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

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.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General-security and protection

Keywords

Security; Cryptography; Oblivious Algorithms

1. INTRODUCTION

It is known that access patterns, to even encrypted data, can leak sensitive information such as encryption keys [26, 56]. Furthermore, this problem of access pattern leakage is prevalent in numerous application scenarios, including cloud data outsourcing [43], design of tamper-resistant secure processors [13, 34, 45, 46, 56], as well as secure multi-party computation [23, 32, 33].

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 [17]. However, the community has started to more seriously investigate the possibility of making ORAMs practical only recently [18, 21, 41, 42, 44, 52]. 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 [42, 44, 52].

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 [28], 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* 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 [28, 44] for two different characterizations of sparse access graphs (see Table 1):

*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.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660314.

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 [44]	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 [28]	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 [13, 30, 34] 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 [14] 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 [44] 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 [2, 54, 55]. 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 [12,22] or efficient circuit structures [37, 53].

The work by Zahur and Evans [53] 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 [47] 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 [36] observe that Oblivious Turing Machine [39] 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 [7] 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 [30]. 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 [9, 11, 15–21, 28, 38, 41, 43, 49–51] 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 [32] 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) [8, 35]. 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 [27] implemented secure oblivious data structures using both the binary-tree ORAM [41], and the Path ORAM [44]. They leverage the technique suggested by Gentry *et al.* [15] 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 [44], 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.* [15] 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 map [9, 15, 41, 44] 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 [44] achieves the least

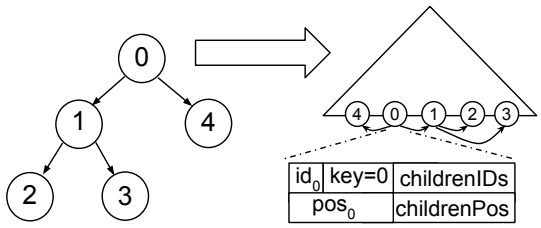


Figure 1: **Static oblivious binary search tree.** A logical binary search tree is on the left, and on the right-hand side is how these nodes are stored in a (non-recursive) position-based ORAM. Every position tag specifies a path to a leaf node on the tree. Every parent points to the position tags of 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.* [15] for performing binary search with ORAM.

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 [41, 44], or a partition [43]) 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 [9, 15, 43, 44] 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 [41, 43] 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.* [15] 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 [48].

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, 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 $O(\log N)$.

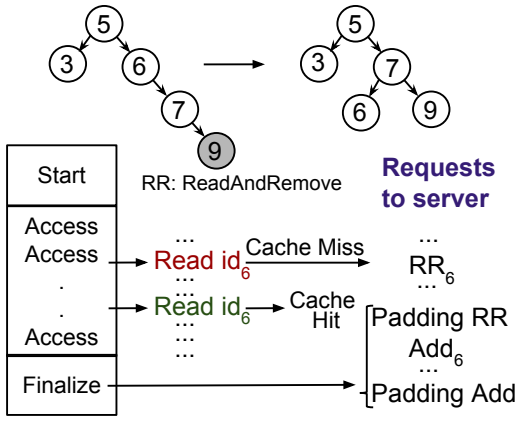


Figure 2: **Operations generated for insertion in an AVL Tree.** Cache hit/miss behavior does not leak information due to the padding we perform to ensure that every operation, regardless of the opcode and operands, has an equal amount of ReadAndRemove and Add calls to the position-based ORAM.

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 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 ReadAndRemove operations and the maximum number of Add operations is $3 \times \lceil 1.45 \log(N+2) \rceil$. It should be noted that ReadAndRemove is padded before a real Add happens and hence all 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 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.

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 [48].

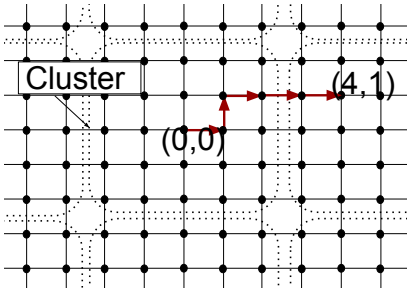


Figure 3: **Accesses for 2 dimensional structure exhibiting locality.** The red arrows indicate the path of the random walk for one set of $\sqrt{\log N}$ accesses.

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. [48].

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 [44]. 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.

- 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 3. 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* [3, 10, 24].

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 [24]). *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.

- *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 [48].

Finally, note that Ren *et al.* [40] 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 5 illustrates this architecture[1].

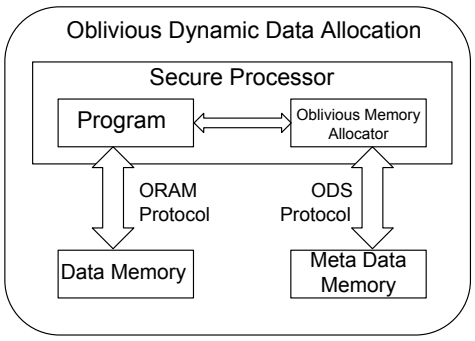


Figure 5: **Oblivious dynamic memory allocation.** For simplicity, imagine that both the program and the oblivious memory allocator reside in the secure processor's instruction cache. Securely executing programs (or a memory allocator) that reside in insecure memory is discussed in orthogonal work [30] and outside the scope here.

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 [6]. 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 [48].

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.

We present an alternative approach using the planar separator theorem [29]. 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.

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

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.**

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. [7], 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 [7], we consider the standard framework by Ford and Fulkerson [14] 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 [7], 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 [7], for sparse graphs, even when $C = 2^{|V|}$.

Random walk on graphs with bounded doubling dimension. In many computer science applications (e.g., [4]), 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

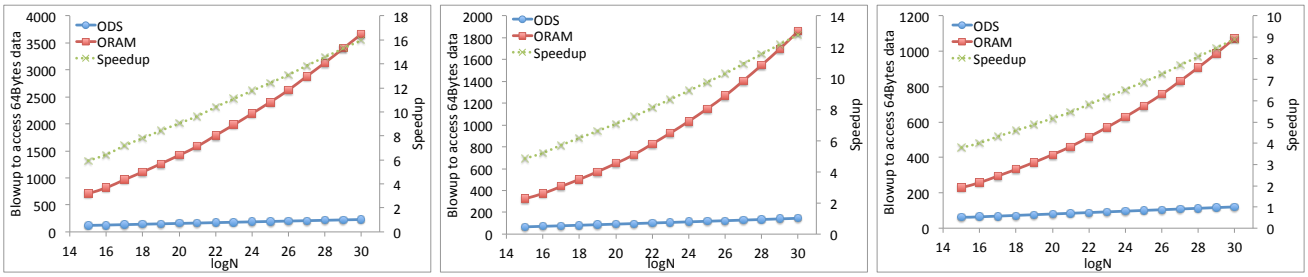
We evaluate the bandwidth blowup for our oblivious data structures with various application scenarios in mind, including cloud outsourcing [43], secure processors [13,34], and secure computation [15,23,31]. Below we explain the metrics we focus on, and our methodology in obtaining the performance benchmarks.

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 [44] 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



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

Figure 6: **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.

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 [31], which further builds on the FastGC garbled circuit implementation by Huang. et al. [25], incorporating frameworks by Bellare*et al.* [5]. We use the binary-tree ORAM by Shi *et al.* [41] in our ODS implementation as the position-based ORAM backend. Similarly, we compare with an implementation of the binary-tree ORAM [41]. 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.* [23] 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 [44] 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., $\lambda = 80$.

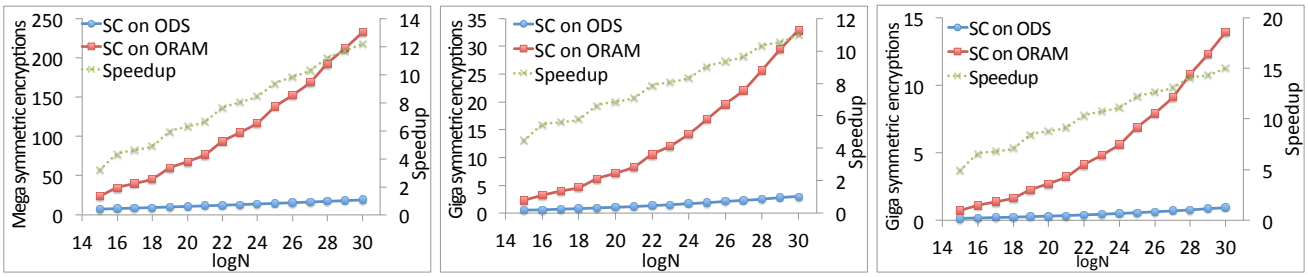
6.2 Oblivious Data Structures

Bandwidth blowup results. Figure 6 shows the bandwidth blowup for different data structures when 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 6, 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 $4\times$–$16\times$ 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 7. 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 7, 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 [48].

Secure computation results. Figure 8 shows the result for our secure computation setting. In these experiments, the payload is 32-bit integers, and for applicable data struc-



(a) Oblivious Stack Operations (b) Oblivious Map/Set Operations (c) Oblivious Priority Queue Operations

Figure 8: **Secure Computation over ODS vs. ORAM** Payload = 32 bits. The speedup curve has the y-axis label on the right-hand side.

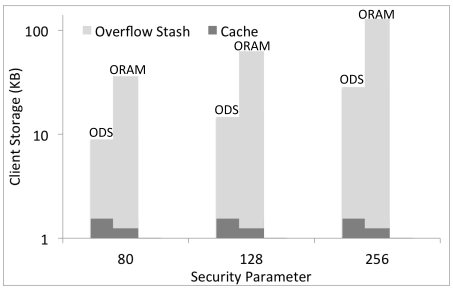


Figure 7: **Client storage for our oblivious map/set and the general ORAM-based approach.** $N = 2^{20}$, payload = 64Bytes. 85%-95% of the client-side storage in our ODS is due to the overflow stash of the underlying position-based ORAM.

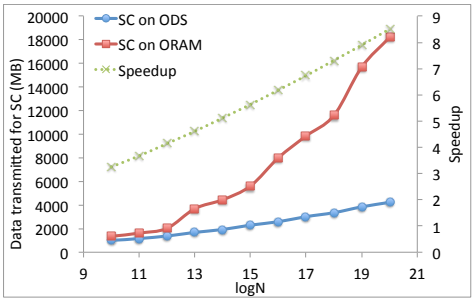


Figure 9: **Secure Computation over Oblivious AVL Tree vs. ORAM.** Here we consider bandwidth for transferring garbled circuits and OT. Payload = 32 bits.

tures, 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 9. For $\log N = 20$, the amount of 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 10a 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 10b 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.

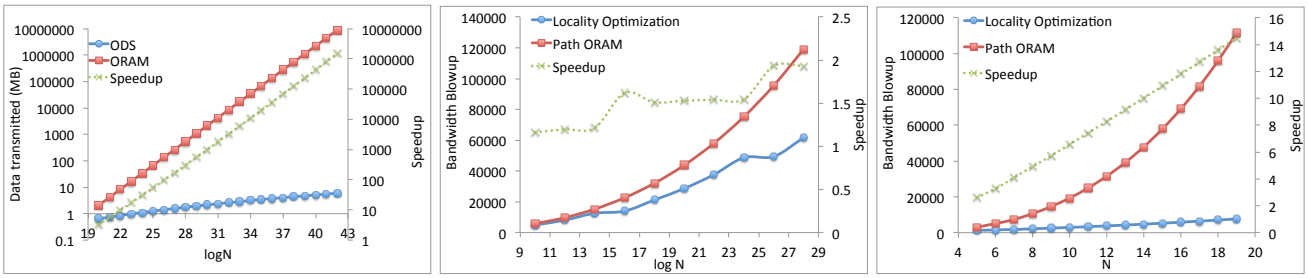
Evaluation for shortest path queries on planar graphs. We evaluate the shortest path query answering problem over a grid, which is a planar graph. The grid is of size $N \times N$, where $N + 1$ is a power of 2. As described in Section 5, we build a separation tree for the planar graph graph, and each 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 10c 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 ac-



(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 10: **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.

knowledge the anonymous reviewers for their insightful comments and suggestions. This work is supported by several funding agencies acknowledged separately on the front page.

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

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

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

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

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

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

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

[9] 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.

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

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

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

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

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

[15] 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).

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

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

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

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

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

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

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

[23] 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).

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

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

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

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

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

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

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

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

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

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

[34] 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).

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

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

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

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

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

[40] 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/.

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

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

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

[44] 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).

[45] 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).

[46] 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).

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

[48] 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/.

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

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

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

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

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

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

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

[56] 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.

APPENDIX

A. PROOFS

Proof 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 $d_G(u_i, u_j) \leq j - i \leq \rho$. □

Proof 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 [44] 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 [44]. □