DYNO: Dynamic Oblivious Primitives with Resizable Memory

***Abstract*—Oblivious RAMs (ORAMs) and Oblivious Data Structures (ODSs) are cryptographic primitives that hide memory access patterns as seen by untrusted cloud providers. Prior state-of-the-art approaches require the user to know the memory capacity upon initialization, denoted by N, and initialize an oblivious memory of size proportional to N. Similarly, access costs are proportional to N. We provide new security definitions for dynamic ORAMs and ODSs, and propose the first efficient dynamic ORAM (DYNORAM) and dynamic ODS (DYNODS) schemes that allocate and de-allocate memory on demand, maintaining storage and access costs proportional to the number of inserted but not deleted entries at any given time, denoted by n_i. All of our constructions achieve asymptotically optimal access and storage costs (i.e., the same as the static state-of-the-art schemes with capacity exactly n_i). They can be directly used to replace the previous static oblivious primitives used by existing state-of-the-art solutions for dynamic encrypted search, which currently require upon initialization to build, store and query oblivious memories of size 257-395GB for a database with maximum capacity 2^{30}. DYNO enables the ability of dynamic and efficient memory allocation and de-allocation for oblivious memory and oblivious data structures, which is de-facto in the plaintext world.**

1. Introduction

With cloud computing gaining popularity, the privacy of users' sensitive data has become a major concern. The benefits of data outsourcing and the importance of privacy have been stressed in numerous earlier works (e.g., [1–6]). Encryption cannot fully protect how the user accesses data (the "access pattern") which can leak sensitive information [7–9]. Oblivious RAMs (ORAMs) [10–12] and Oblivious Data Structures (ODSs) [13–15] allow us to outsource data and computations to untrusted cloud providers while completely hiding the memory access patterns.

All efficient prior ORAM and ODS schemes require users to know the maximum required memory size, denoted by N, in advance. Then, these schemes need to allocate and fill an encrypted memory with size proportional to N during initialization (the memory is filled with encrypted dummy elements) which is essential for the security of these schemes. In addition to storage and initialization costs, the access costs will also be proportional to N. In many real-world applications, however, the maximum size is either not known in advance, or it is reached long after initialization. We use n_i to denote the total number of inserted but not deleted entries at time i. The main question of this paper is:

Can we design efficient ORAM/ODS schemes with storage and access costs proportional to n_i instead of N?

This seems impossible at first, as an ORAM/ODS should hide whether it contains $n_i = 0$ or $n_i = N$ entries. In addition, a dynamic ORAM/ODS that is initially empty (i.e., at capacity 0) requires all operations to have the same impact on the ORAM/ODS for them to be indistinguishable. I.e., a "write" (or insert) operation needs to expand the ORAM/ODS by at least one entry, which implies that the "read" operations will have to pretend that they add new entries to remain indistinguishable from "write" operations. As we will show in Section 4, this means that we have to add some leakage to achieve resizability, and raises the question of how we can add the minimum amount of leakage necessary to achieve our goal efficiently.

One approach to achieve resizability is to create an instance of the ORAM or ODS and rebuild it with double its size when it becomes full. Here, "rebuild" is a *stop-the-world* event, where the client has to download all of the data and initialize an entirely new instance from scratch. We call this the "naive" solution and show that it has prohibitive costs in Section 6.

Our Contributions. In this work, we present the first efficient dynamic oblivious memory (DYNORAM) and dynamic oblivious data structures (DYNODS) that address the above questions. In particular:

1) We introduce formal definitions for DYNORAM and DYNODS with four different types of leakage ordered from the most to the least secure: Type-I is equivalent to standard ORAM/ODS definitions where all operations are indistinguishable. Type-II/Type-III/Type-IV leak only information related to the type of the performed operations: Type-II leaks 2 operation types—for example whether the operation is "insert" or "search/update/delete", where the "search", "update", and "delete" operations remain indistinguishable; Type-III leaks 3 operation types—whether the operation is "insert", "delete", or "search/update"; and Type-IV leaks 4 operation types—all operations are distinguishable. We highlight that the operands (e.g., what was inserted) are not leaked. This is especially useful as many applications, such as Searchable Encryption (SE) [5, 16, 17], already leak the operation types and can leverage the types with more leakage (i.e., leaking more operations) for better efficiency.

2) We propose a variation of PathORAM [12] called LazyPathORAM, and use it as a building block in all of our constructions. LazyPathORAM allows us to instantiate a static ORAM/ODS in constant time and incrementally initialize it (i.e., without filling the ORAM with dummy entries upon initialization). We can extend LazyPathORAM's ideas to all of the Tree-based ORAMs (e.g., [14, 18–20]), and it can replace PathORAM in any application.

3) We propose DYNORAM and DYNODS which are based on the idea of maintaining two static lazy ORAM/ODS instances and moving entries between them. Specifically, we maintain two ORAM/ODS instances of size X (the smaller ORAM/ODS) and 2X (the larger one). During insertions, for instance, the new entry is inserted in the large ORAM/ODS and one entry is moved from the smaller ORAM/ODS X to

the larger one. Whenever the larger ORAM/ODS instance becomes full we discard the (now empty) smaller one and allocate an even larger ORAM/ODS instance with size 4X.

4) In the case of DYNODS, we propose a framework that can facilitate the creation of different dynamic ODS schemes, such as single maps, multi-maps, sorted-multi-maps, min/max heaps, stacks, and queues. For all of these data structures, we use the static state-of-the-art construction as a building block. We create Type-III or Type-IV schemes, explain how their security can be elevated, and empirically show the related costs (see Sections 5.5 and 6.3).

5) All of our proposed constructions (DYNORAM and all different proposed DYNODS schemes) achieve asymptotically optimal access and storage costs, i.e., we asymptotically match the cost of an *ideal* solution that initializes a static scheme with n_i entries at time i—we highlight that this is an *artificial* reference point. We experimentally show that our access costs are 1.9-7.8× times slower than the ideal, and our storage costs are within 1.5-3× higher.

Applications. Below, we provide some applications for our DYNORAM and DYNODS constructions.

- **Oblivious/Dynamic Searchable Encryption (SE):** Existing works on Oblivious Dynamic SE (such as Orion [16]) and Dynamic Searchable Encryption (such as SDD and QOS in [17]) use oblivious RAMs and maps as building blocks to achieve the desired level of privacy. However, in both cases, their storage and access complexities depend on N and not on n_i since they have to initialize oblivious data structures of size N during setup. In all cases, our DYNODS (with security Type-III/IV, since the operation type is leaked in SE anyway) can be used for improving the asymptotic/practical performance. For example, to initialize OSSE [5] with capacity 2^{30}, two oblivious maps need to be initialized with the total size of *395GB, which can be entirely avoided using* DYNO (see Section 6.3).
- **Lack of practical I/O Efficient Oblivious RAMs:** Recently, new ORAM schemes with good I/O performance (a.k.a. locality) have been proposed [21, 22]; however, the proposed solutions are either theoretical approaches [21] or require a super-linear stash size [22]. Our DYNO schemes allow the user to stay within main memory boundaries for as long as their data fits in it; and only then have to store (some of) the data in secondary storage. We empirically evaluate this and show that the effect is significant (see Section 6.3).
- **Dynamic Oblivious Databases:** Our proposed DYNO can be used as building block for supporting oblivious point, range, join, group-by, top-k, and bottom-k queries in dynamic oblivious databases (and/or used in combination with [23–25]).
- **Dynamic ORAM/ODS for hardware enclave platforms:** Recent works [1, 14] propose ORAMs and ODSs for hardware enclave platforms which are useful for several applications, such as private contact discovery in Signal or private retrieval of public keys in Key Transparency. We implement a lazy variant of the doubly-oblivious Path OMap [14] and show that the average access cost of DYNOMap in a write-heavy workload is 2.4× faster than the naive solution for 16M entries.

Prior Works. Numerous works have been proposed with the goal of improving the practical and theoretical performance of ORAMs and oblivious computation [10, 11, 19, 25–45]. Recently, new approaches have been proposed that try to enrich ORAMs with new features, e.g., new ORAM schemes with good I/O performance (a.k.a. locality) [21, 22]; dedicated approaches for hardware enclaves [1, 14, 18, 23]; ORAMs designed for MPC [18, 37]; high-throughout ORAMs [1, 35, 38, 46, 47]; oblivious parallel RAM (OPRAM) constructions [48–55]; and approaches that offer more powerful oblivious primitives, such as oblivious data structures [13–15].

To the best of our knowledge, none of the above approaches offer ORAM/ODS with dynamic memory allocation/de-allocation. The closest to our work is the approach proposed by Moataz et al. [56], which is the only work that offers dynamic/resizable ORAMs. Their proposed approaches are tailored to the tree-based ORAM of Shi et al. [20]. They propose adding layers or nodes to the ORAM tree, analyze and find the best parameters for this setup, and prove the scheme secure. However, their analysis is limited to the original TreeORAM scheme and therefore requires $O(N \log N)$ server storage and has $O(\log^3 N)$ access cost (or $O(\log^2 N)$ assuming a local position map), which is a factor of $\log N$ less efficient than PathORAM in both aspects. DYNO outperforms [56] in all of the above aspects both theoretically and experimentally, achieves asymptotically optimal storage and access costs, can use any Tree-based ORAM scheme (including PathORAM and its derivatives) internally, and can be generalized to offer dynamic ODSs. We empirically show that for $n_i = 2^{30}$ DYNORAM outperforms [56] by *10× in running time, 54× in round trips, and 37× in bandwidth usage* (Section 6.2).

We also note that Wang et al. [13] present an oblivious memory allocator that solves an entirely different problem: management of a fixed-size memory, as an OS memory allocator does. We emphasize that our schemes do not need to hide memory allocations and de-allocations; we achieve our security goals assuming that the adversary can observe the memory allocations and de-allocations.

Technical Roadmap. First, in Section 2 we provide the necessary background, including the prior works we use as building blocks. In Section 3, we present an augmented version of the PathORAM scheme [12] named LazyPathORAM, a key building block for DYNO, which removes the linear initialization cost of PathORAM. Then, in Section 4 we present the new security definitions for dynamic ORAM/ODS primitives with different levels of security; effectively balancing leakage and performance. In Section 5 we present DYNO, a framework for creating dynamic ORAM (DYNORAM) and ODS (DYNODS) schemes: simple maps, multi-maps, sorted multi-maps (Section 5.2), heaps (Section 5.3), and stacks and queues (Section 5.4). We then discuss how we can augment DYNO to achieve any of the security levels introduced in Section 4, covering all of the leakage-performance spectrum (Section 5.5). In section 6

we present empirical analysis of the the costs and benefits of using DYNO, the severity of the problems that DYNO can solve, and the costs of elevating the security of the DYNO schemes. We present complimentary materials in Appendices and the extended version of the paper [57].

2. Preliminaries

We denote by $\lambda \in \mathbb{N}$ a security parameter. PPT stands for probabilistic polynomial-time. Our protocols are executed between two parties, a client and a server. We denote with $(x'; y') \leftrightarrow P(x; y)$ a (multi-round) protocol with input x and output x' for the client, and input y and output y' for the server; $x' \leftarrow \text{Alg}(x)$ indicates the evaluation of an algorithm. We assume an honest but curious adversary.

Negligible function. A function v: $\mathbb{N} \rightarrow \mathbb{R}$ is negligible in λ, denoted by $negl(\lambda)$, if for every positive polynomial $p(\cdot)$ and all sufficiently large λ, $v(\lambda) < 1/p(\lambda)$.

Obliviousness. Intuitively, a memory/algorithm/data-structure is oblivious *iff* for any two same-sized sequences of polynomially many operations, their resulting access patterns (i.e., the sequence of memory accesses while performing the operations) is computationally indistinguishable for anyone but the client, assuming an honest but curious adversary that sees all memory accesses and network communications. Below, we provide the definitions of oblivious memory and data structures.

Oblivious RAM (ORAM). Oblivious RAM (ORAM) [10, 11, 26–40] is a compiler that encodes the memory such that accesses to the compiled memory do not reveal access patterns to the original memory. An ORAM scheme consists of two protocols – ORAM = (ORAMSETUP, ORAMACCESS):

- $(\sigma; R) \leftarrow \text{ORAMSETUP}(1^\lambda, M)$: takes as input the security parameter λ and the memory array M of N values and outputs the secret state σ for the client and the encrypted memory R for the server.
- $(v_i, \sigma'; R') \leftrightarrow \text{ORAMACCESS}(op, i, v_i, \sigma; R)$: is a protocol between the client and server, where the client's input is the type of operation op (read/write), an index i, and the value v_i (for op = "read" client sets $v_i = \perp$). Server's input is the encrypted memory R. The client's output consists of the updated secret state σ and the the old value of $M[i]$ (unchanged if op = "read"). The server's output is the updated encrypted memory R'.

Security of ORAM. An ORAM = (ORAMSETUP, ORAMACCESS) scheme is statistically secure with respect to security parameter λ *iff* there exists a polynomial-time simulator Sim = (SIMORAMSETUP, SIMORAMACCESS), where for any initial memory array and for any polynomial-length sequence of operations chosen by the adversary, SIMORAMSETUP is indistinguishable from ORAMSETUP, and SIMORAMACCESS is indistinguishable from ORAMACCESS. The formal definition is similar to the ODS security definition below, and can be found in the extended version of the paper [57] (Appendix D.1).

Tree-ORAMs. We focus on ORAMs that fall within the tree-based framework, such as [12, 14, 18–20], since they are aligned with our goal to maintain poly-log amortized and worst-case cost in the dynamic setting. Tree-ORAM constructions organize the ORAM storage into a full binary tree with N leaves, where each node contains a fixed number of values, and move values obliviously along the tree paths (from the root to different leaves) to hide the memory access patterns. In this work, we use a slightly modified version of PathORAM, which we explain in Section 3. We use PathORAM since it is the most known variant of the Tree-ORAM constructions; however, DYNO can be adapted to work with other Tree-ORAM constructions in a straightforward way. We assume readers are familiar with the PathORAM construction [12]; but also briefly describe the non-recursive PathORAM in Section 3.

Oblivious Data Structures (ODS). An ODS aims to ensure that for any two sequences of the same number of operations, the resulting access patterns are indistinguishable. This implies that the memory access patterns, including the number of accesses, should not leak any information about the operations or the operands. Below, we provide an abstract interface for ODS and a generic description of the desirable security. Later, however, these interfaces and security definitions will be modified and specialized because we aim to provide a suite of various commonly used ODSs, such as maps, multi-maps, sorted multi-maps, stacks, queues, and min-/max-heaps.

In general, an ODS scheme consists of five algorithms/protocols—ODS=(ODSSETUP, ODSSEARCH, ODSINSERT, ODSUPDATE, ODSDELETE):

- $(\sigma; T) \leftarrow \text{ODSSETUP}(1^\lambda, N)$: Given a security parameter λ and an upper bound N on the number of entries, outputs the secret state σ for the client and the (encrypted) oblivious data structure T for the server.
- $(res, \sigma'; T') \leftrightarrow \text{ODSSEARCH}(searchArgs, \sigma; T)$: Returns the value(s) if exists and the updated σ and T.
- $(\sigma'; T') \leftrightarrow \text{ODSINSERT}(insertArgs, \sigma; T)$: Inserts an entry and returns the updated σ and T.
- $(\sigma'; T') \leftrightarrow \text{ODSUPDATE}(updateArgs, \sigma; T)$: Updates an existing entry and returns the updated σ and T.
- $(res, \sigma'; T') \leftrightarrow \text{ODSDELETE}(deleteArgs, \sigma; T)$: Deletes an entry and returns the corresponding value(s) (if existed) and the updated σ and T.

Note that the arguments ($searchArgs$, $insertArgs$, $updateArgs$, and $deleteArgs$) are ODS-dependent; e.g., a map takes a key and a value for insertion, but a queue only takes a value.

Security of ODS. An ODS = (ODSSETUP, ODSINSERT, ODSSEARCH, ODSUPDATE, ODSDELETE) is statistically secure with respect to security parameter λ *iff* there exists a polynomial-time simulator Sim= (SIMODSSETUP, SIMODSACCESS), such that for any polynomial-length sequence of data structure operations chosen by the adversary, SIMODSSETUP is indistinguishable from ODSSETUP, and SIMODSACCESS is indistinguishable from ODSSEARCH, ODSINSERT, ODSUPDATE, and ODSDELETE. The variants of the leakage function $\mathcal{L}$ are defined in Section 4.

Definition 1. *An ODS scheme Π is $\mathcal{L}$-secure if for any PPT adversary $\mathcal{A}$, there exists a stateful PPT simulator Sim = (*SIMODSSETUP, SIMODSACCESS*) such that:*

$$|\Pr[\mathbf{Real}^{\mathsf{ODS}}_{\mathcal{A},\Pi}(\lambda)=1]-\Pr[\mathbf{Ideal}^{\mathsf{ODS}}_{\mathcal{A},\mathsf{Sim},\mathcal{L}}(\lambda)=1]| \leq \mathsf{negl}(\lambda)$$

The randomness is taken over the random bits used by the algorithms of the ODS scheme, the algorithms of the simulator, and $\mathcal{A}$. For the standard static ODS definition, $\mathcal{L}(op) = \perp$ (Type-I). Experiments $\mathbf{Real}^{\mathsf{ODS}}_{\mathcal{A},\Pi}(\lambda)$ and $\mathbf{Ideal}^{\mathsf{ODS}}_{\mathcal{A},\mathsf{Sim},\mathcal{L}}(\lambda)$ are formally defined in Appendix B (Figure 12).

Dummy Operations. ODSs (and ORAMs) often use dummy operations in which randomly chosen entries are accessed and written back. Since all entries in an ORAM/ODS are encrypted with fresh randomness every time written (using a CPA-secure encryption scheme [58]), dummy operations are indistinguishable from real (or other dummy) operations. We denote this by explicitly noting the dummy nature of an operation, or by passing $\perp$ as argument.

Oblivious Sorted Multi-Map (OSM). Binary trees and specifically AVL trees can be used to implement maps. Wang et al. [13] show how AVL trees can be used obliviously to create Oblivious Maps (OMaps). *The key idea is to store the key and the position of the root of the tree in the client, and store the key and the position of both children of a node in it*. Thus, without a position map, an AVL-path (or even the whole AVL-tree) can be read from the top down, re-mapped to new positions in the underlying ORAM, and written back from the bottom up. Each node's position is updated in its parent, and the root's position is updated in the client state. Recall that AVL trees are self-balancing and have a logarithmic upper-bound on the number of memory accesses in each operation [13]. The AVL tree ensures that the absolute height difference between the child sub-trees of a node remains below one, and this is done through a series of "rotations" after node insertions and deletions. The total number of ORAM accesses in each operation is bounded by $3 \times 1.44 \times \log N$, which comes from the worst-case AVL tree height and the maximum number of nodes accessed per rotation [13]. Furthermore, Mishra et al. [14] showed that by augmenting AVL trees to store some extra metadata in each node, i.e., the number of entries with the same key in each child sub-tree, we can create sorted multi-maps that allow a client to request the i-th smallest value of a key or get the range of the i-th to j-th values of a key efficiently. This scheme allows insertions and deletions of key-value pairs in $O(\log N)$ ORAM accesses, and the search for a range of k entries in $O(\log N + k)$ ORAM accesses. We extend the OSM scheme and use it as our main construction: DYNOMap (see Section 5.2). We cover more details about the OSM scheme, an example of a series of operations, and tight bounds on the constants of the access cost complexities in the extended version (Appendix E).

Path Oblivious Heap. The Path Oblivious Heap scheme of Shi [15] implements an oblivious min (or max) heap with optimal access complexity ($O(\log N)$). We use this scheme as a black-box in our our DYNOHeap scheme (Section 5.3).

3. Lazily Built Static PathORAM

In this section we present LazyPathORAM, a variant of PathORAM [12] where the initialization of the ORAM is piggy-backed on the oblivious accesses. This satisfies a property we need in Section 5 to create all of our dynamic ORAM and ODS schemes (we can derive this property from any tree-based ORAM). Below, we describe the standard (non-recursive) PathORAM version, and then our new variant called LazyPathORAM.

PathORAM. Similar to other Tree-ORAMs, a full binary tree with N leaves is stored in the server. Each node contains a constant number of memory blocks (typically 4 [12]), and each memory block is tagged with a position, which indicates the leaf it belongs to. This structure can contain up to N real memory blocks, and the remaining slots in the nodes are filled with dummy blocks (where their dummy nature is not distinguishable by the server). The crucial invariant is that a block is always on the path from the root to its leaf. When accessing a block, a position map is used to find its leaf tag, and the entire path to the leaf is read to find the block. Then, to not visit the same path for that block in the next access, the block is remapped to another path, chosen uniformly at random (and the position map is updated). The original path is written back, and the block is written to any node that is in the intersection of the old path and the newly chosen one; thus maintaining the invariant. Also, every write is encrypted with fresh randomness, even if the path is written back unchanged. A client stash is used for when the nodes on the accessed path overflow. The process of writing back to the path is called eviction, and PathORAM uses a greedy approach to fill as many blocks as it can in the path (minimizing the stash size)—the stash size is proven to be $\tilde{O}(\log N)$ blocks [12].

LazyPathORAM. We observe that **(A)** each block is always written to a randomly chosen path that differs from the path it will be read from; **(B)** if the ORAM is known (by the adversary) to be empty when created, writing dummy values is pointless, as each node will predictably contain dummy values before its first access; and **(C)** even if the client already has N blocks when creating the ORAM, the blocks can be kept in client storage, and their writing to the ORAM can be deferred to the next N accesses. Therefore, a client can *always* start with an ORAM that is empty, and skip the initialization of the tree nodes. Then, the client can keep the initial values in its stash, and piggy-back them on the next N accesses. We call the above LazyPathORAM and later we use it as a building block for Dyno (in Section 5). There, we always start each instance from an empty state, and therefore do not need to store the initial values in the client state (the client stores only the stash). The full construction (in pseudocode) can be found in Appendix A.

Security. In PathORAM, the position tags are randomly generated for all blocks during setup. But, when writing a previously absent block, the block is not present in its tagged path, and reading any (randomly chosen) path satisfies the protocol. Therefore, the choice (i.e., random sampling) of the "write path" can be deferred to when the block is written

for the first time. Furthermore, the paths and nodes accessed by the client are always observable by the adversary. Following the above observations, we claim that LazyPathORAM has the same security as PathORAM. The scheme details can be found in Appendix A.

Theorem 1. *Assuming ORAM instances are created empty and this is known by the adversary, LazyPathORAM has the same leakage as PathORAM.*

Proof. Since the ORAM is created without any real data, the adversary knows that no node can contain any real values before the first time it is accessed. Therefore, since the only difference between PathORAM and LazyPathORAM is whether or not they write dummy values to all (known to be empty) nodes during initialization, the two schemes have the same leakage. □

Furthermore, even if the client has N initial values, it can add them to its stash after initializing an empty ORAM, and its stash size will go back to $\tilde{O}\log N$ blocks after at most N accesses. Again, DYNO only initializes empty instances, and its stash size will remain upper-bounded by $\tilde{O}\log N$.

Efficiency. The initialization cost of LazyPathORAM is reduced to the cost of allocating (without initialization) the space needed for the server memory, as it only reserves the space during ORAMSETUP. The rest is similar to the non-lazy PathORAM: the access cost is $O(\log N)$ if the client stores the position map, or $O(\log^2 N)$ if the position map is recursively stored in the server.

4. New Dynamic Oblivious Primitives

In this section, we extend the standard definitions of ORAM and ODS to support dynamic memory allocation and de-allocation. Let us assume that N is the maximum required ORAM/ODS capacity for a given application, while n_i is the total number of inserted (and non-deleted) entries/blocks at time i. For an efficient dynamic oblivious primitive with dynamic memory allocation/de-allocation, we desire both the storage and the insert/search/update/delete complexities to be proportional to n_i and not N.

As we already discussed, efficiently achieving dynamic memory allocation/de-allocation with no leakage is impossible. We formalize this claim, and then relax the standard ORAM/ODS definitions to leak some information about the types of the operations.

Lemma 1. *A dynamic ORAM/ODS that hides the performed operation (i.e., insert/search/update/delete) requires $\Omega(N)$ space after N operations (N is the upper bound on the number of required entries).*

In order to prove this very simple lower bound, we need to consider a dynamic ORAM/ODS that begins being empty (i.e., at capacity 0). To hide the operation types, we need to make sure that any operation will have the same effect in the ORAM/ODS. An "insert" operation needs to be indistinguishable from a "search", "update" or "delete" operation. An insert operation needs to expand the ORAM/ODS by at least one entry. Thus, all of the operations (in order to be indistinguishable) will have to pretend that they add a new entry (by adding dummy entries) to the ORAM/ODS until we reach N entries (including "search" operations). After reaching N and based on the fact that N is the maximum required ORAM/ODS capacity, we can use the space filled by dummy entries for future insertions. The above bound implies a lower bound for the cost of all insert, search, update and delete operations (which has to be the same in all of these operations—depending on N and not n_i).

Now, let's consider the case that we perform $N/2$ inserts followed by $N/2$ delete operations—deleting the first $N/2$ entries. According to the above bound, the storage complexity has to be $\Omega(N)$ and not proportional to n_i (0 in this case). Similarly, the insert, search, update, and delete costs have to be proportional to N and not n_i.

Below, we propose the definition for the Dynamic ODS—DYNODS; we provide the definition of the Dynamic ORAM (DYNORAM) and its construction in the extended version (Appendix D.2).

DYNODS. Similar to the ODS definition in Section 2, here, we provide an abstract interface for dynamic ODS, which will be specialized in Section 5 for each of our proposed dynamic data structures (especially, maps and heaps). A DYNODS scheme consists of five algorithms/protocols—DYNODS = (DYNODSSETUP, DYNODSINSERT, DYNODSSEARCH, DYNODSUPDATE, DYNODSDELETE):

- $(\sigma; T) \leftarrow$ DYNODSSETUP(1^λ): Given a security parameter λ, outputs the secret state σ for the client and the (encrypted) oblivious state T for the server.
- $(res, \sigma'; T') \leftrightarrow$ DYNODSSEARCH$(searchArgs, \sigma; T)$.
- $(\sigma'; T') \leftrightarrow$ DYNODSINSERT$(insertArgs, \sigma; T)$.
- $(\sigma'; T') \leftrightarrow$ DYNODSUPDATE$(updateArgs, \sigma; T)$.
- $(res, \sigma'; T') \leftrightarrow$ DYNODSDELETE$(deleteArgs, \sigma; T)$.

DYNODSSEARCH, DYNODSINSERT, DYNODSUPDATE, and DYNODSDELETE are similar to their static counterparts (ODSSEARCH, ODSINSERT, ODSUPDATE, and ODSDELETE; Section 2). They take ODS-specific arguments and the client/server states as input and output the operation result and the updated client/server states.

DYNODS Security. We define the security of DYNODS in the real/ ideal security game. The leakage function $\mathcal{L}$ will allow us to leak controlled information **only** about the type of the operations. A DYNODS = (DYNODSSETUP, DYNODSINSERT, DYNODSSEARCH, DYNODSUPDATE, DYNODSDELETE) is $\mathcal{L}$-secure *iff* there exists a polynomial-time simulator Sim= (SIMDYNODSSETUP, SIMDYNODSACCESS), such that for any polynomial-length sequence of operations chosen by the adversary, SIMDYNODSSETUP is indistinguishable from DYNODSSETUP, and SIMDYNODSACCESS is indistinguishable from DYNODSINSERT, DYNODSSEARCH, DYNODSUPDATE, DYNODSDELETE. The difference with the definition in Section 2 is that Sim takes the output of a newly defined leakage function $\mathcal{L}$ as input. The formal security definition of ODS is given in Section 2. Below, we define three types of an $\mathcal{L}$-secure DYNODS.

Definition 2. *An $\mathcal{L}$-adaptively-secure* DynODs *is*

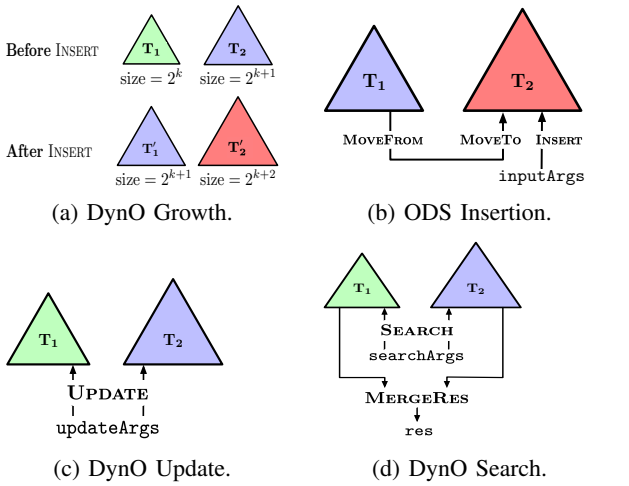
- **DynODs-*Type I:*** *iff $\mathcal{L}(op) = \perp$. It is equivalent with the standard definition in Section 2.*
- **DynODs-*Type II-INS:*** *iff $\mathcal{L}(op) = \{$"insert" or "search/update /delete"$\}$. It leaks two different operations: whether the operation is "insert" or "search/update/delete" ("search", "update", and "delete" operations remain indistinguishable).*
- **DynODs-*Type II-DEL:*** *iff $\mathcal{L}(op) = \{$"delete" or "search/upda-te/insert"$\}$. It leaks two different operations: whether the operation is "delete" or "insert/search/update" ("insert", "search", and "update" operations remain indistinguishable).*
- **DynODs-*Type III:*** *iff $\mathcal{L}(op) = \{$"insert" or "delete" or "search /update"$\}$. It leaks three different operations: whether the operation is "insert", "search/update", or "delete" ("search" and "update" operations remain indistinguishable).*
- **DynODs-*Type IV:*** *iff $\mathcal{L}(op) = \{$"insert" or "delete" or "search" or "update"$\}$. It leaks all of the operations.*

We introduced a formal definition for DynODs with four different types of leakage ordered from most to least secure; Type-I leaks 1 operation type, Type-II leaks 2 operation types, and so on. Different data structures and different applications can afford to leak different information. In the next sections, we provide our dynamic data structures that achieve the Type-III or Type-IV definitions. However, we explain how they can be modified to achieve a stronger definition (at the cost of additional performance overhead). Regarding Type-III vs. Type-IV, when "search" operations return one entry, hiding the difference between the "search" and "update" operations is achieved without any additional cost in the ODS (and ORAM) setting; every "search" operation requires the entry that contains the value to be written back to the server, and every "update" operation must first fetch the entry from the server. But, similar to the static setting, when the result size can vary (e.g., a multi-map returning all values matching a key), the leakage changes. This can be avoided by padding every operation (or just the search operations) to the worst-case size [24, 59–61].

We note that the DynORAM definition (extended version, Appendix D.2) is also organized in four types; our DynORAM scheme can achieve any of the four types, similar to DynODs; and the only related prior work [56] has the same security as DynORAM-Type-III.

5. Our DynO Approach

In this section, we present our DynODs constructions, which are all based on the same idea—we allocate and discard static ODS instances and move entries between them so that we always have the necessary space (for n_i entries) while remaining efficient. We start by presenting a framework that we use as the base of our constructions. We use the same idea to present DynORAM in the extended version (Appendix D.2). While we describe the DynOMap and DynOHeap in Sections 5.2 and 5.3, we use examples about them to explain the framework. DynOMap is based on the OSM of Mishra et al. [14] (see Section 2), and DynOHeap is based on the OHeap of Shi [15] (see Section 5.3). First, we present the two main building blocks of the framework in more detail.



(a) DynO Growth. (b) ODS Insertion.

(c) DynO Update. (d) DynO Search.

Figure 1: DynO Growth, Insertion, Update, and Search. Growth (a) happens every time the ODS reaches size 2^k (for any k), but entries are moved during every insertion (b). Updates and searches are done in both instances (c, d).

Lazy-PathORAM. All schemes use LazyPathORAM (see Section 3) as a building block. This allows us to instantiate the necessary static ODSs that we use without initializing them with dummy entries. Without the lazy variant of PathORAM, we would have to pay a cost linear in n_i whenever we created a new static ODS instance.

Dynamic ODS via two static ODS instances. Internally, a DynODs instance maintains two static ODS instances and moves entries between them such that it always takes 1.5-3 times the space needed to store n_i entries. More specifically, if $2^k < n_i \leq 2^{k+1}$, DynO maintains two ODS instances of size 2^k (the smaller ODS instance) and 2^{k+1} (the larger one). To maintain the invariant, when an entry is inserted, the number of entries in the smaller ODS is decreased by 1 and the number of entries in the larger ODS is increased by 2; and when an entry is deleted, the number of entries in the smaller ODS is increased by 1 and the number of entries in the larger one is decreased by 2.

During insertions, whenever the larger ODS instance becomes full, we discard the smaller one and allocate a new ODS instance with size 2^{k+2} (we mirror this for deletions). This happens as part of an insert, and we achieve the same worst-case and amortized performance by using the lazy variant of the static ODS—it is the same as the non-lazy variant except that it uses the lazy PathORAM, and therefore its OdsSetup algorithm only needs to reserve space.

5.1. DynODs framework

Below, we present the DynODs = (DynOdsSetup, DynOdsInsert, DynOdsSearch, DynOdsUpdate,

$(\sigma; T) \leftarrow$ DynOdsSetup(1^λ)

Set σ_1, σ_2 to be empty static ODS client states
Set T_1, T_2 to be empty static ODS instances
$size \leftarrow 0$
$\sigma \leftarrow (\sigma_1, \sigma_2, size)$
$T \leftarrow (T_1, T_2)$
return $(\sigma; T)$

$(\sigma'; T') \leftrightarrow$ DynOdsInsert$(insertArgs, \sigma; T)$

$(\sigma_1, \sigma_2, size) \leftarrow \sigma$ and $(T_1, T_2) \leftarrow T$
if $size = 0$ **then**
  $(\sigma_2; T_2) \leftarrow$ OdsSetup$(1^\lambda, 1)$
else if $size$ is a power of 2 **then**
  $\sigma_1 \leftarrow \sigma_2; T_1 \leftarrow T_2$
  $(\sigma_2; T_2) \leftarrow$ OdsSetup$(1^\lambda, 2 \times size)$
$(\sigma; T) \leftarrow$ Move$(2, \sigma; T)$
  ▷ first argument is destination
$(\sigma_2; T_2) \leftrightarrow$ OdsInsert$(insertArgs, \sigma_2; T_2)$
$size \leftarrow size + 1$
$\sigma' \leftarrow (\sigma_1, \sigma_2, size)$ and $T' \leftarrow (T_1, T_2)$
return $(\sigma'; T')$

$(\sigma', T') \leftrightarrow$ Move$(d, \sigma; T)$

$(\sigma_1, \sigma_2, size) \leftarrow \sigma$ and $(T_1, T_2) \leftarrow T$
$s \leftarrow \{1, 2\} \backslash \{d\}$ ▷ s: source; d: destination
$(x, \sigma_s; T_s) \leftrightarrow$ MoveFrom$(size, \sigma_s; T_s)$
$(\sigma_d; T_d) \leftrightarrow$ MoveTo$(x, \sigma_d; T_d)$
  ▷ MoveFrom/MoveTo are defined per ODS.
$\sigma' \leftarrow (\sigma_1, \sigma_2, size)$ and $T' \leftarrow (T_1, T_2)$
return $(\sigma'; T')$

Figure 2: Dynamic ODS (**DynO**) Framework: DynOdsSetup and DynOdsInsert.

$(res, \sigma'; T') \leftrightarrow$ DynOdsDelete$(deleteArgs, \sigma; T)$

$(\sigma_1, \sigma_2, size) \leftarrow \sigma$ and $(T_1, T_2) \leftarrow T$
$res \leftarrow$ empty array of size 2
for $i \in 1..2$ **do**
  if $T_i \neq \emptyset$ **then**
    $(res_i, \sigma_i; T_i)$
      $\leftrightarrow$ OdsDelete$(deleteArgs, \sigma_i; T_i)$
$retVal \leftarrow \perp$
if $res_1 \neq \perp \vee res_2 \neq \perp$ **then**
  $size \leftarrow size - 1$
if $res_1 \neq \perp \wedge res_2 \neq \perp$ **then**
  $retVal \leftarrow$ FilterRes$(deleteArgs, res)$
    ▷ FilterRes is defined per ODS.
  $replaceVal \leftarrow \{res_1, res_2\} \backslash \{retVal\}$
  $(\perp, \sigma_2; T_2) \leftrightarrow$ MoveFrom$(\perp, \sigma_2; T_2)$
    ▷ Dummy
  $(\sigma_1, T_1) \leftrightarrow$ MoveTo$(replaceVal, \sigma_1, T_1)$
  $(\sigma; T) \leftrightarrow$ Move$(1, \sigma; T)$
else if $res_1 \neq \perp$ **then**
  $retVal \leftarrow res_1$
  for $i \in 1..2$ **do** $(\sigma; T) \leftrightarrow$ Move$(1, \sigma; T)$
else ▷ $res_2 \neq \perp$
  $retVal \leftarrow res_2$
  $(\perp, \sigma_2; T_2) \leftrightarrow$ MoveFrom$(\perp, \sigma_2; T_2)$
    ▷ Dummy
  $(\sigma_1; T_1) \leftrightarrow$ MoveTo$(\perp, \sigma_1; T_1)$ ▷ Dummy
  $(\sigma; T) \leftrightarrow$ Move$(1, \sigma; T)$
if $size$ is a power of 2 **then**
  $\sigma_2 \leftarrow \sigma_1; T_2 \leftarrow T_1$
  $(\sigma_1; T_1) \leftarrow$ OdsSetup$(1^\lambda, size/2)$
$\sigma' \leftarrow (\sigma_1, \sigma_2, size)$ and $T' \leftarrow (T_1, T_2)$
return $(retVal, \sigma'; T')$

Figure 3: Dynamic ODS (**DynO**) Framework: DynOdsDelete.

DynOdsDelete) framework in detail by describing each protocol—the DynOds framework is presented in Figures 2, 3, and 4. Note that, as we will show later, different DynOds schemes may store different information in their client state. Here, we only cover the information that is needed for the framework itself.

DynOdsSetup. The setup algorithm simply creates the necessary state for DynOds (see DynOdsSetup in Figure 2). We do not need to allocate any space or data structure on the server in the beginning as that is handled during insertions and deletions.

DynOdsInsert. When starting an insertion, DynOds checks whether the larger ODS instance ($2X$) has become full and the smaller one (X) is empty—both happen at the same time. In this case, DynOds discards the smaller ODS instance, replaces it with the larger one, and allocates a new larger instance with twice the capacity (i.e., $4X$). This is shown in Figure 1(a). Now, we know that the larger instance has space. At this point, DynOds inserts the new entry into its larger ODS instance, and moves one entry from the smaller instance to the larger one. To perform the move, the DynOds framework requires two new interfaces in the underlying ODS: MoveFrom, which is used to take an entry from the smaller instance during insertion, and MoveTo, which is used to put that entry into the larger one. These interfaces are also used during deletions, as we will explain. Figure 1(b) shows the steps taken during an insertion, and Figure 2 shows the algorithm in pseudocode.

For example, in a DynOheap, which we will describe in detail in Section 5.3, consider the case where $X = 8$ and $2X = 16$, and X is empty and $2X$ is full. during an insertion, at this point, **(A)** the X instance is discarded and replaced by the larger one ($X = 16$); **(B)** a new larger instance is allocated with capacity $2X = 32$; **(C)** the minimum entry of the 16 heap is removed (MoveFrom) and put in the 32 heap (MoveTo); and **(D)** the new entry is inserted into the 32 heap. In the end, the X heap will have 15 entries, and the $2X$ heap will have 2.

DynOdsDelete. Deletions mirror insertions, but raise a new challenge: the result of a deletion, e.g., the minimum entry in a DynOds min-heap, may be in either ODS instance; forcing us to perform the deletion in both instances. As such, we require a third algorithm to be specified for the DynOds scheme: FilterRes. FilterRes takes the two

$(res, \sigma; T) \leftrightarrow$ DYNODSSEARCH$(searchArgs, \sigma; T)$

1: $(\sigma_1, \sigma_2, size) \leftarrow \sigma$ and $(T_1, T_2) \leftarrow T$
2: Set res to be an empty array of size 2
3: **for** $i \in 1..2$ **do**
4: **if** $T_i \neq \emptyset$ **then**
5: $(res_i, \sigma_i; T_i)$ $\leftrightarrow$ ODSSEARCH$(srchArgs, \sigma_i; T_i)$
6: **else**
7: $res_i \leftarrow \perp$
8: $retVal \leftarrow$ MERGERES$(searchArgs, res)$
9: $\sigma' \leftarrow (\sigma_1, \sigma_2, size)$ and $T' \leftarrow (T_1, T_2)$
10: **return** $(retVal, \sigma'; T')$

$(\sigma'; T') \leftrightarrow$ DYNODSUPDATE$(updateArgs, \sigma; T)$

1: $(\sigma_1, \sigma_2, size) \leftarrow \sigma$ and $(T_1, T_2) \leftarrow T$
2: **for** $i \in 1..2$ **do**
3: **if** $T_i \neq \emptyset$ **then**
4: $(\sigma_i; T_i) \leftrightarrow$ ODSUPDATE$(UpdateArgs, \sigma_i; T_i)$
5: $\sigma' \leftarrow (\sigma_1, \sigma_2, size)$ and $T' \leftarrow (T_1, T_2)$
6: **return** $(\sigma'; T')$

Figure 4: Dynamic ODS (**DynO**) Framework: DYNODSSEARCH and DYNODSUPDATE.

entries returned by the ODS instances as input, and returns the entry that should be returned to the client. DYNODS also passes the arguments of the deletion to FILTERRES as they may be needed in deciding which result should be returned to the client. FILTERRES is performed in the client, and the selection is not leaked to the server. We assume that at least one of the ODS instances contains the entry to be deleted (we will discuss this assumption later). During a deletion, the following steps are taken (pseudocode in Figure 3):
(1.) The deletion is performed in both ODS instances. **(2.)** FILTERRES decides which one gets deleted. **(3.)** The deletion result that FILTERRES did not return is put (back) into the smaller ODS instance. **(4.)** An entry is moved from the larger ODS instance to the smaller one. **(5.)** If only the smaller ODS instance returned a value in (1), we need to move an extra entry from the larger ODS instance to the smaller one to achieve the desired changes in the number of entries in each instance. We always pad the number of accesses to make the different cases (of what was returned by each instance) indistinguishable. **(6.)** We check whether the smaller ODS instance is full and the larger is empty (both happen at the same time). If this is true, we discard the larger ODS instance and allocate a new smaller one with half the capacity of the previously smaller one.

For example, in the case of a min-heap, assume the smaller heap has capacity 16 and contains 15 entries, and the larger one has capacity 32 and contains 2 entries (similar to the result of the insertion example). In a deletion, the following steps are taken: **(A)** The minimum entry of both heaps are extracted; **(B)** The minimum of the two is selected using FILTERRES; **(C)** The other entry is inserted into the smaller heap; **(D)** The last entry of the larger heap is moved to the smaller one. Now, the larger is empty and the smaller is full. So, **(E)** the larger is discarded and replaced by the smaller one, and a new heap with capacity 8 is allocated, replacing the smaller one.

Searches and Updates. Searches and updates do not change the number of entries in either ODS instance. As such, they are performed in both instances. In the case of search, similar to delete, both ODS instances may return results, and an extra step is needed to decide what combination of them should be returned as a result to the client. Here, we need a fourth protocol to be specified for the ODS scheme—MERGERES, similar to FILTERRES, is called with the search arguments and results as input, and returns the final search results that the client wants. At the last step of a search, DYNODS calls the MERGERES protocol and returns its output to the client. Lastly, MERGERES is also executed in the client. Figure 1(c) shows how updates are performed in both instances, and Figure 1(d) shows the process of searching both ODS instances and merging the results. Figure 4 shows both algorithms in pseudocode.

For example, regarding searches, following the insertion example, a search (to find the minimum entry of a heap without deleting it) would **(A)** read the minimum entry of both heaps; and **(B)** choose the minimum using MERGERES.
performance. Let $T(n_i)$, $C(n_i)$, and $S(n_i)$ be the access complexity, client space complexity, and server space complexity of the underlying ODS scheme, for n_i inserted and non-deleted entries. The size of σ (i.e., the client storage complexity) is $O(C(n_i))$; i.e., the client state of DYNODS is asymptotically upper bounded by $O(C(n_i))$, as the σ of DYNODS has size $(C(n_i) + C(2n_i) + 1)$. Similarly, the server storage complexity is $O(S(n_i))$. Finally, assuming MOVEFROM and MOVETO are bounded by $T(n_i)$, the setup complexity of DYNODS is $O(1)$, and insert, search, update, and delete complexity of DYNODS are all $O(T(n_i))$. Note that compared to a hypothetical ideally-sized static instance with capacity n_i, as we will show in Section 6, DYNODS has a concrete overhead of at least $2\times$ as we perform at least 2 static ODS accesses during a DYNODS operation.
Security. Informally, DYNODS is DYNODS-Type III secure if MOVEFROM and MOVETO can be simulated in polynomial time. The underlying ODS is already oblivious, and the remaining step to achieve obliviousness is verifying that the four added algorithms, which are specific to each ODS, are oblivious. The MERGERES and FILTERRES protocols are executed directly in the client, and so, verifying the MOVEFROM and MOVETO is enough. Below, we formalize the aforementioned claim.

Theorem 2. *Assuming the used ODS is a secure oblivious data structure according to Definition 1 (see Section 2) and* MOVEFROM *and* MOVETO *are indistinguishable from* SimMOVEFROM *and* SimMOVETO*, the proposed* DYNODS *is Type III secure based on Definition 2.*

We provide the proof in Appendix B. We highlight that all latencies, memory allocations, and memory accesses are included in the leakage functions and the security model. The rest of this section presents the different data structures that we build using the framework.

Discussion. For deletions, we assume the value to be deleted already exists. This is because a deletion decreases the DYNODS capacity, and lowering the capacity if no entry is removed can lead to data loss. In practice, this can be solved by either always doing a search when unsure whether the entry exists, or, adding a DELETEORNOOP interface to the DYNODS definition. The first approach is simpler. However, the second approach is also very simple as ODS schemes already support it. The only complication is that the added leakage (i.e., an attempted deletion failed) has to be accepted and incorporated into the security definition.

5.2. DYNOMap—Oblivious Map Family

DYNOMap is our most expressive scheme, which can act as: **(1.)** an oblivious sorted multi-map (OSM), **(2.)** an oblivious unsorted multi-map (returns all matching values in a search—see ALLVALUES below), **(3.)** a simple map, allowing at most one value per key. We present DYNOMap first to show the steps of fitting an ODS scheme into the DYNO framework.

This scheme is based on Mishra et. al's oblivious sorted multi-map (OSM) [14] (see Section 2). In the OSM scheme, we start with an AVL tree, allow multiple values for a key, and use the combination of the key and the value to order the nodes. Also, in each node, we store the number of entries in each of its child sub-trees that have the same key. This allows us to find the i-th value for a key with an AVL-OMap lookup that costs $O(\log^2 N)$, and if there are r entries in a range between two entries, we can retrieve all of them in $O(\log^2 N + r \log N)$ time. In this section, when we mention ODS (including DYNODSINSERT, etc.), we mean the corresponding protocols in the OSM scheme.

We add an argument to the DYNODSSETUP algorithm that specifies which of the three variants is needed, store it in the client state, and use it to decide how to perform certain operations throughout the lifetime of the DYNODS instance. Also, this DYNODS scheme does not support DYNODSUPDATE functionality unless it is used as a simple map. In the other cases, the client has to do a DYNODSDELETE followed by a DYNODSINSERT.

Recall that OSM is an oblivious AVL tree, and since all operations start by reading the root node, the client can perform a delete operation to delete the entry contained in the root node without knowing it a priori. In this case, the entry to be deleted is decided after reading the root node (dummy operation if the ODS is empty), and the delete operation is continued from after the read operation. We thus define the DELETEROOT operation, which we use below.

The protocols that define the scheme are as follows:

- **MOVEFROM:** Performs a DELETEROOT operation.
- **MOVETO:** performs an ODSINSERT.
- **FILTERRES:** returns one of the results that is not null (with ties broken arbitrarily).
- **MERGERES:** works on a per-case basis; e.g., if the search query performed is a top-k, the results are sorted to find the top k entries. See the discussion below.

Extensions to Mishra et al.'s OSM. Mishra et al. present a range search function to get the i-th to j-th values of a key [14]. This becomes problematic in our setting with two instances, as performing the same query when the values are arbitrarily spread between two static OSM instances requires processing all of the values in the client. As our main extension **(A)**, we leverage the same binary search tree technique to implement the following queries:

- RANGESEARCH(min, max) returns all key-value pairs (k, v) such that $\langle min.k.min.v \rangle \leq \langle k.v \rangle \leq \langle max.k.max.v \rangle$.
- TOPK(key, k) returns the first k values for the key.
- BOTTOMK(key, k) returns the last k values for the key.
- ALLVALUES(key) returns all values for the key.

The RANGESEARCH query works similar to the i-j range search of the static OSM scheme, but instead of using the in-node counters to find the left and right boundaries of the results (the i-th and j-th values in the original OSM), we use standard binary search tree techniques. We traverse the tree once to find the path from the root to the leftmost entry with $\langle min.k.min.v \rangle \leq \langle k.v \rangle$, and repeat this to find the rightmost entry with $\langle k.v \rangle \leq \langle max.k.max.v \rangle$. Then, we perform a standard BFS from the least common ancestor of the two nodes to return all values that are within the range. For the TOPK and BOTTOMK queries we follow the same process as the original OSM—in our setting, the i-j search only works when looking at the top and bottom values for a key, as we know they are in the union of the top or bottom k values in each instance. Finally, for the ALLVALUES query, we simply return all values that pertain to the same key.

We can easily upper-bound the number of ORAM accesses needed to complete the RANGESEARCH query: each search (for the left-/right-most results) will take at most $(1.44 \log n_i)$ accesses (the worst-case height of an AVL tree), and assuming a result size of r, we will process at most r nodes between the two nodes. Similarly, we can easily bound the number of accesses needed for the other query types, and similar to [14], we achieve oblivious operations. We will discuss the security of this scheme further below.

The next two extensions (**B, C**) are used when the DYNODS instance is created as a simple map: (similar to the original OMap construction of Wang et al. [13]). In **(B)** we extend OSM's ODSINSERT to allow replacing the value of a key, if found; and **(C)** extend its ODSDELETE to support taking only a key as its argument and delete the single value of a key, if found.

We present the pseudocode for MOVEFROM, MOVETO, RANGESEARCH, and MERGERES (only for the case of RANGESEARCH, as different queries have different merge strategies) in Figure 5. The rest are omitted for brevity.

Security. Since MOVEFROM and MOVETO are implemented by normal OSM accesses, they can be simulated by SIMODSACCESS, which we leverage to prove the scheme secure. However, similar to the original OSM construction [14], for simplicity, we assume that all search results are padded to a constant size, and the updates are also padded as much, so that we achieve DYNODS-Type III security.

$(x, \sigma'; T') \leftrightarrow$ MoveFrom$(\sigma; T)$

1: $(k_r, v_r) \leftarrow$ read the root node's key and value
2: $((k_r, v_r), \sigma'; T') \leftrightarrow$ OdsDelete$((k_r, v_r), \sigma; T)$
3: **return** $((k_r, v_r), \sigma'; T')$

$(\sigma; T) \leftrightarrow$ MoveTo$(insertArgs, \sigma; T)$

1: $(\sigma', T') \leftrightarrow$ OdsInsert$(insertArgs, \sigma; T)$
2: **return** $(\sigma'; T')$

$(res, \sigma'; T') \leftrightarrow$ OdsSearch$(Args, \sigma; T)$

1: **if** $type(Args) =$ RangeSearch **then**
2: $l \leftarrow$ find leftmost node such that $\langle min.k.min.v \rangle \leq \langle l.k, l.v \rangle$
3: $r \leftarrow$ find rightmost node such that $\langle r.k, r.v \rangle \leq \langle max.k.max.v \rangle$
4: **else if** $type(Args) =$ TopK **then**
5: $(key, k) \leftarrow Args$
6: $l \leftarrow$ first node with $l.k = key$
7: $r \leftarrow k$-th node with $r.k = key$ ▷ last node, if less than k values
8: **else if** $type(Args) =$ BottomK **then**
9: $(key, k) \leftarrow Args$
10: $l \leftarrow k$-th last node with $l.k = key$ ▷ first node, if less than k values
11: $r \leftarrow$ last node with $r.k = key$
12: **else**
13: $(key) \leftarrow Args$ ▷ AllValues or simple map
14: $l \leftarrow$ first node with $l.k = key$
15: $r \leftarrow$ last node with $r.k = key$
16: $res \leftarrow \{l, r\}$
17: $s \leftarrow$ least common ancestor of l and r
18: $res \leftarrow res \cup BFS(s)$ ▷ only traversing entries between l and r
19: **return** res

$res \leftarrow$ MergeRes$(searchArgs, searchRes)$

1: $(x, y) \leftarrow searchResults$
2: **if** $type(searchArgs) =$ RangeSearch **then**
3: **return** $x \cup y$
4: **if** $type(searchArgs) =$ TopK **then**
5: $(key, k) \leftarrow searchArgs$
6: **return** first k values from $x \cup y$
7: **if** $type(searchArgs) =$ BottomK **then**
8: $(key, k) \leftarrow searchArgs$
9: **return** last k values from $x \cup y$
10: **return** $x \cup y$ ▷ AllValues or simple map

Figure 5: DynOds Oblivious Map Family. We leverage the oblivious AVL tree techniques of [13, 14] to offer multiple query types. See Section 5.2.

Otherwise, if the result sizes are leaked, the leakage function can be easily extended to include the size of the results. The extended leakage function has been already proposed in oblivious searchable encryption, e.g., in the "Horus" scheme of [16]. We discuss this further in Section 5.5.

Efficiency. The complexities of the DynOdsInsert, DynOdsUpdate, and DynOdsDelete protocols are all $O(\log^2 n_i)$ as they perform a constant number of basic AVL tree operations. For DynOdsSearch, the complexity varies per query type. Based on the original OSM scheme's search complexity, for the TopK and BottomK queries we have $O(\log^2 n_i + k \log n_i)$ complexity; and for the RangeSearch and AllValues queries, assuming r results, we have complexity $O(\log^2 n_i + r \log n_i)$.

Discussion. In the case of the simple map, for insertions, we assume that the key does not already exist in the map as this adds the possibility of having one value for the key in each ODS instance. But, in practice, this can be solved by a simple augmentation to the MoveFrom protocol: instead of returning the root of the AVL tree, we search for the key that is being inserted and return that entry if found; and the closest entry in the tree otherwise.

Also, we note that the simple map can be implemented more efficiently. To optimize the simple map, as we only allow one value per key, we can use the simple OMap scheme of Wang et al. [13] internally. This lowers the server storage usage as the child sub-tree counters are omitted from AVL tree nodes. Similarly, when an unsorted multi-map is needed, we can use a simple OMap in combination with an oblivious stack [13] on the same ORAM and thus remove the storage requirement of the counters; however, this would increase the cost of deleting one key-value pair as we would have to process the entire value list for the key in the client.

5.3. DynOHeap

We use Shi's Path Oblivious Heap scheme [15] to create the DynO-Heap scheme. When we mention ODS (DynOdsInsert, etc.) in this section, we mean the corresponding algorithms/protocols in Path Oblivious Heap. The protocols that define the scheme are as follows. Note that we are using the basic ODS interfaces directly, which allows us to use the scheme as a black-box.

- **MoveFrom** performs an OdsDelete (i.e., extracts the min value from the heap).
- **MoveTo** performs an OdsInsert.
- **FilterRes** and **MergeRes** return the min argument.

Security. Similar to Section 5.2, we directly achieve Type III obliviousness as we only use standard ODS interfaces for MoveFrom and MoveTo, which can be simulated by SimOdsAccess.

Efficiency. The complexities of the DynOdsInsert, DynOdsSearch, DynOdsUpdate, and DynOdsDelete are all $O(\log n_i)$ as they perform a constant number of basic Path Oblivious Heap operations.

5.4. Stacks and Queues

We create both of DynOStack and DynOQueue using DynOHeap. For DynOStack, we use the height of the entry on the stack as the heap key. We add a counter to the client state that specifies the next key to be inserted, and using a max heap, the construction is trivial—new entries are always inserted at the top, and deletions and searches always return

the newest undeleted entry. Similarly, for DynOQueue, we follow the same steps as DynOStack, except that we use a min heap—new entries are always inserted at the bottom, while deletions and searches always return the oldest undeleted entry. The security and the efficiency of both schemes ensue from DynOHeap directly.

Discussion 1. While our DynOStack and DynOQueue schemes are both practical, we can achieve DynODs-Type III security using any non-oblivious stack and queue using randomized encryption to hide values. In stacks and queues, given the order of the operations, each deletion or search can be directly mapped to exactly one insertion. Therefore, randomized encryption is enough to reduce the leakage to DynODs-Type III. This is not seen in maps and heaps as the order of the operations is not enough to compute the same mapping. Still, the DynOStack and DynOQueue schemes are useful because **(A)** they do not leak the memory locations of the entries, and **(B)** we cannot achieve DynODs-Type I and II security with non-oblivious constructions. We discuss the second point in Section 5.5.

Discussion 2. We note a challenge that rises in the case of DynOQueue: we have to fix the number of bits used for the heap key, but since we insert to the head but delete from the tail, we have to bound the number of insertions that the scheme supports. In practice, $\log N$ bits are enough to represent the heap key in a static heap of size N. We use the idea of ring buffers: we store an additional counter in the client state that contains the key of the tail of the queue, allow the head counter (that is used for insertions) to overflow, and augment the comparison function of the heap to take this into account. For example, in a queue with capacity 16, after a total of 18 insertions and 10 deletions, the queue will contain the following keys: $\{10, 11, 12, 13, 14, 15, 0, 1\}$; and keys 0-1 should be considered larger than keys 10-15.

5.5. DynODs Leakages

In this section we describe how we can elevate the security of our proposed DynODs schemes. We first discuss a specific case of elevating the security of a DynODs from Type III to Type II-DEL, and then extrapolate the argument to all other cases. The key observation is that all DynODs operations perform random (looking) ORAM accesses internally, and may or may not change the capacity of the DynODs. Finally, we will empirically evaluate the costs discussed below in Section 6.3.

DynODs-Type II-DEL from DynODs-Type III. To reach DynODs-Type II-DEL we need to make the "insert", "search", and "update" protocols indistinguishable. Between the three, "insert" has the highest cost and the largest footprint—not only we perform more calls to the static ODS instances to move entries, we may discard static ODS instances and allocate new ones. Thus, we have to move entries between the ODS instances and grow the DynODs in all three protocols. Furthermore, the OdsInsert, OdsSearch, and OdsUpdate calls should be indistinguishable, and we need to do a similar call to both ODS instances after moving entries. In the case of "insert", we make the actual insert call to the larger ODS instance and a dummy call to the smaller. In the case of "search" and "update", we make the OdsSearch and OdsUpdate calls to both ODS instances, and merge the results in the client.

Note that, similar to the discussion in the beginning of Section 4, after we reach N, we can stop moving entries and re-use the dummy-filled space for insertions.

Generalization. Similar to the discussion above, we can always make a group of DynODs operations indistinguishable by creating a super-operation that encompasses all of them, and for each of them, we do the ORAM accesses that are necessary properly (in a non-dummy manner), and perform dummy ORAM accesses for the rest. Essentially, the super-operation should **(A)** encompass all of the needed ORAM accesses in the DynODs operations, and **(B)** reach the highest capacity that can be resulted from the operations. The first point introduces a simple lower-bound on the number of ORAM accesses needed in all operations to make them oblivious. The second point, on the other hand, results in a lower-bound on the capacity of the DynODs, which can be achieved by moving enough entries around.

Hiding DynODs Type. The same approach as above can be used to hide the type of the ODS that is being used. Since DynODs schemes all use the same ORAM scheme internally, all ODS operations (regardless of the type) can be grouped and made indistinguishable, as discussed above.

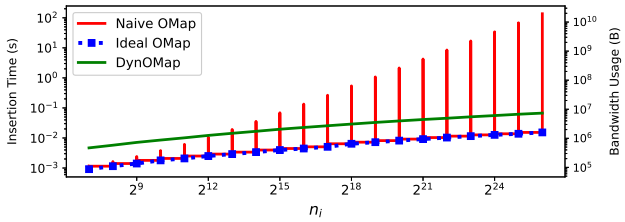


Figure 6: OMap Insertions.

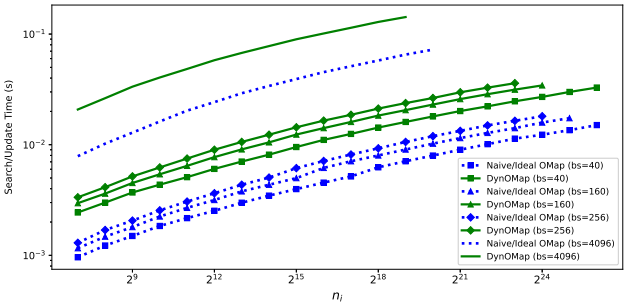


Figure 7: OMap Searches and Updates.

6. Experimental Evaluation

In this section, we evaluate the performance of DynO. We provide the evaluation of insertions, deletions, and searches (updates have the same performance as searches). We implemented the following static state-of-the-art ORAM and ODS schemes, initialized with n_i entries at time i, as baselines: ORAM of [12], OMap of [13, 14], and OHeap

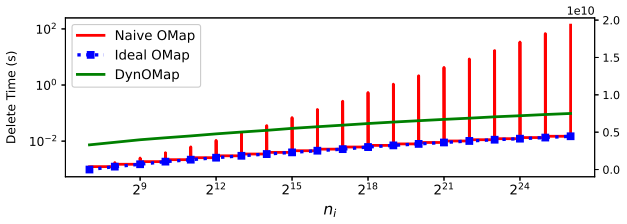


Figure 8: OMap Deletions.

of [15]. We call these "ideal" solutions, since a static ORAM/ODS of capacity n_i has the best performance for n_i entries (we highlight that these are artificial reference points and not competitors of DYNO). We also present a "naive" baseline in which we create an instance of the ORAM/ODS and rebuild it with double its size when it becomes full. Finally, except when discussing the cost of elevating the security, we run DYNO under Security Type III.

Setup. We implemented DYNO and the baselines, as well as the doubly-oblivious ORAM and OMap of Oblix [14] (using Intel SGX) in ~7,900 lines of C++ using OpenSSL [62] (with AES-NI enabled) for cryptographic operations. The doubly oblivious rewrite of Oblix can be of independent interest; Oblix's RAM was reported to have a throughput of 1,153 access/sec for $N = 2$M [1], whereas our rewrite achieves 1,555 access/sec in similar settings. The code will be published upon acceptance. We conducted our experiments on an AWS M6i.2xlarge instance with 8 third generation Intel Xeon cores (Ice Lake 8375C) and 32GB of RAM running Ubuntu 22.04 LTS, with an "st1" throughput-optimized EBS volume for the secondary storage tests. All experiments were instantiated on a single machine. Experiments were repeated $10\times$ and the average is reported. We used synthetic datasets of size $N = 2^8 - 2^{26}$ blocks with ORAM block sizes of 40B, 160B, 256B, and 4KiB (including metadata)—40B block size is often used in SE (see Table 2); 160B for Signal's Private Contact Discovery [14, 63]; 256B for private lookups in Google's Key Transparency [14, 64]; and 4KiB as the standard RAM block size.

6.1. DYNOMap

Insertions. Figure 6 shows the insertion time and bandwidth usage of an entry when varying the size of the database, n_i, with a block size of 40 bytes. As expected, we achieve a predictable insertion cost that scales with n_i, while the "naive" solution has spikes when the structure has to be rebuilt. This clearly shows why we need to avoid *"stop-the-world"* events where in order to resize, the user has to download everything and create a new data structure. DYNOMap only has a constant overhead of 4.4-$5.1\times$ over the ideal costs, where the lowest is seen for the highest n_i. This overhead stems from the multiple accesses to the sub-structures. Considering the bandwidth costs, we highlight that for $n_i = 2^{26}$, the naive solution requires to transfer 19GB while DYNOMap only 5MB.

Searches and Updates. Figure 7 shows the cost of searches and updates with different block sizes when varying n_i. We see that the block size has linear effect on access costs, as expected, and for different block sizes trends are the same. DYNOMap has a constant overhead of 2.2-$2.6\times$ over the ideal cost for search/updates.

Scheme	Time	Round Trips	Bandwidth
DYNORAM	0.87 ms	8	24.8 KB
Moataz et al. [56]	11.94 ms	434	941.4 KB

TABLE 1: Access cost comparison to Moataz et al. for $n_i = 2^{30}$ entries, each 40B (including metadata).

Deletions. Figure 8 shows the deletion time and bandwidth usage of an entry when varying n_i, with a block size of 40 bytes. We see that deletions and insertions follow the same patterns, including the spikes, albeit with different overheads—deletions have a 6.5-$7.8\times$ overhead compared to the ideal solution. The difference in overheads is due to the different number of sub-structure calls in the different DYNO operations—an insertion only performs three static OMap operations, whereas a deletion performs 6.

Discussion. We observe the same patterns across different operations and schemes (e.g., the insertions follow the same patterns in ORAMs, maps, heaps, stacks, and queues). For brevity, we only present the aforementioned plots for the OMap, which is the most expressive and has the highest complexity and therefore the highest costs; and present the rest of the DYNODSs in Appendix C.

For insertions and deletions, we see DYNO's highest advantage over the naive solutions when crossing size boundaries, as the naive solution needs to resize. Naturally, we see the lowest advantage in the remaining points when the naive solution has the same cost as the ideal solution (the constant overheads are mentioned above). In addition, we note that DYNO's overhead for crossing the size boundaries, i.e., when a new sub-structure is allocated and an old one is dropped, is negligible (less than 1%), as allocating a new instance is very cheap with LazyPathORAM. Therefore, even if we keep crossing the boundary in consecutive operations (insert-delete-insert-etc.) DYNO's overhead is negligible.

6.2. DYNORAM

As DYNOMap is more expressive than DYNORAM, we defer the basic evaluation of DYNORAM (similar to above) to the extended version (Appendix D.2).

Comparison to Moataz et al. Here, we present an empirical evaluation of DYNORAM and Moataz et al.'s work [56], which presents the only other resizable ORAM schemes, and has a slowdown of $\Omega(\log N)$ compared to DYNORAM as it uses the original Tree-ORAM scheme [20] internally (see Section 1). We empirically evaluate DYNORAM and the best scheme of Moataz et al.'s work (strategy 3 in [56]) on running time, number of round-trips, and bandwidth usage. Moataz et al. have variable ORAM node sizes based on multiple factors, but list concrete node sizes for $N = 2^{30}$. We use their analysis directly and focus on the same capacity. We assume an in-memory position map for both schemes, thus reducing the number of round-trips by $O(\log N)$. We simulate a generous lower-bound on the time it takes for

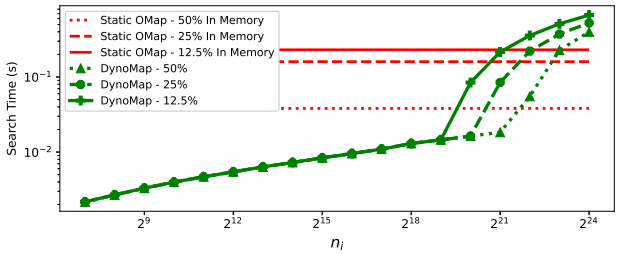


Figure 9: OMap Search on disk. Fixing $N = 2^{24}$ for the static OMap and varying the percentage of the entire server state of the static OMap that fits in memory.

Scheme	# ORAMs	Block Sizes	Total Size
OSSE [5]	2	49B, 37B	395GB
LLSE [5]	2	45B, 37B	378GB
SDD [17]	31	37B each	343GB
QOS [17]	2	37B each	343GB
Orion [16]	2	37B each	343GB
Horus [16]	2	37B, 16B	257GB

TABLE 2: The required OMap/ORAM initialization costs for $N = 2^{30}$ in multiple recent Searchable Encryption schemes, which can be avoided using DYNO.

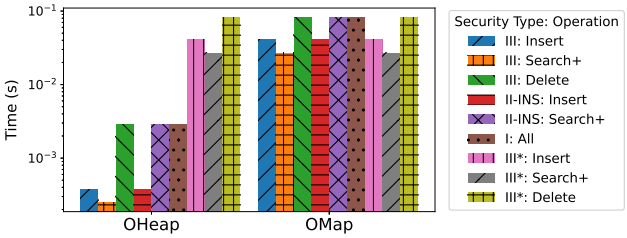


Figure 10: The cost of elevating the security level of DYNOHeap and DYNOMap. Type III* also hides ODS type.

an insertion in that scheme, but use exact measurements for DYNORAM. To calculate the lower-bound, we only account for the encryption and decryption costs of that scheme, and run both with a block size of 40 bytes including metadata. The results are in Table 1. Our scheme has an advantage of **10× in running time (excluding network round-trips), 54× in the number of round trips, and 37× in bandwidth usage**. In the above running times, (in favor of [56],) we exclude the network delays. DYNORAM is expected to be more efficient than [56], since it transfers, encrypts and decrypts $O(\log N)\times$ less blocks on each access.

6.3. Additional Experiments

Secondary Storage. Figure 9 shows the search/update cost of DYNOMap compared to a static OMap while varying n_i, when all of the data does not fit in memory. We assume $N = 2^{24}$ and look at cases where 50%, 25%, and 12.5% of the static OMap of size N can fit in memory, keeping as much of the OMaps in memory as possible. While the static OMap has a constant access cost, DYNOMap can avoid the disk and stay in memory while it has not outgrown it. As a result, especially for workloads where the database is not always at capacity, DYNO has a clear advantage as it can answer more queries with in-memory speeds.

Searchable Encryption (SE). In many SE schemes [5, 16, 17], oblivious RAMs and Maps are used to achieve the desired level of privacy. However, without dynamically resizable oblivious primitives, the client has to allocate and initialize the needed ORAMs (note that OMaps use ORAMs too) during setup, even with an empty initial database. As we show in Table 2, this cost is far from negligible. We consider multiple recent and state-of-the-art SE schemes and highlight the size of the ORAMs they create and initialize with dummy values, assuming $N = 2^{30}$. This ranges from 257GB (Horus [16]) to 395BGB (OSSE [5]). Using DYNO, these costs can be entirely avoided. This is especially interesting since in SE, *the nature of the operations is leaked anyway*, and using DYNO does not add any leakage.

Private Contact Discovery and Hardware Enclave Platforms. Similar to SE, in private contact discovery, since many users may join and leave the directory daily, the cost of stopping the database to rebuild it becomes prohibitive. Furthermore, the insertions/deletions need to be applied immediately, because, when a new user joins, they expect to be able to use the service immediately. We evaluated DYNOMap based on our implementation of Oblix's doubly-oblivious static OMap [65] against a naive solution built the same way, and observed a speedup of 2.4× for a write-heavy workload, inserting 16M entries.

Elevating DYNO Security. As described in Section 5.5, operation types and even ODS types can be hidden. Figure 10 shows the cost of doing operations of each security level in DYNOHeap and DYNOMap. The overhead of only hiding the operation is the similar in both cases. Type III operations are the cheapest, as they do not perform additional accesses. The accesses of type II are more expensive, as searches and updates need to reach the same cost as the more expensive insertions (Type II-INS) or deletions (Type II-DEL). Type I is even more expensive, as for all operations, the cost of the most expensive operation (i.e., deletion) has to be paid. Hiding the nature of the ODS is free in DYNOMap as it has the most number of accesses anyway. For DYNOHeap, however, the cost is substantial as it pays an additional $O(\log N)$ factor to match the number of ORAM accesses done by DYNOMap. Type III* denotes hiding the ODS type while achieving Type III security.

7. Conclusion

In this work we present the importance of dynamically resizable Oblivious RAMs (ORAMs) and Oblivious Data Structures (ODSs). We provide the necessary security definitions from most secure (i.e., the standard ORAM/ODS definitions) to least secure (i.e., leaking all operation types) and present the first set of efficient dynamically resizable ORAM and ODS constructions that achieve asymptotically optimal access and storage costs proportional to the total number of inserted (and non-deleted) entries at any time.

References

[1] E. Dauterman, V. Fang, I. Demertzis, N. Crooks, and R. A. Popa, "Snoopy: Surpassing the scalability bottleneck of oblivious storage," in *SOSP*, 2021.

[2] A. Deshpande, "Sypse: Privacy-first data management through pseudonymization and partitioning.," in *CIDR*, 2021.

[3] J. Bater, X. He, W. Ehrich, A. Machanavajjhala, and J. Rogers, "Shrinkwrap: efficient sql query processing in differentially private data federations," *PVLDB*, 2018.

[4] C. Wang, J. Bater, K. Nayak, and A. Machanavajjhala, "Dp-sync: Hiding update patterns in secure outsourced databases with differential privacy," in *SIGMOD*, 2021.

[5] J. G. Chamani, D. Papadopoulos, M. Karbasforushan, and I. Demertzis, "Dynamic searchable encryption with optimal search in the presence of deletions," *USENIX*, 2022.

[6] S. Oya and F. Kerschbaum, "Ihop: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization," in *USENIX*, 2022.

[7] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *CCS*, 2015.

[8] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks," *NDSS*, 2020.

[9] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range series," in *CCS*, 2018.

[10] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *STOC*, 1987.

[11] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, 1996.

[12] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path Oram: An Extremely Simple Oblivious Ram Protocol," in *CCS*, 2013.

[13] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *CCS*, 2014.

[14] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *IEEE SP*, 2018.

[15] E. Shi, "Path oblivious heap: Optimal and practical oblivious priority queue," in *IEEE SP*, 2020.

[16] J. Ghareh Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *CCS*, 2018.

[17] I. Demertzis, J. Ghareh Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," in *NDSS*, 2020.

[18] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *CCS*, 2015.

[19] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious ram," in *USENIX*, 2015.

[20] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o ((logn) 3) worst-case cost," in *AsiaCrypt*, 2011.

[21] G. Asharov, T.-H. Hubert Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Locality-preserving oblivious ram," in *CRYPTO*, 2019.

[22] A. Chakraborti, A. J. Aviv, S. G. Choi, T. Mayberry, D. S. Roche, and R. Sion, "roram: Efficient range oram with o (log2 n) locality.," in *NDSS*, 2019.

[23] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform.," in *NSDI*, 2017.

[24] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "Seal: Attack mitigation for encrypted databases via adjustable leakage," in *USENIX*, 2020.

[25] S. Krastnikov, F. Kerschbaum, and D. Stebila, "Efficient oblivious database joins," *PVLDB*, 2020.

[26] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward," in *CCS*, 2015.

[27] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *STOC*, 2012.

[28] C. W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, and S. Devadas, "Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram," in *ASPLOS*, 2015.

[29] C. Gentry, S. Halevi, C. Jutla, and M. Raykova, "Private database access with he-over-oram architecture," in *ACNS*, 2015.

[30] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *IEEE SP*, 2015.

[31] S. Lu and R. Ostrovsky, "Distributed oblivious ram for secure two-party computation," in *TCC*, 2013.

[32] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "Phantom: Practical oblivious computation in a secure processor," in *CCS*, 2013.

[33] R. Ostrovsky and V. Shoup, "Private information storage," in *STOC*, 1997.

[34] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *ISCA*, 2013.

[35] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," in *IEEE SP*, 2013.

[36] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," in *NDSS*, 2012.

[37] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi, "Scoram: oblivious ram for secure computation," in *CCS*, 2014.

[38] P. Williams, R. Sion, and A. Tomescu, "Privatefs: A parallel oblivious file system," in *CCS*, 2012.

[39] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square-root oram: efficient random access in multi-party computation," in *IEEE SP*, 2016.
[40] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "Optorama: Optimal oblivious ram," in *EUROCRYPT*, 2020.
[41] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, "Taostore: Overcoming asynchronicity in oblivious data storage," in *IEEE SP*, 2016.
[42] Z. Chang, D. Xie, S. Wang, and F. Li, "Towards practical oblivious join," in *SIGMOD*, 2022.
[43] S. Eskandarian and M. Zaharia, "Oblidb: Oblivious query processing for secure databases," *PVLDB*, 2019.
[44] M. Vuppalapati, K. Babel, A. Khandelwal, and R. Agarwal, "Shortstack: Distributed, fault-tolerant, oblivious data access," in *OSDI*, 2022.
[45] Z. Chang, D. Xie, and F. Li, "Oblivious ram: A dissection and experimental evaluation," *PVLDB*, 2016.
[46] A. Chakraborti and R. Sion, "Concuroram: High-throughput stateless parallel multi-client oram," *NDSS*, 2019.
[47] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi, "Obladi: Oblivious serializable transactions in the cloud," in *OSDI*, 2018.
[48] E. Boyle, K.-M. Chung, and R. Pass, "Oblivious parallel ram and applications," in *TCC*, 2016.
[49] T.-H. H. Chan, K.-M. Chung, and E. Shi, "On the depth of oblivious parallel ram," in *ASIACRYPT*, 2017.
[50] T.-H. H. Chan, Y. Guo, W.-K. Lin, and E. Shi, "Oblivious hashing revisited, and applications to asymptotically efficient oram and opram," in *ASIACRYPT*, 2017.
[51] T.-H. Hubert Chan and E. Shi, "Circuit opram: Unifying statistically and computationally secure orams and oprams," in *TCC*, 2017.
[52] B. Chen, H. Lin, and S. Tessaro, "Oblivious parallel ram: improved efficiency and generic constructions," in *TCC*, 2016.
[53] K. Nayak and J. Katz, "An oblivious parallel ram with $O(\log^2 N)$ parallel runtime blowup," *ePrint*, 2016.
[54] T.-H. H. Chan, K. Nayak, and E. Shi, "Perfectly secure oblivious parallel ram," in *TCC*, 2018.
[55] G. Asharov, I. Komargodski, W.-K. Lin, E. Peserico, and E. Shi, "Optimal oblivious parallel ram," in *SODA*, 2022.
[56] T. Moataz, T. Mayberry, E.-O. Blass, and A. H. Chan, "Resizable tree-based oblivious ram," in *FC*, 2015.
[57] "Dyno: Dynamic oblivious primitives with resizable memory - extended version." https://github.com/ExtVersionOfDyno/Dyno-Extended, 2023.
[58] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC press, 2020.
[59] S. Kamara and T. Moataz, "Encrypted multi-maps with computationally-secure leakage.," *ePrint*, 2018.
[60] S. Kamara and T. Moataz, "Computationally volume-hiding structured encryption," in *CRYPTO*, 2019.
[61] S. Patel, G. Persiano, K. Yeo, and M. Yung, "Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing," in *CCS*, 2019.
[62] "OpenSSL: The open source toolkit for SSL/TLS." https://www.openssl.org/, 2021. Version 3.0.
[63] M. Marlinspike, "Technology preview: Private contact discovery for signal."
[64] R. Hurst and G. Belvin, "Google's key transparency."
[65] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *Oblix: An Efficient Oblivious Search Index*, SP, 2018.

Appendix A. LazyPathORAM

In this section we present the full LazyPathORAM construction. In the extended version (Appendix D) we present the ORAM security definition, present our dynamic ORAM construction (DynORam), prove it secure with respect to the security definition, and show its empirical evaluation.

We present the pseudocode of the LazyPathORAM scheme in Figure 11. The difference with the original PathORAM [12] is that we store two extra bits in each ORAM tree node to specify whether each of its children have been initialized. We also store one bit in the client to track whether the root has been initialized. During an access, when reading the path, we read the nodes from the root down, and stop reading when the rest of the path is uninitialized; and when writing the path back, we write to all nodes in the path, initializing the ones that have not been previously initialized.

Appendix B. DynOds Security

We prove the security of the DynOds schemes in the real/ideal game of Definition 1 (Figure 12). We use static ODS instances as our building blocks. For a static ODS scheme, we have a polynomial-time simulator Sim_{ODS} = (SimOdsSetup, SimOdsAccess) that follows Definition 1 (ODS security). This makes the task of creating the DynOds simulator Sim = (SimDynOdsSetup, SimDynOdsAccess) straightforward; we show that given the leakage function $\mathcal{L}$ (see Section 4), we can use Sim_{ODS} to build Sim and simulate the DynOds framework.

Below, we show how we create the SimDynOdsSetup, SimDynOdsInsert, SimDynOdsDelete, and SimDynOdsSearchUpdate simulators to achieve DynOds-Type III security. The simulator is shown in Figure 13.

For **SimDynOdsSetup**, the simulator has to output a T with the same size and format as the real algorithm (Algorithm SimDynOdsSetup, Figure 13).

For **SimDynOdsInsert**, we can simulate the ODS accesses just by knowing the size of the DynOds instance. We know the size based on the sequence of operations, which is kept in the simulator state. We replace the OdsSetup calls in lines 3 and 6 of DynOdsInsert (Figure 2) with calls to

$(\sigma; R) \leftrightarrow$ OramSetup$(1^\lambda, N)$

1: $R \leftarrow$ allocate space for $4 \times (2N-1)$ blocks on server
2: $S \leftarrow$ empty stash; $pos \leftarrow$ empty array of size N
3: $isRootValid \leftarrow false$
4: $\sigma \leftarrow (S, pos, isRootValid)$
5: **return** $(\sigma; R)$

$(res, \sigma'; R') \leftrightarrow$ OramAccess$(op, i, data, \sigma; R)$

1: $(S, pos, isRootValid) \leftarrow \sigma$
2: $valid \leftarrow$ empty set
3: **if** $isRootValid$ **then** Add 0 to $valid$
4: $x \leftarrow pos[i]$
5: **if** $x = \perp$ **then**
6: $\quad x \leftarrow$ UniformRandom$(0...N-1)$
7: $pos[i] \leftarrow$ UniformRandom$(0...N-1)$
8: **for** $l = 0, \dots L$ **do**
9: $\quad (B, valid; R) \leftrightarrow$ ReadBucketLazy$(P(x,l), valid; R)$
10: $\quad S \leftarrow S \cup B$
11: $res \leftarrow$ read block i from S
12: **if** $op = write$ **then**
13: $\quad S \leftarrow S \backslash \{(i, res)\} \bigcup \{(i, data)\}$
14: **for** $l = L, \dots 0$ **do**
15: $\quad B \leftarrow \{(i', data') \in S : P(x,l) = P(pos[i'], l)\}$
16: $\quad$ Limit B to at most Z blocks
17: $\quad S \leftarrow S \backslash B$
18: $\quad (valid; R) \leftarrow$ WriteBucketLazy$(P(x,l), B, valid; R)$
19: $\sigma \leftarrow (\sigma, pos, isRootValid)$
20: **return** $(res, \sigma; R)$

$(B, valid'; R) \leftrightarrow$ ReadBucketLazy$(id, valid; R)$

1: **if** $id \notin valid$ **then**
2: $\quad$ **return** $(\perp, valid; R)$
3: $(B; R) \leftrightarrow$ ReadBucket$(id; R)$
4: **if** $B.LChildValid$ **then**
5: $\quad valid \leftarrow valid \bigcup \{((2 \cdot id) + 1)\}$
6: **if** $B.RChildValid$ **then**
7: $\quad valid \leftarrow valid \bigcup \{((2 \cdot id) + 2)\}$
8: **return** $(B, valid; R)$

$(valid'; R) \leftrightarrow$ WriteBucketLazy$(id, B, valid; R)$

1: **if** $((2 \cdot id) + 1) \in valid$ **then**
2: $\quad B.LChildValid = true$
3: **if** $((2 \cdot id) + 2) \in valid$ **then**
4: $\quad B.RChildValid = true$
5: $valid \leftrightarrow valid \bigcup \{id\}$
6: $(\perp; R) \leftrightarrow$ WriteBucket$(id, B; R)$
7: **return** $(valid; R)$

Figure 11: LazyPathORAM.

$bit \leftarrow$ **Real**$^{ODS}_{\mathcal{A},\Pi}(\lambda)$:

1: $M_0 \leftarrow \mathcal{A}(1^\lambda)$
2: $(\sigma_0, EM_0) \leftarrow \Pi$.OdsSetup$(1^\lambda, M_0)$
3: **for** $k = 1$ to q **do**
4: $\quad (op_k, i_k, v_k) \leftarrow \mathcal{A}(1^\lambda, EM_0, \dots EM_{k-1}, m_1, \dots, m_{k-1})$
5: $\quad (v_{i_k}, \sigma_k; EM_k) \leftrightarrow \Pi$.OdsAccess$(op_k, i_k, v_k, \sigma_{k-1}; EM_{k-1})$
6: $bit \leftarrow \mathcal{A}(1^\lambda, EM_0, \dots, EM_k, m_1, \dots, m_k)$
7: **return** bit

$bit \leftarrow$ **Ideal**$^{ODS}_{\mathcal{A},\mathsf{Sim},\mathcal{L}}(\lambda)$:

1: $M_0 \leftarrow \mathcal{A}(1^\lambda)$
2: $(st_{\mathcal{S}}, EM_0) \leftarrow$ SimOdsSetup$(1^\lambda, |M_0|)$
3: **for** $k = 1$ to q **do**
4: $\quad (op_k, i_k, v_k) \leftarrow \mathcal{A}(1^\lambda, EM_0, \dots EM_{k-1}, m_1, \dots, m_{k-1})$
5: $\quad (v_{i_k}, st_{\mathcal{S}}; EM_k) \leftrightarrow$ SimOdsAccess$(\mathcal{L}(op_k), st_{\mathcal{S}}; EM_{k-1})$
6: $bit \leftarrow \mathcal{A}(1^\lambda, EM_0, \dots, EM_k, m_1, \dots, m_k)$
7: **return** bit

$EM_0, \dots EM_k$ include the server state and the memory accesses to it. $m_1, \dots m_k$ are the messages exchanged in OdsAccess and SimOdsAccess.

Figure 12: Real and ideal experiments for ODS schemes.

SimOdsSetup, and replace the OdsInsert call of line 8 with a SimOdsAccess. Also, we replace the MoveFrom and MoveTo calls with calls to SimDynOdsMoveFrom and SimDynOdsMoveTo.

For **SimDynOdsDelete**, again, we can simulate the accesses just by knowing the size, which is computable from the simulator state. Again, we replace every OdsDelete and OdsInsert call with a call to SimOdsAccess, and replace the calls to MoveFrom and MoveTo with calls to SimMoveFrom and SimMoveTo. We emphasize that every possible data-dependent execution path of DynOdsDelete has the same sequence of accesses to its ODS instances, and we simulate all by simulating one. Finally, the FilterRes algorithm is done in the client, and therefore doesn't need any simulation.

For **SimDynOdsSearchUpdate**, we first note that DynOdsSe-arch and DynOdsUpdate are indistinguishable. The two algorithms only access the ODS instances in lines 3-5 of DynOdsSearch and lines 2-4 of DynOdsUpdate in Figure 4, which are indistinguishable. We simply replace the OdsSearch and OdsUpdate calls with SimOdsAccess. Finally, we stress that the MergeRes call in line 9 of the DynOdsSearch protocol is done in the client and therefore isn't simulated. □

Appendix C. Additional Experiments

DynOram We evaluate DynOram similar to Section 6.1. Figure 14 shows the running time and bandwidth usage of

$(st_s; T) \leftarrow$ SIMDYNODSSETUP(1^λ)

$ops \leftarrow$ an empty vector
ist_1, ist_2 to be empty static ODS simulator states
$T_1, T_2 \leftarrow$ empty simulated static ODS instances
$st_s \leftarrow (ops, ist_1, ist_2)$, $T \leftarrow (T_1, T_2)$
return (st_s, T)

$(st_s; T) \leftarrow$ SIMDYNODSINSERT$(st_s; T)$

$(ops, ist_1, ist_2) \leftarrow st_s; (T_1, T_2) \leftarrow T$
$size = count_{insert}(ops) - count_{delete}(ops)$
Append an *insert* to *ops*
if $size = 0$ **then**
  $(ist_2, T_2) \leftarrow$ SIMODSSETUP$(1^\lambda, 1)$
else if *size* is a power of 2 **then**
  $ist_1 \leftarrow ist_2; T_1 \leftarrow T_2$
  $(ist_2, T_2) \leftarrow$ SIMODSSETUP$(1^\lambda, 2 \times size)$
$(st_s; T) \leftarrow$ SIMMOVE$(2, st_s; T)$
▷ first argument is destination
$(ist_2, T_2) \leftarrow$ SIMODSACCESS(ist_2, T_2)
$st_s \leftarrow (ops, ist_1, ist_2); T \leftarrow (T_1, T_2)$
return (st_s, T)

$(st_s; T) \leftarrow$ SIMDYNODSDELETE$(st_s; T)$

$(ops, ist_1, ist_2) \leftarrow st_s; (T_1, T_2) \leftarrow T$
Append a *delete* to *ops*
$size = count_{insert}(ops) - count_{delete}(ops)$
for $i \in 1..2$ **do**
  if $T_i \neq \emptyset$ **then**
    $(ist_i; T_i) \leftarrow$ SIMODSACCESS$(ist_i; T_i)$
for $i \in 1..2$ **do**
  $(st_s; T) \leftarrow$ SIMMOVE$(1, st_s; T)$
if *size* is a power of 2 **then**
  $ist_2 \leftarrow ist_1; T_2 \leftarrow T_1$
  $(ist_1; T_1) \leftarrow$ SIMODSSETUP$(1^\lambda, size/2)$
$st_s \leftarrow (ops, ist_1, ist_2); T \leftarrow (T_1, T_2)$
return (st_s, T)

$(\sigma, T) \leftarrow$ SIMMOVE$(d, \sigma; T)$

$(ops, ist_1, ist_2) \leftarrow st_s; (T_1, T_2) \leftarrow T$
$s \leftarrow \{1, 2\} \backslash \{d\}$ ▷ s: source; d: destination
$(ist_s; T_s) \leftarrow$ SIMDYNODSMOVEFROM$(ist_s; T_s)$
$(ist_d; T_d) \leftarrow$ SIMDYNODSMOVETO$(ist_d; T_d)$
$st_s \leftarrow (ops, ist_1, ist_2); T \leftarrow (T_1, T_2)$
return (st_s, T)

$(\sigma, T) \leftarrow$ SIMDYNODSSEARCHUPDATE$(\sigma; T)$

$(ops, ist_1, ist_2) \leftarrow st_s; (T_1, T_2) \leftarrow T$
for $i \in 1..2$ **do**
  if $T_i \neq \emptyset$ **then**
    $(ist_i; T_i) \leftarrow$ SIMODSACCESS$(ist_i; T_i)$
$st_s \leftarrow (ops, ist_1, ist_2); T \leftarrow (T_1, T_2)$
return (st_s, T)

Figure 13: DYNODS Simulator.

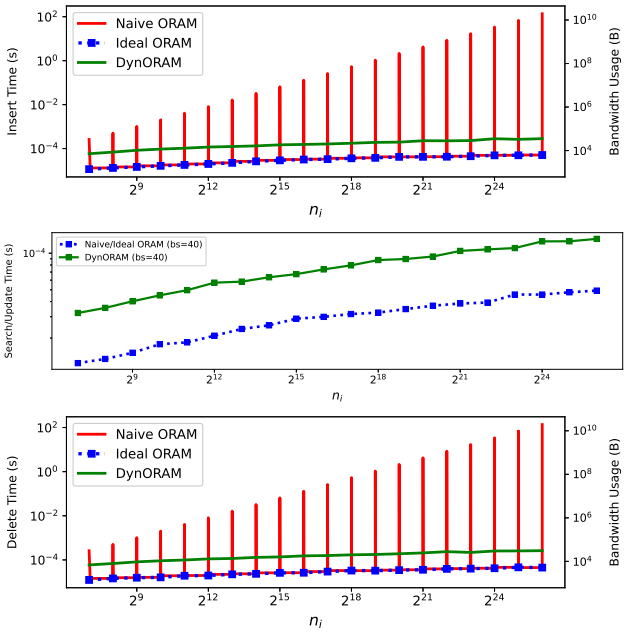


Figure 14: ORAM Insertions, Searches/Updates, Deletions.

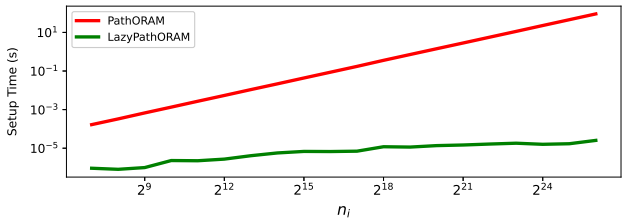


Figure 15: ORAMSETUPin LazyPathORAM and Normal PathORAM.

insertions and deletions, and the running time of searches and updates in DYNORAM, compared to the ideal and the naive solutions. In all three figures, we vary the size of the database n_i with a block size of 40B. We see the same patterns as for DYNOMap; in insertions, searches, and deletions, DYNORAM has an overhead of 4.7-5.9×, 2.3-2.7×, and 4.7-6.1×, respectively, over the ideal ORAM (i.e., [12]). We highlight that, contrary to DYNOMap, insertions and deletions have similar overheads because DYNORAM does not need an extra pair of ORAMACCESSs to find the entry to be deleted (see Figure 18). Finally, Figure 15 shows the setup cost of LazyPathORAM compared to the non-lazy variant. As expected, the setup times are significantly different as LazyPathORAM does not need to initialize any ORAM nodes, and only reserves space.

DYNOHeap, DYNOStack, and DYNOQueue We also present the experimental evaluation of the DYNOHeap, DYNOStack, and DYNOQueue. We only show one set of plots for all of them as they have the same complexities and performances. We show the same three plots (similar to DYNOMap and DYNORAM) in Figure 16. We highlight that

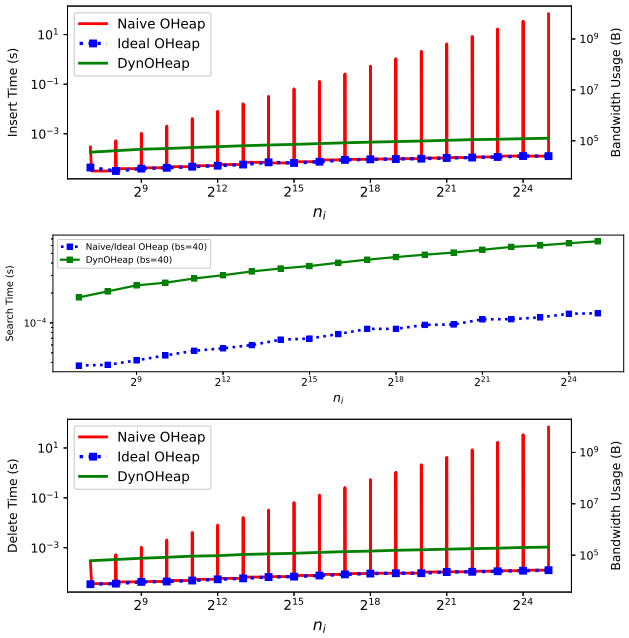


Figure 16: OHeap, OStack, and OQueue—Insertions, Searches/Updates, Deletions.

the patterns are similar in all schemes. Again, in insertions, searches, and deletions, the three schemes have an overhead of 4.1-6.6×, 4.8-5.7×, and 8.0-9.5×, respectively, over their ideal counterparts. We highlight that similar to DynOMap, DynO has a higher overhead for deletions compared to insertions, as deletions require two more accesses to determine the entry to be deleted.

Extended Version

Appendix D. Oblivious RAM (ORAM)

In this section we present the ORAM security definition. We then present our dynamic ORAM scheme (DYNORAM), prove it secure with respect to the security definition, and show its empirical evaluation.

D.1. ORAM Security Definition

We use a single security definition for both static and dynamic ORAMs. Similar to DYNODS security (Section 4), we define the security of dynamic ORAMs as a set of four types, where the most secure (Type-I) is equivalent to the classic ORAM security definition. We define the security of an ORAM in the real/ideal game which, for static ORAMs, is presented in Section 2. We start by presenting the security types.

Definition 3. *An $\mathcal{L}$-adaptively-secure ORAM is*

- **DYNORAM-*Type I:*** *iff $\mathcal{L}(op) = \perp$. It is equivalent with the standard ORAM security definition in Section 2.*
- **DYNORAM-*Type II-INS:*** *iff $\mathcal{L}(op) = \{$"insert" or "search/up-date/delete"$\}$. It leaks two different operations: whether the operation is "insert" or "search/update/ delete" ("search", "update", and "delete" operations remain indistinguishable).*
- **DYNORAM-*Type II-DEL:*** *iff $\mathcal{L}(op) = \{$"delete" or "search/update/insert"$\}$. It leaks two different operations: whether the operation is "delete" or "search/update/insert" ("search", "update", and "insert" operations remain indistinguishable).*
- **DYNORAM-*Type III:*** *iff $\mathcal{L}(op) = \{$"insert" or "delete" or "search/ update"$\}$. It leaks three different operations: whether the operation is "insert", "delete", or "search/update" ("search" and "update" operations remain indistinguishable).*
- **DYNORAM-*Type IV:*** *iff $\mathcal{L}(op) = \{$"insert" or "delete" or "search" or "update"$\}$. It leaks all of the four operation types.*

The leakage function $\mathcal{L}$ will allow us to leak controlled information **only** about the type of the operations. We state the security definition for dynamic ORAMs next, which is a super-set of the static variant. A DYNORAM = (DYNORAMSETUP, DYNORAMINSERT, DYNORAMSEARCH, DYNORAMUPDATE, DYNORAMDELETE) is $\mathcal{L}$-secure *iff* there exists a polynomial-time simulator Sim= (SIMDYNORAMSETUP, SIMDYNORAMACCESS), such that for any polynomial-length sequence of operations chosen by the adversary, SIMDYNORAMSETUP is indistinguishable from DYNORAMSETUP, and SIMDYNORAMACCESS is indistinguishable from DYNORAMINSERT, DYNORAMDELETE, DYNORAMSEARCH, DYNORAMUPDATE. The difference compared with the static definition in Section 2 is that here, SIMDYNORAMACCESS takes the output of the newly defined leakage function $\mathcal{L}$ as input, whereas the leakage is fixed (to Type-I) in the other version.

$bit \leftarrow \mathbf{Real}^{ORAM}_{\mathcal{A},\Pi}(\lambda)$:

1: $M_0 \leftarrow \mathcal{A}(1^\lambda)$
2: $(\sigma_0, EM_0) \leftarrow \Pi.\text{ORAMSETUP}(1^\lambda, M_0)$
3: **for** $k = 1$ to q **do**
4: $\quad (op_k, i_k, v_k) \leftarrow \mathcal{A}(1^\lambda, EM_0 \dots EM_{k-1}, m_1, \dots, m_{k-1})$
5: $\quad (v_{i_k}, \sigma_k; EM_k) \leftrightarrow \Pi.\text{ORAMACCESS}(op_k, i_k, v_k, \sigma_{k-1}; EM_{k-1})$
6: $bit \leftarrow \mathcal{A}(1^\lambda, EM_0, \dots, EM_k, m_1, \dots, m_k)$
7: **return** bit

$bit \leftarrow \mathbf{Ideal}^{ORAM}_{\mathcal{A},\text{Sim},\mathcal{L}}(\lambda)$:

1: $M_0 \leftarrow \mathcal{A}(1^\lambda)$
2: $(st_{\mathcal{S}}, EM_0) \leftarrow \text{SIMORAMSETUP}(1^\lambda, |M_0|)$
3: **for** $k = 1$ to q **do**
4: $\quad (op_k, i_k, v_k) \leftarrow \mathcal{A}(1^\lambda, EM_0, \dots EM_{k-1}, m_1, \dots, m_{k-1})$
5: $\quad (v_{i_k}, st_{\mathcal{S}}; EM_k) \leftrightarrow \text{SIMORAMACCESS}(\mathcal{L}(op_k), st_{\mathcal{S}}; EM_{k-1})$
6: $bit \leftarrow \mathcal{A}(1^\lambda, EM_0, \dots, EM_k, m_1, \dots, m_k)$
7: **return** bit

$EM_0, \dots EM_k$ include the server state and the memory accesses to it. $m_1, \dots m_k$ are the messages exchanged in ORAMACCESS and SIMORAMACCESS.

Figure 17: Real and ideal experiments for ORAM schemes.

Finally, we present the formal security definition for both static and dynamic ORAMs.

Definition 4. *An ORAM scheme Π is $\mathcal{L}$-secure if for any PPT adversary $\mathcal{A}$, there exists a stateful PPT simulator Sim = (SIMORAMSETUP, SIMORAMACCESS) such that:*

$$|\Pr[\mathbf{Real}^{\text{ORAM}}_{\mathcal{A},\Pi}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\text{ORAM}}_{\mathcal{A},Sim,\mathcal{L}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

where experiments $\mathbf{Real}^{\text{ORAM}}_{\mathcal{A},\Pi}(\lambda)$ and $\mathbf{Ideal}^{\text{ORAM}}_{\mathcal{A},Sim,\mathcal{L}}(\lambda)$ are defined in Figure 17. The randomness is taken over the random bits used by the algorithms of the ORAM scheme, the algorithms of the simulator, and $\mathcal{A}$. For the standard static ORAM definition, $\mathcal{L}(op) = \perp$ (Type-I).

D.2. DYNORAM

Definition. Similar to the DYNODS definition in Section 4, we provide an abstract interface for dynamic ORAM, and then provide the scheme. A DYNORAM scheme consists of five algorithms/protocols—DYNORAM = (DYNORAMSETUP, DYNORAMINSERT, DYNORAMDELETE, DYNORAMSEARCH, DYNORAMUPDATE):

- $(\sigma; R) \leftarrow$ DYNORAMSETUP(1^λ): Given a security parameter λ, it outputs the secret state σ for the client and an instance of the ORAM R for the server.

- $(\sigma'; R') \leftarrow$ DYNORAMINSERT$(id, val, \sigma; R)$: Inserts a new entry at index id with value val.
- $(res, \sigma'; R') \leftarrow$ DYNORAMDELETE$(id, \sigma; R)$: Deletes the entry at index id and returns its value.
- $(res, \sigma'; R') \leftarrow$ DYNORAMSEARCH$(id, \sigma; R)$: Returns the entry at index id.
- $(\sigma'; R') \leftarrow$ DYNORAMUPDATE$(id, val, \sigma; R)$: Updates the value of the entry at index id to val.

DYNORAMINSERT, DYNORAMDELETE, DYNORAMSEARCH, and DYNORAMUPDATE also take the client secret state σ and server state R as input, and return the updated client and server states σ' and R'.

Construction. Our DYNORAM scheme is a four tuple—DYNORAM = (DYNORAMSETUP, DYNORAMINSERT, DYNORAMDELETE, DYNORAMSEARCHUPDATE). Note that we merge the search and update operations into one protocol. It is built similarly to the DYNODS schemes. We maintain two static LazyPathORAM instances with sizes 2^k and 2^{k+1}; move a block from the smaller to the larger during insertions; and deletions mirror insertions. Here, the insertions increase the capacity of the ORAM by increasing the range of the indexes it contains by one—at size s, the client can only insert to index $s+1$. In this scheme, we move entries between the static instances deterministically—an iterator is maintained on an index in a static instance, and that entry is moved to the other static instance when necessary. The iterator of the smaller ORAM starts at the first entry and advances forward, and the iterator of the larger instance mirrors this procedure.

Similar to DYNODS , when the larger static ORAM is full and the iterator of the smaller instance has passed the last entry in it (these happen simultaneously), we discard the smaller ORAM, replace it with the larger one, and create a new larger ORAM with double the capacity of the now-smaller ORAM. Finally, given the deterministic nature of the iterators, each block that is within the range of the ORAM is exactly in one of the ORAMs. As such, "read" and "update" operations to that block go to the static ORAM that holds it, while the other ORAM performs a dummy operation. Figures 18 and 19 show the scheme in pseudocode.

Security. Informally, DYNORAM is DYNORAM-Type-III secure. Below, we formalize this.

Theorem 3. *Assuming that the used static ORAM is a secure oblivious RAM according to Definition 4 then the proposed* DYNORAM *is secure according to Definition 3 Type III.*

Proof. We prove this theorem similar to the security proof of the DYNODS schemes in Appendix B. We create a simulator for the DYNORAM scheme using the available simulator of ORAM—sim_{ORAM} = (SIMORAMSETUP, SIMORAMACCESS)— by replacing the setup and access calls to the static ORAM with calls to its simulator. We show that given the leakage function $\mathcal{L}$, we can use sim_{ORAM} to simulate DYNORAM.

Below, we show how we create the DYNORAM simulator Sim= (SIMDYNORAMSETUP, SIMDYNORAMINSERT, SIMDYNORAMDELETE, SIMDYNORAMSEARCHUPDATE)

$(\sigma; R) \leftarrow$ DYNORAMSETUP(1^λ)

$\sigma_1, \sigma_2 \leftarrow$ empty LazyPathORAM client states
$R_1, R_2 \leftarrow$ empty LazyPathORAM server states
$size \leftarrow 0$
$\sigma \leftarrow (\sigma_1, \sigma_2, size); R \leftarrow (R_1, R_2)$
return (σ, R)

$(\sigma'; R') \leftrightarrow$ DYNORAMINSERT$(id, val, \sigma; R)$

$(\sigma_1, \sigma_2, size) \leftarrow \sigma; (R_1, R_2) \leftarrow R$
if $size = 0$ **then**
  $(\sigma_2; R_2) \leftarrow$ ORAMSETUP$(1^\lambda, 1)$
else
  if $size$ is a power of 2 **then**
    $\sigma_1, R_1 \leftarrow \sigma_2, R_2$
    $(\sigma_2; R_2) \leftarrow$ ORAMSETUP$(1^\lambda, 2 \times size, \emptyset)$
  $mId \leftarrow size \mod |R_1|$ ▷ index to move
  $(mov, \sigma_1; R_1) \leftrightarrow$ ORAMACCESS$(read, mId, \perp, \sigma_1; R_1)$
  $(\sigma_2; R_2) \leftrightarrow$ ORAMACCESS$(write, mId, mov, \sigma_2; R_2)$ ▷ entry moved
$size \leftarrow size + 1$
$(\sigma_2; R_2) \leftrightarrow$ ORAMACCESS$(write, size, val, \sigma_2; R_2)$
$\sigma' \leftarrow (\sigma_1, \sigma_2, size); R' \leftarrow (R_1, R_2)$
return $(\sigma'; R')$

$(res, \sigma'; R') \leftrightarrow$ DYNORAMDELETE$(id, \sigma; R)$

$(\sigma_1, \sigma_2, size) \leftarrow \sigma; (R_1, R_2) \leftarrow R$
$(res, \sigma_2, R_2) \leftrightarrow$ ORAMACCESS$(read, size, \perp, \sigma_2, R_2)$
$size \leftarrow size - 1$
$mId \leftarrow size \mod |R_1|$ ▷ index to move
$(mov, \sigma_2, R_2) \leftrightarrow$ ORAMACCESS$(read, mId, \perp, \sigma_2, R_2)$
$(\sigma_1, R_1) \leftarrow$ ORAMACCESS$(write, mId, mov, \sigma_1, R_1)$ ▷ entry moved
if $size$ is a power of 2 **then**
  $\sigma_2, R_2 \leftarrow \sigma_1, R_1$
  $(\sigma_1, R_1) \leftarrow$ ORAMSETUP$(1^\lambda, |R_2|/2)$
$\sigma' \leftarrow (\sigma_1, \sigma_2, size); R' \leftarrow (R_1, R_2)$
return $(res, \sigma'; R')$

ORAMSETUP and ORAMACCESS are LazyPathORAM's protocols.

Figure 18: DYNORAM: Setup, Insert, and Delete.

$(res, \sigma'; R') \leftrightarrow$ DYNORAMSEARCHUPDATE$(op, id, val, \sigma; R)$

$(\sigma_1, \sigma_2, size) \leftarrow \sigma$; $(R_1, R_2) \leftarrow R$
$id_1 \leftarrow id \mod |R_1|$ and $id_2 \leftarrow id$
$res \leftarrow \perp$
if $id \geq |R_1| \vee id < (size - |R_1|)$ **then**
  $(\perp, \sigma_1; R_1) \leftrightarrow$ ORAMACCESS$(read, id_1, \perp, \sigma_1; R_1)$
  $(res, \sigma_2; R_2) \leftrightarrow$ ORAMACCESS$(op, id_2, val, \sigma_2; R_2)$
else
  $(res, \sigma_1; R_1) \leftrightarrow$ ORAMACCESS$(op, id_1, val, \sigma_1; R_1)$
  $(\perp, \sigma_2; R_2) \leftrightarrow$ ORAMACCESS$(read, id_2, \perp, \sigma_2; R_2)$
$\sigma' \leftarrow (\sigma_1, \sigma_2, size)$; $R' \leftarrow (R_1, R_2)$
return $(res, \sigma'; R')$

ORAMSETUP and ORAMACCESS are LazyPathORAM's protocols.

Figure 19: DYNORAM: SearchUpdate.

to achieve DYNORAM-Type III security. The simulator is shown in Figure 20.

For **SIMDYNORAMSETUP**, everything happens on the client-side, and nothing needs to be simulated (algorithm SIMORAMSETUP, Figure 20).

For **SIMDYNORAMINSERT, SIMDYNORAMDELETE**, we only need the size of the ORAM to know the semantics of the operation. We calculate the size from the sequence of operations, stored in the simulator state. Then, we replace the ORAMSETUP and ORAMACCESS calls (Figure 18) with SIMORAMSETUP and SIMORAMACCESS calls, respectively (Figure 20).

Similarly, for **SIMDYNORAMSEARCHUPDATE**, we replace the two ORAMACCESS calls with SIMORAMACCESS.

Efficiency. The complexities are similar to the DYNODS schemes—the complexity of DYNORAMSETUP is $O(1)$, as it only creates the necessary state; and the complexities of DYNORAMINSERT, DYNORAMDELETE, DYNORAMSEARCH, and DYNORAMUPDATE is $O(\log n_i)$, as they perform a constant number of ORAMACCESS operations. But, this scheme requires $O(n_i)$ client storage as it needs position maps for both PathORAM instances. To achieve sub-linear client storage complexity, we recommend the use of DYNOMap as a position map, or to use a recursive version of LazyPathORAM internally; either solution increases the cost of the DYNORAM accesses to $O(\log^2 n_i)$. We empirically evaluate DYNORAM in Appendix C.

Appendix E. Oblivious Sorted Multi-Map

OSM construction. To create the OSM, Mishra et al. [14] start from an AVL tree (similar to the OMap scheme of

$(st_s; T) \leftarrow$ SIMDYNORAMSETUP(1^λ)

Set ops to be an empty vector
ist_1, ist_2 to be empty static Oram simulator states
Set T_1, T_2 to be empty simulated static Oram instances
$st_s \leftarrow (ops, ist_1, ist_2)$, $T \leftarrow (T_1, T_2)$
return (st_s, T)

$(st_s; T) \leftarrow$ SIMDYNORAMINSERT$(st_s; T)$

$(ops, ist_1, ist_2) \leftarrow st_s$ and $(T_1, T_2) \leftarrow T$
$size = count_{insert}(ops) - count_{delete}(ops)$
Append an $insert$ to ops
if $size = 0$ **then**
  $(ist_2, T_2) \leftarrow$ SIMORAMSETUP$(1^\lambda, 1)$
else if $size$ is a power of 2 **then**
  $T_1 \leftarrow T_2$
  $(ist_2, T_2) \leftarrow$ SIMORAMSETUP$(1^\lambda, 2 \times size)$
$(ist_1, T_1) \leftarrow$ SIMORAMACCESS(ist_1, T_1)
$(ist_2, T_2) \leftarrow$ SIMORAMACCESS(ist_2, T_2)
$(ist_2, T_2) \leftarrow$ SIMORAMACCESS(ist_2, T_2)
$st_s \leftarrow (ops, ist_1, ist_2)$ and $T \leftarrow (T_1, T_2)$
return (st_s, T)

$(st_s; T) \leftarrow$ SIMDYNORAMDELETE$(st_s; T)$

$(ops, ist_1, ist_2) \leftarrow st_s$ and $(T_1, T_2) \leftarrow T$
Append a $delete$ to ops
$size = count_{insert}(ops) - count_{delete}(ops)$
$(ist_2, T_2) \leftarrow$ SIMORAMACCESS(ist_2, T_2)
$(ist_2, T_2) \leftarrow$ SIMORAMACCESS(ist_2, T_2)
$(ist_1, T_1) \leftarrow$ SIMORAMACCESS(ist_1, T_1)
if $size$ is a power of 2 **then**
  $ist_2 \leftarrow ist_1$; $T_2 \leftarrow T_1$
  $(ist_1; T_1) \leftarrow$ SIMORAMSETUP$(1^\lambda, size/2)$
$st_s \leftarrow (ops, ist_1, ist_2)$ and $T \leftarrow (T_1, T_2)$
return (st_s, T)

$(\sigma, T) \leftarrow$ SIMDYNORAMSEARCHUPDATE$(\sigma; T)$

$(ops, ist_1, ist_2) \leftarrow st_s$ and $(T_1, T_2) \leftarrow T$
$(ist_1, T_1) \leftarrow$ SIMORAMACCESS(ist_1, T_1)
$(ist_2, T_2) \leftarrow$ SIMORAMACCESS(ist_2, T_2)
$st_s \leftarrow (ops, ist_1, ist_2)$ and $T \leftarrow (T_1, T_2)$
return (st_s, T)

Figure 20: DYNORAM Simulator.

[13]), and augment it as follows: **(A)** allow multiple values for a key, e.g., key 4 in Figure 21(d) that has the following values: $\{b, c, u\}$; **(B)** order entries first by keys and then by values so that the values of the same key are sorted, e.g., $(4 \rightarrow c) < (4 \rightarrow u)$; and **(C)** in each node, store the number of entries in each child sub-tree with the same key as that node (see Figure 22). E.g., the node that contains $(4 \rightarrow c)$ in Figure 21(c) has 1 entry with the same key in its left child sub-tree and none on its right child sub-tree.

Figure 21 shows a series of **insertions** into the sorted multi-map:

1) The tuple $(4 \rightarrow b)$ is inserted, which becomes the new

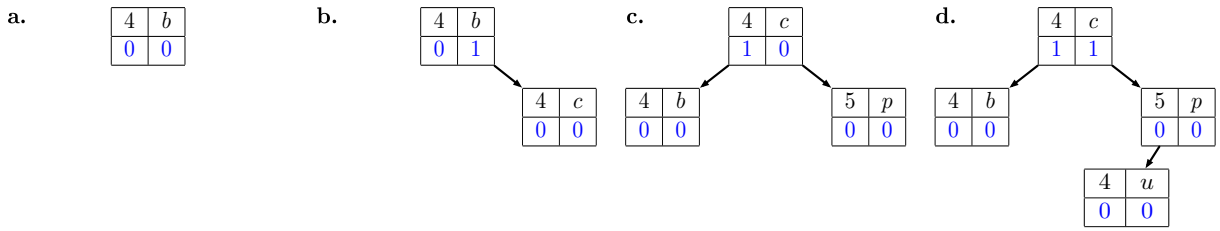


Figure 21: Sorted Multi-Map Insertion Example. The top row of a node contains its key and value, and the bottom contains the number of entries with same key in its left and right sub-trees. The following key-value pairs are inserted: a. (4, b), b. (4, c), c. (5, p), and d. (4, u).

Key	Val
$\texttt{Key}_\texttt{l}$	$\texttt{Key}_\texttt{r}$
$\texttt{Pos}_\texttt{l}$	$\texttt{Pos}_\texttt{r}$
$\texttt{SameKeyCount}_\texttt{l}$	$\texttt{SameKeyCount}_\texttt{r}$

Figure 22: AVL Tree-based map data structure and the additional information needed for the sorted multi-map (last row, in blue). Each AVL-node is stored in an ORAM block. Each block stores a key-value pair (first row) and the key and the ORAM position of its left and right children (second and third row). In sorted multi-maps, the number of entries in each child sub-tree that have the same key is also stored (fourth row).

root of an empty AVL tree.

2) The tuple $(4 \to c)$ is inserted, which becomes the right child of the first node, increasing the right-side counter of the first node by one.
3) The tuple $(5 \to p)$ is inserted, which becomes the right child of the second node. Now, the tree has become unbalanced as the root's child sub-trees have heights 0 and 2, with difference greater than 1. This triggers a rotation which makes the second node the new root, the first node its left child, and the third its right child. The nodes' counters are also updated, and the only non-zero counter is now the root's left counter, which is 1.
4) The tuple $(4 \to u)$ is inserted, which becomes the left child of the third node, increasing the root's right counter by one, as it also has key 4.

Note that similar to [13], each of the above operations needs to be padded to have a total of $3 \times 1.44 \times \log N$ ORAM accesses.

In this scheme, **searches** follow AVL tree semantics. For example, if a search is done for the third value of key 4 in the OSM of Figure 21(d), the following steps are taken: **(A)** the root is visited, which has the same key as searched (4). There is one entry with the same key in its left child sub-tree, and therefore the third entry searched is in its right sub-tree. **(B)** Node $(5 \to p)$ is visited, which has a larger key than searched. Therefore, its left child is visited next. **(C)** Node $(4 \to u)$ is visited, which has the same key as searched (4). There are no entries with the same key in its left child sub-tree, and hence it contains the third entry. **(D)** The value u is returned. **(E)** All of the visited nodes are written to new positions in the underlying ORAM, making the next access to them safe.

Deletions follow AVL tree semantics too, taking a key and a value as arguments. For example, if a delete is done for the tuple $(5, p)$ in the OSM of Figure 21(d), the following steps are taken: **(A)** Node $(5 \to p)$ is found similar to the first two steps of the search example. **(B)** The node is removed and replaced by its only child (node $(4 \to u)$). **(C)** All of the visited nodes are written to new positions in the underlying ORAM, making the next access to them safe.

Bounding the constants of Oblivious Sorted Multi-Map Accesses. For the sorted multi-map, the number of ORAM accesses for retrieving r values for a key is $T(n, r) = O(\log n + r)$.

When fetching the range $i..j$ of the values of a certain key, the algorithm does the following:

1) Traverse the tree to the node containing the i-th value. Let v_l denote this node and P_l denote the path from root to v_l.
2) Do the same for the j-th value; defining v_r and P_r.
3) Find the last (lowest) common node of P_l and P_r. Let s denote this node
4) Do a BFS starting at s, only going down the tree between the two paths. The algorithm can decide whether a child of a node whould be processed based on the topology of the tree (i.e., by comparing the node's key to the queried key, and then checking the left and right same-key counters of the node).

Claim. $T(n, r) \leq 2 \times \lceil 3 \times 1.44 \times \log n \rceil + (r - 2)$.

Proof. Step **1** and **2** each take exactly $\lceil 3 \times 1.44 \times \log n \rceil$ ORAM accesses. Step **3** needs no ORAM accesses as P_l and P_r are already in client stash.

For each node u that step **4** traverses, we *know* that

$$\langle key(v_l).val(v_l) \rangle \leq \langle key(u).val(u) \rangle \leq \langle key(v_r).val(v_r) \rangle$$

Furthermore, we know that such u's are the only nodes between v_l and v_r. This follows directly from the tree being sorted. Let U denote the set of all such u nodes. Each $u \in U$ is reachable from s. Furthermore, the path to any $u \in U$ consists of nodes that are members of $P_l \cup P_r \cup U$. Therefore, since $P_l \cup P_r$ is already in cache, the number of ORAM accesses that step **4** needs is at most $|U| = (r - 2)$. □