Oblivious Data Structures*

Xiao Shaun Wang[1], Kartik Nayak[1], Chang Liu[1], T-H. Hubert Chan[2],
Elaine Shi[1], Emil Stefanov[3], and Yan Huang[4]

[1]UMD [2]HKU [3]UC Berkeley [4]IU Bloomington
[1]{wangxiao,kartik,liuchang,elaine}@cs.umd.edu
[2]hubert@cs.hku.hk [3]emil@cs.berkeley.edu [4]yh33@indiana.edu

Abstract

We design novel, asymptotically more efficient data structures and algorithms for programs whose data access patterns exhibit some degree of predictability. To this end, we propose two novel techniques, a *pointer*-based technique and a *locality*-based technique. We show that these two techniques are powerful building blocks in making data structures and algorithms oblivious. Specifically, we apply these techniques to a broad range of commonly used data structures, including maps, sets, priority-queues, stacks, deques; and algorithms, including a memory allocator algorithm, max-flow on graphs with low doubling dimension, and shortest-path distance queries on weighted planar graphs. Our oblivious counterparts of the above outperform the best known ORAM scheme both asymptotically and in practice.

1 Introduction

It is known that access patterns, to even encrypted data, can leak sensitive information such as encryption keys [27, 57]. Furthermore, this problem of access pattern leakage is prevalent in numerous application scenarios, including cloud data outsourcing [44], design of tamper-resistant secure processors [14,35,46,47,57], as well as secure multi-party computation [24, 33, 34].

Theoretical approaches for hiding access patterns, referred to as Oblivious RAM (ORAM) algorithms, have existed for two and a half decades thanks to the ground-breaking work of Goldreich and Ostrovsky [18]. However, the community has started to more seriously investigate the possibility of making ORAMs practical only recently [19, 22, 42, 43, 45, 53]. Encouragingly, recent progress in this area has successfully lowered the bandwidth blowup of ORAM from a factor of tens of thousands to $10X$ – $100X$ range [43, 45, 53].

Since generic Oblivious RAM can support arbitrary access pattern, it is powerful and allows the oblivious simulation of any program. However, state-of-the-art ORAM constructions incur moderate bandwidth blowup despite the latest progress in this area. In particular, under constant or polylogarithmic client-side storage, the best known scheme achieves $O(\frac{\log^2 N}{\log \log N})$ asymptotic blowup [29], i.e., for each effective bit read/written, $O(\frac{\log^2 N}{\log \log N})$ bits must be in reality accessed for achieving obliviousness. We remark that under large block sizes, Path ORAM can achieve $O(\log N)$ bandwidth blowup under poly-logarithmic client-side storage – however, the large block size assumption is often not applicable for data structures or algorithms that operate on integer or floating point values.

It will be beneficial to have customized, asymptotically more efficient constructions for a set of common algorithms which exhibit some degree of predictability in their access patterns. The *access pattern graph*

*This research is partially funded by the National Science Foundation under grant CNS-1314857, a Sloan Research Fellowship, Google Faculty Research Awards, Defense Advanced Research Projects Agency (DARPA) under contract FA8750-14-C-0057, as well as a grant from Hong Kong RGC under the contract HKU719312E. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US government.

for an algorithm has memory cells as nodes, and two cells can be accessed in succession only if there is a directed edge between the corresponding nodes. Hence, for general RAM programs, their access pattern can be a complete graph. Our key insight is that common data structures have a sparser access pattern graph than generic RAM programs that make arbitrary random accesses to data. For example, for a binary search tree or heap, memory accesses can only go from one tree node to an adjacent one. Therefore, we should be able to gain some efficiency (compared to ORAM) by not hiding some publicly known aspects of the access patterns.

In this work, we are among the first to investigate oblivious data structures (ODS) for sparse access pattern graphs. We achieve asymptotic performance gains in comparison with generic ORAM schemes [29,45] for two different characterizations of sparse access graphs (see Table 1):

Technique	Example Applications	Client-side storage	Blowup
Pointer-based for rooted tree access pattern graph	`map/set`, `priority_queue`, `stack`, `queue`, oblivious memory allocator	$O(\log N) \cdot \omega(1)$	$O(\log N)$
Locality-based for access pattern graph with doubling dimension dim	maximum flow, random walk on sparse graphs; shortest-path distance on planar graphs; `doubly-linked list`, `deque`	$O(1)^{\mathsf{dim}} \cdot O(\log^2 N) + O(\log N) \cdot \omega(1)$	$O(12^{\mathsf{dim}} \log^{2-\frac{1}{\mathsf{dim}}} N)$
Path ORAM [45]	All of the above	$O(\log N) \cdot \omega(1)$	$O(\frac{\log^2 N + \chi \log N}{\chi})$ for block size $\chi \log N$
ORAM in [29]	All of the above	$O(1)$	$O(\frac{\log^2 N}{\log \log N})$

Table 1: **Asymptotic performance of our schemes in comparison with generic ORAM baseline.** For the locality-based technique and Path ORAM, the bandwidth blowup hold for a slight variant of the standard ORAM model where non-uniform block sizes are allowed.
A note on the notation $g(N) = O(f(N))\omega(1)$: throughout this paper, this notation means that for any $\alpha(N) = \omega(1)$, it holds that $g(N) = O(f(N)\alpha(N))$.

- *Bounded-degree trees.* We show that for access pattern graphs that are rooted trees with bounded degree, we can achieve $O(\log N)$ bandwidth blowup, which is an $\tilde{O}(\log N)$ factor improvement in comparison with the best known ORAM.
- *Graphs with low doubling dimensions.* Loosely speaking, graphs with low *doubling dimension* are those whose local neighborhoods grow slowly. Examples of graphs having low doubling dimension include paths and low dimensional grids. Let dim denote the doubling dimension of the access pattern graph. We show how to achieve $O(1)^{\mathsf{dim}} \cdot O(\log^{2-\frac{1}{\mathsf{dim}}} N)$ amortized bandwidth blowup while consuming $O(1)^{\mathsf{dim}} \cdot O(\log N) + O(\log^2 N) \cdot \omega(1)$ client-side storage. As a special case, for a two-dimensional grid, we can achieve $O(\log^{1.5} N)$ blowup.

Applications. These two characterizations of sparse access pattern graphs give rise to numerous applications for commonly encountered tasks:

1. *Commonly-encountered data structures.* We derive a suite of efficient oblivious data structure implementations, including the commonly used `map/set`, `priority_queue`, `stack`, `queue`, and `deque`.
2. *Oblivious memory allocator.* We use our ODS framework to design an efficient oblivious memory allocator Our oblivious memory allocator requires transmitting $O(\log^3 N)$ bits per operation (notice that this is the exact number of bits, not bandwidth blowup). We show that this achieves exponential savings in comparison with the baseline chunk-based method. In particular, in the baseline approach,

to hide what fraction of memory is committed, each memory allocation operation needs to scan through $O(N)$ memory.

Our oblivious memory allocator algorithm can be adopted on ORAM-capable secure processor [14, 31, 35] for allocation of oblivious memory.

3. *Graph algorithms.* We achieve asymptotic improvements for operations on graphs with low doubling dimension, including random walk and maximum flow. We consider an oblivious variant of the Ford-Fulkerson [15] maximum flow algorithm in which depth-first-search is used to find an augmenting path in the residual network in each iteration. We also consider shortest-path distance queries on planar graphs. We make use of the planar separator theorem to create a graph data structure and make it oblivious.

Practical performance savings. We evaluated our oblivious data structures with various application scenarios in mind. For the outsourced cloud storage and secure processor settings, bandwidth blowup is the key metric; whereas for a secure computation setting, we consider the number of AES encryptions necessary to perform each data structure operation. Our simulation shows an order of magnitude speedup under moderate data sizes, in comparison with using generic ORAM. Since the gain is shown to be asymptotic, we expect the speedup to be even greater when the data size is bigger.

Techniques. We observe two main techniques for constructing oblivious data structures for sparse access pattern graphs.

- *Pointer-based technique.* This is applied when the access graph is a rooted tree with bounded degree. The key idea is that each parent keeps pointers for its children and stores children's position tags, such that when one fetches the parent node, one immediately obtains the position tags of its children, thereby eliminating the need to perform position map lookups (hence a logarithmic factor improvement).

  We also make this basic idea work for dynamic data structures such as balanced search trees, where the access pattern structure may change during the life-time of the data structure. Our idea (among others) is to use a cache to store nodes fetched during a data structure operation, and guarantee that for any node we need to fetch from the server, its position tag already resides in the client's cache.

- *Locality-based technique.* This is applied when the access pattern graph has low doubling dimension. The nodes in the graph are partitioned into clusters such that each cluster keeps pointers to only $O(1)^{\mathsf{dim}}$ neighboring clusters where dim is an upper bound on the doubling dimension. The intuition is that each cluster contains $O(\log N)$ nodes and can support $O(\log^{\frac{1}{\mathsf{dim}}} N)$ node accesses. As we shall see, each cluster has size $\Omega(\log^2 N)$ bits, and hence each cluster can be stored as a block in Path ORAM [45] with $O(\log N)$ bandwidth blowup.

  Since each cluster only keeps track of $O(1)^{\mathsf{dim}}$ neighboring clusters, only local information needs to be updated when the access pattern graph is modified.

Non-goals, limitations. Like in almost all previous work on oblivious RAM, we do not protect information leakage through the timing channel. Mitigating timing channel leakage has been treated in orthogonal literature [3, 55, 56]. Additionally, like in (almost) all prior ORAM literature, we assume that there is an *a'priori* known upper bound N on the total size of the data structure. This seems inevitable since the server must know how much storage to allocate. Similar to the ORAM literature, our oblivious data structures can also be resized on demand at the cost of 1-bit leakage.

1.1 Related Work

Oblivious algorithms. Customized oblivious algorithms for specific funtionalities have been considered in the past, and have been referred to by different names (partly due to the fact that the motivations of

these works stem from different application settings), such as oblivious algorithms [13, 23] or efficient circuit structures [38, 54].

The work by Zahur and Evans [54] may be considered as a nascent form of oblivious data structures; however the constructions proposed do not offer the full gamut of common data structure operations. For example, their stacks and queues support special conditional update semantics; and their associative map supports only batched operations but not individual queries (supporting batched operations are significantly easier). Toft [48] also studied efficient construction of oblivious priority queue. Their construction reveals the type of operations, and thus the size of the data structure. Our proposed construction only reveals number of operations with the same asymptotic bound.

Mitchell and Zimmerman [37] observe that Oblivious Turing Machine [40] can also be leveraged to build oblivious stacks and queues yielding an amortized $O(\log N)$ blowup (but worst-case cost is $O(N)$); however, the $O(\log N)$ oblivious TM-based approach does not extend to more complex data structures such as maps, sets, etc.

Blanton, Steele and Alisagari [8] present oblivious graph algorithms, such as breadth-first search, single-source single-destination (SSSD), minimum spanning tree and maximum flow with nearly optimal complexity on dense graphs. Our work provides asymptotically better algorithms for special types of sparse graphs, and for repeated queries. See Section 5 for details.

Oblivious program execution. The programming language community has started investigating type systems for memory-trace obliviousness, and automatically compiling programs into their memory-trace oblivious counterparts [31]. Efficiency improvements may be achieved through static analysis. The main idea is that many memory accesses of a program may not depend on sensitive data, e.g., sequentially scanning through an array to find the maximal element. In such cases, certain arrays may not need to be placed in ORAMs; and it may be possible to partition arrays into multiple ORAMs without losing security. While effective in many applications, these automated compiling techniques currently only leverage ORAM as a blackbox, and cannot automatically generate asymptotically more efficient oblivious data structures.

Generic oblivious RAM. The relations between oblivious data structures and generic ORAM [10, 12, 16–22, 29, 39, 42, 44, 50–52] have mostly been addressed earlier. We add that in the secure computation setting involving two or more parties, it is theoretically possible to employ the ORAM scheme by Lu and Ostrovsky [33] to achieve $O(\log N)$ blowup. However, their ORAM does not work for a cloud outsourcing setting with a single cloud, or a secure processor setting. Further, while asymptotically non-trivial, their scheme will likely incur a large blowup in a practical implementation due to the need to obliviously build Cuckoo hash tables.

History-independent data structures. Oblivious data structures should not be confused with history-independent data structures (occasionally also referred as oblivious data structures) [9,36]. History-independent data structures require that the resulting state of the data structure reveals nothing about the histories of operation; and do not hide access patterns.

Concurrent work. In concurrent work, Keller and Scholl [28] implemented secure oblivious data structures using both the binary-tree ORAM [42], and the Path ORAM [45]. They leverage the technique suggested by Gentry *et al.* [16] in the implementation of the shortest path algorithm, achieving $O(\log N)$ asymptotic saving. They do not generalize this approach for dynamic data structures, nor do they consider other access pattern graphs.

2 Problem Definition

A data structure $\mathcal{D}$ is a collection of data supporting certain types of operations such as insert, del, or lookup. Every operation is parameterized by some operands (e.g., the key to look up). Intuitively, the goal of oblivious data structures is to ensure that for any two sequences each containing k operations, their resulting access patterns must be indistinguishable. This implies that the access patterns, including the number of accesses, should not leak information about both the op-code and the operand.

Definition 1 (Oblivious data structure)**.** *We say that a data structure $\mathcal{D}$ is oblivious, if there exists a polynomial-time simulator $\mathcal{S}$, such that for any polynomial-length sequence of data structure operations $\vec{\mathsf{ops}} = ((\mathsf{op}_1, \mathsf{arg}_1), \ldots, (\mathsf{op}_M, \mathsf{arg}_M))$*

$$\mathsf{addresses}_{\mathcal{D}}(\vec{\mathsf{ops}}) \stackrel{c}{\equiv} \mathcal{S}(\mathcal{L}(\vec{\mathsf{ops}}))$$

where $\mathsf{addresses}_{\mathcal{D}}(\vec{\mathsf{ops}})$ is the physical addresses generated by the oblivious data structure during a sequence of operations $\vec{\mathsf{ops}}$; and $\mathcal{L}(\vec{\mathsf{ops}})$ is referred to as the leakage function. Typically we consider that $\mathcal{L}(\vec{\mathsf{ops}}) = M$, i.e., the number of operations is leaked, but nothing else.

Intuitively, this definition says that the access patterns resulting from a sequence of data structure operations should reveal nothing other than the total number of operations. In other words, a polynomial-time simulator $\mathcal{S}$ with knowledge of only the total number of operations, can simulate the physical addresses, such that no polynomial-time distinguisher can distinguish the simulated addresses from the real ones generated by the oblivious data structure.

Note that directly using standard ORAM may not be able to satisfy our definition, since information may leak through the number of accesses. Specifically, some data structure operations incur more memory accesses than others; e.g., an AVL tree deletion will likely incur rotation operations, and thus incur more memory accesses than a lookup operation. Therefore, even if we were to employ standard ORAM, padding might be needed to hide what data structure operation is being performed.

2.1 Model and Metric

Bandwidth blowup. In order to achieve obliviousness, we need to access more data than we need. Roughly speaking, the *bandwidth blowup* is defined as the ratio of the number of bytes transferred in the oblivious case over the non-oblivious baseline.

Since we hide the type of the data structure operation, the number of bytes transferred in the oblivious case is the same across all operations of the same data structure instance. However, the number of bytes transferred in the non-oblivious case may vary across operation. For most of the cases we consider, the number of bytes transferred for each operation are asymptotically the same. Further, the average-case cost and worst-case cost in the non-oblivious case are also asymptotically the same. In these cases, we do not specify which individual operation is considered when we mention bandwidth blowup.

Model. Results using our pointer-based techniques apply to the standard RAM model with *uniform* block sizes. Results relying on our locality-based techniques apply to a slight variant of the standard RAM model, where blocks may be of *non-uniform* size. This assumption is the same as in previous work such as Path ORAM [45], which relied on a "big data block, little metadata block" trick to parameterize the recursions on the position map.

3 Rooted Trees with Bounded Degree

Numerous commonly used data structures (e.g., stacks, map/set, priority-queue, B-trees, etc.) can be expressed as a rooted tree with bounded degree. Each data access would start at the root node, and traverse the nodes along the tree edges. In these cases, the access pattern graph would naturally be a bounded-degree tree as well. In this section, we describe a pointer-based technique that allows us to achieve $O(\log N)$ blowup for such access pattern graphs. In comparison, the best known ORAM scheme has $O(\frac{\log^2 N}{\log \log N})$ blowup. Our ideas are inspired by those of Gentry *et al.* [16] who showed how to leverage a position-based ORAM to perform binary search more efficiently.

3.1 Building Block: Non-Recursive Position-based ORAM

To construct oblivious data structures, an underlying primitive we rely on is a position-based ORAM. Several ORAM schemes have been proposed in the recent past that rely on the client having a position

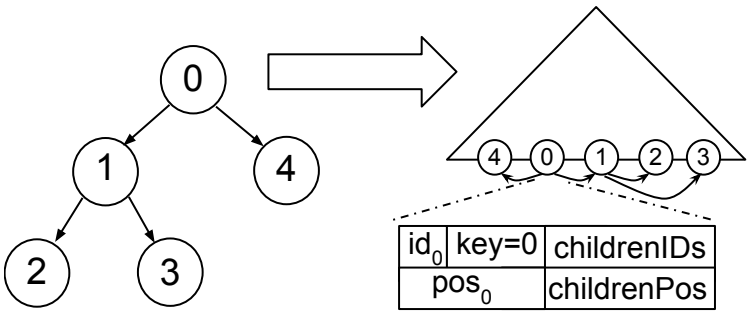


Figure 1: **Static oblivious binary search tree.** A logical binary search tree is on the left, and on the right-hand side is how these nodes are stored in a (non-recursive) position-based ORAM. Every position tag specifies a path to a leaf node on the tree. Every parent points to the position tags of its children such that we can eliminate the need for position map lookups, thus saving an $O(\log N)$ factor in comparison with generic ORAM. This is a generalization of the techniques described by Gentry *et al.* [16] for performing binary search with ORAM.

map [10, 16, 42, 45] that stores a location label for each block. By *recursive*, we mean the position map is (recursively) stored on the server, as opposed to *non-recursive*, where the position map is entirely stored by the client. So far, Path ORAM [45] achieves the least blowup of $O(\log N)$ for the non-recursive case, or the recursive case when the data block size is $\Omega(\log^2 N)$ bits. We shall use Path ORAM by default.

For a client to look up a block with identifier id, the client must first look up its locally stored position map to obtain a location pos for block id. This pos records the rough location (e.g., either a tree path [42,45], or a partition [44]) of block id. Knowing pos, the client will then know from which physical addresses on the server to fetch blocks.

When a data block is read, it will be removed from the ORAM and assigned a new location denoted pos′. The block is then written back to the server attached with its new location tag, i.e., data||pos′. If this is a read operation, data is simply a re-encryption of the fetched block itself; if this is a write operation, data will be the newly written block.

We use the following abstraction for a position-based ORAM scheme. Although any (non-recursive) position-based ORAM [10, 16, 44, 45] will work for our construction (potentially resulting in different performance); for simplicity, we often assume that Path ORAM is the underlying position-based ORAM.

ReadAndRemove(id, pos): fetches and removes from server a block identified by id. pos is the position tag of the block, indicating a set of physical addresses where the block might be.

Add(id, pos, data): writes a block denoted data, identified by id, to some location among a set of locations indicated by pos.

In standard ORAM constructions, to reduce the client storage required to store the position map, a recursion technique has been suggested [42, 44] to recursively store the position map in smaller ORAMs on the server. However, this will lead to a blowup of $O(\log^2 N)$.

Inspired by the binary search technique proposed by Gentry *et al.* [16] in the context of RAM-model secure computation, we propose a technique that allows us to eliminate the need to perform recursive position map lookups, thus saving an $O(\log N)$ cost for a variety of data structures.

3.2 Oblivious Data Structure

We first explain our data representation and how oblivious AVL tree can be implemented securely. We show that a factor of $O(\log N)$ can be saved by eliminating the need to recursively store the position map. The detailed general framework with dynamic access algorithms can be found in [49].

Node format. In an oblivious data structure, every node is tagged with some payload denoted data. If there is an edge from vertex v to vertex w in the access pattern graph, then it means that one can go from v to w during some data structure operation.

In the oblivious data structure, every node is identified by some identifier id along with its position tag pos. The node also stores the position tags of all of its children. Therefore, the format of every node will be

$$\mathsf{node} := (\mathsf{data}, \mathsf{id}, \mathsf{pos}, \mathsf{childrenPos})$$

In particular, $\mathsf{childrenPos}$ is a mapping from every child id to its position tag. We write

$$\mathsf{childrenPos}[\mathsf{id}_c]$$

to denote the position tag of a child identified by id_c. In the rest of the paper, unless otherwise specified, a node will be the basic unit of storage.

All nodes will be (encrypted and) stored in a (non-recursive) position-based ORAM on the server. The client stores only the position tag and identifier of the root of the tree.

Oblivious map insertion example. To illustrate dynamic data structures, we rely on AVL tree (used to implement oblivious map) insertion as a concrete example. Figure 2 illustrates this example. The insertion proceeds as follows: first, find the node at which the insertion will be made; second, perform the insertion; and third, perform rotation operations to keep the tree balanced.

During this operation $O(\log N)$ nodes will be accessed, and some nodes may be accessed twice due to the rotation that happens after the insertion is completed. Further, among the nodes accessed, the graph structure will change as a result of the rotation.

Notice that once a node is fetched from the server, its position tag is revealed to the server, and thus we need to generate a new position tag for the node. At the same time, this position tag should also be updated in its parent's children position list. Our key idea is to rely on an $O(\log N)$-sized client-side *cache*, such that all nodes relevant for this operation are fetched only once during this entire insertion operation. After these nodes are fetched and removed from the server, they will be stored in the client-side cache, such that the client can make updates to these nodes locally before writing them back. These updates may include insertions, removals, and modifying graph structures (such as rotations in the AVL tree example). Finally, at the end of the operation, all nodes cached locally will be written back to the server. Prior to the write-back, all fetched nodes must be assigned random new position tags, and their parent nodes must be appropriately modified to point to the new position tags of the children.

This approach gives us $O(\log N)$ blowup at the cost of a client-side cache of size $O(\log N)$. Since the client-side cache allows us to read and write this $O(\log N)$-length path only once for the entire insertion operation, and each node requires $O(\log N)$ cost using (non-recursive) Path ORAM as the underlying position-based ORAM, the bandwidth overhead is $O(\log N)$ – since in the oblivious case, the total I/O cost is $O(\log^2 N)$ whereas in the non-oblivious case, the total I/O cost is $O(\log N)$.

Padding. During a data structure operation, cache misses will generate requests to the server. The number of accesses to the server can leak information. Therefore, we pad the operation with dummy $\mathsf{ReadAndRemove}$ and Add accesses to the maximum number required by any data structure operation. For example, in the case of an AVL tree, the maximum number of $\mathsf{ReadAndRemove}$ operations and the maximum number of Add operations is $3 \times \lceil 1.45 \log(N+2) \rceil$. It should be noted that $\mathsf{ReadAndRemove}$ is padded before a real Add happens and hence all $\mathsf{ReadAndRemove}$ calls in an operation happen before the first Add.

A generalized framework for oblivious data structures. We make the following observations from the above example:

- The algorithm accesses a path from the root down to leaves. By storing the position tag for the root, all positions can be recovered from the nodes fetched. This eliminates the need to store a position map for all identifiers.
- After the position tags are regenerated, the tag in every node's $\mathsf{childrenPos}$ must also be updated. So we require that every node has only one parent, i.e., the access pattern graph is a bounded-degree tree.

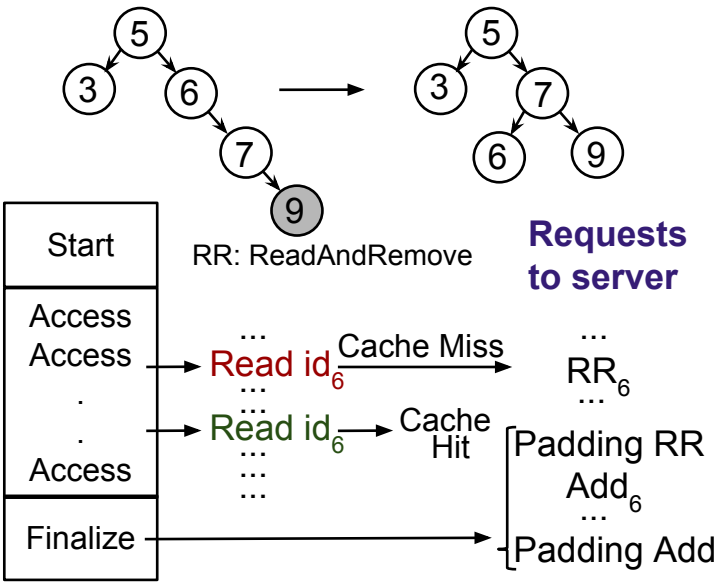


Figure 2: **Operations generated for insertion in an AVL Tree.** Cache hit/miss behavior does not leak information due to the padding we perform to ensure that every operation, regardless of the opcode and operands, has an equal amount of ReadAndRemove and Add calls to the position-based ORAM.

The above technique can be generalized to provide a framework for designing dynamic oblivious data structures whose access pattern graphs are bounded-degree trees. Based on this framework for constructing oblivious data structures, we derive algorithms for oblivious map, queue, stack, and heap. Due to space constraints, we defer the detailed algorithms, implementations of different oblivious data structures as well as some practical consideration to the online full version of this paper [49].

Theorem 1. *Assuming that the underlying (non-recursive) position-based ORAM is secure, our oblivious stack, queue, heap, and map/set are secure by Definition 1.*

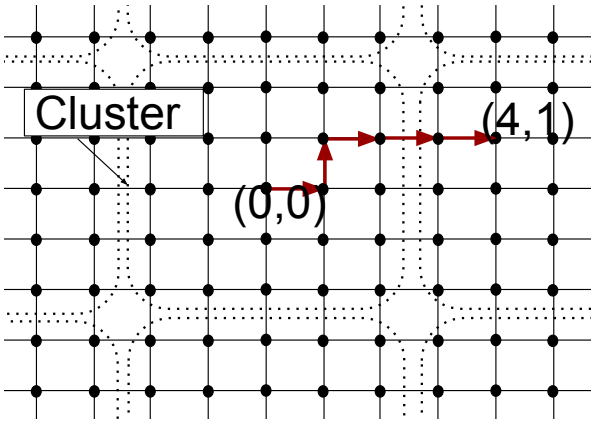
Proof can be found in the online version. [49].

4 Access Pattern Graphs with Low Doubling Dimension

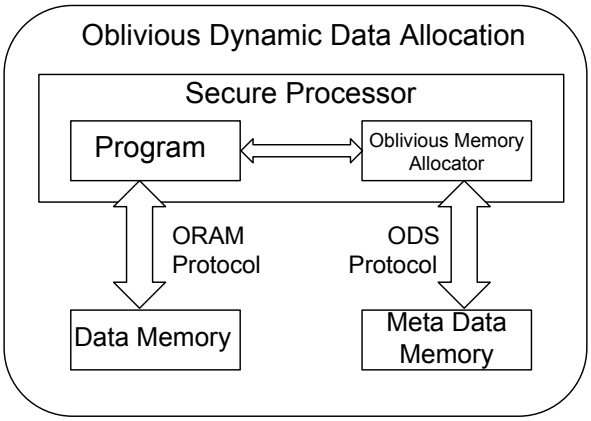
As seen in Section 3, we can achieve $O(\log N)$ blowup when the access pattern graph is a rooted tree where edges are directed towards children. In this section, we consider another class of access pattern graphs whose locality property will be exploited. Recall that the access pattern graph is directed, and we consider the underlying undirected graph $G = (V, E)$, which we assume to be static.

Intuition. The key insight is that if the block size is $\Omega(\log^2 N)$ bits, then (recursive) Path ORAM can achieve $O(\log N)$ blowup [45]. Hence, we shall partition V into *clusters*, each of which contains at most $\log N$ nodes. To make the data structure oblivious, we need to pad a cluster with dummy nodes such that each cluster contains exactly $\log N$ nodes. Assuming that each node has size $\Omega(\log N)$ bits, each cluster corresponds to $\Omega(\log^2 N)$ bits of data and will be stored as a generic ORAM block.

Hence, by accessing one cluster, we are storing the data of $\log N$ nodes in the client cache, each with a bandwidth blowup of $O(\log N)$. If these nodes are useful for the next T iterations, then the amortized blowup is $O(\frac{\log^2 N}{T})$. To understand how these clusters can be helpful, consider the following special cases of access pattern graphs.



(a) **Accesses for 2 dimensional structure exhibiting locality.** The red arrows indicate the path of the random walk for one set of $\sqrt{\log N}$ accesses.



(b) **Oblivious dynamic memory allocation.** For simplicity, imagine that both the program and the oblivious memory allocator reside in the secure processor's instruction cache. Securely executing programs (or a memory allocator) that reside in insecure memory is discussed in orthogonal work [31] and outside the scope here.

- For doubly-linked list and deque, the access pattern graph is a (1-dimensional) path. In this case, each cluster is a contiguous sub-path of $\log N$ nodes. Observe that starting at any node u, after $\log N$ iterations, we might end up at a node v that might not be in the same cluster as u. However, node v must be in a *neighboring* cluster to the left or the right of u. This suggests that we should pre-fetch the data from the 2 neighboring clusters, in addition to the cluster of the current node. This will make sure that in the next $\log N$ iterations, the cache will already contain the data for the accessed nodes.

- If the access pattern graph is a 2-dimensional grid, then each cluster is a sub-grid of dimensions $\sqrt{\log N}$ by $\sqrt{\log N}$ as shown in Figure 3a. Observe that by pre-fetching the 8 neighboring clusters surrounding the cluster containing the current node, we can make sure that the cache contains data of the accessed nodes in the next $\sqrt{\log N}$ iterations. Therefore, the blowup for those $\sqrt{\log N}$ iterations is $9 \cdot (\sqrt{\log N} \cdot \sqrt{\log N}) \cdot O(\log N)$, which gives an amortized blowup of $O(\log^{1.5} N)$. It is not difficult to see that for the d-dimensional grid, each cluster can be a hypercube with $\log^{\frac{1}{d}} N$ nodes on a side, and the number of neighboring clusters (including itself) is 3^d. Hence, the amortized blowup is $O(3^d \log^{2-\frac{1}{d}} N)$.

The d-dimensional grid is a special case of a wider class of graphs with bounded *doubling dimension* [4, 11, 25].

Doubling dimension. We consider the underlying undirected graph $G = (V, E)$ of the access pattern graph in which we assign each edge with unit length. We use $d_G(u, v)$ to denote the shortest path distance between u and v. A *ball* centered at node u with radius r is the set $B(u, r) := \{v \in V : d_G(u, v) \leq r\}$. We say that graph G has *doubling dimension* at most dim if every ball is contained in the union of at most 2^{dim} balls of half its radius. A subset $S \subset V$ is an *r-packing* if for any pair of distinct nodes $u, u' \in S$, $d(u, u') > r$. We shall use the following property of doubling dimension.

Fact 1 (Packings Have Small Size [25])**.** *Let $R \geq r > 0$ and let $S \subseteq V$ be an r-packing contained in a ball of radius R. Then, $|S| \leq (\frac{4R}{r})^{\mathsf{dim}}$.*

Blowup analysis sketch. We shall see that Fact 1 implies that pre-fetching $O(1)^{\mathsf{dim}}$ clusters is enough to support $\Theta(\log^{\frac{1}{\mathsf{dim}}} N)$ node accesses. Hence, the amortized blowup for each access is $O(1)^{\mathsf{dim}} \cdot O(\log^{2-\frac{1}{\mathsf{dim}}} N)$.

Setting up oblivious data structure on server. We describe how the nodes in V are clustered, and what information needs to be stored in each cluster.

In iteration i, **Access**(u_i, op, data) is performed, where node u_i is accessed, op is either Read or Write, and data is written in the latter case. We maintain the invariant that cache contains the cluster containing u_i.

1: $\rho = \left\lfloor \frac{1}{4} \log^{\frac{1}{\mathsf{dim}}} N \right\rfloor$
2: $i = 0$
3: Look up cluster C_0 containing u_0 from the cluster map.
4: Read the ORAM block corresponding to C_0 from the server into cache.
5: **for** i from 0 **do**
6: **Access**(u_i, op, data):
7: **if** $i \mod \rho = 0$ **then**
8: C_i is the cluster containing u_i, which is currently stored in cache by the invariant.
9: L_i is the list of neighboring clusters of C_i.
10: Read clusters in L_i from the ORAM server into cache; perform dummy reads to ensure exactly 12^{dim} clusters are read.
11: Write back non-neighboring clusters of C_i from cache to ORAM server; perform dummy writes to ensure exactly 12^{dim} clusters are written back.
12: **end if**
13: **if** op = Read **then**
14: Read data of node u_i from cache.
15: **else if** op = Write **then**
16: Update data of node u_i in cache.
17: **end if**
18: **return** data
19: **end for**

Figure 4: **Algorithm for node accesses.**

- *Greedy Partition.* Initially, the set U of *uncovered nodes* contains every node in V. In each iteration, pick an arbitrary uncovered node $v \in U$, and form a cluster $C_v := \{u \in U : d_G(u, v) \leq \rho\}$ with radius $\rho := \left\lfloor \frac{1}{4} \log^{\frac{1}{\mathsf{dim}}} N \right\rfloor$; remove nodes in C_v from U, and repeat until U becomes empty. From Fact 1, the number of clusters is at most $\frac{O(\Delta)^{\mathsf{dim}}}{\log N}$, where $\Delta := \max_{u,v} d_G(u, v)$.
- *Padding.* By Fact 1, each cluster contains at most $\log N$ nodes. If a cluster contains less than $\log N$ nodes, we pad with dummy nodes to achieve exactly $\log N$ nodes.
- *Cluster Map.* We need a generic ORAM to look up in which cluster a node is contained. However, this is required only for the very first node access, so its cost can be amortized by the subsequent accesses. As we shall see, all necessary information will be in the client cache after the initial access.
- *Neighboring Clusters.* Each cluster corresponds to an ORAM block, which stores the data of its nodes. In addition, each cluster also stores a list of all clusters whose centers are within 3ρ from its center. We say that two clusters are *neighbors* if the distance between their cluster centers is at most 3ρ. From Fact 1, each cluster has at most 12^{dim} neighboring clusters.

Algorithm for node accesses. We describe the algorithm for node accesses in Figure 4. An important invariant is that the node to be accessed in each iteration is already stored in cache when needed.

Lemma 1 (Cache Invariant)**.** *In each iteration i, the ORAM block corresponding to the cluster containing u_i is already stored in* cache *in lines 13 to 17.*

Theorem 2. *Algorithm in Figure 4 realizes oblivious data structures that are secure by Definition 1. When the number of node accesses is at least $\rho = \Theta(\log^{\frac{1}{\mathsf{dim}}} N)$, it achieves amortized $O(12^{\mathsf{dim}} \log^{2-\frac{1}{\mathsf{dim}}} N)$ bandwidth blowup. Moreover, the client memory stores the data of $O(12^{\mathsf{dim}} \log N) + O(\log^2 N) \cdot \omega(1)$ nodes.*

Proof. The proofs can be found in Appendix A. □

Dynamic access pattern graph. Our data structure can support limited insertion or deletion of nodes in the access pattern graph, as long as the doubling dimension of the resulting graph does not exceed a preset upper bound dim and some more detailed conditions are satisfied. We defer details to the online version [49].

Finally, note that Ren *et al.* [41] proposed a new technique to reduce the block size assumption to $\Omega(\frac{\log^2 N}{\log \log N})$. Their technique is also applicable here to slightly improve our bounds — however, due to the use of PRFs, we would now achieve computational rather than statistical security. Further, this technique would be expensive in the secure computation context due to the need to evaluate the PRF over a secure computation protocol such as garbled circuit.

5 Case Studies

Oblivious dynamic memory allocator. We apply our pointer-based technique for ODS to an important operating system task: memory management. A memory management task consists of two operations: dynamically allocating a portion of memory to programs at their request, and freeing it for reuse. In C convention, these two operations are denoted by `malloc`(l) and `free`(p). The security requirement of the oblivious memory allocator is that the physical addresses accessed to the allocator's working memory should not reveal any information about the opcode (i.e., `malloc` or `free`) or the operand (i.e., how many bytes to allocate and which address to free).

Since the total number of memory accesses may still leak information, naive extensions to existing memory management algorithms using ORAM are not secure. Padding in this case results in a secure but inefficient solution. We develop a new memory management algorithm which stores metadata in a tree-like structure, which can be implemented as an ODS, so that each memory allocation operation can be executed in the same time as one ORAM operation.

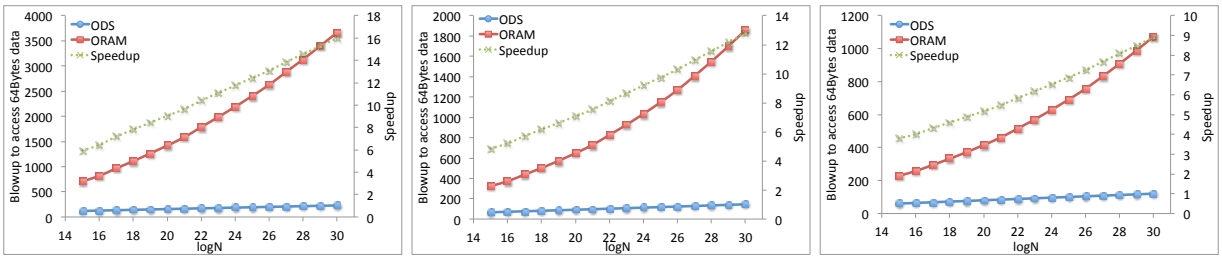
We develop a new oblivious memory allocator to achieve $O(\log^2 N)$ memory accesses per operation. The oblivious memory allocator's working memory consists of metadata stored separately from the data ORAM. Figure 3b illustrates this architecture[1].

The intuition is that if we treat the memory as a segment of size N, then the malloc and free functionalities are extracting a segment of a given size from the memory, and inserting a segment into the memory respectively. We construct a tree structure with nodes corresponding to segments, which is also called segment tree [7]. The root of the tree corresponds to the segment $[0, N)$. Each node corresponding to $[a, b)$ where $a + 1 < b$ has two children corresponding to $[a, \frac{a+b}{2})$ and $[\frac{a+b}{2}, b)$ respectively. Nodes corresponding to $[a, a + 1)$ are leaf nodes. It is easy to see that the height of a segment tree of size N is $O(\log N)$. The metadata memory is stored with these segments respectively. For each access, we will travel along one or two paths from the root to leaf, which can fit into our ODS framework. We state our result and include the details in the full version [49].

Theorem 3. *The oblivious memory allocator described above requires transmitting $O(\log^3 N)$ bits per operation.*

Shortest path on planar graph. We consider shortest distance queries on a weighted undirected planar graph. A naive approach is to build an all-pairs shortest distance matrix offline, so that each online query can be performed by a matrix lookup. On storing the matrix in an ORAM, the query can be computed within $O(\log^2 N)$ runtime, where N is the number of vertices in the graph. Recall that a planar graph is sparse and has $O(N)$ edges. By using Dijkstra's algorithm to compute the shortest distances from each single source, the total runtime for the offline process is within $O(N^2 \log N)$, while the space is $O(N^2)$. We notice on a large graph, i.e. $N \geq 2^{20}$, the space and offline runtime is impractical.

[1]For simplicity, we assume the program fits in the instruction cache which resides in the secure processor



(a) Oblivious Map/Set (b) Oblivious Stack/Queue (c) Oblivious Deque/List

Figure 5: **Bandwidth blowup for various oblivious data structures in comparison with general ORAM.** Payload = 64Bytes. The speedup curve has the y-axis label on the right-hand side.

We present an alternative approach using the planar separator theorem [30]. Using our locality-based approach, we can achieve oblivious versions of the construction with $O(N^{1.5} \log^3 N)$ offline processing time, $O(N^{1.5})$ space complexity, at the cost of $O(\sqrt{N} \log N)$ online query time.

Max flow on sparse graphs. We apply our oblivious data structure to solve the problem of maximum flow, and compare our approach with that of Blanton et al. [8], which focuses on the case when the graph is dense. We assume that the underlying undirected graph $G = (V, E)$ is fixed, and has doubling dimension at most dim. Since the degree of each node is at most 2^{dim}, the graph is sparse, and we assume that each block stores information concerning a node and its incident edges. A one-time setup phase is performed to construct the oblivious data structure in the ORAM server with respect to the graph G. This can support different instances of maximum flow in which the source, the sink, and the capacities of edges (in each direction) can be modified.

Similar to [8], we consider the standard framework by Ford and Fulkerson [15] in which an augmenting path is found in the residual network in each iteration. There is a strongly polynomial-time algorithm that takes $O(|E| \cdot |V|)$ path augmentations, in which each augmenting path is found by BFS. In [8], the adjacency matrix is permuted for every BFS, which takes time $O(|V|^2 \log |V|)$; hence, their running time is $O(|V|^3 \cdot |E| \log |V|)$.

However, we can perform only DFS using our locality-based technique. Hence, we consider a *capacity scaling* version of the algorithm in which the capacities are assumed to be integers and at most C. Hence, the running time of the (non-oblivious) algorithm is $O(|E|^2 \log C)$, which is also an upper bound on the number of node accesses. By Theorem 2, our running time is $O(12^{\mathsf{dim}} \cdot |E|^2 \log C \log^{2-\frac{1}{\mathsf{dim}}} |V|)$; in particular, our algorithm is better than [8], for sparse graphs, even when $C = 2^{|V|}$.

Random walk on graphs with bounded doubling dimension. In many computer science applications (e.g., [5]), random walk is used as a sub-routine. Since a random walk on a graph with bounded doubling dimension only needs to keep local information, our locality-based techniques in Theorem 2 is applicable to achieve a blowup of $O(12^{\mathsf{dim}} \log^{2-\frac{1}{\mathsf{dim}}} |V|)$ for every step of the random walk.

6 Evaluation

6.1 Methodology and Metrics

We evaluate the bandwidth blowup for our oblivious data structures with various application scenarios in mind, including cloud outsourcing [44], secure processors [14, 35], and secure computation [16, 24, 32]. Below we explain the metrics we focus on, and our methodology in obtaining the performance benchmarks.

Bandwidth blowup. Recall that bandwidth blowup is defined as the ratio of the number of bytes transferred in the oblivious case over the non-oblivious baseline. Bandwidth blowup is the most important metric

for the secure processor and the outsourced cloud storage applications, since bandwidth will be the bottleneck in these scenarios.

For our evaluation of bandwidth blowup, we use Path ORAM [45] to be our underlying position-based ORAM. We parameterize Path ORAM with a bucket size of 4, and the height of the tree is set to be $\log N$ where N is the maximum number of nodes in the tree.

We compare our oblivious data structures against Path ORAM (with a bucket size of 4 and tree height $\log N$), where the position map is recursively stored. For recursion of position map levels, we store 32 position tags in each block. We note while parameters of the Path ORAM can potentially be further tuned to give better performance, it is outside the scope of the paper to figure out the optimal parameters for Path ORAM.

Number of encryptions for secure computation. We also consider a secure computation application. This part of the evaluation focuses on an encrypted database query scenario, where Alice stores the (encrypted, oblivious) data structure, and Bob has the input $(\mathsf{op}, \mathsf{arg})$ pairs. After the computation, Alice gets the new state of oblivious data structure without knowing the operation and Bob gets the result of the operation.

We use the semi-honest, RAM-model secure computation implementation described in [32], which further builds on the FastGC garbled circuit implementation by Huang. et al. [26], incorporating frameworks by Bellare*et al.* [6]. We use the binary-tree ORAM by Shi *et al.* [42] in our ODS implementation as the position-based ORAM backend. Similarly, we compare with an implementation of the binary-tree ORAM [42]. For the general ORAM baseline, the position map recursion levels store 8 position tags per block, until the client stores $1,000$ position tags (i.e., roughly 1KB of position tags). We follow the ORAM encryption technique proposed by Gordon *et al.* [24] for implementing the binary-tree ORAM: every block is xor-shared while the client's share is always an output of client's secret cipher. This adds 1 additional cipher operation per block (when the length of an ORAM block is less than the width of the cipher). We note specific choices of ORAM parameters in related discussion of each application.

In secure computation, the bottleneck is the cost of generating and evaluating garbled circuits (a metric that is related to bandwidth blowup but not solely determined by bandwidth blowup). We therefore focus on evaluating the cost of generating and evaluating garbled circuits for the secure computation setting.

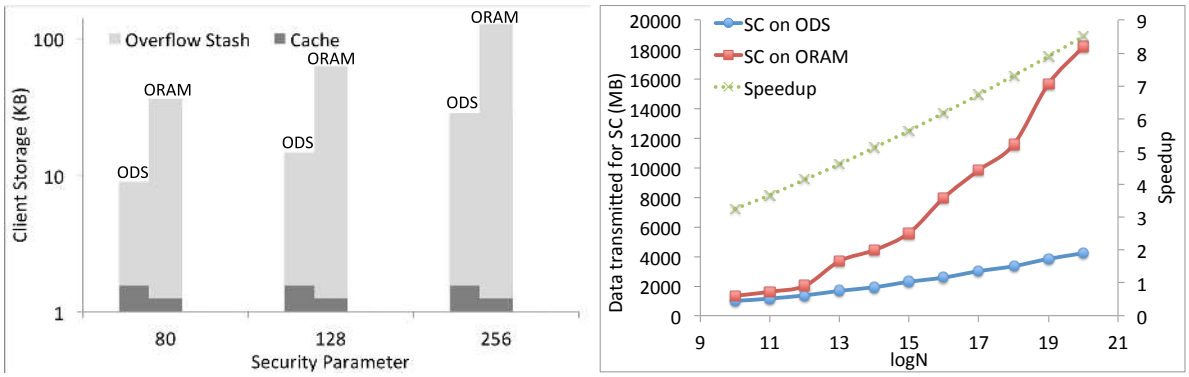
We use the *number of symmetric encryptions* (AES) as performance metric. Measuring the performance by the number of symmetric encryptions (instead of wall clock time) makes it easier to compare with other systems since the numbers can be independent of the underlying hardware and ciphering algorithms. Additionally, in our experiments these encryption numbers also represent the bandwidth consumption since every ciphertext will be sent over the network in a Garbled Circuit backend. Modern processors with AES support can compute 10^8 AES-128 operations per second [1].

We assume that the oblivious data structure is setup once at the beginning, whose cost hence can be amortized to later queries. Therefore, our evaluation focuses on the online part. The online cost involves three parts: i) the cost of preparing input, which involves Oblivious Transfer (OT) for input data; ii) the cost of securely computing the functionality of the position-based ORAM; and iii) securely evaluating the computation steps in data structure operations (e.g., comparisons of lookup keys).

Methodology for determining security parameters. We apply the standard methodology for determining security parameters for tree-based ORAMs [45] for our oblivious data structures. Specifically, we warm-up our underlying position-based ORAM with 16 million accesses. Then, due to the observation that time average is equal to ensemble average for regenerative processes, we simulate a long run of 1 billion accesses, and plot the stash size against the security parameter λ. Since we cannot simulate "secure enough" values of λ, we simulate for smaller ranges of λ, and extrapolate to the desired security parameter, e.g., $\lambda = 80$.

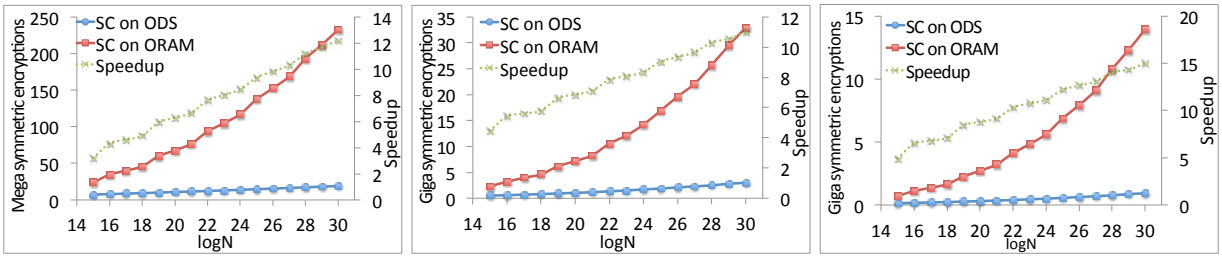
6.2 Oblivious Data Structures

Bandwidth blowup results. Figure 5 shows the bandwidth blowup for different data structures when



(a) **Client storage for our oblivious map/set and the general ORAM-based approach.** $N = 2^{20}$, payload = 64Bytes. 85%-95% of the client-side storage in our ODS is due to the overflow stash of the underlying position-based ORAM.

(b) **Secure Computation over Oblivious AVL Tree vs. ORAM.** Here we consider bandwidth for transferring garbled circuits and OT. Payload = 32 bits.



(a) Oblivious Stack Operations

(b) Oblivious Map/Set Operations

(c) Oblivious Priority Queue Operations

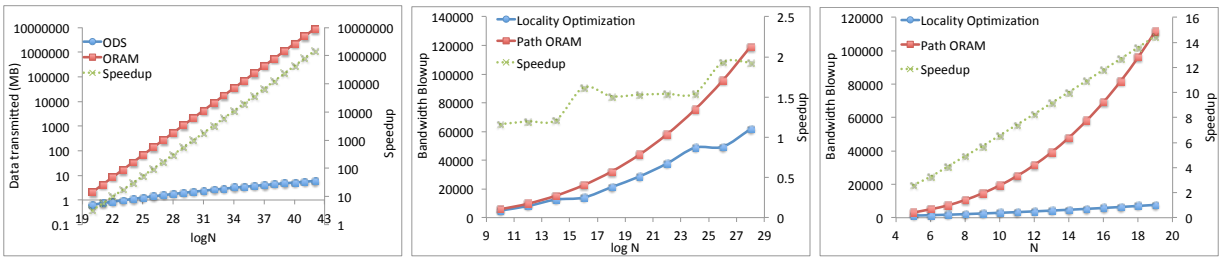
Figure 7: **Secure Computation over ODS vs. ORAM** Payload = 32 bits. The speedup curve has the y-axis label on the right-hand side.

payload is 64 Bytes. Note that because of the similarity in the implementation of Oblivious Stack and Oblivious Queue, their bandwidth blowup is the same; and therefore they share the same curve.

As shown in the Figure 5, the blowup for our ODS grows linear with $\log N$, which confirms the result shown in Table 1. The red curves show the blowup of naively building data structures over recursive ORAM. The graphs show that our oblivious data structure constructions achieve 4×–16× speedup in comparison with Path ORAM, for a data structure containing $N = 2^{30}$ nodes.

The aforementioned bandwidth blowup results are obtained while consuming a small amount of client-side storage. We evaluated the client size storage with maximum 2^{20} nodes and payload 64 Bytes for different oblivious data structures, and the results are shown in Figure 6a. The client-side storage is split between 1) the cache consumed by the ODS client (dark grey part in the bar plot); and 2) the overflow stash required by the underlying Path ORAM (light grey part in the bar plot), which depends on the security parameter. In Figure 6a, we present client storage needed with security parameter of 80, 128 and 256. For each value, we plot the space needed for oblivious data structure and naively building over recursive ORAM side by side. Result for other data structures can be found in online full version [49].

Secure computation results. Figure 7 shows the result for our secure computation setting. In these experiments, the payload is 32-bit integers, and for applicable data structures, the key is also 32-bit integers. The results show that our oblivious data structures achieve 12×–15× speedup in comparison with a general ORAM-based approach. We also compare the amount of data transmitted in secure computation for our oblivious AVL tree and AVL tree built over ORAM directly in Figure 6b. For $\log N = 20$, the amount of



(a) **Data transferred in for dynamic memory allocation.**

(b) **Bandwidth blowup for Random Walk on Grid.** Payload size is 16 bits

(c) **Bandwidth blowup for shortest path queries on planar graphs.** Payload size is 64 bits

Figure 8: **Applications of oblivious data structures in comparison with general ORAM.** The y-axis of speedup curve is on right side. Figures evaluate the bandwidth consumption for a secure processor or cloud outsourcing application.

data transmitted is reduced by 9×.

6.3 Evaluation Results for Case Studies

In this section, we provide results for additional case studies. In Section 5, we have explained the algorithms for these case studies, and their asymptotic performance gains.

Evaluation for dynamic memory allocation. We empirically compare the cost of our dynamic memory allocator with the baseline chunk-based approach. As explained in Section 5, our construction achieves exponential savings asymptotically, in comparison with the naive approach.

Figure 8a plots the amount of data transferred per memory allocation operation. We observe that our oblivious memory allocator is 1000 times faster than the baseline approach at memory size 2^{30} (i.e., 1 GB).

Evaluation for random walk. For the random walk problem, we generate a grid of size $N \times N$, where N goes from 2^{10} to 2^{20}. Each edge weight is a 16-bit integer.

Figure 8b plots the bandwidth blowup versus $\log N$. We observe a speedup from 1× to 2× as N varies from 2^{10} to 2^{28}. The irregularity at $\log N = 16, 25$ is due to the rounding-up for computing $\sqrt{\log N}$. At $\log N = 16, 25$, $\sqrt{\log N}$ is an integer, i.e., no rounding.

Evaluation for shortest path queries on planar graphs. We evaluate the shortest path query answering problem over a grid, which is a planar graph. The grid is of size $N \times N$, where $N + 1$ is a power of 2. As described in Section 5, we build a separation tree for the planar graph graph, and each tree node corresponding to a subgraph. By choosing N such that $N+1$ is a power of 2, it is easy to build the separation tree so that all subgraphs corresponding to nodes at the same height in the tree have the same size, which are either rectangles or squares. We build the tree recursively until the leaf nodes correspond to squares of size smaller than $\log^2(N + 1)$.

We compare our optimized approach as described in Section 5 with the baseline approach which uses Path ORAM directly. Figure 8c illustrates the bandwidth blowup for the two approaches as well as the speedup for $\log(N + 1)$ ranging from 5 to 19. We can observe a speedup from 2× to 14×.

7 Conclusion and Future Work

We propose oblivious data structures and algorithms for commonly encountered tasks. Our approach outperforms generic ORAM both asymptotically and empirically. Our key observation is that real-world program has predictability in its access patterns that can be exploited in designing the oblivious counterparts. In our future work, we plan to implement the algorithms proposed in this paper. In particular, we wish to offer an oblivious data structure library for a secure multi-party computation.

Acknowledgments. We would like to thank Jonathan Katz for numerous helpful discussions. We gratefully acknowledge the anonymous reviewers for their insightful comments and suggestions. This work is supported by several funding agencies acknowledged separately on the front page.

References

[1] Hardware AES showdown - via padlock vs intel AES-NI vs AMD hexacore. http://grantmcwilliams.com/tech/technology/item/532-hardware-aes-showdown-via-padlock-vs-intel-aes-ni-vs-amd-hexacore.

[2] Arbitman, Y., Naor, M., and Segev, G. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Automata, Languages and Programming* (2009), S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. Nikoletseas, and W. Thomas, Eds., vol. 5555 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 107–118.

[3] Askarov, A., Zhang, D., and Myers, A. C. Predictive black-box mitigation of timing channels. In *CCS* (2010), pp. 297–307.

[4] Assouad, P. Plongements lipschitziens dans $\mathbf{R}^n$. *Bull. Soc. Math. France 111*, 4 (1983), 429–448.

[5] Bar-Yossef, Z., and Gurevich, M. Random sampling from a search engine's index. *J. ACM* (2008).

[6] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *S & P* (2013).

[7] Berg, M. d., Cheong, O., Kreveld, M. v., and Overmars, M. *Computational Geometry: Algorithms and Applications*. 2008.

[8] Blanton, M., Steele, A., and Aliasgari, M. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS* (2013).

[9] Blelloch, G. E., and Golovin, D. Strongly history-independent hashing with applications. In *FOCS* (2007).

[10] Chung, K.-M., Liu, Z., and Pass, R. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. `http://arxiv.org/abs/1307.3699`, 2013.

[11] Clarkson, K. L. Nearest neighbor queries in metric spaces. *Discrete Comput. Geom. 22*, 1 (1999), 63–93.

[12] Damgård, I., Meldgaard, S., and Nielsen, J. B. Perfectly secure oblivious RAM without random oracles. In *TCC* (2011).

[13] Eppstein, D., Goodrich, M. T., and Tamassia, R. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS* (2010).

[14] Fletcher, C. W., Dijk, M. v., and Devadas, S. A secure processor architecture for encrypted computation on untrusted programs. In *STC* (2012).

[15] Ford, Jr., L. R., and Fulkerson, D. R. Maximal flow through a network. In *Canadian Journal of Mathematics* (1956).

[16] Gentry, C., Goldman, K. A., Halevi, S., Jutla, C. S., Raykova, M., and Wichs, D. Optimizing ORAM and using it efficiently for secure computation. In *PETS* (2013).

[17] Goldreich, O. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC* (1987).

[18] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious RAMs. *J. ACM* (1996).

[19] Goodrich, M. T., and Mitzenmacher, M. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP* (2011).

[20] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. Oblivious RAM simulation with efficient worst-case access overhead. In *CCSW* (2011).

[21] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. Practical oblivious storage. In *CODASPY* (2012).

[22] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA* (2012).

[23] Goodrich, M. T., Ohrimenko, O., and Tamassia, R. Data-oblivious graph drawing model and algorithms. *CoRR abs/1209.0756* (2012).

[24] Gordon, S. D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., and Vahlis, Y. Secure two-party computation in sublinear (amortized) time. In *CCS* (2012).

[25] Gupta, A., Krauthgamer, R., and Lee, J. R. Bounded geometries, fractals, and low-distortion embeddings. In *FOCS* (2003), pp. 534–543.

[26] Huang, Y., Evans, D., Katz, J., and Malka, L. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium* (2011).

[27] Islam, M., Kuzu, M., and Kantarcioglu, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS* (2012).

[28] Keller, M., and Scholl, P. Efficient, oblivious data structures for mpc. Cryptology ePrint Archive, Report 2014/137, 2014. `http://eprint.iacr.org/`.

[29] Kushilevitz, E., Lu, S., and Ostrovsky, R. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA* (2012).

[30] Lipton, R. J., and Tarjan, R. E. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics* (1979).

[31] Liu, C., Hicks, M., and Shi, E. Memory trace oblivious program execution. In *CSF* (2013).

[32] Liu, C., Huang, Y., Shi, E., Katz, J., and Hicks, M. Automating efficient RAM-model secure computation. In *IEEE S & P* (May 2014).

[33] Lu, S., and Ostrovsky, R. Distributed oblivious RAM for secure two-party computation. In *TCC* (2013).

[34] Lu, S., and Ostrovsky, R. How to garble ram programs. In *EUROCRYPT* (2013).

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

[36] Micciancio, D. Oblivious data structures: applications to cryptography. In *STOC* (1997), ACM.

[37] Mitchell, J. C., and Zimmerman, J. Data-Oblivious Data Structures. In *STACS* (2014).

[38] Nikolaenko, V., Ioannidis, S., Weinsberg, U., Joye, M., Taft, N., and Boneh, D. Privacy-preserving matrix factorization. In *CCS* (2013).

[39] Ostrovsky, R., and Shoup, V. Private information storage (extended abstract). In *STOC* (1997).

[40] Pippenger, N., and Fischer, M. J. Relations among complexity measures. In *J. ACM* (1979).

[41] Ren, L., Fletcher, C., Yu, X., Kwon, A., van Dijk, M., and Devadas, S. Unified oblivious-ram: Improving recursive oram with locality and pseudorandomness. Cryptology ePrint Archive, Report 2014/205, 2014. `http://eprint.iacr.org/`.

[42] Shi, E., Chan, T.-H. H., Stefanov, E., and Li, M. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT* (2011).

[43] Stefanov, E., and Shi, E. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy (S & P)* (2013).

[44] Stefanov, E., Shi, E., and Song, D. Towards practical oblivious RAM. In *NDSS* (2012).

[45] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S. Path ORAM – an extremely simple oblivious ram protocol. In *CCS* (2013).

[46] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS* (2003).

[47] Thekkath, D. L. C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. Architectural support for copy and tamper resistant software. *SIGOPS Oper. Syst. Rev.* (2000).

[48] Toft, T. Secure data structures based on multi-party computation. In *PODC* (2011), pp. 291–292.

[49] Wang, X. S., Nayak, K., Liu, C., Chan, T.-H. H., Shi, E., Stefanov, E., and Huang, Y. Oblivious data structures. Cryptology ePrint Archive, Report 2014/185, 2014. `http://eprint.iacr.org/`.

[50] Williams, P., and Sion, R. Usable PIR. In *NDSS* (2008).

[51] Williams, P., and Sion, R. Round-optimal access privacy on outsourced storage. In *CCS* (2012).

[52] Williams, P., Sion, R., and Carbunar, B. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS* (2008).

[53] Williams, P., Sion, R., and Tomescu, A. Privatefs: A parallel oblivious file system. In *CCS* (2012).

[54] Zahur, S., and Evans, D. Circuit structures for improving efficiency of security and privacy tools. In *S & P* (2013).

[55] Zhang, D., Askarov, A., and Myers, A. C. Predictive mitigation of timing channels in interactive systems. In *CCS* (2011), pp. 563–574.

[56] Zhang, D., Askarov, A., and Myers, A. C. Language-based control and mitigation of timing channels. In *PLDI* (2012), pp. 99–110.

[57] Zhuang, X., Zhang, T., and Pande, S. Hide: an infrastructure for efficiently protecting information leakage on the address bus. *SIGARCH Comput. Archit. News 32*, 5 (Oct. 2004), 72–84.

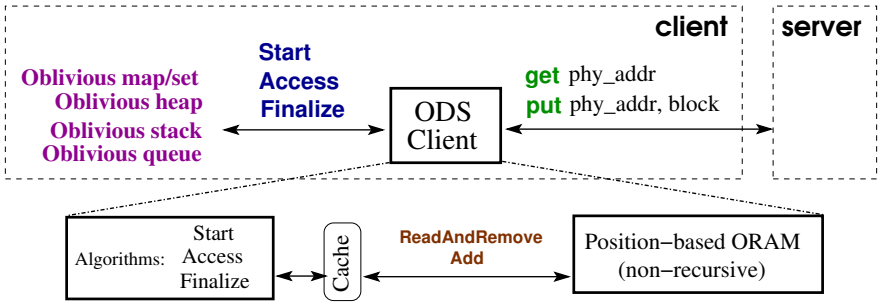


Figure 9: **Architectural overview.** Algorithms for oblivious map/set, heap, stack, queue, and others are implemented using calls to the ODS client. The ODS.Access call supports four types of operations, $(\mathsf{Read}, \mathsf{id})$, $(\mathsf{Write}, \mathsf{id}, \mathsf{data}^*)$, $(\mathsf{Insert}, \mathsf{id}, \mathsf{data}^*)$, and $(\mathsf{Del}, \mathsf{id})$,

A Proofs

of Lemma 1. We prove by induction on i. The initialization in lines 3 and 4 ensures that it holds for $i = 0$.

Suppose that the invariant holds for some $i \mod \rho = 0$. It suffices to show that the invariant holds for all iterations $j \in [i, i+\rho]$. Lines 8 to 11 ensure that all neighboring clusters of C_i are stored in the cache. Recall that these are the clusters whose centers are at distance at most 3ρ from the center of C_i. Hence, it suffices to show that for each $j \in [i, i+\rho]$, the center of the cluster containing j is at distance at most 3ρ from the center of C_i. This follows from the triangle inequality readily, because each node is at distance at most ρ from its cluster center, and $d_G(u_i, u_j) \leq j - i \leq \rho$. □

of Theorem 2. Observe that exactly 12^{dim} ORAM blocks are read from and written to the server at iteration $i \mod \rho = 0$. Hence, the security of the algorithm reduces to the security of the underlying ORAM.

Since we assume that each node contains at least $\Omega(\log N)$ bits, each cluster contains $\Omega(\log^2 N)$ bits. Since Path ORAM [45] has $O(\log N)$ blowup when the block size is at least $\Omega(\log^2 N)$ bits, it follows that the bandwidth blowup for our algorithm over $\rho = \Theta(\log^{\frac{1}{\mathsf{dim}}} N)$ iterations is $O(12^{\mathsf{dim}} \log^2 N)$. Observe that the $O(\log^2 N)$ initialization blowup in lines 3 and 4 can also be absorbed in this term. Therefore, the amortized blowup per access is $O(12^{\mathsf{dim}} \log^{2-\frac{1}{\mathsf{dim}}} N)$, and from the algorithm description, cache stores at most $2 \cdot 12^{\mathsf{dim}}$ clusters, which correspond to at most $O(12^{\mathsf{dim}} \log N)$ nodes. An additional client memory to store $O(\log^2 N) \cdot \omega(1)$ nodes is required for the stash in the underlying Path ORAM [45]. □

B ODS Architecture and Framework

In this section we will explain the general framework for dynamic oblivious data structures. Before that let's first define the memory abstraction used.

B.1 Memory abstraction.

The memory provides space to store the nodes in the aforementioned data structure graph. Each node $(\mathsf{id}, \mathsf{data})$ in this graph is an atomic unit of storage, where id can be regarded as the node's address in memory. For reasons that will be obvious later, we define the following four types of memory operations:

$$\mathsf{Read}(\mathsf{id}), \quad \mathsf{Write}(\mathsf{id}, \mathsf{data}^*), \quad \mathsf{Insert}(\mathsf{id}, \mathsf{data}^*), \quad \mathsf{Del}(\mathsf{id})$$

Any data structure algorithm can be expressed using these four types of memory operations plus computation steps in between memory operations (e.g., comparison of key values).

Note that in comparison, standard, generic ORAM typically supports only $\mathsf{read}(\mathsf{id})$ and $\mathsf{write}(\mathsf{id}, \mathsf{data}^*)$ operations as its memory abstraction, but not $\mathsf{insert}(\mathsf{id}, \mathsf{data}^*)$ $\mathsf{del}(\mathsf{id})$. The additional $\mathsf{insert}(\mathsf{id}, \mathsf{data}^*)$ $\mathsf{del}(\mathsf{id})$

operations are for the convenience of expressing dynamic data structure operations such as insertions and deletions.

B.2 ODS Architecture

ODS client. In the plain, non-oblivious version of data structures, memory operations go directly to memory, thereby leaking information through access patterns. In our oblivious data structures, we will replace the memory with a specially-designed "oblivious memory", referred to as the ODS client (see Figure 9). This ODS client's role is similar to that of an ORAM client; it offers exactly the same interface as memory (i.e., supporting the aforementioned four operations) from the perspective of the data structure program. Internally, the ODS client translates the logical memory requests to a sequence of obfuscated memory accesses, and reveals the obfuscated physical addresses to the untrusted server.

Obviously, one naive but inefficient way is to simply use a generic ORAM client as the ODS client. However, we will construct a customized ODS client that brings asymptotic speedup.

Untrusted server. The untrusted server supports a simple interface of getphys_addr and putphys_addr, data. The server should not be able to learn any information from the sequence of physical addresses accessed.

Figure 9 gives a high-level overview of our oblivious data structure constructions. Note that we use the client/server terminology in this paper for simplicity, while in other contexts such as secure processors, the parties will alternatively be referred as the trusted CPU and the untrusted memory.

B.3 Dynamic access

By using this framework, it is very easy to build oblivious data structures. We give a fully detailed version in Figure 11 how ODS framework works. As shown in Figure 10, every (dynamic) data structure operation is implemented as follows:

- **A single** ODS.Start **call** to the ODS prepares the ODS for the beginning of this operation.
- **A sequence of** ODS.Access **calls to the ODS**. This tells ODS what nodes to fetch and remove from the server, and what local updates to make to those nodes. These nodes (up to $O(\log N)$ of them) are kept in the ODS client's cache before the ODS.Finalize call.

  The ODS.Access calls generate ReadAndRemove operations to the underlying position-based ORAM, and hence the physical accesses observed by the server are oblivious. Not every ODS.Access call will generate a ReadAndRemove operation, since nodes in cache can be directly returned. (Only lines 8, 17 and 25 in the ODS.Access algorithm generate ReadAndRemove operations. All other operations are served by cache.) However, such cache hit/miss behaviour does not leak information since later in the ODS.Finalize stage, the ODS client performs padding to ensure that every operation results in the same number of ReadAndRemove and Add calls to the underlying position-based ORAM.
- **A single** ODS.Finalize **call to the ODS.** At this point, the ODS client performs three tasks: 1) generates new pos for each of the cached nodes and updates childrenPos of all the cached nodes based on the (id, pos) of each node (lines 1 - 3 in ODS.Finalize); 2) write back the cached nodes to the server through Add calls to the position-based ORAM (line 7 in ODS.Finalize); and 3) perform padding to ensure that every data structure operation results in the same number of ReadAndRemove (line 4 in ODS.Finalize) and Add operations (line 9 in ODS.Finalize) to the underlying position-based ORAM.

Remark 1. *For this algorithm to work, it must hold that whenever a node is fetched from the server, its position tag must already exist in the client's cache, and thus no position map lookups are needed. Furthermore, as in the static case, every node must have only one parent, i.e., the access pattern graph is a bounded-fanout tree.*

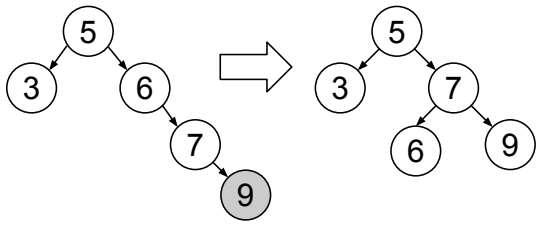
of Theorem 1. The proof is straightforward. Due to our padding strategy described in Section 3.2 any data structure operation, regardless of the opcode and operand, generates the same number of ReadAndRemove

```
rootPos:  contains the root of the data structure
data:  contains the data_node and childrenIDs
childrenPos:  map from id to pos for all children of a node

ODS.Start():
 1: Update cache to contain rootPos
ODS.Access(op, id, data*):
 1: if op = Insert then
 2:     cache.insert(id, data*, null, null) // No pos, childrenPos yet.  Updated during Finalize()
 3: else if op = Read then
 4:     if cache.contains(id) then
 5:         (id, data, pos, childrenPos) := cache.get(id)
 6:     else
 7:         pos := get pos from cache using childrenPos entries
 8:         (id, data, pos, childrenPos) := ReadAndRemove(id, pos)
 9:         cache.insert(id, data, pos, childrenPos)
10:     end if
11:     return data
12: else if op = Write then
13:     if cache.contains(id) then
14:         cache.update(id, data*) // pos, childrenPos remain the same
15:     else
16:         pos := get pos from cache using childrenPos entries
17:         (id, data, pos, childrenPos) := ReadAndRemove(id, pos)
18:         cache.insert(id, data, pos, childrenPos)
19:     end if
20: else if op = Del then
21:     if cache.contains(id) then
22:         cache.remove(id)
23:     else
24:         pos := get pos from cache using childrenPos entries
25:         ReadAndRemove(id, pos)
26:     end if
27: end if
ODS.Finalize(rootID, padVal):
 1: Generate UniformRandom pos for all nodes in cache
 2: Update childrenPos of all nodes based on their id and new pos
 3: Update rootPos based on the new rootID
 4: Pad ReadAndRemove() to padVal
 5: while not cache.empty() do
 6:     (id, data, pos*, childrenPos*) := cache.deleteNextElement()
 7:     Add(id, data, pos*, childrenPos*)
 8: end while
 9: Pad Add() to padVal
```

Figure 10: **ODS client algorithm for dynamic data structures**. The cache stores (id, data, pos, childrenPos) for all the nodes accessed during an operation. The cache is indexed on id and also allows queries on childrenPos maintained in each of the entries.

calls and Add calls to the underlying (non-recursive) position-based ORAM. Therefore, it is obvious that security reduces to that of the underlying position-based ORAM. □



(a) Inserting key 9 in the tree

1: ReadAndRemove($\mathsf{id}_5, \mathsf{pos}_5$)
2: ReadAndRemove($\mathsf{id}_6, \mathsf{pos}_6$)
3: ReadAndRemove($\mathsf{id}_7, \mathsf{pos}_7$)
4: `Pad ReadAndRemove() to maximum number of nodes required by any data structure operation`
5: Add($\mathsf{id}_6, ((6,0,0),(\mathsf{null},\mathsf{null})), \mathsf{pos}_6^*, (\mathsf{null},\mathsf{null})$)
6: Add($\mathsf{id}_9, ((9,0,0),(\mathsf{null},\mathsf{null})), \mathsf{pos}_9^*, (\mathsf{null},\mathsf{null})$)
7: Add($\mathsf{id}_7, ((7,1,1),(\mathsf{id}_6,\mathsf{id}_9)), \mathsf{pos}_7^*, (\mathsf{pos}_6^*, \mathsf{pos}_5^*)$)
8: Add($\mathsf{id}_5, ((5,1,2),(\mathsf{id}_3,\mathsf{id}_7)), \mathsf{pos}_5^*, (\mathsf{pos}_3, \mathsf{pos}_7^*)$)
9: `Pad Add() to maximum number of nodes required by any data structure operation`

(b) Requests sent by ODS.Access(), ODS.Finalize() to the server due to a cache miss

$$\begin{aligned}\mathsf{data} &= (\mathsf{data}_{\mathsf{node}}, \mathsf{childrenIDs}) \\ &= ((\mathsf{key}, \mathsf{l\_height}, \mathsf{r\_height}), (\mathsf{l\_id}, \mathsf{r\_id}))\end{aligned}$$

1: ODS.Start()
2: `// Operations to find the node`
3: **ODS.Access(Read, id_5, (null))**
4: **ODS.Access(Read, id_6, (null))**
5: **ODS.Access(Read, id_7, (null))**
6: `// Inserting the node`
7: ODS.Access(Insert, $\mathsf{id}_9, ((9,0,0),(\mathsf{null},\mathsf{null}))$)
8: `// Operations to balance`
9: ODS.Access(Read, $\mathsf{id}_9, (\mathsf{null})$)
10: ODS.Access(Read, $\mathsf{id}_7, (\mathsf{null})$)
11: ODS.Access(Read, $\mathsf{id}_6, (\mathsf{null})$)
12: ODS.Access(Read, $\mathsf{id}_7, (\mathsf{null})$)
13: ODS.Access(Read, $\mathsf{id}_6, (\mathsf{null})$)
14: ODS.Access(Read, $\mathsf{id}_7, (\mathsf{null})$)
15: ODS.Access(Write, $\mathsf{id}_6, ((6,0,0),(\mathsf{null},\mathsf{null}))$)
16: ODS.Access(Write, $\mathsf{id}_7, ((7,1,1),(\mathsf{id}_6,\mathsf{id}_9))$)
17: ODS.Access(Read, $\mathsf{id}_6, (\mathsf{null})$)
18: ODS.Access(Read, $\mathsf{id}_5, (\mathsf{null})$)
19: **ODS.Finalize(id_5, padVal = 10)**

(c) Operations generated for ODS.Access(). Bold fonts indicate requests to the server.

Figure 11: **Operations generated for insertion in an AVL Tree**

C Oblivious Data Structure Implementation

In this appendix, we detail the implementation of oblivious data structures including stack, queue, heap-based priority queue, AVL-tree-based map, doubly linked list, deque, and hash table.

C.1 Oblivious Stack

The oblivious stack implementation follows the same logic as linked list based stack in non-oblivious case. As shown in the algorithm Figure 12, top can be regarded as a pointer that is always pointing to the top of the stack.

C.2 Oblivious Queue

The implementation of oblivious queue is similar to oblivious stack. Here we also use the aforementioned pre-selection technique. Here tail contains the id from where the next element will be retrieved. Insertion happens at head. Details are shown in Figure 13.

Push($\mathsf{data}_{\mathsf{node}}$):
1: ODS.Start()
2: id := nextid()
3: ODS.Access(Insert, id, ($\mathsf{data}_{\mathsf{node}}$, top))
4: top := id
5: ODS.Finalize(top, padVal = 1)

Pop():
1: ODS.Start()
2: ($\mathsf{data}_{\mathsf{node}}$, nextid) := ODS.Access(Read, top, null)
3: ODS.Access(Del, top, null)
4: top := nextid
5: ODS.Finalize(top, padVal = 1)
6: **return** $\mathsf{data}_{\mathsf{node}}$

Figure 12: **Oblivious Stack**

head/tail: head/tail of the queue
Enqueue($\mathsf{data}_{\mathsf{node}}$):
1: ODS.Start()
2: head^* := nextid()
3: ODS.Access(Insert, head, ($\mathsf{data}_{\mathsf{node}}$, head^*))
4: head := head^*
5: ODS.Finalize(tail, padVal = 1)

Dequeue():
1: ODS.Start()
2: ($\mathsf{data}_{\mathsf{node}}$, tail^*) := ODS.Access(Read, tail, null)
3: ODS.Access(Del, tail, null)
4: tail := tail^*
5: ODS.Finalize(tail, padVal = 1)
6: **return** $\mathsf{data}_{\mathsf{node}}$

Figure 13: **Oblivious Queue**

C.3 Oblivious Priority Queue

In the implementation of oblivious priority queue, shown in Figure 14 and Figure 15, we follow the logic of array based heap implementation in non-oblivious case. One key difference is that, we do not have enough random accesses so accessing the first empty slot is done using the function readPath. However, this does not worsen the asymptotic performance.

C.4 Oblivious Map

We construct oblivious map using an AVL tree. Oblivious map would be similar to oblivious heap, except that there may be changes in the access pattern graph due to rotations. Essentially, rotations involve a number of node accesses where the parent of the next node to be accessed is always accessed earlier. Hence, we can use our general framework. The details of rotation are included in Figure 16.

The algorithm for insertion and find are similar to the non-oblivious case, although they result in oblivious accesses to the server. Details of can be found in Figure 17.

```
ReadPath():
 1: nodeid := 1 level := 1 list := ()
 2: while nodeid ≤ size do
 3:     (key) := ODS.Access(Read, nodeid, null)
 4:     append (key, nodeid) at the end of list
 5:     if level-th least significant bit in nodeid is 0 then
 6:         nodeid := nodeid ∗ 2
 7:     else
 8:         nodeid := nodeid ∗ 2 + 1
 9:     end if
10:     level := level + 1
11: end while
12: return list
Insert(key):
 1: ODS.Start()
 2: list := ReadPath()
 3: append (key, size) at the end of list
 4: size := size + 1
 5: for i := length(list) downto 2 do
 6:     if list[i].key > list[i − 1].key then
 7:         swap(list[i].key, list[i − 1].key)
 8:     end if
 9: end for
10: for i : length(list) downto 1 do
11:     ODS.Access(Write, list[i].nodeid, list[i].key)
12: end for
13: ODS.Finalize(1, padVal = 3 log N)
```

Figure 14: **Oblivious Heap: ReadPath and Insert**

C.5 Oblivious Doubly Linked List

If only one iterator is used in doubly-linked list at a time, we can achieve $O(\log N)$ blowup using Oblivious Stack with $O(\log N)$ blowup. Here the intuition is that we use one stack holding elements on the left side of the iterator and another stack to hold elements that are on the right side of the iterator and the client will hold the current element that the iterator is pointing to.

Following this idea, we can move the iterator left or right by doing push() and pop() on two stacks. In order to hide whether we are moving left or right, we may need to pad dummy accesses. In the worst case we need to do two stack operations i.e. two ReadAndRemove and Add, as shown in Figure 18.

Further optimization can reduce the number of position-based ORAM accesses to one based on the observation that one push operation and one pop operation can be done together using one position-based ORAM accesses, which corresponds to only one ReadAndRemove and one Add. Here, we also need to put the two stacks into one ORAM so that we can hide which stack we are operating on without doing dummy operations.

C.6 Deque

While using a position based ORAM, once a node is accessed, its position pos needs to be modified. Hence, the techniques discussed in Section 3.2 only work when all ancestor's of a node are stored in the client.

A deque primarily performs 4 operations - pushBack, pushFront, popBack, popFront. As a node can be accessed from both ends of deque, each node maintains a

```
ExtractMin():
 1: ODS.Start()
 2: list := ReadPath()
 3: key* := list[1].key
 4: list[1].key := list[length(list)].key
 5: size := size − 1
 6: for i : length(list) − 1 downto 1 do
 7:     ODS.Access(Write, list[i].nodeid, list[i].key)
 8: end for
 9: nodeid := 1
10: while nodeid ≤ size do
11:     key := ODS.Access(Read, nodeid, null)
12:     lkey := ODS.Access(Read, 2nodeid, null)
13:     rkey := ODS.Access(Read, 2nodeid + 1, null)
14:     if rkey < lkey then
15:         if key ≥ rkey then
16:             ODS.Access(Write, nodeid, rkey)
17:             nodeid := nodeid ∗ 2 + 1
18:         end if
19:     else if lkey ≤ rkey then
20:         if key ≥ lkey then
21:             ODS.Access(Write, nodeid, lkey)
22:             nodeid := nodeid ∗ 2
23:         end if
24:     end if
25:     ODS.Access(Write, nodeid, key)
26: end while
27: ODS.Finalize(1, padVal = 3 log N)
28: return key*
```

Figure 15: **Oblivious Heap: ExtractMin**

link to the previous and the next node. Thus, using position based ORAM would imply storing all the nodes on the client, which is not feasible.

Using recursion based ORAM: Storing all ancestors on the client is not a precondition for ORAM. Hence, we resort to using an ORAM for deque. However, if each node is stored separately on server, it would incur an $O(\log^2 N)$ blowup. We note that, for a deque, for any k consecutive operations, only the next k entries on either sides of the deque would be accessed. Thus, we group k deque nodes as one ORAM block. For reasons that will be clear later, we use $k = O(\log N)$. Hence, the original deque can be thought of as a smaller sized deque containing blocks.

The server stores most of the deque blocks. A few blocks (< 5) from either sides of the deque are cached by the client as a 'working set'. Both caches contain enough blocks to serve the next k operations. At the end of every k operations, if either cache is small (and cannot serve the next k operations), we read another block from the server and append it to the cache. However, if the cache size is too large, i.e. there are more than two blocks, we write a blocks to the server. Otherwise, we perform a dummy access on the server. For details, refer to the algorithm in Figure 20.

Security For every k operations of deque, the server observes 1 access on the ORAM, independent of what the k operations are. Hence, the server cannot distinguish between different operations performed on deque.

```
RotLC(id):
1: (key, ht1, ht2, l, r) := ODS.Access(Read, id, null)
2: (key′, ht1′, ht2′, l′, r′) := ODS.Access(Read, l, null)
3: ODS.Access(Write, id, ((key, ht2′, ht2), (r′, r)))
4: new_h := max{ht2′, ht2} + 1
5: ODS.Access(Write, id′, ((key′, ht1′, new_h), (l′, id)))
6: new_h := max{h1′, new_h} + 1
7: return (id′, new_h)
DbRotLC(id):
1: (key, ht1, ht2, l, r) := ODS.Access(Read, id, null)
2: (r′, h′) := RotRC(l)
3: ODS.Access(Write, id, ((key, ht1, h′), (l, r′)))
4: (new_id, new_h) := RotLC(id)
5: return (id′, new_h)
Balance(id):
1: (key, ht1, ht2, l, r) := ODS.Access(Read, id, null)
2: if ht1 − ht2 > 1 then
3:     (key, ht1l, ht2l, ll, rl) :=
       ODS.Access(Read, l, null)
4:     if ht1l ≥ ht1r then
5:         (new_node, new_h) := RotLC(id)
6:     else (new_node, new_h) := DbRotLC(id)
7:     end if
8: else if ht2 − ht1 > 1 then
9:     (key, ht1r, ht2r, lr, rr) :=
       ODS.Access(Read, r, null)
10:    if ht2l ≥ ht2r then
11:        (new_node, new_h) := RotRC(id)
12:    else (new_node, new_h) := DbRotRC(id)
13:    end if
14: end if
15: return (new_node, new_h)
```

Figure 16: **Balance functions for AVL Tree**

Bandwidth Blowup: We assume each node is of size $\Omega(\log N)$. As $k = O(\log N)$ nodes are grouped into 1 block, the ORAM block size is $O(\log^2 N)$. For blocks with size $O(\log^2 N)$, recursion based Path ORAM has a blowup of $O(\log N)$ as discussed in [45]. Thus, the total bandwidth for k blocks is $O(\log^3 N)$. In the non oblivious scenario, the total bandwidth for accessing $O(\log N)$ nodes is $O(\log^2 N)$. Hence, the bandwidth blowup for each node is $O(\log N)$.

C.7 Hash Table

Hash table with key as memory index and value as memory content is as powerful as RAM. Hence, improving the asymptotic performance of oblivious hash table will improve the asymptotic bound for ORAM, which is out of the scope of this paper. Here we want to design oblivious hash table with blowup close to blowup of ORAM. Here we describe a solution that gives an Oblivious Hash Table that is 3 times slower than the underlying ORAM used using cuckoo hash table.

In the originally proposed cuckoo hash table, a rehash can happen with non-negligible probability, which is not secure. However, [2] shows that by using a pending queue for elements to be inserted, we can do

```
rootID: id of the root
Find(key):
 1: ODS.Start()
 2: (key′, (l, r)) := FindPriv(key, rootID)
 3: ODS.Finalize(rootID, padVal = 1.44 ∗ 3 log N)
 4: return (key′, (l, r))
FindPriv(key, rootID):
 1: if rootID = null then
 2:     return null
 3: end if
 4: (key′, (l, r)) := ODS.Access(Read, rootID, null)
 5: if key = key′ then
 6:     return (key′, (l, r))
 7: else if key < key′ then
 8:     return FindPriv(key, l)
 9: else
10:     return FindPriv(key, r)
11: end if
Insert(key, rootID):
 1: ODS.Start
 2: (key′, (l, r)) := InsertPriv(key, rootID)
 3: ODS.Finalize(rootID, padVal = 1.44 ∗ 3 log N)
 4: return rootID
InsertPriv(key, rootID):
 1: if rootID = null then
 2:     id := nextid()
 3:     ODS.Access(Insert, id, ((key, 0, 0), (null, null)))
 4:     return (id, 0, 0)
 5: end if
 6: (key′, ht1, ht2, l, r) := ODS.Access(Read, id, null)
 7: if key < key′ then
 8:     ((ht1, ht2), id) := InsertPriv(key, l)
 9: else if key > key′ then
10:     ((ht1, ht2), id) := InsertPriv(key, r)
11: else
12:      ODS.Access(Write, id, ((key, ht1, ht2), (l, r)))
13: end if
14: return Balance(id)
```

Figure 17: **AVL Tree**

```
MoveLeft():
 1: if not cache.empty then
 2:     OStackR.Push(cache)
 3: end if
 4: cache := OStackL.Pop()
InsertLeft(data):
 1: OStackL.Push(data)
RemoveLeft():
 1: OStackL.Pop(data)
```

Figure 18: **Doubly-Linked List with one Iterator.** Uses two oblivious stacks defined in Appendix C.1

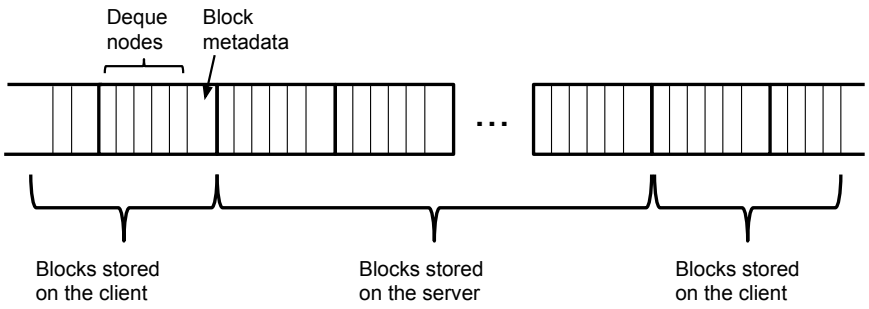


Figure 19: **Oblivious Deque**

constant amount of work for every insert and amortize the cost of each insert to a very small number. In practice, doing three accesses is enough for every operation. Combining this result with ORAM, we get an oblivious hashtable where each operation is translated to 3 ORAM accesses.

D Oblivious Dynamic Memory Allocator

In this section, we discuss an application of ODS to an important operating system task: memory management. We shall show that naive extensions to existing memory management algorithms using ORAM are not secure, since the total number of memory accesses may still leak information. We also show that padding in this case results in a secure but inefficient solution. Due to these reasons, we develop a new memory management algorithm which stores meta data in a tree-like structure, which can be implemented as an ODS, so that each memory allocation operation can be executed in the same time as one ORAM operation.

Scenario. The need for an oblivious memory allocator arises from the community's recent efforts at compiling programs to run on ORAM-capable secure processors [14, 35]. Programs may need to dynamically allocate (and free) memory, and then rely on the processor's ORAM features to obliviously access the allocated memory. Figure 3b illustrates this application scenario.

Problem and security requirement. A memory management task consists of two operations: dynamically allocating a portion of memory to programs at their request, and freeing it for reuse. In C convention, these two operations are denoted by `malloc`(l) and `free`(p). In some programs, the trace of memory allocation/free operations made will depend on sensitive inputs, e.g., due to conditionals with secret guards. Therefore, the security requirement of the oblivious memory allocator is that the physical addresses accessed to the allocator's working memory should not reveal any information about the opcode (i.e., `malloc` or `free`) or the operand (i.e., how many bytes to allocate and which address to free).

```
blocklength := log N
opcount := 0
dequeOperation(optype, data):
 1: opcount := opcount + 1
 2: if optype = insertL then
 3:     cacheL.PushNodeFront(data)
 4: else if optype = removeL then
 5:     data := cacheL.PopNodeFront()
 6: else if optype = insertR then
 7:     cacheR.PushNodeFront(data)
 8: else data := cacheR.PopFront()
 9: end if
10: if opcount = blocklength then
11:     opcount := 0
12:     if cacheL.BlockLength() < blocklength then
13:         block :=
           ORAMRead(cacheL[cacheL.size()].id_next)
14:         cacheL.PushBlockEnd(block)
15:     else if cacheL.BlockLength() > 2 * blocklength then
16:         id := nextid()
17:         cacheL[cacheL.size() − 1].id := id
18:         cacheL[cacheL.size() − 1].id_next :=
           cacheL[cacheL.size()].id
19:         cacheL[cacheL.size()].id_next := id
20:         ORAMWrite(cacheL[cacheL.size()])
21:         cacheL.PopBlockEnd(cacheL.size)()
22:     else Perform a dummy ORAM access
23:     end if
24:     // Similar case for cacheR
25: end if
26: if optype = removeL or optype = removeR then
27:     return data
28: end if
```

Figure 20: **Oblivious Deque.** Here, ORAMRead() and ORAMWrite() operations refer to recursion based ORAM.

Why naive methods fail. Most commonly adopted algorithms are based on chunks. The basic idea is to maintain a list of free chunks of memory using a doubly linked list. At the head of each chunk, there are several bytes (i.e. m bytes) to store some meta information, such as the size of the chunk, the pointers to the previous and next chunks. If two chunks are consecutive to each other, they can be merged into a larger chunk. The malloc(l) can be implemented as follows: the algorithm searches through the free chunk list for a chunk of size greater or equal to $l + m$. On failure, the algorithm simply returns NULL. On success, if the chunk's size is equal to $l + m$, then return that chunk. Otherwise, split the chunk into two with the first one having size $l + m$. The algorithm returns the first chunk, and add the second one back to the free chunk list. The free(p) operation simply adds the chunk corresponding to p into the free chunk list, and performs merging when appropriate. For simplicity, we assume that only one thread possesses the CPU, and so we do not consider issues raised by multi-thread execution.

Clearly, the free chunk list's size implicitly depends on the input of malloc and free function calls. Therefore an oblivious implementation should hide this information. Putting the memory into a big ORAM is insecure, since the malloc function call may scan through the whole free chunk list, and thus the number of its

memory accesses may leak its length. To make it oblivious, we should insert dummy memory accesses linear to the maximal free chunk list's size, which can be proportional to the whole memory size. This implies that such a naive approach requires memory accesses linear to the whole memory size to make it secure, and thus is very inefficient.

Construction. The algorithms are given in Figure 21 and Figure 22. The intuition is that if we treat the memory as a segment of size N, then the malloc and free functionalities are extracting a segment of a given size from the memory, and inserting a segment into the memory respectively. We construct a tree structure, whose nodes correspond to segments. The root of the tree corresponds to the segment $[0, N)$. Each node corresponding to $[a, b)$ where $a + 1 < b$ has two children corresponding to $[a, \frac{a+b}{2})$ and $[\frac{a+b}{2}, b)$ respectively. Nodes corresponding to $[a, a + 1)$ are leaf nodes. It is easy to see that the height of a segment tree of size N is $O(\log N)$. To implement the two operations, denoted as extract(l) and insert(a, b) respectively, we store auxiliary information (m, m_l, m_r) on each node corresponding to $[a, b)$, where m is the length of the longest unextracted segment in $[a, b)$, m_l is the length of the longest unextracted segment to the left of $[a, b)$, i.e. $[a, a + m_l)$ is unextracted, m_r is the length of the longest unextracted segment to the right of $[a, b)$, and p indicates whether the entire segment $[a, b)$ is extracted. Figure 23 illustrates a segment tree of size 8. Close to each none-leaf node, a tag in the form $[a, b)$ (m, m_l, m_r) indicates the segment and auxiliary information. A solid black nodes illustrate that the corresponding segments are extracted in full, while the hollow black nodes illustrate that the corresponding segments are not extracted in full. Notice if a node (e.g. node $[0, 2)$) is extracted, then all its successors should be extracted as well. In this case, the solid black node contain all the information of its successors. Therefore, we shallow those node illustrating that their information are not tracked until the segment is inserted back.

The extract(l) function contains two phases: the first phase searches for an unextracted segment of size l, and the second phase marks this segment as extracted and update the corresponding auxiliary information. The first phase first checks on root, if $m < l$, then extraction is impossible, so return null result directly. Otherwise, we know there must exist a segment of size $\geq l$. Suppose the node i corresponding to $[a, b)$ and the subtree rooted at i has such a segment (e.g. initially i is the root). Then there are three cases: (1) i's left children has a segment no shorter than l; (2) i's right children has a segment no shorter than l; (3) the longest segment overlaps both the left and right children of i. In the first two cases, the algorithm recursively search on the subtree. In the last case, the segment $[\frac{a+b}{2} - m_{li_1}, \frac{a+b}{2} - m_{li_1} + l)$ is unextracted, and thus the algorithm returns that immediately. Since at each level of recursion, the algorithms either goes one level deeper in the tree, or return the final result. Therefore the first phase requires $O(\log N)$ ODS accesses.

Suppose the found unextracted segment is $[a, b)$. The second phase is composed of two steps: (1) all segments covered by $[a, b)$ should be marked as extracted, i.e. set (m, m_l, m_r) to $(0, 0, 0)$; and (2) suppose the extraction happens at node i, then the auxiliary information of all ancestors of i should be updated. For the first step, as we mentioned earlier, if one segment is marked as extracted, then all its successors' auxiliary are not tracked until a later insert operation. Therefore the first step updates the auxiliary information of at most $O(\log N)$ nodes. Since one node's ancestors' number is not larger than the height of the tree, the second step can also be completed in $O(\log N)$ updates. Therefore, the second phase can be completed within $O(\log N)$ ODS accesses. Therefore the total number of ODS accesses for extract is $O(\log N)$, which leads to a time complexity of $O(\log^2 N)$.

Notice that the segment information is not changed during program's execution. Therefore the memory allocator can be implemented as either a static ODS or a dynamic ODS.

E Practical Considerations

In this appendix, we discuss further practical issues while implementing oblivious data structures.

E.1 When to Separate Payload and Index

For data structures such as AVL trees, each node may contain a key for performing comparisons, and some opaque payload data. If we attach the payload directly to every data structure node, it may incur extra

bandwidth blowup during the lookup – due to the need to retrieve and write back the payload for every intermediate node encountered during a lookup.

Therefore, when the payload is sufficiently large, we separate the storage of the index structures from the actual payload. The actual payload will be stored in another position-based ORAM separate from the the index ORAM. The index ORAM will store only keys used for comparison, and a position tag and an identifier for the actual payload (stored in the other ORAM). At the end of each lookup, the payload is requested and a separate call to the position-based ORAM is made to fetch the actual payload.

When the payload data is small, we directly attach the payload to the index structure, since storing the payload in a separate payload ORAM will incur the overhead of storing the id twice for each node, once in the index structure, and the other time in the payload structure.

E.2 Initializing and Resizing

As in all prior ORAM work, an upper bound N on the data structure size must be supplied by the program (or the caller of the ODS library) at initialization. Revealing an upper-bound on the size of the storage seems inevitable since the server must actually allocate storage to store the data. Dynamically resizing (increasing or shrinking size) during the lifetime of the data structure can also be supported, albeit potentially at the price of 1-bit leakage each time resizing is done – this is also similar to prior works on ORAM. One suggested method is to increase the storage by a factor of 2 when it becomes full or nearly full, or half its size when the utilization goes below a half. Resizing can be supported using techniques for resizing the underlying (non-recursive) position-based ORAM [45].

E.3 Choice of Underlying Position-based ORAM

As mentioned earlier, any (non-recursive) position-based ORAM will work in our constructions. In pratice, we can choose the underlying ORAM based on the specific application scenario. In a secure processor setting where client-side storage is stringent, Path ORAM [45] or other binary-tree based ORAM variants [10,16,42] may be more appropriate; while in a cloud outsourcing setting where the client can expend more local storage, one may choose a flat partitioning-based ORAM [43, 44] to tradeoff local storage for reduced bandwidth blowup.

E.4 Choice of Underlying Data Structure

Some data structures, such as maps or sets, can have various instantiations, e.g., using AVL tree, B+ tree, etc. In this paper, we chose AVL tree since we focus on optimizing the bandwidth blowup. However, it is conceivable that other implementations such as B+ tree might be more suitable for systems with different characterizations. For example, in scenarios where disk accesses are the bottleneck, it might be better to use B+ tree to best utilize the characteristics of a block-oriented storage.

E.5 Offering Various Levels of Security

An oblivious data structure library can offer options to the program to enable different levels of security. Depending on the context, sometimes a weaker security notion may suffice. Other times, stronger security notions are necessary.

Revealing type of operation. In some cases, we may not care about revealing what data structure operation is being performed. For example, consider a very simple straight-line program as below:

```
secret int key, val;
map<int, int> M;
int oldval = M[key];
M[key] = val;
```

Regardless of what the secret inputs (`key`, `val`) are, this program always makes a lookup operation to the map data structure, and then an insert operation (overwriting the previous value). In this case, revealing the type of data structure operation does not leak information about sensitive data.

This relaxed security notion can be formally defined by redefining the leakage function $\mathcal{L}$ in Definition 1 to include the type of data structure operations in addition to the total length M. If this relaxed security notion suffices, the oblivious data structure only needs to pad ReadAndRemove and Add operations with respect to the maximum of each type of operation, instead of across all operations.

Hiding between data structure instances. In some other scenarios, more stringent security requirements are necessary. Sometimes it may be necessary to hide which data structure instance is being operated on. For example, consider the following program that conditions on a secret variable `v`.

```
secret int v;
set<int> S;
priority_queue<int,int> Q;
if (v > 32)
    S.lookup(v);
else
    int a = Q.pop();
```

When hiding which data structure instance is necessary, we can do the following:

- Instead of separating each data structure instance into a different position-based ORAM instance, store all data structures in a single position-based ORAM. Because of this modification, all data structures nodes must now have unique ids across all data structure instances. One tricky issue arising from this is what block size to use, since each data structure instance may be storing various sized records. We give recommendations on this below in Section E.6.
- When performing padding, we must now pad each of ReadAndRemove and Add operations to the maximum across all data structures instances and all operations.

  However, this can potentially lead to an asymptotically worse performance. For example, we want to hide whether we are accessing an Oblivious Stack or Oblivious Map, then we have to pad every operation up to the worst case of Map operations. In this case, the bandwidth blowup of Oblivious Stack increases from $O(\log N)$ to $O(\log^2 N)$.

E.6 ORAM Block size for ODS

As mentioned, we maintain a position-based ORAM to store structure nodes for each data structure. By default, we set the block size of position-based ORAM to be the same as the size of a structure node. Using a smaller size for ORAM block will force us to split structure nodes, and thus we need to store and transfer extra id's with similar reason when we discuss where to put payload. If we use larger size for ORAM block, then there will be extra space wasted in each of the ORAM block.

If strong security is needed where we want to hide which data structure is being accessed, then we need to put all structure nodes into one position-based ORAM and all payload into another one position-based ORAM. We notice that all data structure nodes has a very small size, typically around 20 Bytes, and the difference in size among oblivious data structures is not big. So in case we have more than one type of data structures, we use the greatest size of structure node as ORAM block size.

F Dynamic Access Pattern Graph For Graphs With Low Doubling Dimensions

Our data structure can support limited insertion or deletion of nodes in the access pattern graph, as long as the doubling dimension of the resulting graph does not exceed a preset upper bound dim, and the following

conditions are satisfied.

1. *Node insertion.* A new node (together with its incident edges) can be inserted only as a neighbor of the current node in the access pattern graph; moreover, distances between existing nodes do not decrease.

2. *Node deletion.* A deletion only happens at the current node and a neighbor of the deleted node becomes the current node; moreover, distances between existing nodes do not increase.

Updates to Clustering. We describe how the clustering data structure is updated as a result of node insertion/deletion in the access pattern graph.

1. *Node insertion.* Since our assumption implies that distances between existing nodes do not change, we just need to consider which cluster contains the new node. Let u be the current node, and the new node x is added as a neigbor of u in the access pattern graph G. Suppose node v was visited the last time the cache is updated, and v is contained in the cluster with center z. Observe that $d_G(u, v) \leq \rho - 1$ and $d_G(v, z) \leq \rho$. Hence, if the new node x is within distance ρ from some existing cluster center $\widehat{z}$, we have $d_G(z, \widehat{z}) \leq d_G(z, v) + d_G(v, u) + d_G(u, x) + d_G(x, \widehat{z}) \leq \rho + (\rho - 1) + 1 + \rho = 3\rho$; therefore, we can add x to the cluster with center $\widehat{z}$ which is already in the cache. Note that since the doubling dimension does not increase, we are guaranteed that the cluster centering at $\widehat{z}$ is not full, and we replace a dummy node with x.

   On the other hand, if the new node x is not covered by any existing cluster center (because it is at distance more than ρ from every cluster center), then we can make x a new cluster center to cover itself. Observe that since $d_G(x, z) \leq 3\rho$, and the doubling dimension does not increase, we can replace a dummy cluster in the cache with the the cluster containing x.

2. *Node deletion.* Suppose x is the node to be deleted Recall that our assumption implies that deleting x does not change the distances between remaining nodes in the access pattern graph. If x is not a cluster center, we can simply remove x from the cluster containing x, which is already in the cache.

   If x is a cluster center, then we have to reassign nodes that are originally covered by x. Doing a similar analysis as in node insertion, suppose z is the cluster center of the visited node during the last cache update. Then, $d_G(x, z) \leq 2\rho$. Hence, if w is a node originally covered by x, then w might need to be covered by another center $\widehat{z}$, where $d_G(\widehat{z}, z) \leq d_G(\widehat{z}, w) + d_G(w, x) + d(x, z) \leq 4\rho$.

   Therefore, to ensure that all the modifications can be carried out in the cache, we modify the notion of neighboring clusters such that two clusters are neighbors if their centers are within distance 4ρ; this increases the number of clusters to be stored in the cache to $16^{\dim}$.

G Additional Evaluation Results

G.1 Bandwidth Blowup

As shown, all of them shows a linear speedup compared with naively building data structures over ORAM.

We show the bandwidth blowup for payload = 256 Bytes in Figure 24, and payload = 512 Bytes in Figure 25.

G.2 Client Storage

Here we show the client storage for different oblivious structures. As we can see from the figures, oblivious data structures usually use less client storage because we do not need to do recursions on ORAM. However, in the case of deque, where we are utilizing locality by having big cache, we have a greater client storage.

Update(id, L):

1: $\mathsf{left} \leftarrow id \times 2$; $\mathsf{right} \leftarrow id \times 2 + 1$
2: $(\mathsf{m}^1, \mathsf{m}^1_\mathsf{l}, \mathsf{m}^1_\mathsf{r}) := \mathsf{ODS.Access}(\mathsf{Read}, \mathsf{left})$
3: $(\mathsf{m}^2, \mathsf{m}^2_\mathsf{l}, \mathsf{m}^2_\mathsf{r}) := \mathsf{ODS.Access}(\mathsf{Read}, \mathsf{right})$
4: $\mathsf{m} \leftarrow \max\{\mathsf{m}^1, \mathsf{m}^2, \mathsf{m}^1_\mathsf{r} + \mathsf{m}^2_\mathsf{l}\}$;
5: $\mathsf{m}_\mathsf{r} := \mathsf{m}^2_\mathsf{r} = \mathsf{L}\ ?\ \mathsf{L} + \mathsf{m}^1_\mathsf{r} : \mathsf{m}^2_\mathsf{r}$
6: $\mathsf{m}_\mathsf{l} := \mathsf{m}^1_\mathsf{l} = \mathsf{L}\ ?\ \mathsf{L} + \mathsf{m}^2_\mathsf{l} : \mathsf{m}^1_\mathsf{l}$
7: $\mathsf{ODS.Access}(\mathsf{Write}, \mathsf{id}, (\mathsf{m}, \mathsf{m}_\mathsf{l}, \mathsf{m}_\mathsf{r}))$

PushDown1(a, b, a', b', id)

1: **if** $a' \le a \wedge b' \ge b$ **then**
2: $\quad \mathsf{ODS.Access}(\mathsf{Write}, id, (0, 0, 0))$
3: $\quad$ **return**
4: **end if**
5: **if** $a' > a \wedge b' < b$ **then**
6: $\quad$ **return**
7: **end if**
8: $\mathsf{PushDown1}(a, (a+b)/2, a', b', id \times 2)$
9: $\mathsf{PushDown1}((a+b)/2, b, a', b', id \times 2 + 1)$
10: $\mathsf{Update}(id, (b-a)/2)$;

PushDown2(a, b, a', b', id):

1: $(m, m_l, m_r) \leftarrow \mathsf{ODS.Access}(\mathsf{Read}, 1)$
2: **if** $a' \le a \wedge b' \ge b$ **then**
3: $\quad \mathsf{ODS.Access}(\mathsf{Write}, \mathsf{id}, (\mathsf{b}-\mathsf{a}, \mathsf{b}-\mathsf{a}, \mathsf{b}-\mathsf{a}))$
4: $\quad$ **return**
5: **end if**
6: **if** $a' > a \wedge b' < b$ **then**
7: $\quad$ **return**
8: **end if**
9: $\mathsf{PushDown2}(a, (a+b)/2, a', b', id \times 2)$
10: $\mathsf{PushDown3}((a+b)/2, b, a', b', id \times 2 + 1)$
11: $\mathsf{Update}(id, (b-a)/2)$

Insert(a, b):

1: $\mathsf{ODS.Start}()$;
2: $\mathsf{PushDown2}(0, N, a, b, 1)$;
3: $\mathsf{ODS.Finalize}(1, \mathsf{padVal} = 3 \log N)$

Figure 21: **Oblivious Memory Allocator (Part 1)**

```
Search(a, b, id, l):
 1: if a + 1 = b then
 2:     ODS.Access(Write, id, (0, 0, 0))
 3: return [a, b)
 4: end if
 5: (m, m_l, m_r) := ODS.Access(Read, id)
 6: left := id × 2; right := id × 2 + 1
 7: (m^1, m^1_l, m^1_r) := ODS.Access(Read, left)
 8: (m^2, m^2_l, m^2_r) := ODS.Access(Read, right)
 9: if m = b − a then
10:     L := (b − a)/2;
11:     ODS.Access(Write, left, (L, L, L))
12:     ODS.Access(Write, right, (L, L, L)
13: end if
14: if m^1 ≥ l then
15:     [a_r, b_r) := Search(a, (a + b)/2, left, l)
16: return [a_r, b_r)
17: else if m^2 ≥ l then
18:     [a_r, b_r) := Search((a + b)/2, b, right, l)
19: return [a_r, b_r)
20: else
21:     a′ := (a + b)/2 − m^1_r b′ := (a + b)/2 + l − m^1_r
22:     PushDown1(a, b, a′, b′, id) return [a′, b′)
23: end if
Extract(l):
 1: ODS.Start()
 2: (m, m_l, m_r) := ODS.Access(Read, 1)
 3: if m < l then
 4:     ODS.Finalize(1, padVal = 3 log N)
 5: return null
 6: end if
 7: [a, b) = Search(0, N, 1, l)
 8: ODS.Finalize(1, padVal = 3 log N)
 9: return [a, b);
```

Figure 22: **Oblivious Memory Allocator (Part 2)**

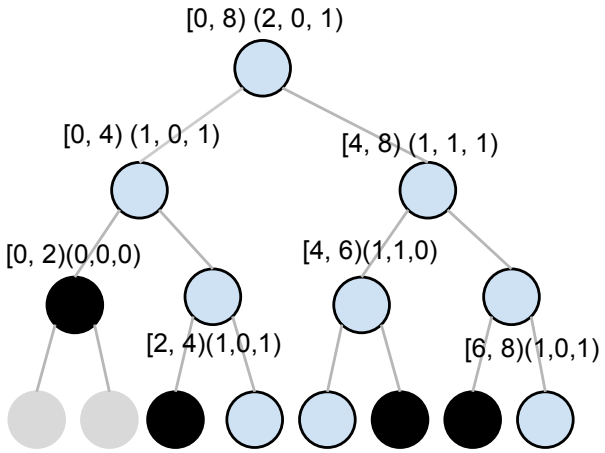
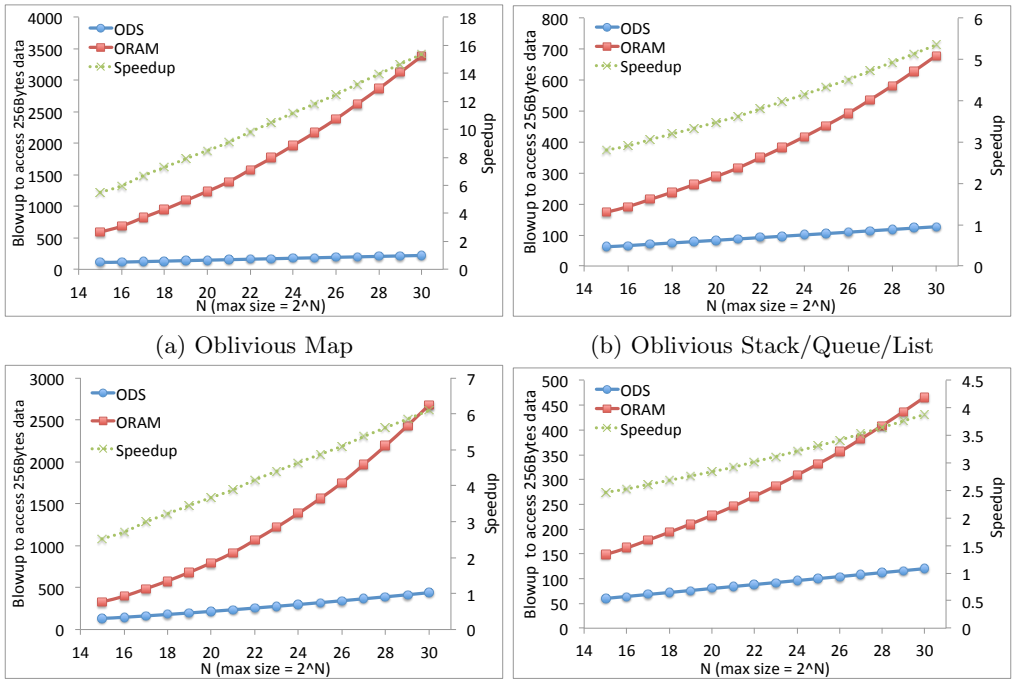


Figure 23: **Segment tree.** The segments as well as the auxiliary information are labeled close to the corresponding nodes. Solid nodes are extracted, while hollow ones are not. The two nodes in the left corner are in shade, since their parent is extracted, which implies they are extracted as well.



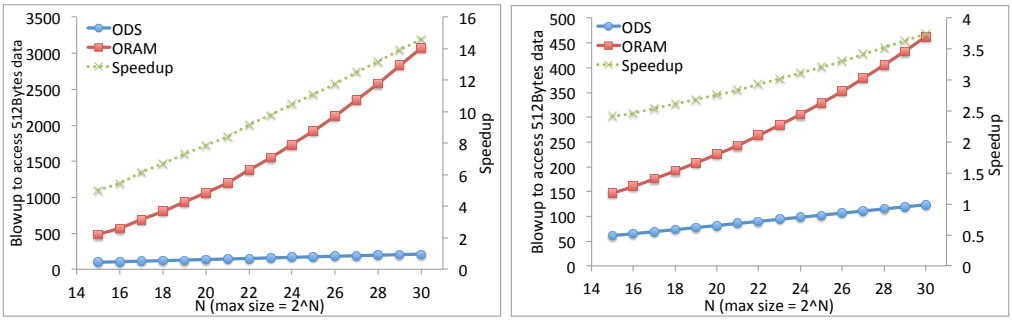
(a) Oblivious Map

(b) Oblivious Stack/Queue/List

(c) Oblivious Priority Queue

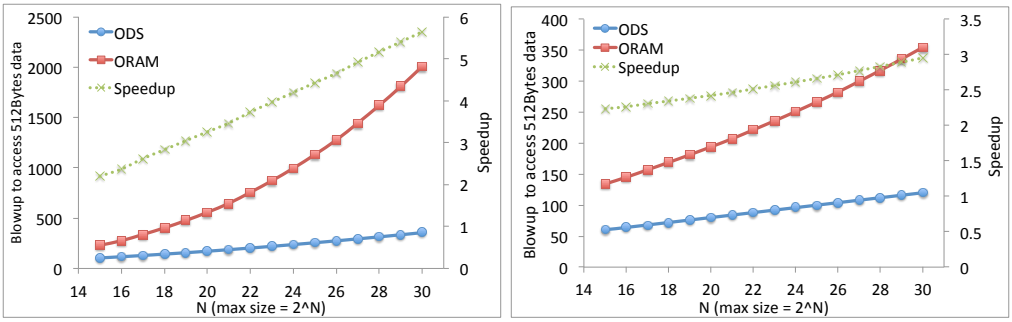
(d) Oblivious Deque

Figure 24: **Bandwidth blowup of various oblivious data structures in comparison with naive ORAM.** Payload = 256 Bytes. The speedup curve has the y-axis label on the right-hand side.



(a) Oblivious Map

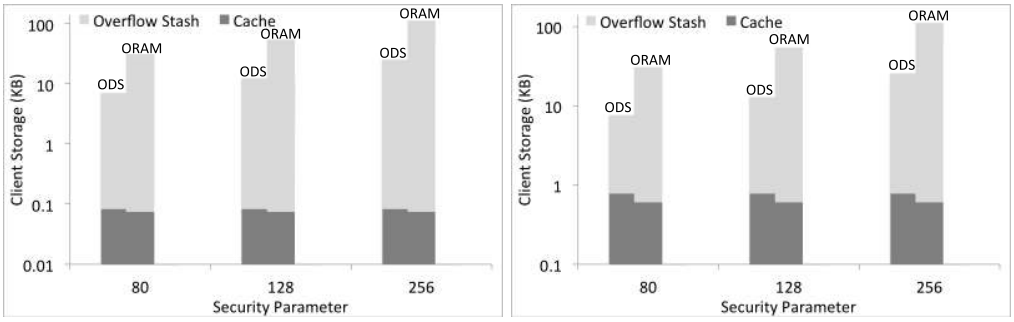
(b) Oblivious Stack/Queue/List



(c) Oblivious Priority Queue

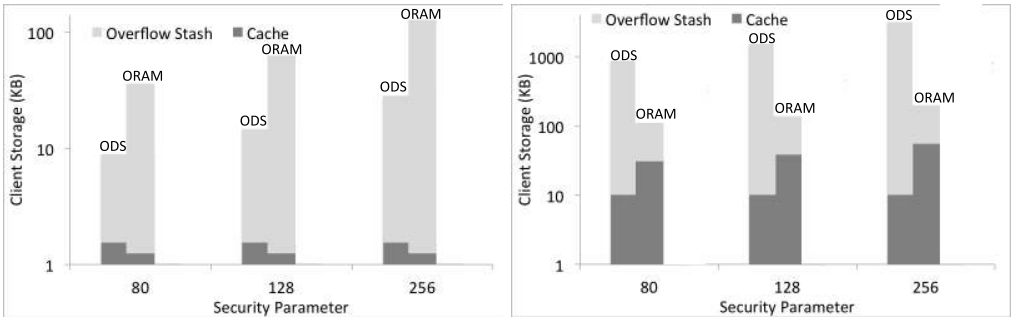
(d) Oblivious Deque

Figure 25: **Bandwidth blowup of various oblivious data structures in comparison with naive ORAM.** Payload = 512 Bytes. The speedup curve has the y-axis label on the right-hand side.



(a) Oblivious Stack/Queue/Doubly-Linked List

(b) Oblivious Priority Queue



(c) Oblivious Map/Set

(d) Oblivious Deque

Figure 26: **Client Storage with $N = 2^{20}$, payload = 64 Bytes**