lemma 6.2 (Failure and stash estimate). *Fix replica i and a level $\ell \geq 1$. Write $p := q^{\ell-1}$ for the length of a level- ℓ clone-reset epoch and $m := q^\ell = qp$ for the number of level- ℓ logical nodes. Fix a contiguous interval J contained in one such epoch. Let A contain the distinct addresses of real calls to replica i from that epoch's reset through the end of J , ordered by their first real occurrence since the reset; thus $|A| \leq p$. Recall that $r = \lceil (\log N)^\epsilon \rceil$ and $s = \lceil (\log N)^{1+2\epsilon} \rceil$. Let $R_i(J)$ count the real calls to replica i occurring in J , and let $\text{Fail}_\ell(J)$ count those calls whose search in this replica first fails at level ℓ . There is an absolute constant C_1 such that, except with probability $2^{-\Omega(s)}$,*

$$\text{Fail}_\ell(J) \leq C_1 \left[s + R_i(J) \left(\frac{e|A|}{rm} \right)^{r/2} \right]. \quad (6)$$

If J starts at the epoch's reset, then, except with probability $2^{-\Omega(s)}$,

$$\text{Fail}_\ell(J) \leq C_1 |A| \left(\frac{e|A|}{rm} \right)^{r/2}. \quad (7)$$

If $|A|(e|A|/(rm))^{r/2} < s$, then, except with probability $2^{-\Omega(s)}$, $\text{Fail}_\ell(J) = 0$: all complete nodes ever requiring stashing from the reset through the end of J fit together in the one-block stash.

Proof. Let E denote the interval from the epoch's reset through the end of J , so that $A = A_i(E)$ and $J \subseteq E$. For brevity in the proof, write $\eta = (e|A|/(rm))^{r/2}$.

Failures are late arrivals. Let $D_J \subseteq A$ contain the addresses whose first real occurrence lies in J . There are at most $R_i(J)$ such addresses. By Fact 5.1, an address that first fails at level ℓ has at least q earlier addresses of A in its bin. Since $r \leq q$, it is counted by $\text{Late}(D_J)$. We select addresses from D_J because we are counting failures that occur in J . However, we measure whether each

selected address is late relative to all of A , since the earlier addresses in its bin may have arrived before J .

Applying Lemma 6.1 and using $|D_J| \leq R_i(J)$ proves (6). Applying it instead with $D = A$ gives

$$\text{Late}(A) \leq C_0(s + |A|\eta). \quad (8)$$

If $|A|\eta \geq s$, the right-hand side is at most $2C_0|A|\eta$. When J starts at the reset, $J = E$ and $\text{Fail}_\ell(J) \leq \text{Late}(A)$, proving (7) in this case.

The sparse case. It remains to consider $0 < |A|\eta < s$; the case $A = \emptyset$ is immediate. Bound (8) alone would allow $O(s)$ late arrivals. We show that all nodes responsible for those arrivals fit together in the stash, so none of them causes an actual failure.

Call a bin *hot* if it contains at least $r+1$ keys of A , and let J_0 be the number of hot bins. Every node that can exhaust its q clones is hot, because $r < q$. Moreover, each hot bin contains at least one key counted by $\text{Late}(A)$. Therefore,

$$J_0 \leq \text{Late}(A) \leq C_0(s + |A|\eta) = O(s),$$

where the second inequality follows from Lemma 6.1 and the last uses $|A|\eta < s$. Hence there are at most C_2s hot bins, for an absolute constant C_2 , except with probability $2^{-\Omega(s)}$.

Next we count how many addresses of A lie in the hot bins. If L_b is the A -load of bin b , then the first r keys in a hot bin are not late and all its remaining keys are late. Consequently,

$$\sum_{b: L_b \geq r+1} L_b = rJ_0 + \text{Late}(A) = O(rs). \quad (9)$$

Here we used $|A|\eta < s$ in (8). Thus the items in the hot nodes whose addresses belong to $W_E \cap A$ occupy $O(rs)$ words.

Records introduced by fake calls. The hot nodes can also contain records whose addresses are not in A , because fake reads are followed by real adds. Put $U = W_E \setminus A$. This is a fixed set disjoint from A , with $|U| \leq m$. We must bound the number of U -keys that hash into bins made hot by A . Lemma 6.1 does not count these keys, since they do not belong to A .

We cannot first reveal the hot bins and then treat them as fixed, since their identities depend on the same hash function. Instead, count both events in one witness. Set $\mu = m\eta^2$. Since $|A|/m \leq 1/q$ and $q \geq r > 2$,

$$\mu = |A|\eta \frac{\eta}{|A|/m} \leq sq \left(\frac{e}{rq} \right)^{r/2} \leq sr \left(\frac{e}{r^2} \right)^{r/2} = o(s).$$

Let $w = \lceil C_3rs \rceil$, and let $\mathcal{W}$ be the event that, for some $j \leq C_2s$, there are j bins containing r distinct A -keys each and w distinct U -keys in their union. If w U -keys lie in the hot bins, then $\mathcal{W}$ occurs. Each such witness contains $rj + w = O(rs) \leq k$ distinct keys, and a union bound gives

$$\begin{aligned} \Pr[\mathcal{W}] &\leq \sum_{j=1}^{C_2s} \left(\frac{e\mu}{j} \right)^j \left(\frac{ej}{w} \right)^w \\ &\leq \exp[-\Omega(rs \log r)]. \end{aligned} \quad (10)$$

The first factor counts the evidence that the j bins are hot, and the second forces the chosen U -keys into their union. For the last inequality, $(e\mu/j)^j \leq e^\mu = \exp(o(s))$, whereas $ej/w = O(1/r)$ and

$w = \Theta(rs)$. Thus, with probability at least $1 - \exp[-\Omega(rs \log r)]$, only $O(rs)$ keys of U lie in hot bins. Impossible witnesses contribute zero.

The hot nodes fit in the stash. There is at most one item, including its depth marker, per supporting address. Equations (9)–(10) therefore bound the items in all hot nodes by $O(rs)$ words. Their child pointers add $O(q^2 J_0) = O(q^2 s)$ words. Hence all complete hot nodes together occupy

$$O(rs + q^2 s) = O(q^2 s) = o(B)$$

words, which is below the one-block stash capacity for sufficiently large N .

The stash is empty at the reset. By Fact 5.1, a node that may later exhaust its clones first has a valid q th visit, when the algorithm attempts to copy the complete node into the stash. Each node is attempted only once during the epoch. Since the sum of the complete sizes of all nodes that may be attempted is below the stash capacity, every atomic insertion succeeds. Subsequent visits use the complete stash copy, so no failure occurs in E , and hence none occurs in J . Combining the exceptional events above gives probability $2^{-\Omega(s)}$. This proves the sparse case and completes the lemma. $\square$

6.4 Rapid Decrease on Every Time Interval

For any interval I , let $R_i(I)$ count real calls to replica i , and let $\text{Fail}_{i,\ell}(I)$ count those calls whose search first fails at level ℓ . A real call reaches replica $i + 1$ exactly when it fails in replica i , and every failing call has one first failing level. Therefore, in the hybrid, the full ORAM satisfies the deterministic identity

$$R_{i+1}(I) = \sum_{\ell=1}^{L-1} \text{Fail}_{i,\ell}(I). \quad (11)$$

The root contributes zero. No same-level relation between successive replicas is assumed.

We prove a bound for every I , because an interval aligned with one level's epochs need not be aligned with another level's longer epochs. For a fixed level, partition I into complete epochs and at most two *fragments*: the intersections with its boundary epochs. A fragment need not start at a reset. Its earlier epoch history can cause failures inside I , but failures outside I should not be charged to I . This is precisely what the selected subset in Lemma 6.2 handles.

Lemma 6.3 (Rapid decrease on all intervals). *For sufficiently large absolute constants C_a, C_r , set*

$$\begin{aligned} a &= C_a L s, & \rho_1 &= 1, & \delta(\rho) &= \left(\frac{2e\rho}{rq} \right)^{r/2}, \\ \rho_{i+1} &= C_r L \rho_i \delta(\rho_i). \end{aligned}$$

Except with probability $\text{negl}(N)$, simultaneously for all replicas $i \leq \gamma$ and every contiguous time interval I ,

$$R_i(I) \leq a + \rho_i |I|.$$

Moreover $\rho_\gamma = N^{-\omega(1)}$.

Proof. The base case follows from $R_1(I) \leq |I|$. Equation (4) implies $L\delta(1) = o(1)$, so the recurrence has $0 < \rho_{i+1} \leq \rho_i \leq 1$ for sufficiently large N . Fix the simulated request sequence and preceding hashes satisfying the induction hypothesis for replica i . Its input sequence is fixed and H_i retains its original independence distribution.

Fix I and a level with epoch length p . First consider any complete epoch, or the reset prefix through the end of a boundary fragment, for which $\rho_i p < a$. Its candidate set has

$$v \leq \min\{p, 2a\}, \quad v\eta(v, qp) \leq 2a \left(\frac{e}{rq}\right)^{r/2} = o(s).$$

Here $a = C_a Ls$, $q \geq r$, and $r \log r = \omega(\log L)$. The small-load case of Lemma 6.2 makes all these pieces contribute zero failures.

For a complete epoch with $\rho_i p \geq a$, the candidate count is $v \leq 2\rho_i p$, so

$$\eta(v, qp) \leq \left(\frac{2e\rho_i}{rq}\right)^{r/2} = \delta(\rho_i).$$

Equation (7) bounds its failures by $C\rho_i p \delta(\rho_i)$. Summing full epochs over all levels gives

$$\text{full-epoch contribution} \leq CL\rho_i \delta(\rho_i) |I|. \quad (12)$$

For a remaining boundary fragment J , take the epoch prefix E through its right endpoint. Again $|A_i(E)| \leq 2\rho_i p$. Apply (6) to J , whose candidate set is $A_i(E)$. The bound is $C(s + R_i(J)\delta(\rho_i))$. Across all levels there are at most $2L$ such fragments. Since $R_i(J) \leq a + \rho_i |J|$, their sum is at most

$$CLs + CL\delta(\rho_i)a + CL\rho_i \delta(\rho_i) |I|. \quad (13)$$

Choose C_a so the first term is at most $a/4$. For large N the second is at most $a/4$, since $CL\delta(\rho_i) \leq CL\delta(1) = o(1)$. Choose C_r to cover the two density terms in (12) and (13). Identity (11) now gives $R_{i+1}(I) \leq a + \rho_{i+1} |I|$.

Only polynomially many estimates are used: for example, $O(\gamma LT^3)$ choices of replica, level, epoch prefix, and interval suffice for a length- T execution. Each conditional error probability is $2^{-\Omega(s)}$, uniformly in the preceding hashes. Since $s = \omega(\log N)$, sequential conditioning and a union bound give a negligible total error. In particular, we do not assume that current-hash tail events are independent or condition on those events before applying the occupancy lemma.

To solve the recurrence, write $x_i = \log(1/\rho_i)$. Then

$$\begin{aligned} x_{i+1} &= (1 + r/2)x_i + (r/2) \log(rq/(2e)) - \log(C_r L) \\ &\geq (r/2)x_i + (r/2) \log r \end{aligned}$$

for large N , using $q \geq r$ and $r \log r = \omega(\log L)$. Thus $x_i \geq (r/2)^{i-1} \log r$ for $i \geq 2$. Since $r \geq (\log N)^\epsilon$ and $\gamma = \lceil 2/\epsilon \rceil + 2$, we have $\epsilon(\gamma - 1) \geq 2 + \epsilon > 1$, and hence $x_\gamma = \omega(\log N)$, proving the claim. $\square$

Lemma 6.4 (Success of the last replica). *Except with probability $\text{negl}(N)$, replica γ never fails in the hybrid execution.*

Proof. Every nonroot epoch has length $p \leq q^{L-2} < N$. Lemma 6.3 bounds the real calls in each complete epoch and each final observed reset prefix by $a + \rho_\gamma p \leq a + 1$. Condition on the preceding hashes, which fix the incoming requests to replica γ independently of H_γ . Its candidate count satisfies

$$v\eta(v, qp) \leq (a + 1) \left(\frac{e}{rq}\right)^{r/2} = o(s).$$

Lemma 6.2 gives zero failures there. Union bound over the epochs and use that the root cannot fail. The whole burst a need not fit in a stash: only complete nodes with at least q candidate visitors are stashed. They form a subset of the nodes with at least r candidates, whose combined mass is $O(q^2 s) = o(B)$. $\square$

6.5 Correctness, Obliviousness, and Efficiency

Theorem 6.5 (Full ORAM). *Fix $0 < \epsilon < 1/48$ and $B \geq (\log N)^{1+6\epsilon}$. The construction above is correct and statistically oblivious for polynomially many adaptive requests, except with probability $\text{negl}(N)$. With the space reduction of Section A and the caching strategy of Section C, it stores N words using $O_\epsilon(N)$ server words and, for $B \leq M < N$, incurs*

$$O(\epsilon^{-2}(1 + \log_B(N/M))) \quad (14)$$

amortized block transfers per request, after an initialization costing $O_\epsilon(N)$ block transfers.

Proof. In this section, we first prove a block transfer bound of $O(\epsilon^{-2}(1 + \log_B N))$. Section C later shows that we can improve the bound to $O(\epsilon^{-2}(1 + \log_B(N/M)))$ by caching the smallest levels of the data structure in the client's private cache. We transfer the hybrid analysis to finite buckets and collision sets, then account for space and block transfers.

Buckets and collision sets. Apply Lemma 4.3. At any rebuilt level, items have supporting addresses in the fixed set W_E of (5), with at most one item per address. The original bucket proof therefore applies to this superset, with a constant-factor increase in variable-record size; the q^2 child pointers and independent clone placement are unchanged. The unchanged hash independence supplies all required moments. Increase the bucket constants accordingly and union bound over the polynomial execution.

For clarity, the collision-set estimate can be shown directly. Fix a request and a level $\ell \geq 1$, and write $p := q^{\ell-1}$ for its clone-reset epoch length and $m := q^\ell = qp$ for its number of logical nodes. Take all at most p distinct real-call addresses in the epoch as candidates. Excluding the target itself, let X count those sharing its length- ℓ prefix. For s specified candidates and the target, independence gives collision probability m^{-s} , so

$$\mathbb{E} \binom{X}{s} \leq \binom{p}{s} m^{-s} \leq \left(\frac{e}{qs} \right)^s.$$

For any fixed $c > 0$, since $s = o(B)$,

$$\Pr[X \geq cB] \leq \frac{(e/(qs))^s}{(cB/(2s))^s} = \left(\frac{2e}{cqB} \right)^s = \text{negl}(N).$$

If fewer than s candidates exist the event is impossible. Allowing the target itself adds at most one, and depth filtering only reduces the set. Choose c to account for the constant-size address–depth records. Filter while inserting into col , so intermediate updates do not form an unfiltered union. Union bound as in Section 6.2.

Correctness and statistical obliviousness. On these capacity events and success of the oblivious primitives, the real implementation agrees with the hybrid. Rebuilds preserve the freshest data values and, independently, the maximum marker depths. Valid-node searches therefore return correct data, and Lemma 6.4 ensures that some replica succeeds. The silent oracle is never needed. The hybrid simulator from Section 6.2 transfers to the real execution by a union bound on these exceptional events. This proves adaptive statistical obliviousness and correctness, without assuming independence between bucket overflow and stash events.

Cost of the full ORAM on N addresses. Reads cost $O(L)$ probes per replica. A level- ℓ rebuild processes $O(q^{\ell+3})$ words, with whole-block padding, and occurs every $q^{\ell-1}$ operations. To retain both fields, apply the oblivious deduplication primitive of Section 3.3 to two temporary copies of the items, ordered by address and respectively by decreasing freshness and decreasing depth. An oblivious sort by address brings the two retained entries together; a padded scan combines the freshest value and maximum depth into one item. This uses a constant number of sorts and scans. The usual rebuild calculation gives

$$O(L) + O\left(\sum_{\ell=1}^{L-1} \frac{q^{\ell+3} \log N}{Bq^{\ell-1}}\right) = O(L) + O\left(\frac{Lq^4 \log N}{B}\right) = O(L). \quad (15)$$

The first term covers whole-block scans and the small levels. One replica uses $O(q^{L+2} + LB) = O(Nq^4 + LB)$ words, including stashes and bucket rounding. Here we retain the possible factor q from rounding q^{L-1} up above N . Its seed occupies $O(B)$ words; seeds for inactive replicas can be kept in fixed server locations and loaded in $O(1)$ probes. Multiplying by $\gamma = O(1/\epsilon)$ gives $O(\epsilon^{-2}(1 + \log_B N))$ amortized probes and $O(\epsilon^{-1}(Nq^4 + LB))$ server words before the linear-space wrapper. Finally, initialization performs one rebuild of every level in each replica, plus the initial bin placement of Section A. Since $q^{L+2} < q^4 N$, (4) bounds both by $O(N)$ transfers.

Linear space and larger client memory. A straightforward analysis based on Section 4.4 would show a space requirement of $O_\epsilon(B^{3\epsilon}N)$ where $O_\epsilon(\cdot)$ hides an $O(1/\epsilon)$ factor coming from the number of replicas. In Section A, we shall describe additional tricks for making the ORAM’s space consumption at most $O_\epsilon(N)$.

Section C shows that by caching the smaller levels of the data structures in the client’s private cache, we can replace the $O(\epsilon^{-2}(1 + \log_B N))$ with $O(\epsilon^{-2}(1 + \log_B(N/M)))$ in the stated block transfer bounds. $\square$

7 Oblivious Search Tree

In this section, we present our oblivious ordered dictionary (henceforth also called an oblivious ST), supporting exact **Lookup**, strict **Successor** and **Predecessor** queries, **Insert**, and **Delete** at an amortized cost of $O(\log_B N)$ per operation whenever $B \geq \log^{2+\epsilon} N$. We refer the reader to Section 2.2 for an informal overview of the construction, and to Figure 2 for a worked example. At a high level, we store a logical search tree directly inside a hierarchical data structure of geometrically growing levels. Unlike our ORAM, we do not clone nodes; instead, every node that is touched is consumed, and its entire contents are reinserted at the top of the hierarchy. The duplication of keys along parent–child edges — reminiscent of the fractional cascading used in the lookahead array [BFCF⁺07] — is what makes these reinsertions cheap to reconcile.

7.1 Parameters and Notation

Throughout the section, N denotes an upper bound on the number of keys that are simultaneously stored, and $\epsilon \in (0, 1)$ is an arbitrarily small constant such that $B \geq \log^{2+\epsilon} N$. We assume that keys are distinct and that a key, a value, or a pointer each fit in a single word. Instances that fit in private memory are handled by fixed-length scans there. Further, we assume N is at least a sufficiently large constant; smaller N can be handled by a linear scan.

Arity. Define the *arity parameter*

$$q := B^\eta, \quad \eta := \frac{\epsilon}{3(2 + \epsilon)},$$

rounded to the nearest integer, so that

$$\frac{B}{q^3} = \Theta(B^{1-3\eta}) = \Theta\left(B^{\frac{2}{2+\epsilon}}\right). \quad (16)$$

Every node of the logical tree holds exactly $2q$ slots, hence fits in $O(1)$ blocks.

Levels. Let $L := \lceil \log_q N \rceil$ be the number of levels of the hierarchy, numbered $0, 1, \dots, L - 1$. Level 0 consists of a single node. Letting

$$m_\ell := \min\left(q^\ell, \lceil N/(2q) \rceil\right),$$

level $\ell > 0$ has m_ℓ ordinary node slots in the meta-algorithm. The oblivious implementation adds further node slots, as described below. Since $q^{L-1} \geq N/q \geq \lceil N/(2q) \rceil$ for $N \geq 2q$, we always have $m_\ell \leq q^\ell$, and $m_{L-1} = \lceil N/(2q) \rceil$.

Rebuild period. Let

$$L^* := \lceil \log_q(10Lq) \rceil = O(1), \quad s := \min(L^*, L - 1).$$

Note that $q^{L^*} \geq 10Lq$ and $q^{L^*} \leq 10Lq^2$. Levels 0 through s are rebuilt on *every* operation; each level $\ell > s$ is rebuilt every $q^{\ell-L^*}$ operations. Concretely, let $j(t)$ be the largest $j \geq 0$ such that q^j divides t . On operation t , rebuild level

$$\ell^*(t) := \min(L - 1, L^* + j(t)). \quad (17)$$

This schedule can be kept public, as it depends only on the operation count.

7.2 Data Structure

Item. An item is a key-value pair $\text{item} := (k, v)$. We reserve the key ∞ , which lies outside the application's key space, for *padding* items. We write $\text{item}.k$ and $\text{item}.v$ for the key and the value.

Logical tree, entry and node. An *entry* is a pair $(\text{item}, \text{ptr})$ consisting of an item and a (possibly null) pointer. A pointer stores the pair $(\text{level}, \text{index})$ identifying a node slot in the hierarchy. A *node* is a list of exactly $2q$ entries

$$\text{item}_1, \text{ptr}_1, \text{item}_2, \text{ptr}_2, \dots, \text{item}_{2q}, \text{ptr}_{2q}.$$

Note that a node carries as many pointers as items, which deviates from the usual B -tree layout; the duplication invariant below is what makes this work, and it leads to a simpler rebuild.

Within a node, real entries come first; the remaining slots are padding entries whose item has key ∞ and whose pointer is null. We require the following invariants of the logical tree.

- *(I1) Orderedness.* The real keys of a node are strictly increasing: $\text{item}_1.k < \text{item}_2.k < \dots < \text{item}_r.k$ where $r \leq 2q$ is the number of real entries. (Padding keys are all ∞ and are exempt from strictness.) Additionally, if ptr_i is not null, all real keys in the subtree rooted at the node pointed to by ptr_i lie in the interval $[\text{item}_i.k, \text{item}_{i+1}.k)$, letting $\text{item}_{2q+1}.k = \infty$.

- (I2) *Duplication*. If ptr_i is non-null and points to a node Y , then $item_i.k = item_1^Y.k$, that is, $item_i$ and the first item of Y are two copies of the same key.
- (I3) *Layout*. If a node at level ℓ points to a node at level ℓ' , then $\ell' > \ell$; and every logical node other than the unique level-0 node has exactly one parent. Here logical nodes are the unconsumed nodes containing real entries, together with the root; nodes containing only padding entries are not logical nodes.

Hierarchical storage. In the meta-algorithm, level 0 consists of one ordinary node, and each level $\ell > 0$ consists of m_ℓ ordinary nodes. An ordinary node is either a real node, containing at least one real entry, or a padding node, containing only padding entries. Each ordinary node additionally carries a one-bit *consumed* flag, cleared when the level is rebuilt and set when the node is fetched. Consumed nodes have no incoming pointers in the logical tree. Their flags additionally exclude their old contents from later rebuilds, including old copies of keys that have since been deleted. The total space is $\sum_{\ell=0}^{L-1} O(q \cdot m_\ell) = O(N)$ words, i.e. $O(N/B)$ blocks.

In the oblivious implementation of Section 7.5, each nonroot level is augmented with m_ℓ filler nodes, giving $2m_\ell$ physical node slots. The ordinary and filler nodes are randomly interleaved and are indistinguishable to the adversary. Each physical node carries a bit indicating whether it is a filler, and a filler additionally records the location of the next filler in its level.

7.3 Non-Oblivious Meta-Algorithm

As before, we first describe a meta-algorithm capturing the operational semantics, and then give its oblivious realization in Section 7.5.

Search(k, μ), where $\mu \in \{\text{eq}, \text{succ}, \text{pred}\}$

Output: the visited path Path , and the best candidate $item^{\text{ans}}$ for exact lookup, strict successor, or strict predecessor, according to μ .

Algorithm: Initialize $\text{Path} := \emptyset$, $item^{\text{ans}} := \perp$, and let ptr point to the unique node in level 0. Repeat:

1. Let X be the node at ptr ; append X to Path and mark its slot consumed.
2. Among the real items of X , let $item_i$ be the one with the largest key satisfying $item_i.k < k$ if $\mu = \text{pred}$, and $item_i.k \leq k$ otherwise (if one exists).
3. Select a real candidate $item'$ from X as follows, if one exists:

μ	candidate key
eq	k
succ	$\min\{item_j.k : item_j.k > k\}$
pred	$\max\{item_j.k : item_j.k < k\}$.

If a candidate exists, replace $item^{\text{ans}}$ when it is $\perp$, or when the candidate key is smaller for *succ* or larger for *pred*. For *eq*, keep the first exact match. (*Ties are never updated: the first copy seen along the path is the authoritative, freshest one.*)

4. If no $item_i$ exists in Step 2, or ptr_i is null, then stop. Otherwise set $ptr \leftarrow ptr_i$ and repeat.

**Lookup(k), Successor(k), Predecessor(k),
Insert(k, v), Delete(k)**

Run **Search**(k, μ) with $\mu = \text{succ}$ for **Successor**, $\mu = \text{pred}$ for **Predecessor**, and $\mu = \text{eq}$ for **Lookup**, **Insert**, and **Delete**, obtaining **Path** and item^{ans} . Flatten **Path** into a list **Relns** of entries, tagging every entry of the node fetched from level ℓ with $\text{tag} := \ell$ and supplying $2q$ padding entries for every skipped level. Then:

- **Lookup**(k), **Successor**(k), **Predecessor**(k): append one padding entry to **Relns**; the output is $\text{item}^{\text{ans}}.v$, or $\perp$ if $\text{item}^{\text{ans}} = \perp$.
- **Insert**(k, v): append to **Relns** the entry $((k, v), \text{null})$ with $\text{tag} := -1$; the output is $\perp$.
- **Delete**(k): replace every entry of **Relns** whose item has key k by a padding entry, and append one further padding entry; the output is $\perp$.

In every case **Relns** has exactly $2Lq + 1$ entries. Finally, let $\ell^* := \ell^*(t)$ as in (17), where t is the index of the present operation, and call **RebuildPrefix**(ℓ^*, Relns).

RebuildPrefix(ℓ^*, Relns)

1. *Collect.* Let **Arr** be the concatenation of **Relns** with the entries of all ordinary nodes of levels $0, 1, \dots, \ell^*$, where an entry taken from level ℓ gets $\text{tag} := \ell$. Replace every entry from a consumed node by a padding entry; existing padding entries remain padding.
2. *Sort.* Sort **Arr** by the key $\text{item}.k$ as the primary ascending key and tag as the secondary ascending key. Padding entries (with key ∞) are moved to the end.
3. *Deduplicate.* For each group of consecutive entries sharing a real key k , emit a single entry whose item is the item of the *first* entry of the group (smallest tag, i.e. the authoritative copy) and whose pointer is the pointer of the *last* entry of the group (largest tag), and delete the remaining entries of the group. Compact the survivors to the front of **Arr**, padding the rest.
4. *Truncate.* Truncate **Arr** to its first $2q \cdot m_{\ell^*}$ entries. (Lemma 7.2 shows that no real entry is ever lost.)
5. *Rebuild.* Call **BuildLevels**(ℓ^*, Arr).

BuildLevels(ℓ^*, Arr)

Input: a sorted, deduplicated array **Arr** of $2q \cdot m_{\ell^*}$ entries.

Algorithm: Cut **Arr** into m_{ℓ^*} consecutive chunks of $2q$ entries each and install them as the nodes of level ℓ^* ; clear all consumed flags of that level. Then for $\ell = \ell^*$ downto 1:

1. For each node i of level ℓ whose first item is real, produce the entry $(\text{item}_1^{(i)}, \text{ptr})$, where ptr points to node i of level ℓ ; this *duplicates* the node's smallest item one level up. Nodes whose first item is padding produce a padding entry.
2. Let these entries, in increasing order of i , be the array of level $\ell - 1$; pad it to $2q \cdot m_{\ell-1}$ entries and cut it into $m_{\ell-1}$ nodes of $2q$ entries. Clear the consumed flags of level $\ell - 1$.

Why the prefix always fits. Level ℓ holds at most $m_\ell \leq q^\ell$ nodes, so level $\ell - 1$ receives at most m_ℓ entries, which occupy at most $\lceil m_\ell / (2q) \rceil \leq m_{\ell-1}$ nodes. Iterating, the number of used nodes shrinks by a factor of $2q$ per level, so level 0 ends up with a single node, as required.

Initialization. The empty data structure is obtained by calling **BuildLevels**($L - 1, \text{Arr}$) on an Arr consisting entirely of padding entries. Every level then holds padding nodes only, the canonical traversal is empty, and (I1)–(I3) hold vacuously.

7.4 Correctness of the Meta-Algorithm

We now show that the meta-algorithm is correct. Concretely, we will show by induction that our data structure represents a key–value store that is consistent with all insertions and deletions. We first define a canonical traversal, which is used to generate the key–value store the data structure represents.

Canonical traversal. For a node X , define $T(X)$ to be the concatenation, over $i = 1, \dots, 2q$, of

- nothing, if item_i is a padding item;
- the single item item_i , if ptr_i is null; and
- item_i followed by $T(Y)$ with its first item removed, if ptr_i points to Y .

We say that the data structure *represents* a key–value store S if $T(\text{root})$ is exactly the sequence of key–value pairs of S , listed in increasing key order. We say that the data structure is *correctly represented* if, after initialization and after every operation, the key–value store it represents consists of exactly those keys that have been inserted and not subsequently deleted, each paired with its most recently inserted value.

Our first lemma is a simple result that establishes all pointers aimed at nodes being rebuilt are discarded.

Lemma 7.1 (Surviving pointers leave the rebuilt region). *Assume (I1)–(I3) hold at the start of an operation, and consider a call to **RebuildPrefix**(ℓ^*, ReIns). Let R be the set of nodes contributing to Arr , namely the unconsumed real nodes of levels $0, \dots, \ell^*$ together with the nodes of **Path**. Then for every key k surviving Step 3, the pointer retained for k is either null or points to a node outside R .*

Proof. Let ptr be the retained pointer, contributed by the copy of k with the largest tag in Arr , say the copy residing in a node at level a . Suppose for contradiction that ptr points to a node $Y \in R$, say at level b . By (I3), $b > a$. By (I2), item_1^Y is another copy of k , and since $Y \in R$ all of Y 's entries are in Arr with tag $b > a$, contradicting maximality. $\square$

We next show that truncation does not impact the correctness of the algorithm. To do this, we define a *truncation-free* modification of the meta-algorithm with the following changes:

- Arrays have unbounded capacity, and the truncate step of **RebuildPrefix** is skipped.
- **BuildLevels** discards padding and packs the real entries into as many consecutive chunks of $2q$ entries as needed, padding only the last chunk. It indexes their minima in the same way at each nonroot level, without imposing the bounds m_ℓ .

- At level 0, the entire remaining entry list is stored in a single root, padded to at least $2q$ entries but allowed to exceed that size. Searches and the canonical traversal process all root entries, and **Relns** retains the entire search path even if its length exceeds $2Lq + 1$. The entry ranges in (I1)–(I2) are extended accordingly for the root.

The next lemma shows that if the truncation-free modification is correct, then the meta-algorithm (with truncation) is correct. We define the canonical traversal and correctness for the truncation-free variant in the same way as for the original variant.

Lemma 7.2 (Truncation preserves correctness). *If the truncation-free meta-algorithm is correctly represented and satisfies (I1)–(I3) after initialization and after every operation, so does the meta-algorithm.*

Proof. Run both variants on the same operation sequence. We show inductively that corresponding nodes have identical real entries and pointers, differing only in padding. This holds at initialization. Suppose it holds before operation t , and write $\ell^* := \ell^*(t)$. By the induction hypothesis, both calls to **Search**(k, μ) follow the same path. The collected real entries, and consequently the survivors after Step 3, agree. It suffices to show that there are at most $2qm_{\ell^*}$ such survivors.

If $m_{\ell^*} = \lceil N/(2q) \rceil$, every surviving key occurs in the canonical traversal after the truncation-free rebuild: **BuildLevels** retains every real input entry and connects every nonempty chunk to the single, possibly oversized root by duplicating chunk minima. By the assumed correctness, these are keys of the store after operation t , which has at most N keys. Since Step 3 deduplicates by key, there are at most $N \leq 2qm_{\ell^*}$ real survivors.

Otherwise $m_{\ell^*} = q^{\ell^*} < \lceil N/(2q) \rceil$, so $L^* \leq \ell^* < L - 1$. Let $t_0 < t$ be the last operation that rebuilt a prefix ending at a level greater than ℓ^* , or let $t_0 = 0$ if there was no such operation. Immediately after t_0 , levels $0, \dots, \ell^*$ contain at most q^{ℓ^*+1} distinct real keys: if $t_0 > 0$, their keys are copies of minima of the at most $m_{\ell^*+1} \leq q^{\ell^*+1}$ nodes at level $\ell^* + 1$; if $t_0 = 0$, they contain no real keys. Until operation t , a rebuild within this prefix only rearranges or duplicates keys already there, apart from the at most $2Lq + 1$ keys supplied by **Relns** on each operation. Moreover, (17) gives $t - t_0 \leq q^{\ell^*+1-L^*}$. Thus the number of distinct real keys collected, and hence of real survivors, is at most

$$q^{\ell^*+1} + (t - t_0)(2Lq + 1) \leq q^{\ell^*+1} \left(1 + \frac{2Lq + 1}{10Lq} \right) < 2q^{\ell^*+1} = 2qm_{\ell^*}.$$

In either case truncation removes only padding. The rebuilt level has at most m_{ℓ^*} nonempty nodes, and each higher index level also fits, since $m_{\ell} \leq qm_{\ell-1} \leq 2qm_{\ell-1}$. Padding therefore does not change the real chunks or their index entries in **BuildLevels**. This completes the induction, so both variants have the same canonical traversal after every operation, and (I1)–(I3) transfer to the meta-algorithm. $\square$

Lemma 7.3 (Invariants and representation). *The meta-algorithm is correctly represented and satisfies (I1)–(I3) after initialization and after every operation.*

Proof. By Lemma 7.2, it suffices to show that the truncation-free meta-algorithm is correctly represented and satisfies (I1)–(I3) after initialization and after every operation. We prove by induction on the number of operations that it represents the correct store and satisfies (I1)–(I3). This holds at initialization, when the store is empty. For the induction step, suppose the structure represents the correct store S and satisfies (I1)–(I3) before the current operation. Let S' be S updated according to that operation.

We first reason about R , as in Lemma 7.1. Define a *fold* of a nonempty child Y into its parent X at position i by replacing $(\text{item}_i, \text{ptr}_i)$ with

$$(\text{item}_i, \text{ptr}_1^Y), \quad (\text{item}_2^Y, \text{ptr}_2^Y), \quad \dots, \quad (\text{item}_r^Y, \text{ptr}_r^Y),$$

where r is the number of real entries of Y . Thus the first entry keeps the parent's item and value, and the child's first pointer. The intermediate list may exceed a node's physical capacity; folding is only a conceptual operation on lists. Since $\text{item}_i.k = \text{item}_1^Y.k$, its traversal is exactly the old item_i followed by $T(Y)$ with its first item removed. A fold therefore preserves both the sorted canonical traversal and every subtree hanging off Y , including the subtree of ptr_1^Y .

The set R is ancestor-closed and contains the root. Indeed, the parent of a node in the rebuilt prefix has a smaller level and is either an unconsumed prefix node or a node of **Path**; the parent of a path node is its predecessor on the path. Repeatedly folding R into the root consequently produces a single sorted list. For each key, folding its chain of copies keeps the item of its shallowest copy in R and the pointer of its deepest copy in R . Ancestor closure ensures that the shallowest copy in R is authoritative. These are precisely the choices made by sorting and deduplicating with the level tags. Every retained non-null pointer leaves R by Lemma 7.1, and each untouched subtree keeps exactly one incoming pointer.

Now consider the request. Any of the three lookup operations leaves this folded list unchanged. Insertions and deletions use **Search**(k, eq). If k is present, this search visits its authoritative copy and follows its entire copy chain. Thus every copy of k belongs to **Relns**, and the entry for k in the folded list has a null pointer. An insertion's tag -1 replaces its item with (k, v) ; a deletion removes it. Neither operation discards another key or an untouched subtree. A deletion of an absent key does nothing. If an inserted key k is absent, its position in the folded list does not split any untouched subtree: otherwise the search would follow the pointer to that subtree, contradicting that the entire search path is in R . The new null-pointer entry therefore inserts exactly (k, v) at its sorted position. Removing copies of k before deduplication gives the same list as this conceptual fold followed by deletion. The resulting list represents S' and contains at most one entry for each key of S' .

BuildLevels cuts the list into consecutive chunks and indexes each nonempty chunk by duplicating its first item. This is the inverse of folding and preserves the canonical traversal. Sorted chunks and their increasing minima give (I1), and the duplicated minima give (I2). Every new index pointer goes from level $\ell - 1$ to level ℓ . Every old pointer that survives leaves R , hence targets an untouched node at a level greater than ℓ^* , while its source is now at level ℓ^* . Each nonempty new node receives one parent pointer, and each untouched subtree retains its unique parent. This proves (I3) and completes the induction. $\square$

Lemma 7.4. *The meta-algorithm correctly realizes $\mathcal{F}_{\text{st}}$.*

Proof. By Lemma 7.3, before each operation the meta-algorithm satisfies (I1)–(I3) and represents the correct store S . The same lemma proves that each operation updates the represented store correctly. It remains to prove that **Search** returns the right answer. By definition of representation, $T(\text{root})$ lists exactly the key–value pairs of S in increasing key order. By (I3), every logical node belongs to the tree rooted at the unique root, and a search terminates because each followed pointer leads to a strictly deeper level. By (I2), removing a child's first item in the canonical traversal only suppresses a duplicate of its parent entry. Thus every real key encountered by the search belongs to S , and every key of S occurs in the tree.

Let k be the queried key. For **eq**, (I1) implies that if k occurs in the current subtree but not in the current node, it must lie in the child selected by the largest key at most k . Starting at the

root and applying this observation at each step, the search encounters k if and only if $k \in S$. By (I1)–(I2), after an exact match any deeper copy of k lies in the child pointed to by that matching entry. Continuing the search therefore consumes the whole copy chain, as required by insertions and deletions.

For **succ**, let item_i be the largest real key at most k in the current node, if one exists. By (I1), subtrees reached through ptr_j with $j < i$ contain only keys below $\text{item}_i.k \leq k$. By (I1)–(I2), each subtree reached through ptr_j with $j > i$ has minimum key $\text{item}_j.k > k$. Step 3 records the smallest real key in the current node greater than k , so those subtrees cannot improve on the recorded candidate. Only the selected subtree can contain a smaller eligible key. If every key in the current node exceeds k , Step 3 already considers every child’s minimum key; hence at termination, by representation, the saved candidate is the strict successor in S , or $\perp$ if none exists.

For **pred**, select instead the item item_i with the largest real key strictly below k . This item is itself a candidate. By (I1), subtrees reached through ptr_j with $j < i$ contain only keys below $\text{item}_i.k$, whereas every real entry with $j > i$ and its subtree have keys at least k . Thus only the selected child can improve the candidate. In particular, when k equals a separator, this rule searches the preceding interval. By (I2), a nonroot node entered through a key strictly below k has that same key as its first item, so only the root can lack a key strictly below k , in which case no predecessor exists. At a null pointer the recorded candidate is, by representation, the largest key in S below k . An empty tree returns $\perp$ in every mode.

Finally, we check values as well as keys. The canonical traversal suppresses only parent–child duplicates, whose keys agree by (I2). Since $T(\text{root})$ contains each key of S exactly once, all copies of a key must therefore form a single chain in the tree given by (I3). If a copy is encountered on the search path, all its shallower copies are on that path too. The tie rule keeps the first such copy, which is also the copy retained by the canonical traversal. Because the structure represents S , its value is the current value in S . This proves the required lookup semantics. $\square$

7.5 Oblivious Implementation of the Meta-Algorithm

Intuition. The oblivious implementation makes three main modifications.

1. *Permuted levels.* Whenever a nonroot level is rebuilt, each of its nodes is assigned a uniformly random destination slot, the pointers stored one level up are written to point at those slots, and the level is then physically shuffled into them.
2. *Filler nodes.* The m_ℓ ordinary nodes of a rebuilt nonroot level are augmented with m_ℓ filler nodes, which the permutation mixes in with the ordinary nodes. A level skipped by the search (or visited after the search has stopped) is accessed nonetheless, at its next unused filler. Fillers are chained: each filler records the destination slot of the next one, and NextFiller_ℓ stores the head of the chain. Thus every operation reads exactly one node slot at each of the L levels.
3. *Data-independent processing.* Within each fetched node, selections such as “the largest key at most k ” scan all $2q$ entries using conditional assignments. During a rebuild, equal-key groups are identified by complete scans of the sorted array, and the surviving entries are compacted using oblivious sorting. All processed arrays have publicly known lengths.

We make additional modifications to each phase to accommodate these changes, but, as we will show, after each operation the two algorithms have identical logical trees, up to permutation of node addresses.

Read phase. $\text{Search}(k, \mu)$ is implemented as follows. Maintain $(\ell_{\text{cur}}, ptr, \text{done})$ in client memory, initialized to $(0, \text{root}, \text{false})$. For $\ell = 0, 1, \dots, L - 1$:

1. Let $\text{active} := (\neg \text{done} \wedge \ell = \ell_{\text{cur}})$. For $\ell = 0$, read the fixed root slot; here active is always true. For $\ell > 0$, conditionally select ptr or NextFiller_ℓ according to active , and read that slot. Always read and write the level's filler-head record, advancing it to the fetched filler's successor only when active is false. In either case, write the fetched slot back with its consumed flag set, leaving its entries and other metadata unchanged.
2. Scan all $2q$ entries of the fetched node. Using conditional assignments, maintain the entry (item_i, ptr_i) that determines the next node to visit and the best answer candidate item' found in the current node. As in the **Search** procedure of the meta-algorithm in Section 7.3, the query mode μ determines the comparison predicates, but every mode uses the same full scan and memory-access pattern. If active is true and a candidate exists, conditionally update item^{ans} .
3. Append the $2q$ entries to Relns with $\text{tag} := \ell$, converting them to padding entries when active is false.
4. Conditionally on active , set done if the node has no key at most k (strictly below k if $\mu = \text{pred}$) or the pointer paired with the largest such key is null; otherwise set $(\ell_{\text{cur}}, ptr) \leftarrow ptr_i$. A filler access preserves this search state, so a pointer that skips levels remains pending.

After the loop, perform the following:

- Extend Relns by one entry: append $((k, v), \text{null})$ with $\text{tag} := -1$ for **Insert** (k, v) , and append a padding entry for every other operation.
- Scan all of Relns , replacing an entry by padding exactly when the operation is **Delete** (k) and the entry has key k .
- Let $\ell^* := \ell^*(t)$ according to (17), where t is the index of the current operation, and call **RebuildPrefix** (ℓ^*, Relns) .

In the above read-phase algorithm, Step 1 accesses one slot at every level that has not been visited since the level was initialized or most recently rebuilt. At the root, this is the unique fixed slot. At every nonroot level, the accessed location is uniformly distributed among the remaining unvisited locations (before its first rebuild, the visits simply follow a public initial schedule); see also the initialization and rebuild algorithms, and the proof of Lemma 7.7.

Rebuild phase. **RebuildPrefix** (ℓ^*, Relns) is implemented as follows.

1. *Collect.* Scan every physical node slot in levels $0, 1, \dots, \ell^*$, using the fixed root slot at level 0. Tag each entry from level ℓ with $\text{tag} := \ell$ and attach a live bit that is true exactly when the node is ordinary and unconsumed and the entry is real. Give real entries of Relns a true live bit and padding entries a false one. Retain the original keys of non-live ordinary entries through the next step. Entries from filler nodes have $\text{live} := \text{false}$; their entry slots already contain padding entries, and their filler bits are not modified. The combined public

length is

$$n := 2Lq + 1 + 2q \left(1 + 2 \sum_{\ell=1}^{\ell^*} m_\ell \right) = O(q^{\ell^*+1} + Lq).$$

2. *Sort.* Concatenate the collected entries with **Relns** and obviously sort the resulting fixed-length array by **(item.k, tag)**. Call the sorted result **Arr**; it includes both live and non-live entries.
3. *Deduplicate.* With one forward and one backward scan of **Arr**, accomplish the following: for each group of consecutive entries with the same key, use its first live entry as the *representative*: retain that entry's item, replace its pointer by the pointer of the group's last live entry. A group with no live entry has no representative. Obviously and stably compact the representatives to the front of **Arr** using the sorting primitive of Section 3.3, and replace all other entries by padding.
4. *Truncate* to $2q \cdot m_{\ell^*}$ entries, a public length.
5. *Rebuild* the levels 0 to ℓ^* by calling the **BuildLevels**(ℓ^* , **Arr**) algorithm below where **Arr** is the outcome of the above procedure.

Oblivious **BuildLevels**(ℓ^* , **Arr**)

Cut **Arr** into m_{ℓ^*} nodes and place them in level ℓ^* , in sorted order. For $\ell = \ell^*$ downto 1:

- 5a. *Reinitialize.* Append m_ℓ filler nodes to the m_ℓ ordinary nodes of level ℓ , obtaining an array A_ℓ of $2m_\ell$ nodes still in sorted order, where fillers appear last. A filler node has the same size as an ordinary node and consists of padding entries plus a field for the location of the next filler. Mark the first m_ℓ nodes as ordinary and the last m_ℓ as fillers. Clear the consumed flags of all $2m_\ell$ nodes.
- 5b. *Draw the destinations.* Sample a uniformly random permutation π_ℓ of $[2m_\ell]$ and annotate each node i of A_ℓ with its destination slot $\pi_\ell(i)$, without moving any node. Concretely, apply an oblivious random permutation to the array of *indices* $(1, 2, \dots, 2m_\ell)$, and let $\pi_\ell(i)$ be the position of index i in the resulting array. Obviously route each $\pi_\ell(i)$ to $A_\ell[i]$.
- 5c. *Chain the fillers.* In one scan of A_ℓ , the $(j+1)$ -st filler hands its destination $\pi_\ell(m_\ell + j + 1)$ to the j -th filler, which records it; the last filler records a null successor. Set $\text{NextFiller}_\ell := \pi_\ell(m_\ell + 1)$.
- 5d. *Build the level above.* Form the array of level $\ell - 1$: scanning A_ℓ in order, each ordinary node $i \leq m_\ell$ emits the entry consisting of that node's first item paired with a pointer to slot $\pi_\ell(i)$ of level ℓ (null if the item is padding). Pad to $2q \cdot m_{\ell-1}$ entries and cut into $m_{\ell-1}$ nodes. This array is sorted, since A_ℓ is.
- 5e. *Shuffle.* Obviously sort A_ℓ by the destination annotation, which moves each node i to slot $\pi_\ell(i)$; discard the destination annotations to obtain the new level ℓ .

Finally, install the resulting singleton level-0 node at the fixed root address and clear its consumed flag.

Initialization. Initialize the root and all nonroot node slots to padding in one sequential pass, with null entry pointers and cleared consumed flags. At each nonroot level mark the first m_ℓ slots as ordinary and the last m_ℓ slots as fillers, and chain the fillers in that public order. Until a level's first scheduled rebuild, no logical pointer can reach it: it starts empty, and rebuilding a shallower prefix creates no pointer to an empty deeper level. Its reads therefore follow the public filler chain. Its first rebuild then installs the usual randomly permuted layout. Initialization writes $O(N)$ words sequentially and costs $O(N/B)$ transfers in the nontrivial regime $N > B$.

Fact 7.5. *The algorithm never runs out of filler nodes.*

Proof. The root needs no filler. A nonroot level $\ell \leq s$ is rebuilt on every operation and has $m_\ell \geq 1$ fillers. For $\ell > s$, its rebuild period is $p_\ell = q^{\ell-L^*}$. Additionally, m_ℓ either equals q^ℓ or $\lceil N/(2q) \rceil \geq q^{\ell-2}$ by construction. Since $L^* \geq 2$, it follows that ℓ does not run out of filler nodes. $\square$

Lemma 7.6. *Conditioned on every oblivious random permutation and every oblivious sort producing correct output, the oblivious implementation produces the same outputs as the meta-algorithm on every request sequence.*

Proof. The differences from the meta-algorithm are the following, and none of them changes the semantics. First, ordinary nodes are permuted and interleaved with fillers. Step 5d of **BuildLevels** writes the destination $\pi_\ell(i)$ into the parent pointer and Step 5e moves node i to exactly that slot, so every pointer still resolves to the correct node, and the search follows the same logical path. Second, a nonroot level that the meta-algorithm skips is here accessed at NextFiller_ℓ . Both initialization and Step 5c give each filler its successor, and by Fact 7.5 a fresh filler always exists, so this access always lands on a filler. A filler node contains only padding entries, so appending its entries to Relns contributes no live entry. These padding entries are dropped by the deduplication step; hence their presence in Arr does not change the outcome of any rebuild. Third, all selections are computed by conditional assignment over complete scans, which computes the same functions as the early-exit descriptions in Section 7.3. Fourth, a logical node's consumed flag is set exactly when the meta-algorithm removes it from the tree. The global sort carries each entry's live bit with it, and deduplication ignores precisely the entries excluded by the meta-algorithm's collection step. Non-live entries can interleave with live entries of the same key but affect neither the selected item nor the selected pointer. Stable compaction therefore produces the same sorted array for **BuildLevels**. Finally, the initial public layout represents the empty tree; all nonroot accesses before a level's first rebuild go to its filler chain, so the initialization shortcut has the same logical semantics. $\square$

Lemma 7.7. *The implementation satisfies statistical obliviousness.*

Proof. Consider the hybrid experiment in which every oblivious random permutation is replaced by an ideal functionality that applies a uniformly random permutation to the array of nodes, every oblivious sort is replaced by an ideal functionality that returns the sorted array, and in both cases the adversary observes only the simulated access patterns of these building blocks. By Section 3.3 it suffices to show that the hybrid experiment is correct with probability 1 — which is Lemma 7.6 — and that its access pattern is simulatable.

The rebuild phase is triggered at operation t with the publicly computable parameter $\ell^*(t)$, and consists of scans, oblivious sorts, and random permutations of arrays of publicly known lengths. Its access pattern is therefore simulatable given the ideal building blocks. For the read phase, every operation accesses exactly one node slot at each of the L levels, followed by a data-independent sequence of operations on Relns and on the filler-head records. The root access has a fixed address. Before a nonroot level's first rebuild, its accesses follow its public initial filler chain; whether that

rebuild has occurred depends only on the operation count. These accesses are therefore directly simulatable.

Fix a nonroot level ℓ after its first rebuild and consider the interval between two consecutive rebuilds of it. At the start of the interval the $2m_\ell$ slots are placed by a uniformly random permutation that is independent of everything the adversary has seen, and their real/filler status is independent of their position. Within the interval, a real node is accessed at most once: its parent is on the same fetched path, and deduplication retains no pointer into the consumed region (Lemma 7.1). Its old contents are excluded from later rebuilds by the consumed flag, so no stale incoming pointer is reintroduced. A filler is likewise accessed at most once, since `NextFiller $_\ell$` walks the filler chain, and Fact 7.5 guarantees the chain does not run out. The choice of a logical node or of the next filler does not depend on the unused physical positions. Conditional on the preceding transcript, their assignments remain a uniform bijection to the unvisited slots. The simulator can therefore sample the next slot uniformly without knowing the request. The permutations of different levels are independent, so the same argument applies successively to every level and every adaptively chosen request. $\square$

7.6 Efficiency Analysis

Cost of the read phase. A node holds $2q$ entries and hence occupies $O(1)$ blocks. The read phase reads and writes back exactly one node per level, so it costs

$$O(L) = O\left(\frac{1}{\epsilon} \log_B N\right)$$

block transfers, plus $O(Lq/B) = O(L)$ transfers for assembling `Relns`.

Cost of the rebuild phase. We amortize directly over operations, each of which reinserts the same public number $2Lq + 1$ of entries. A rebuild whose maximum level is $k \geq s$ happens at most once every q^{k-s} operations.

Rebuilding levels 0 through k processes $O(q^{k+1})$ entries. Collecting and globally sorting the entries, compacting the survivors, and installing the new permutations use oblivious sorts, random permutations, and linear scans on arrays of total length $O(q^{k+1} + Lq) = O(q^{k+1})$. The arrays used to shuffle the individual levels grow geometrically, apart from at most two capped levels, so their costs sum to the cost at the largest size plus $O(L)$ for whole-block rounding. Here the reinsertion term is absorbed because $Lq \leq Lq^3 = O(LB/\log^2 N) = O(B)$ for sufficiently large N . By Section 3.3, the total is

$$O\left(L + \frac{q^{k+1}}{B} \log N\right)$$

block transfers, where the additive $O(L)$ accounts for rounding each level smaller than one block up to a whole block transfer. Consequently the amortized rebuild cost per operation is at most

$$\begin{aligned} \sum_{k=s}^{L-1} \frac{1}{q^{k-s}} \cdot O\left(L + \frac{q^{k+1}}{B} \log N\right) &= O\left(L + \frac{Lq^{s+1}}{B} \log N\right) \\ &= O\left(L + \frac{L^2 q^3}{B} \log N\right) \\ &= O\left(\frac{1}{\epsilon} \log_B N + \frac{1}{\epsilon^2} \cdot \frac{(\log_B N)^2 \log N}{B^{2/(2+\epsilon)}}\right), \end{aligned}$$

where the second line uses $q^{s+1} \leq q^{L^*+1} \leq 10Lq^3$ (including the rounding in $L^* = \lceil \log_q(10Lq) \rceil$), and the third uses $L = O(\frac{1}{\epsilon} \log_B N)$ together with (16).

Finally, when $B \geq \log^{2+\epsilon} N$ we have $B^{2/(2+\epsilon)} \geq \log^2 N \geq \log N \cdot \log_B N$, so the second term is $O(\frac{1}{\epsilon^2} \log_B N)$ as well. The amortized cost per operation is therefore $O(\frac{1}{\epsilon^2} \log_B N) = O(\log_B N)$ for any fixed ϵ .

Space. Level ℓ occupies $O(q \cdot m_\ell)$ words. Summing the geometric series and using the cap $m_\ell \leq \lceil N/(2q) \rceil$ at the deepest levels, the total is $O(N)$ words, i.e. $O(N/B)$ blocks. Filler markers and successor fields add only $O(1)$ words per node. Rebuild workspace is $O(N)$ words on the server. The client stores $O(L)$ filler-head records, $O(Lq)$ reinsertion entries, and the buffers for sorting and routing, for a total of $O(B)$ words. Accesses to these arrays also follow the fixed scans or networks described above.

7.7 Putting It Together

Theorem 7.8 (Optimal statistically secure oblivious ST). *Let $\epsilon \in (0, 1)$ be an arbitrarily small constant and suppose $B \geq \log^{2+\epsilon} N$ and $M \geq B$. The construction of this section obviously realizes $\mathcal{F}_{\text{st}}$ with statistical security, consumes $O(N/B)$ blocks of server space, and, for $M < N$, incurs*

$$O\left(\frac{1}{\epsilon^2} \left(1 + \log_B \frac{N}{M}\right)\right)$$

amortized block transfers per operation. If the logical search tree fits in client memory, it can instead be stored and accessed there by fixed-length scans.

Proof. Correctness of the meta-algorithm is Lemma 7.4; the oblivious implementation inherits it by Lemma 7.6, conditioned on the oblivious sorts and permutations succeeding, which happens with all but negligible probability by Section 3.3. Obliviousness is Lemma 7.7, and the space bound follows from Section 7.6. That section also proves an amortized transfer bound of $O(\epsilon^{-2} \log_B N)$ when $M = \Theta(B)$. The caching analysis in Section C.2 improves this to $O(\epsilon^{-2}(1 + \log_B(N/M)))$ for general M , while preserving word-level obliviousness. $\square$

Acknowledgements

The human authors are responsible for the main contribution of the paper, including devising the schemes and the proof ideas. We interacted with ChatGPT 5.2 and Gemini 3 to gain some insights on how to prove some probabilistic bounds with limited independence in the ORAM part. We then wrote an initial draft of the paper ourselves, using ChatGPT 5.2 and Gemini 3 only for polishing and for doing local calculations. Later versions of AI including ChatGPT 5.6, ChatGPT 6, Claude Opus 5, and Claude Opus 5.5 audited our draft, and rewrote some parts of the scheme description and proofs with human hand-holding, fixing some minor gaps and improving clarity in the process.

This work is in part supported by an ONR grant, a Packard Fellowship, a JP Morgan research award, and an Amazon research award. Andrew Krapivin was supported by the Jeanne B. and Richard F. Berdik ARCS Pittsburgh Endowed Scholar Award. We gratefully acknowledge Michael Bender for helpful discussions on external-memory algorithms.

References

- [ACMR95] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. In *Proceedings of the Twenty-Seventh Annual ACM*

- Symposium on Theory of Computing*, STOC '95, page 238–247, New York, NY, USA, 1995. Association for Computing Machinery.
- [ACN⁺20] Gilad Asharov, T-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *Symposium on Simplicity in Algorithms (SOSA)*, 2020.
 - [AEKL25] Gilad Asharov, Eliran Eiluz, Ilan Komargodski, and Wei-Kai Lin. Megablocks: Breaking the logarithmic i/o-overhead barrier for oblivious RAM. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, pages 4692–4706. ACM, 2025.
 - [AHR24] Ananya Appan, David Heath, and Ling Ren. Oblivious single access machines - a new model for oblivious computation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS '24*, page 3080–3094, New York, NY, USA, 2024. Association for Computing Machinery.
 - [Ajt10] Miklós Ajtai. Oblivious rams without cryptographic assumptions. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*, STOC '10, pages 181–190, New York, NY, USA, 2010. ACM.
 - [AKL⁺20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMA: Optimal Oblivious RAM. In *Advances in Cryptology - EUROCRYPT 2020*, 2020.
 - [AKM23] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. Futorama: A concretely efficient hierarchical oblivious ram. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23*, page 3313–3327, New York, NY, USA, 2023. Association for Computing Machinery.
 - [AV88] Alok Aggarwal and S. Vitter, Jeffrey. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, September 1988.
 - [BFCF⁺07] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM.
 - [BM70] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery.
 - [BR94] M. Bellare and J. Rompel. Randomness-efficient oblivious sampling. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, page 276–287, USA, 1994. IEEE Computer Society.
 - [BS80] Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.
 - [BSA13] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIA CCS*, 2013.

- [CFM⁺98] Richard Cole, Alan Frieze, Bruce M. Maggs, Michael Mitzenmacher, Andréa W. Richa, Ramesh Sitaraman, and Eli Upfal. On balls and bins with deletions. In *Randomization and Approximation Techniques in Computer Science*, volume 1518 of *Lecture Notes in Computer Science*, pages 145–158. Springer, 1998.
- [CGLS17] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *ASIACRYPT*, pages 660–690, 2017.
- [Che19] Xue Chen. Derandomized balanced allocation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '19*, page 2513–2526, USA, 2019. Society for Industrial and Applied Mathematics.
- [Con] Graeme Connell. Technology deep dive: Building a faster oram layer for enclaves. <https://signal.org/blog/building-faster-oram/>.
- [CS17] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure ORAMs and OPRAMs. In *Theory of Cryptography Conference, (TCC)*, 2017.
- [CW77] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, STOC '77*, page 106–112, New York, NY, USA, 1977. Association for Computing Machinery.
- [Dem02] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, University of Aarhus, Denmark, June 27–July 1 2002.
- [DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.
- [DMWS12] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *International Symposium on Computer Architecture (ISCA)*, 2012.
- [EGT10] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*, 2010.
- [FNR⁺15] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth oblivious RAM. Cryptology ePrint Archive, Paper 2015/1065, 2015.
- [GKK⁺12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 513–524. ACM, 2012.
- [GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.
- [GMOT11] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *CCSW*, 2011.

- [GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.
- [GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.
- [Gol87] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.
- [Goo14] Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(N \log N)$ time. In *STOC*, 2014.
- [GOT12] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.
- [GWT⁺25] Tianyao Gu, Yilei Wang, Afonso Tinoco, Bingnan Chen, Ke Yi, and Elaine Shi. Flexway o-sort: Enclave-friendly and optimal oblivious sorting. In *Usenix Security*, 2025.
- [HS24] Nicholas Harvey and Arvin Sahami. Explicit orthogonal arrays and universal hashing with arbitrary parameters, 2024. arXiv:2405.08787.
- [JLS21] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. Optimal oblivious priority queues. In *Proceedings of the Thirty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’21, page 2366–2383, USA, 2021. Society for Industrial and Applied Mathematics.
- [KL21] Ilan Komargodski and Wei-Kai Lin. A logarithmic lower bound for oblivious ram (for all parameters). In *CRYPTO*, page 579–609, Berlin, Heidelberg, 2021. Springer-Verlag.
- [KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.
- [KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *Asiacrypt*, 2014.
- [LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *CRYPTO*, 2018.
- [LSX19] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the $n \log n$ barrier for oblivious sorting? In *SODA*, 2019.
- [LWN⁺15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *S & P*, 2015.
- [met] Meta’s oram implementation. <https://github.com/facebook/oram>.
- [Mit01] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.
- [MPC⁺18] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy*, 2018.

- [MS26] Arya Maheshwari and Elaine Shi. Oblivious priority queue and single-source shortest path in the external memory setting. In *Advances in Cryptology – CRYPTO 2026: 46th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17–20, 2026, Proceedings, Part VIII*, page 329–360, Berlin, Heidelberg, 2026. Springer-Verlag.
- [NWI⁺15] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP ’15, page 377–394, USA, 2015. IEEE Computer Society.
- [PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious ram with logarithmic overhead. In *FOCS*, 2018.
- [pri] Private communication with Rolfe Schmidt.
- [RFK⁺15] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security*, 2015.
- [RS21] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *SPAA*, 2021.
- [RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, New York, NY, USA, 2009. ACM.
- [SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.
- [SS13] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *S & P*, 2013.
- [SSS95] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discret. Math.*, 8(2):223–250, May 1995.
- [SSS12] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.
- [SvDS⁺13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.
- [SZE⁺16] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *S & P*, 2016.
- [TGS23] Afonso Tinoco, Sixiang Gao, and Elaine Shi. Enigmap: External-memory oblivious map for secure enclaves. In *Usenix Security*, 2023.
- [vEBKZ76] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10(1):99–127, 1976.

- [Vöc03] Berthold Vöcking. How asymmetry helps load balancing. *Journal of the ACM*, 50(4):568–589, 2003.
- [WCS15] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*, 2015.
- [WNL⁺14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *CCS*, 2014.
- [ZE13] Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *S & P*, 2013.
- [ZZD⁺25] Leqian Zheng, Zheng Zhang, Wentao Dong, Yao Zhang, Ye Wu, and Cong Wang. H2o2ram: a high-performance hierarchical doubly oblivious ram. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

A Achieving Linear Space

Our earlier ORAM construction uses $O_\epsilon(Nq^4 + LB)$ words of space for N logical addresses. Here O_ϵ hides an $O(1/\epsilon)$ factor from the $O(1/\epsilon)$ replicas of the semi-functional ORAM. In this section, we describe a modification that preserves the query cost while reducing the space to $O_\epsilon(N)$. We first consider $B \leq M < N$; if the logical memory fits in client memory, we store it there directly. As in Theorem 6.5, assume $0 < \epsilon < 1/48$ and $B \geq (\log N)^{1+6\epsilon}$.

A.1 Blueprint

Our blueprint is the following:

1. Treat every group of $g = q^4$ words with consecutive addresses as a *super-word*, padding the final group if necessary. There are $n = \lceil N/g \rceil$ super-words.

$$g = q^4, \quad n = \lceil N/g \rceil.$$

2. Use the construction of Sections 4 and 5, with its size parameter N replaced by n , as a *pointer ORAM*. For each super-word, it stores a constant-size position identifier into an externally indexed bucket ORAM. Its space is $O_\epsilon(nq^4 + LB) = O_\epsilon(N)$. Since $q = B^\epsilon$ and $B < N$, we have $N^{1-4\epsilon} \leq n \leq N$; thus negligible failure in n remains negligible in N , and an execution of polynomial length in N also has polynomial length in n .
3. Store the super-words in the externally indexed bucket ORAM described below, following the Bucket ORAM blueprint [FNR⁺15]. Given a super-word's position identifier, this structure reads or writes the super-word, additionally choosing and returning a fresh position identifier. The caller saves this new identifier in the pointer ORAM. The data structure uses $O(N)$ space and $O(\epsilon^{-1}(1 + \log_B N))$ amortized block probes per super-word access.

The pointer ORAM chooses its number of levels L to be the smallest integer such that $q^{L-1} \geq n$. Since q^{L-1} can be nearly qn , the geometric space bound $O(q^{L+2})$ for each replica becomes $O(nq^4)$. Grouping q^4 words per super-word therefore gives $O(N)$ space per replica and $O(N/\epsilon)$ space overall.

A.2 Externally Indexed Bucket ORAM for Super-Words

We adapt the basic Bucket ORAM construction of Fletcher et al. [FNR⁺15] to store super-words, using our pointer ORAM as its position map. We recall the construction and give the capacity and obliviousness arguments for this variant. Section A.3 describes its block layout and analyzes its external-memory cost.

We use a binary tree of buckets without a recursive position map; the pointer ORAM supplies the position identifiers. Each bucket holds super-logarithmically many super-words, but occupies substantially less than one block.

Data structure. Set

$$\tau = \lceil (\log N)^{1+\epsilon} \rceil.$$

Here τ measures the bucket capacity in super-words and also sets the number of requests between flushes of the root. Let H be the smallest nonnegative integer with $2^H \geq n/\tau$. Use a complete binary tree with 2^H leaves, with levels numbered $0, \dots, H$. We use h for a level index and H for the height of this externally indexed bucket ORAM, to distinguish them from the pointer ORAM's level index ℓ and number of levels L . Every bucket has $C_P \tau$ slots. A slot contains a super-word, its address, and its assigned leaf identifier, or a dummy record. Thus a bucket occupies

$$z = \Theta(\tau g) \quad \text{words,}$$

including metadata, for a sufficiently large absolute constant C_P .

Path invariant. Every super-word has an independently sampled uniform leaf identifier and exactly one live copy, stored somewhere on the path to that leaf. Initially, place the super-words in their assigned leaf buckets using oblivious bin placement, and initialize the pointer ORAM with their identifiers. The initialization accesses must not reveal these identifiers. All buckets are padded to their fixed capacities.

Handling requests. For a read or write of a logical word, perform the following:

1. *Read and remove.* Obtain the containing super-word's leaf identifier from the pointer ORAM. Scan the indicated path and remove the requested super-word. Scan all words of the super-word to select or update the requested offset by conditional moves; the scan does not reveal the offset. Reads and writes use the same access pattern.
2. *Assign a new path.* Sample a fresh independent uniform leaf identifier and insert the super-word into the root bucket. Save the new identifier in the pointer ORAM. Each logical request therefore uses a constant number of pointer-ORAM operations on a fixed schedule.
3. *Periodic flush.* Every τ requests, at time τk , let j be the largest integer such that 2^j divides k . For $h = 0, \dots, \min\{j, H - 1\}$ in increasing order, flush every bucket at level h into its two children and empty the parent. Route each record according to its assigned leaf. Thus level $h < H$ is emptied once every $\tau 2^h$ requests. If $H = 0$, omit flushes.

A flush processes one parent and its two children at a time. Read and write all their padded slots, using fixed scans and conditional moves in working memory to route the records. The server accesses depend only on the public schedule, and no whole-level oblivious sort is needed.

Bucket capacity and obliviousness. Consider an execution with unbounded buckets and a private position map, using the pointer ORAM’s simulator for its observable accesses. At operation boundaries, the current leaf identifiers are independent uniform values conditional on the preceding public view. Indeed, even revealing the accessed super-word’s old leaf leaves all other identifiers independent and uniform; replacing that identifier by a fresh hidden value restores the invariant. Adaptive requests depend on the preceding view, and hence do not reveal the fresh identifier. Flushes have a public access pattern and reveal no further information. The same argument simulates every accessed path by a fresh uniform leaf.

At a nonleaf level h , every record present before its next emptying comes from one of the at most $\tau 2^h$ insertions since its previous emptying. This includes records arriving during the flush of lower levels. The independently sampled insertion labels hit any fixed bucket with probability 2^{-h} , so its load is dominated by a sum of independent indicators with mean at most τ . Repeated requests can only remove older records from this bound. At a leaf, its load is at most the number of live super-words assigned there; the invariant above gives independent uniform labels and mean $n/2^H \leq \tau$. If $H = 0$, the root holds all $n \leq \tau$ super-words directly.

For a sufficiently large constant C_P , the Chernoff bound gives

$$\Pr[\text{a fixed bucket needs more than } C_P \tau \text{ slots}] \leq \exp(-\Omega(\tau)) = \text{negl}(N).$$

This also bounds the load when a flush merges a parent into its children; for nonleaf buckets, use the same bound based on insertions since the last emptying; for leaf buckets, use the bound on the number of super-words currently assigned to that leaf. Union bound over all buckets and intermediate states in a polynomial execution. Except with negligible probability, all buckets fit and the finite construction agrees with the simulated execution. Together with the pointer ORAM’s guarantee, this proves correctness and statistical obliviousness for adaptive requests.

A.3 Block Layout for the Externally Indexed Bucket ORAM

Block layout and read cost. The bucket size satisfies

$$\frac{z}{B} = O\left(\frac{(\log N)^{1+\epsilon}}{B^{1-4\epsilon}}\right) = O(B^{-\eta}), \quad \eta = \frac{\epsilon - 24\epsilon^2}{1 + 6\epsilon} = \Theta(\epsilon) > 0,$$

where the last assertion uses $\epsilon < 1/48$. For sufficiently large N , let $\beta = \lfloor \log_2(B/z+1) \rfloor = \Omega(\epsilon \log B)$. A binary subtree with β levels occupies at most $(2^\beta - 1)z \leq B$ words. Divide the tree into bands of β levels and store each subtree within a band contiguously, keeping each bucket contiguous as well. Concatenate the subtrees without padding each one to a block boundary; each intersects at most two blocks. A path crosses at most one subtree per band, so reading and writing it costs

$$O(1 + H/\beta) = O(\epsilon^{-1}(1 + \log_B N))$$

block probes. This also covers removing the old copy and inserting the new copy at the root. The additive 1 handles a tree with only one band or a single bucket. Bounded-size instances can be handled by a direct padded scan, with constants depending on ϵ .

Flush cost. Each parent and child bucket occupies $z < B$ contiguous words and thus intersects at most two blocks. Flushing level h therefore costs $O(2^h)$ block probes, whether or not the level is contiguous. Since the flush occurs every $\tau 2^h$ requests, the amortized cost is

$$O\left(\sum_{h=0}^{H-1} \frac{2^h}{\tau 2^h}\right) = O(H/\tau) = O(1).$$

The saving comes from buffering $\Theta(\tau)$ super-words per bucket, not from dividing the number of buckets by the block size. A constant number of working blocks suffices to process a parent and its children; constant working-memory factors can be accommodated by block rescaling.

Space and total cost. The tree has $O(2^H)$ buckets, so its space, including final block rounding, is

$$O(2^H \tau g + B) = O(n g + \tau g + B) = O(N).$$

Here $n g = O(N)$ and $\tau g = o(B) \leq N$. The pointer ORAM uses $O_\epsilon(N)$ words as shown above. A logical request makes a constant number of pointer-ORAM accesses and one super-word access, giving $O(\epsilon^{-2}(1 + \log_B N))$ amortized block probes after initialization. Section C applies caching to the pointer ORAM and the bucket tree to obtain $O(\epsilon^{-2}(1 + \log_B(N/M)))$ for $B \leq M < N$.

B Extension to Unordered Dictionary

Our statistically secure ORAM can easily be extended to an oblivious unordered dictionary preserving the same costs. An unordered dictionary supports the following operations:

- **Get(key)**: returns the value associated with **key**, and returns $\perp$ if not found;
- **Set(key, val)**: sets the value associated with **key** to the specified **val**, and inserts the element if **key** does not exist in the data structure;
- **Delete(key)**: removes the element with **key** from the data structure if one exists.

B.1 Construction

Data structure. Recall that $q = B^\epsilon$. Assume at most N keys are live at any time. We use a 2-tier hashing scheme, in part inspired by Chan et al. [CGLS17]. Let

$$k = \lceil (\log N)^{1+3\epsilon} \rceil.$$

Sample $\tilde{H}_0(\cdot)$, $\tilde{H}_1(\cdot)$, and $\tilde{H}_2(\cdot)$ independently from a k -wise independent hash family, and independently of all underlying ORAM randomness. The first hash is used for balls-and-bins hashing in the first tier, and the last two are used for choice-2 hashing in the second tier. They act on the $O(1)$ -word encoding of a key. A standard degree- $(k-1)$ polynomial construction over an $O(1)$ -word finite field describes each hash in $O(k)$ words. Thus the three descriptions occupy $O(k) = O(B)$ words for $B \geq \log^{1+6\epsilon} N$.

Let $c = q^4$, and let m be the smallest power of two at least $64N/c$. Each hash table has $m = \Theta(N/q^4)$ bins of capacity c . For constant-size key-value records, each bin occupies $O(q^4)$ words, including metadata. Allocate each bin a fixed constant number of whole q^4 -word super-words in the space-optimized ORAM of Section A, padding unused words. Distinct bins, including bins in different tiers, have disjoint super-word storage. The total allocation is $O(m q^4) = O(N)$ words. Accessing a bin uses a fixed padded number of super-word accesses.

Set(key, val). To insert an element with **key**, perform the following:

1. Compute $h_i = \tilde{H}_i(\text{key})$ for $i = 0, 1, 2$.
2. Query the ORAM of Section A for the super-words containing the h_0 -th bin of the first-tier table and the h_1 -th and h_2 -th bins of the second-tier table.

3. If **key** is found in any of these bins, then overwrite its value with **val**. Else, insert the element (**key**, **val**) into the matched bin of the first-tier. If the corresponding bin is full, then insert (**key**, **val**) into one of the chosen second-tier bins with lighter load, breaking ties by favoring the bin to the left [Mit01]. All if branches follow the conventions described in Remark 2, where both branches are padded with fake accesses to have the same word-level access patterns.
4. Finally, write the super-words of all three selected bins back to the underlying ORAM.

In the above Steps 3 and 4, if $h_1 = h_2$, apply the update only once and make both buffered copies contain the same updated contents, using a fixed access pattern. Perform the reads and writes through the underlying ORAM for both bins even when they coincide, writing the same final contents back both times.

Get(key). Compute the three hash locations and read the same three bins as in **Set(key, val)**. Then write all three bins back without changing their contents, whether or not **key** is found. Return the value associated with **key** or return $\perp$ if **key** is not found.

Delete(key). Call **Set(key, $\perp$)**, where $\perp$ is a reserved deletion symbol, henceforth called a *tombstone*.³ Tombstone entries count toward bin loads for the two-choice hashing. A **Get** returns $\perp$ for such an entry, and a subsequent **Set** on the same key replaces the tombstone with the new value.

Periodic rebuild. Every N dictionary operations, we rebuild on a fixed public schedule. Read every logical bin through the underlying ORAM and collect all its entries, including dummy entries. Perform oblivious deduplication and oblivious sorting to preserve the freshest value of each key and remove dummy entries and entries marked with $\perp$. The oblivious deduplication and oblivious sorting cost $O(\frac{N}{B} \log N)$ block transfers, which is bounded by $O(N)$ when $B \geq \log^{1+6\epsilon} N$. Pad the remaining entries with dummies to a list of length exactly N .

Initialize empty tables with fresh dictionary hashes and a new underlying ORAM. Reinsert all N entries, by calling **Set** where dummy entries adhere to the same access patterns except that it writes back a dummy. Since rebuilding occurs every N operations, performing $O(N)$ insertions during each rebuild does not increase the asymptotic amortized cost per operation.

B.2 Proofs

We will need the following lemma that analyzes two-choice hashing.

Lemma B.1 (Bounded-independence two-choice allocation [Vöc03, Che19]). *There is an absolute constant C with the following property. Let S be any fixed sequence of $z \leq m/2$ distinct keys, and let H_1, H_2 be independently sampled k -wise independent hashes from S to $[m]$. Allocate the keys sequentially to the less loaded of their two bins, breaking ties to the left. For every $2 \leq s \leq \log \log m$ satisfying*

$$k \geq C(\log m)2^{Cs},$$

the maximum bin load is at most

$$\log_2 \log m + C(s + z/m)$$

except with probability at most m^{-s} .

³Tombstones can instead be replaced by dummy entries, freeing their slots. Since in our ORAM construction, the adversary is unaware of the hash functions when choosing the request sequence, we can adapt the analysis using ideas from Cole et al. [CFM⁺98] to handle deletions in the two-tier hashing.

Proof sketch. This is Lemma 3.6 of the full version of Chen [Che19] (arXiv:1702.03375), due to Vöcking [Vöc03], with $d = 2$ choices and $c = s$; Remark 3.7 there observes that the argument needs only the independence of the hash values within one witness. Specifically, if a bin reaches the stated load, the backward witness-tree construction gives either a pruned witness tree of height $\log_2 \log m + \Theta(s + z/m)$ or a witness with sufficiently many collisions. Each such witness involves at most $O((\log m)2^{O(s)})$ distinct hash evaluations. Hence the assumed k -wise independence makes all hash values appearing in each witness fully independent. The standard enumeration of pruned trees and collision patterns then gives total probability at most m^{-s} . Lemma 3.6 is stated for constant c , but its proof applies verbatim for $2 \leq c \leq \log \log m$: constancy is used only to bound the number of witness configurations, $(2^{l+1})^{O(c)}$ with $l = \log_2 \log m + O(c)$, by m . $\square$

First analyze a hybrid with unbounded second-tier capacity and an ideal private implementation of the underlying ORAM. The access schedule is padded, so its simulated trace is independent of the dictionary hashes. Fix the simulated trace and the adversary's coins. During a period with fresh dictionary hashes, at most N retained keys are reinserted and at most N further operations occur. Let A be the resulting ordered list of distinct inserted keys, in order of first insertion. Thus $v = |A| \leq 2N$, and A is independent of the fresh first-tier hash. Updates and tombstone deletions do not free slots before the next rebuild. Consequently, the number Z of distinct keys sent to the second tier is exactly the number of keys of A with at least c predecessors in their first-tier bin. We bound Z using Lemma 6.1.

Write $r = (\log N)^{1+\epsilon}$. If $c \geq r$, fix a bin and let X be its A -load. For $t = \min\{c, k\}$, limited independence gives

$$\Pr[X \geq c] \leq \frac{\mathbb{E}\binom{X}{t}}{\binom{c}{t}} \leq \frac{\binom{v}{t}m^{-t}}{\binom{c}{t}} \leq \left(\frac{ev}{cm}\right)^t \leq (e/32)^t.$$

Here $t \geq r$ for sufficiently large N . A union bound over the bins shows that no key overflows, except with negligible probability.

If $c < r$, apply (3) with threshold c and moment order $t = \lfloor k/(c+1) \rfloor$, and select $D = A$. Then $(c+1)t \leq k$, $t = \Theta(k/c)$, and $\eta = (ev/(cm))^{c/2} \leq (e/32)^{c/2}$. Taking $z = \lceil m/c \rceil \geq 2t$ gives

$$\begin{aligned} \Pr[Z \geq z] &\leq \left(\frac{4e^2 v(t/m + \eta)}{z} \right)^t \\ &\leq \left(O(kc/m) + O(c^2 e^{-\Omega(c)}) \right)^t \\ &\leq \exp(-\Omega(t \min\{c, \log N\})) \leq \exp(-\Omega(\log^{1+2\epsilon} N)). \end{aligned}$$

Indeed, in this case c and k are polylogarithmic while $m = \Omega(N/c)$, and $c = q^4 = \omega(\log \log N)$. For $c \leq \log N$, the exponent is $\Omega(k)$; for $\log N < c < r$, it is at least $\Omega(k \log N/r)$. Since $c \rightarrow \infty$ and $m \rightarrow \infty$, the resulting bound $Z < \lceil m/c \rceil$ implies $Z = o(m) = o(N/q^4)$. Union bound over the polynomially many rebuild periods.

We next bound the second-tier load. Condition on the first-tier hash, which fixes the Z keys that reach the second tier, independently of $\tilde{H}_1$ and $\tilde{H}_2$. By the preceding bound, $Z < m/c < m/2$. Let C be the constant from Lemma B.1, choose a sufficiently small constant $\delta > 0$ with $C\delta < 3\epsilon$, and set $s = \lfloor \delta \log \log N \rfloor$. Since $B < N$ and $\epsilon < 1/48$, we have $m = N^{\Omega(1)}$ and hence $\log m = \Theta(\log N)$. It follows that

$$k = \log^{1+3\epsilon} N \geq C(\log m)2^{Cs}$$

for sufficiently large N . Lemma B.1 therefore gives maximum second-tier load $O(\log \log N) < c$, except with probability

$$m^{-s} = \text{negl}(N).$$

Here $c = q^4 \geq (\log N)^{4\epsilon(1+6\epsilon)} = \omega(\log \log N)$. This bound is uniform in the fixed first-tier hash and the simulated request sequence. A union bound therefore covers the polynomially many rebuild periods.

On the event that the second-tier tables and the underlying ORAM succeed, the real execution agrees with this hybrid. The simulated trace and a union bound over these exceptional events therefore give correctness and statistical obliviousness for the real dictionary.

Theorem B.2 (Statistically oblivious unordered dictionary). *Fix $0 < \epsilon < 1/48$, $B \geq (\log N)^{1+6\epsilon}$, and $B \leq M < N$. Assume that each key-value pair can be stored in $O(1)$ words. Then, there exists a statistically oblivious unordered dictionary supporting **Get**, **Set**, and **Delete** operations using $O_\epsilon(N)$ server words and*

$$O(\epsilon^{-2}(1 + \log_B(N/M)))$$

amortized block transfers per operation, where N is an a-priori known upper bound on the number of keys live at any point of time.

Proof. The two tables occupy $O(N)$ logical words, and each dictionary operation uses a constant number of super-word accesses. Thus the construction analyzed so far uses $O_\epsilon(N)$ server words and incurs $O(\epsilon^{-2}(1 + \log_B N))$ amortized block transfers per operation, including the periodic rebuilds. In Section C, we will show how to use the client memory to cache part of the underlying ORAM, improving this bound to $O(\epsilon^{-2}(1 + \log_B(N/M)))$. The first-tier overflow bound, the second-tier bound stated above, and the underlying ORAM's guarantee give correctness and statistical obliviousness by the preceding hybrid argument and a union bound over any polynomial-length execution. $\square$

C Extension to General Private Cache Size M

Our block transfer bounds for both ORAM and search tree can be improved when $M \gg B$, replacing the $\log_B N$ factor with $\log_B(N/M)$ instead. The idea for achieving this improvement is simple and standard: simply place the smallest levels of the data structure inside the client's private cache. For self-containedness, we provide a more detailed description and analysis below.

C.1 Oblivious RAM and Dictionary

Reserve a constant fraction of client memory for working storage, allocate another constant fraction to the externally indexed Bucket ORAM, and divide a third fraction equally among the $\gamma = \lceil 2/\epsilon \rceil + 2$ pointer replicas of the full ORAM. In each structure, cache the largest complete prefix of levels that fits in its allocation. These prefixes are determined by the public parameters and remain cached throughout the execution. Accesses within a cached prefix are executed locally with their original word-level-oblivious pattern. A rebuild crossing the cache boundary likewise uses oblivious primitives on publicly known array lengths. Thus the complete access pattern, including accesses within the cache, remains simulatable.

In one pointer replica, the levels through level ℓ_c occupy $O(q^{\ell_c+3} + \ell_c B)$ words: the first term is the sum of the geometric level arrays, and the second accounts for whole-block rounding and stashes. Ignoring the additive second term for the moment, an allocation $\Theta(M/\gamma)$ leaves at most

$$U_{\text{ptr}} = O(1 + \log_q(N\gamma/M)) = O(\epsilon^{-1}(1 + \log_B(N/M)))$$

uncached levels. Choose C so that $C\gamma LB$ bounds the total rounding and stash space. If $M \geq 2C\gamma LB$, reserve this space first, leaving at least half of M for the geometric parts. Otherwise

$M = O_\epsilon(B(1 + \log_B N))$. When $\log_B N$ is unbounded, we have

$$\log_B(N/M) \geq \log_B N - 1 - O_\epsilon(1) - \log_B(1 + \log_B N) = \Omega(\log_B N),$$

and when $\log_B N = O_\epsilon(1)$ the additive 1 already bounds all the levels. Thus the same bound on U_{ptr} holds in either case.

A read incurs transfers only at the uncached levels; accesses to cached levels are still performed locally. A rebuild contained in the cached prefix is performed obliviously in client memory and causes no transfers. For a rebuild ending at an uncached level, the cached input is supplied locally, and the geometric sum of all smaller levels is bounded by the size of the largest level. Consequently the calculation in (15), restricted to the uncached levels, gives $O(U_{\text{ptr}})$ amortized transfers per replica. Multiplying by $\gamma = O(1/\epsilon)$ gives

$$O(\epsilon^{-2}(1 + \log_B(N/M)))$$

for all γ pointer-ORAM replicas.

For the externally indexed Bucket ORAM, cache the topmost complete bands that fit in its allocation. A prefix through height h occupies $O(2^h z + B)$ words. Since $2^H z = \Theta(N)$ when $H > 0$, rounding the cached prefix down to a band boundary leaves $O(\beta + \log(N/M))$ levels. Its path cost is therefore

$$O\left(1 + \frac{\log(N/M)}{\beta}\right) = O(\epsilon^{-1}(1 + \log_B(N/M))).$$

The amortized flush cost remains $O(1)$ even if cached nodes are charged. Combining the Bucket-ORAM and pointer costs proves the bound in Theorem 6.5. The same caching analysis supplies the general- M bound used by Theorem B.2, since the dictionary adds only a constant number of underlying ORAM accesses and $O(1)$ amortized block transfers outside the ORAM per operation.

C.2 Oblivious Search Tree

Reserve a constant fraction of M for rebuilding, including the $O(B)$ words used for reinsertion entries and working buffers, and use another constant fraction to cache a complete prefix of the search-tree hierarchy. When $M = O(B)$, use the original small-cache analysis. Otherwise, let ℓ_c be the largest cached level and let

$$U := L - \ell_c - 1$$

be the number of uncached levels. Level ℓ has public size $\Theta(qm_\ell)$ words, including its ordinary and filler nodes. These sizes grow geometrically except at the last one or two capped levels. It follows directly from the size of the prefix that

$$U = O(1 + \log_q(N/M)) = O(\epsilon^{-1}(1 + \log_B(N/M))).$$

The same bound follows from the original $U \leq L$ estimate in the small-cache case. The read phase costs $O(U)$ transfers. It still performs the prescribed data-independent accesses to the cached prefix locally.

It remains to account for rebuilding. Recall that a rebuild ending at level $\ell \geq s$ occurs once every $q^{\ell-s}$ operations. Rebuilds with $\ell \leq \ell_c$ use the same oblivious primitives entirely in client memory. When $\ell > \ell_c$, the cached input is supplied locally, and the analysis of Section 7.6 bounds the transfer cost by

$$O\left((\ell - \ell_c) + \frac{q^{\ell+1}}{B} \log N\right).$$

Here $\ell - \ell_c$ accounts for whole-block rounding only at the uncached levels; the $O(Lq)$ reinsertion list is already part of the reserved working memory. The amortized rebuild cost is consequently

$$\sum_{\ell=\max\{s, \ell_c+1\}}^{L-1} \frac{1}{q^{\ell-s}} O\left((\ell - \ell_c) + \frac{q^{\ell+1}}{B} \log N\right) = O\left(U + \frac{Uq^{s+1}}{B} \log N\right).$$

The parameter choices in Section 7.1 give $q^{s+1} \leq 10Lq^3$. They also give $B/q^3 = \Omega(B^{2/(2+\epsilon)}) \geq \Omega(\log^2 N)$, and hence

$$\frac{q^{s+1}}{B} \log N = O(L/\log N) = O(1/\epsilon).$$

Together with the bound on U , this proves an amortized cost of

$$O(\epsilon^{-2}(1 + \log_B(N/M)))$$

per operation, proving the transfer bound in Theorem 7.8. Because the same word-level-oblivious processing is used on both cached and uncached data, the simulation covers the complete memory-access pattern.