Oblivious Single Access Machines

A New Model for Oblivious Computation

Ananya Appan, David Heath, and Ling Ren
{aappan2,daheath,renling}@illinois.edu

ABSTRACT

Oblivious RAM (ORAM) allows a client to securely outsource memory storage to an untrusted server. It has been shown that no ORAM can simultaneously achieve small bandwidth blow-up, small client storage, and a single roundtrip of latency.

We consider a weakening of the RAM model, which we call the *Single Access Machine* (SAM) model. In the SAM model, each memory slot can be written to at most once and read from at most once. We adapt existing tree-based ORAM to obtain an *oblivious SAM* (OSAM) that has $O(\log n)$ bandwidth blow-up (which we show is optimal), small client storage, and a single roundtrip.

OSAM unlocks improvements to oblivious data structures/algorithms. For instance, we achieve oblivious unbalanced binary trees (e.g. tries, splay trees). By leveraging splay trees, we obtain a notion of *caching ORAM*, where an access in the worst case incurs amortized $O(\log^2 n)$ bandwidth blow-up and $O(\log n)$ roundtrips, but in many common cases (e.g. sequential scans) incurs only amortized $O(\log n)$ bandwidth blow-up and $O(1)$ roundtrips. We also give new oblivious graph algorithms, including computing minimum spanning trees and single source shortest paths, in which the OSAM client reads/writes $O(|E| \cdot \log |E|)$ words using $O(|E|)$ roundtrips, where $|E|$ is the number of edges. This improves over prior custom solutions by a log factor.

At a higher level, OSAM provides a general model for oblivious computation. We construct a programming interface around OSAM that supports arbitrary pointer-manipulating programs such that dereferencing a pointer to an object incurs $O(\log d \log n)$ bandwidth blowup and $O(\log d)$ roundtrips, where d is the number of pointers to that object. This new interface captures a wide variety of data structures and algorithms (e.g., trees, tries, doubly-linked lists) while matching or exceeding prior best asymptotic results. It both unifies much of our understanding of oblivious computation and allows the programmer to write oblivious algorithms combining various common data structures/algorithms and beyond.

1 INTRODUCTION

Oblivious RAM allows a client to outsource memory to an untrusted server while hiding both the data being accessed and the memory access pattern, and thus provides a general framework for oblivious computation. The most important efficiency metrics of ORAM are the bandwidth blow-up and the number of roundtrips. Bandwidth blow-up is the number of blocks transferred between the client and the server for every block (unit of memory access) requested. One roundtrip is defined as a batch of read-then-write operations that can be dispatched in parallel. These costs are heavily affected by the block size and client storage assumed. In the typical setting, the client storage is small compared to the total memory size n, and the block size is $\Theta(\log n)$ bits.

Table 1: Comparison of our construction with existing ORAMs for a block size of $\Theta(\log n)$ bits. No existing ORAM construction simultaneously achieves optimal bandwidth blow-up, roundtrips and client storage.

	Bandwidth blow-up	Round-trips	Client storage	Server compute	Stat. secure
Circuit OSAM	$O(\lambda)$	$O(1)$	$O(1)$	✗	✓
Path OSAM	$O(\log n)$	1	$O(\lambda)$	✗	✓
Circuit ORAM [24]	$O(\lambda \log n)$	$O(\log n)$	$O(1)$	✗	✓
Path ORAM [23]	$O(\log^2 n)$	$O(\log n)$	$O(\lambda)$	✗	✓
OptORAMa [1]	$O(\log n)^\dagger$	$O(\log n)$	$O(1)$	✗	✗
SR-ORAM [26]	$O(\log^3 n)$	1	$O(1)$	✓	✗
$\sqrt{n}$-ORAM [10]	$\tilde{O}(\sqrt{n})$	1	$O(1)$	✗	✗

† With a hidden constant of $\approx 2^{228}$, later reduced to ≈ 9400.
- λ (typically 128) is the statistical security parameter. Some prior works set their statistical security parameters to $\omega(1) \log n$ for a negligible in n probability of failure. We choose to write it explicitly as λ to make the failure probability concrete.
- $\tilde{O}(\cdot)$ hides poly-logarithmic factors.

First proposed by Goldreich and Ostrovsky [10], numerous efforts have been made towards reducing the cost of ORAM, and the community has made encouraging progress [1, 2, 11, 12, 14, 17, 20, 23, 24]. But an overall efficient scheme remains elusive. Table 1 summarizes costs incurred by existing works. The recent breakthrough work of OptORAMa [1] achieves a bandwidth blow-up of $O(\log n)$, which is asymptotically optimal but has a very large hidden constant. The more practical tree-based ORAMs [23, 24] incur a sub-optimal bandwidth blow-up of $O(\log^2 n)$ (for $\Theta(\log n)$ block size) with small hidden constants. Moreover, all the above schemes require $O(\log n)$ roundtrips. Existing ORAM schemes that achieve constant roundtrips [8, 9, 26], on the other hand, require expensive server computation and incur high bandwidth blow-up with $\Theta(\log n)$ block size. A lower bound has been shown that a single-roundtrip ORAM (without server computation) must incur $\Omega(\sqrt{N})$ bandwidth blow-up or $\Omega(\sqrt{N})$ client storage [5] .

In sum, it is difficult to construct an ORAM that is optimal in *every* aspect. Thus, while ORAM provides a general framework for oblivious computation, it does not serve as an *efficient* general framework.

Our contribution. We introduce a new model of computation called the Single Access Machine (SAM) model. In short, a single access machine is a RAM, with the restriction that each memory address can be written to at most once, and read from at most once.

Because the capabilities of SAM are strictly weaker than that of RAM, OSAM is easier to instantiate than ORAM. We show that a straightforward simplification of existing tree-based ORAMs achieves OSAM that *is* optimal in every aspect: a single roundtrip, $O(\log n)$ bandwidth blow-up with small hidden constants, small client storage, and no server computation.

Table 2: Amortized bandwidth blowups and roundtrips of our OSAM-based solution as compared to existing practical oblivious solutions to the same problems. In all considered cases, our approach matches or exceeds the asymptotic performance of prior work.

Algo	Ours		Prior Work		
	Bandwidth	Rounds	Bandwidth	Rounds	Ref
DLL	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	[25]
BBST	$O(\log^2 n)$	$O(\log n)$	$O(\log^2 n)$	$O(\log n)$	[25]
Splay Tree	$O(\log^2 n)$	$O(\log n)$	$O(\log^2 n)^\dagger$	$O(\log^2 n)$	[1]
Trie	$O(\ell \cdot \log n)$	$O(\ell)$	$O(\ell \cdot \log n)^\dagger$	$O(\ell \cdot \log n)$	[1]
DFS/ BFS	$O(\lvert E\rvert \log \lvert E\rvert)$	$O(\lvert E\rvert)$	$O(\lvert E\rvert \log \lvert E\rvert)^\dagger$	$O(\lvert E\rvert \log \lvert E\rvert)$	[1]
SSSP	$O(\lvert E\rvert \log \lvert E\rvert)^\star$	$O(\lvert E\rvert)$	$O(\lvert E\rvert \log^2 \lvert E\rvert)$	$O(\lvert E\rvert \log \lvert E\rvert)$	[15]
MST	$O(\lvert E\rvert \log \lvert E\rvert)^\star$	$O(\lvert E\rvert)$	$O(\lvert E\rvert \log^2 \lvert E\rvert)$	$O(\lvert E\rvert \log \lvert E\rvert)$	[15]

- Acronyms - DLL : Doubly Linked List, BBST : Balanced Binary Search Tree, SSSP : Single Source Shortest Path, MST : Minimum Spanning Tree
- Symbol † indicates solving the problem with OptORAMa. In this case, the hidden constant is ≈ 9400.
- Symbol ★ denotes that we extend OSAM with a priority queue [19].
- A trie enables lookup of strings of arbitrary length; we use ℓ to denote the length of the search string.
- For graph algorithms, we consider arbitrary graphs, i.e., with arbitrary degrees and where $|E|$ and $|V|$ are independent.

Although more restrictive, a surprising variety of oblivious data structures and algorithms can be efficiently implemented in the SAM model. Table 2 summarizes some of our results.

Linear data structures. Oblivious stacks, queues, deques, linked lists, and doubly-linked lists can all be implemented using only $O(1)$ SAM operations per data structure operation, leading to optimal $O(\log n)$ bandwidth and a single roundtrip.

Balanced trees, arrays, and connections to ORAM. Tree-based data structures can also be implemented in the SAM model. For instance, balanced binary search trees can be implemented with $O(\log n)$ SAM operations per insertion/update/lookup. This also implies that we can use OSAM to implement an oblivious RAM at $O(\log^2 n)$ bandwidth blow-up and $O(\log n)$ roundtrips, matching Path ORAM.

Unbalanced trees and caching ORAM. More interestingly, OSAM can implement *unbalanced* binary trees with only $O(\log n)$ bandwidth blow-up. This allows us to achieve oblivious data structures including tries, as well as the fascinating *splay tree* [22]. Splay trees are known to have good *locality* properties (see discussion in Appendix C), where, for example, recently accessed elements can be more efficiently accessed a second time.

By using OSAM to implement a splay tree, we achieve a notion of *caching ORAM* that (1) has worst-case amortized $O(\log^2 n)$ bandwidth blow-up and $O(\log n)$ roundtrips, (2) has amortized $O(\log n)$ bandwidth blow-up and $O(1)$ roundtrips for many "common" access patterns, (3) is statistically secure, and (4) has constant factors similar to the best tree-based ORAMs.

Graph algorithms. We show that the SAM model extends beyond trees and captures common graph algorithms (for arbitrary graphs), including depth-first search (DFS) and breadth-first search (BFS). By augmenting the OSAM model with an oblivious priority queue from [19], we obtain new oblivious algorithms for the minimum spanning tree (MST) problem and the single source shortest path problem (SSSP, most famously solved by Dijkstra's algorithm). In all four of our oblivious graph algorithms, we incur a bandwidth-blowup of $O(|E| \log |E|)$ and $O(|E|)$ roundtrips, where $|E|$ is the number of edges. These algorithms outperform prior best custom solutions by a log factor and match commonly-used non-oblivious algorithms for those problems.

General pointer manipulating programs. More generally, the SAM model admits arbitrary pointer-manipulating programs. Dereferencing a pointer to access an object that has d incoming pointers incurs a cost of $O(\log d)$ SAM operations. When compiled to an OSAM program, the bandwidth blow-up and roundtrips are respectively $O(\log d \log n)$ and $O(\log d)$, which are significantly less than the $O(\log^2 n)$ bandwidth blow-up and $O(\log n)$ roundtrips incurred by practical tree-based ORAM (typically $d \ll n$).

Writing pointer-manipulating programs starting from bare-bones SAM operations can be tedious, so we provide an interface – which we call smart pointers – that handles the tedious details of enforcing the single-access rules and makes OSAM programs almost identical to their non-oblivious counterparts. In short, the smart pointer abstraction automatically handles the details needed to properly maintain more than one pointer to the same object.

2 BACKGROUND AND RELATED WORK

2.1 Oblivious RAM

Oblivious RAM (ORAM) allows a client to outsource its main memory to an untrusted server [10]. An ORAM can be thought of as a *compiler* that translates *logical* memory queries into *physical* queries to the server's memory, with the crucial security property that the server learns nothing other than the number of logical queries. At the highest level, ORAM clients achieve security by continually shuffling the server's encrypted memory content.

Tree-based ORAM and position maps. The ORAM constructions most related to our work are *tree-based ORAMs* [6, 17, 21, 24]. In state-of-the-art tree-based ORAMs [17, 23, 24], server memory is organized as a binary tree where each tree node holds up to a constant number of physical blocks. Each logical block is mapped to a leaf in the tree. The crucial invariant of a tree-based ORAM is that each logical block must reside somewhere on the path from the root to its mapped leaf. To read a logical block, the client reads that entire length-$O(\log n)$ path from the server to find the block of interest; this is guaranteed to succeed by the invariant. Once the read finishes, the client remaps the block to a fresh random leaf such that the same block can be securely queried again later. Lastly, the client performs an *eviction step*, where blocks in client memory are sent back to the server. The eviction step is carefully designed to have the same $O(\log n)$ asymptotic cost as reading a path.

The client stores which leaf each logical block is mapped to in a data structure called the *position map*. Ignoring the position map, state-of-art tree-based ORAM like Path ORAM incur $O(\log n)$ blow-up and a single roundtrip. However, the position map has linear (in n) size, so it is too big for the client to store. The solution to this issue is to *recursively* store the position map in another tree-based ORAM until the final position map is small enough to fit in client memory. This recursion step pushes the bandwidth blow-up of tree-based ORAM from $O(\log n)$ to $O(\log^2 n)$, and the roundtrips from

$O(1)$ to $O(\log n)$. Looking ahead, our OSAM saves a log factor in both metrics over a tree-based ORAM precisely because the weaker capability of single accesses obviates the need for a position map, and hence avoids recursion and its associated cost.

Hierarchical ORAM. While this work focuses on tree-based ORAM, the other major ORAM paradigm, known as *hierarchical ORAM*, is also interesting as it gave rise to OptORAMa [1], the first ORAM construction with asymptotically optimal $O(\log n)$ bandwidth blow-up. Its concrete performance, however, is prohibitive due to its use of an impractical primitive called "linear oblivious tight compaction". [7] improved tight compaction's hidden constant from $\approx 2^{228}$ to ≈ 9400, but the approach remains expensive. Recently, [3] showed practical improvements to OptORAMa, but at the cost of greatly increasing client storage.

2.2 Special-Purpose Oblivious Computations

In this section, we review prior works that construct special-purpose oblivious computations. [18] provides a good survey.

Oblivious Data Structures. [27] was one of the first works to study custom oblivious data structures, i.e., without using ORAM. They showed that stacks and queues can be implemented as small Boolean circuits, which can be handled in an oblivious manner.

[25] studied oblivious data structures using tree-based ORAM, and their work is closely related to ours. [25] also investigates cases where the position map can be removed. They give constructions for data structures based on linked lists and balanced binary trees, such as sets, maps, stacks, queues, and priority queues. They also show algorithms for graphs of *low doubling dimension*, which roughly means that the graph is a grid in a low dimensional space. Our approach is more general and handles *unbalanced* trees and *arbitrary* graphs. We discuss details of the [25] approach in Section 3.

Oblivious priority queue. Recently, [19] used tree-based ORAM techniques to construct an efficient oblivious priority queue. The author shows that each priority queue operation can be achieved at only $O(\log n)$ blow-up and $O(1)$ roundtrips.

It is easy to combine the priority queue of [19] with our OSAM construction since our OSAM is instantiated with a tree-based ORAM. By augmenting our OSAM with a priority queue, we attain efficient algorithms for graph SSSP and MST. We remark that it is the *combination* of OSAM and priority queues that enables these improved results.

Other works on special-purpose oblivious computation. [4] gave oblivious graph algorithms for BFS, DFS, SSSP, and MST at a bandwidth cost of $O(|V|^2)$ words. This is optimal for *dense* graphs where $|E| = \Theta(|V|^2)$ but not for general graphs where $|V|$ and $|E|$ can be independent.

[15] built a programming framework for secure computation. As an application of their framework, they implement oblivious algorithms for MST and SSSP with a blow-up of $O(|E| \log^2 |V|)$ and $O(|E| \log |E|)$ roundtrips, and for DFS with a bandwidth blow-up of $O(|V|^2 \log |V|)$ and $O(|V| \log |V|)$ roundtrips. Our oblivious algorithms for all of these incur $O(|E| \log |E|)$ blow-up and $O(|E|)$ roundtrips.

[16] presented a framework for implementing secure parallel algorithms for a class of data analytic algorithms such as computing a histogram using MapReduce, matrix factorization, and PageRank, but they do not solve the common graph traversal problems that we consider in this paper.

2.3 Notation

- λ denotes a statistical security parameter.
- n denotes the memory size in words.
- w denotes the *word size*. We set $w = \Theta(\log n)$ to ensure that words are big enough to index a memory while keeping communication low.
- A *block* is a unit of memory of size $\Theta(w)$ stored on the server.
- Jumping ahead, we distinguish a *single access machine* (SAM) from a *SAM program*. The program issues memory requests, and the machine satisfies them; see Sections 3 and 4.
- m denotes the number of memory requests issued. We assume $m = \text{poly}(n)$, and hence $\log m = \Theta(\log n) = \Theta(w)$.
- A *pointer* points to a *pointee*. A pointer has one pointee, but a pointee may have many pointers.

3 OVERVIEW

In this section, we sketch our techniques at a level sufficient to demonstrate the usefulness of the SAM model. Subsequent sections formalize all the details of our approach.

A point on notation: we will routinely use tree-based ORAM to implement tree-like data structures, so the terms "tree" and "path" are overloaded. Unless otherwise stated, the words "tree" and "path" will henceforth refer to those in the logical data structure to be implemented, not to those in the ORAM. That is, we abstract ORAM and ignore its tree-based structure. We will use the term *ORAM position* to abstractly represent a set of physical addresses on the server that a logical block is mapped to and may reside in.

3.1 Avoiding Position Maps; Review of [25]

Recall from Section 2.1 that in existing tree-based ORAM, the client maintains a structure called the *position map*, which maps each logical block to an ORAM position. The position map is linear in size and is recursively instantiated. This recursive position map blows up bandwidth and roundtrips by a log factor.

For particular oblivious computations, the position map can be removed, and a *non-recursive* ORAM suffices. Such cases were first studied in detail by [25]. The authors noticed that when the end goal of an oblivious computation is to implement a constant-degree rooted tree, the position map is not needed. The idea is to augment nodes in the tree such that each parent node stores a *pointer* to each of its children, and each pointer carries the ORAM position of the child. The client, who holds a pointer to the root, can traverse a tree path by simply chasing pointers stored in each node and storing the path in local memory.

When the client completes its traversal, it must write the path back to the server so that those same nodes can be accessed later. However, the security requirements of the ORAM force the client to write each node back to a *fresh* ORAM position. Thus, existing pointers to those nodes holding the old ORAM positions are *invalidated.* [25] observe that for tree-like structures, it is easy to eliminate invalid pointers, since all newly invalidated pointers lie on the path itself. Hence, the client simply writes back nodes starting from the leaf, and as it proceeds up the path, it updates pointers to each node with the *updated* ORAM positions of its children.

Limitations. While [25]'s approach opened the door to many improvements in oblivious computations, their approach is not fully general. The main limitation is that they cannot generate two pointers that point to the *same* memory block. In particular, imagine we would like to implement a graph-like structure where two nodes A and B each hold a pointer to some *shared* node C.

The challenge here is that if the client traverses a path from, say, A to C, the client must write A and C back to fresh ORAM positions so that they can be accessed again. But if the client does not also update B, then B holds an *invalid pointer* to C. If the client attempts to dereference the invalid pointer from B to C, it will *not* obtain the latest copy of C. Even worse, this dereference is *not secure*, since the server will observe two accesses to the same ORAM position. On the other hand, if the client *does* naïvely update B, then it must also update all predecessors of B with the new location of B, and this can cascade and require that the client access essentially all of memory. Note that tree-like structures circumvent this problem, since each node has only one incoming pointer.

Beyond the inability to handle shared pointers to nodes, the [25] approach is also limited in that they can only handle *balanced* trees. This second limitation emerges from the fact that the client stores entire tree paths in local memory, which must be small.

More generally, [25]'s approach only works for linked-list-like data structures and balanced trees. In this work, we are interested in a rich class of general pointer-manipulating programs.

3.2 SAM and OSAM

Our SAM model extends the capabilities of prior work. This section explains the interesting aspects of the model.

The SAM model centers on an interaction between a *SAM program* and the machine that it runs in. The SAM program itself is an arbitrary randomized algorithm, with the restriction that it runs in a small amount of space, e.g. $O(1)$ or $O(\lambda)$ words. If the program needs more memory, it must issue memory requests to the machine. The machine can hold any polynomial amount of memory. Looking ahead, the machine component of our *oblivious* SAM will further outsource all memory requests to an untrusted random access memory (i.e., the server).

The limitation of the SAM model is that for each of the machine's logical memory addresses i, the program can write to i at most once, and it can read from i at most once. This constraint is meant to capture limitations imposed by an ORAM: we can only write/read each ORAM position once. Before the SAM program can read or write a value, we insist that it first ask the machine to *allocate* an address. The machine can respond with an *arbitrary* fresh address (in our OSAM instantiation, an address encodes an ORAM position). We will see how this preallocation of addresses is useful shortly.

Jumping ahead, our definition of OSAM will require that any sequence of *Read*/*Write* operations (of the same length) should be indistinguishable, and our OSAM construction will require that for each *Read*/*Write* operation, the client will read/write $\Theta(\log n)$ words from the server.

A basic example; stacks. The basic way a SAM program can use machine memory is by allocating an address, writing to that address, and then later reading from it:

$$addr \leftarrow Alloc(\,), \ldots, Write(addr, val), \ldots, val \leftarrow Read(addr)$$

As an example, we can easily implement a program that achieves a stack data structure. To do so, the program can maintain a pointer to the top of the stack. Pushing/popping from the stack is a simple matter of issuing appropriate calls to *Alloc*/*Read*/*Write* and renaming variables:

```
def push(x):
    top' ← SAM.Alloc( )
    SAM.Write(top', { x, top })
    top ← top'

def pop( ):
    { x, top' } ←
        SAM.Read(top)
    top ← top'
    return x
```

Similarly, we can implement binary trees in SAM by storing pointers to child nodes in the parent nodes, as was done in [25].

Allocating before writing; unbalanced trees. So far, we have not shown additional capabilities as compared to prior work. Suppose we want to build an unbalanced binary tree. In [25], it was not possible to traverse an arbitrary path through such a tree, since client memory is bounded, and the path can be of arbitrary length.

In the SAM model, we can traverse paths of arbitrary length. The key to this is our decoupling of the *allocation* of an address from the *writing* to that address. Recall that the challenge of ORAM-based path traversal is that we must *rebuild* the path after we traverse it, since each pointer along the path will be invalidated. In the SAM model, we can rebuild the path *as we go*. More specifically:

- Suppose we (the program) start with an address *addr* that points to the tree root. We allocate a fresh address: $addr' \leftarrow Alloc(\,)$ to store the updated root.
- We call *Read*(*addr*) to load the root from machine memory, which invalidates *addr*. The machine returns a block that, in particular, holds addresses of child nodes.
- Depending on the details of the traversal algorithm, we choose some child address to read. Before we read that child we (1) allocate a new address $addr'' \leftarrow Alloc(\,)$ where the updated accessed child will be saved, (2) update the content of the root node to point to $addr''$, and (3) write the root node to $addr'$. Thus, we have proactively rebuilt the root node by updating it to point to where its child node *will be*, before anything actually resides there.
- From here, we can recursively traverse the child node, and so on, resulting in a full traversal of the target path.

The crucial point is that the program can traverse an arbitrary path through a tree while maintaining only a constant number of words of local memory; the program only needs to keep data corresponding to the current node under consideration.

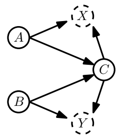
Section 6 formalizes our ability to handle unbalanced trees. Because we can handle arbitrary trees, we are able to handle oblivious tries and oblivious splay trees with better efficiency than prior work. Oblivious splay trees allow us to achieve an interesting notion of caching ORAM; see Section 6.

Sharing. Perhaps somewhat surprisingly, we allow a SAM program to read from an allocated address without writing to it first. In other words, the sequence below is valid.

$$addr \leftarrow Alloc(\,), \ldots, val \leftarrow Read(addr)$$

When such a sequence occurs, the machine responds to the *Read* by returning a distinguished symbol *None*. A slight adjustment to tree-based OSAM can easily handle read-without-write: the OSAM client scans a path through the OSAM tree, and if the desired address is not present, the lookup returns *None*.

The ability to read without write is surprisingly powerful. The crucial point is that the program can use the *None* symbol to branch its execution, depending on whether or not a particular address has been written. Recall from Section 3.1 our discussion of two nodes A and B, each of which holds a pointer to some node C. This problem is difficult for prior work, but by using read-without-write, we can solve it. Consider the following picture:



Here, we indeed give to A and B a pointer to C, and we also give each of these a pointer to an auxiliary address, X and Y respectively. These auxiliary addresses are also given to C and are initially *allocated, but not written.* When a SAM program wishes to traverse from A to C, it first reads A's pointer to X. Per SAM semantics, this read returns *None*, which the program interprets as an indication that it is safe to read C. The memory cell pointed to by C now resides in SAM program local memory, but B's pointer to C is now invalid. Since C is in local memory, the program holds a pointer to B's auxiliary address Y. The program writes a value to Y, indicating that a traversal from B to C is not safe.

By using these auxiliary memory addresses, we can use just a few SAM operations to convey a single bit of information – whether a pointer is valid or not – and this is sufficient to enable C to alert B without updating B in memory. This means that we avoid the need to recursively update B and all of its predecessors, which would in the worst case lead to updating all of memory.

Building on this basic technique, we can not only alert B that its pointer to C is invalid, we can also tell it where the new version of C resides. To achieve this, we implement a simple queue of addresses between a pointer (e.g. B) and the pointee (e.g. C). The pointee can push to the end of a queue to indicate its new location. The pointer can traverse this queue from beginning to end; it knows it has reached the end of the queue when it reads *None* from memory, and it uses the last address in the queue to fetch the latest copy of the pointee; see a full description in Section 5.

Smart pointers. SAM's ability to manage multiple pointers to one node, described above, is relatively intricate. It involves managing and creating queues between nodes that must be updated carefully.

In light of this intricacy, we build a pointer model on top of the basic operations of SAM. We call the pointers in this model 'smart pointers'. The idea is to provide a small number of operations on smart pointers: (1) given a value, we can construct a pointer to a value, (2) we can make an explicit copy of a pointer, (3) we can delete a pointer, and (4) we can dereference a pointer.

The implementations of smart pointer operations are non-trivial. For instance, copying a smart pointer involves setting up a new queue between the new copy and the pointee. With these details worked out, it becomes much easier to reason about SAM programs. Algorithms/data structures written using these smart pointer operations tend to look very similar to their standard implementation in the RAM model. We show that each of the smart pointer operations reduces to (amortized) $O(\log d)$ SAM operations, where d is the number of pointers pointing to the pointee being dereferenced.

Smart pointers enable us to handle a broad class of pointer-manipulating programs. Because of the ease with which smart pointers can be used, we implement all of our oblivious data structures and algorithms on top of them; see Section 6.

Graph algorithms; priority queues. Dereferencing a pointer incurs $O(\log d)$ SAM operations. This immediately reduces bandwidth blow-up and roundtrips while handling graphs of constant degree, but does not do so for graphs of arbitrary degree. Despite this, we achieve breadth-first search and depth-first search with asymptotics that outperform prior works. We achieve this improvement by emulating a graph of arbitrary degree via a graph of constant degree. The considered algorithms traverse the *entire* graph, and we exploit this to circumvent overhead imposed by the emulation.

Achieving our efficient algorithms for SSSP and minimum spanning tree is more nuanced. To achieve our stated costs (Table 2), we need to integrate in our OSAM the oblivious priority queue operations of [19]. This amounts to mainly adding two additional operations to the SAM model: *Insert*, which inserts an element to a global priority queue, and *Pop*, which extracts the element of highest priority. We note that it is the *combination* of SAM and priority queue operations that achieve our stated $O(|E| \cdot \log |E|)$ efficiency, and it is not clear how to achieve this cost with only one or the other. See Section 7 and Appendix D for details on our graph algorithms and priority queue integration.

4 OBLIVIOUS SINGLE ACCESS MACHINES

This section formalizes our definitions of SAM and OSAM, we give our OSAM construction. We refer the reader to Section 3 for an informal explanation of our model. Our construction is achieved by removing the position map from tree-based ORAM.

Definition 4.1 (Single Access Machine (SAM)). A **Single Access Machine** (SAM) is a memory storing a polynomial number of addressable memory cells, each of some specified bit-width w. The machine responds to three types of memory requests:

- $addr \leftarrow Alloc()$: The machine responds with a fresh memory address (i.e., an address that has not been chosen before). The machine may choose addresses in any arbitrary manner.
- $Write(addr, val)$: The machine writes value $val \in \{0,1\}^w$ to address $addr$. If (1) $addr$ was not allocated by the machine or (2) $addr$ was already written to, then the machine instead halts and outputs $\bot$.

- *val* ← *Read*(*addr*): The machine responds with the value written to *addr*, or it responds with *None* if nothing is written. If (1) *addr* was not allocated or (2) *addr* was already read from, then the machine instead halts and outputs ⊥.

A **SAM program** is an interactive, randomized algorithm that issues memory requests to the machine. A program is **valid** if it never issues a request that causes the machine to output ⊥. From here on, we only consider valid SAM programs (i.e., programs that properly allocate memory addresses and read/write each address at most once).

For simplicity, we consider SAM programs that use at most $O(w)$ bits of local space. Looking forward, we will instantiate *oblivious* SAM by leveraging two tree-based ORAM techniques: Path ORAM [23] – which requires that the client have $O(\lambda \cdot w)$ bits of space – and Circuit ORAM [24] – which requires that the client have $O(w)$ bits of space. Thus, the compilation of our SAM programs by our OSAM compiler will use either $O(w)$ bits of space or $O(w \cdot \lambda)$ bits of space, depending on the underlying ORAM technique.

An *oblivious* single access machine (OSAM) is formally a *compiler* that translates SAM memory requests into requests to a standard *random* access memory (allowing repeated accesses to an address). In an OSAM protocol, these requests are sent to the server, which satisfies each request. The crucial security property is that these requests can be *simulated.* This, in particular, means that the server learns nothing more than the number of read/write SAM requests:

Definition 4.2 (Oblivious Single Access Machine (OSAM)). A single access machine compiler Π is a poly-time, online algorithm that implements the single access machine interface (it correctly responds to *Alloc*/*Read*/*Write* requests) and issues random access memory requests. We say that Π is an **oblivious single access machine** (OSAM) if there exists a poly-time simulator $\mathcal{S}$ such that for any polynomial-length sequence of requests $\mathcal{R}$ that form a **valid SAM program**, the following ensembles are statistically close (in λ):

$$\Pi(1^\lambda, \mathcal{R}) \overset{s}{=} \mathcal{S}(1^\lambda, \mathcal{L}(\mathcal{R}))$$

Above, $\mathcal{L}(\mathcal{R})$ denotes the number of *Read*/*Write* requests (i.e., non-*Alloc* requests). In other words, the RAM requests issued by the OSAM can be simulated given only the *total number* of *Read*/*Write* requests in the underlying SAM program.

4.1 Our OSAM Construction

Figure 1 formalizes our tree-based OSAM. We present three algorithms – *Alloc*, *Read*, and *Write* – that respectively formalize how we compile the corresponding SAM operation into RAM operations. At a high level, our construction follows the handling of existing tree-based ORAMS [23, 24], except that we have no need for a position map – the underlying SAM program is responsible for keeping track of ORAM positions. We leave two algorithms – *ReadAndRm* and *Evict* – unspecified. These can be instantiated using tree-based ORAM techniques in [23, 24]. Our compiler (i.e., our OSAM client) maintains the common tree-based ORAM structure *stash* that temporarily holds a small number of blocks.

Alloc allocates fresh addresses by sampling a uniformly random leaf position, and then concatenating this with a global and monotonically increasing counter to ensure that each address is unique.

```
counter ← 0
stash ← empty-list

def Alloc() → addr :
    leaf ←$ [N]  // Uniformly sample a leaf
    a ← counter ⊔ leaf
    // symbol ⊔ denotes concatenation
    counter ← counter + 1
    return a

def Read(i : addr) → val | None :
    v ← ReadAndRm(i)
    // None if no such address written to previously
    Evict()
    return v

def Write(i : addr, v : val) :
    ReadAndRm(Alloc())
    // Read a dummy address
    stash.append({ i, v })
    Evict()

def ReadAndRm(i : addr) → val | None :
    interpret addr as counter ⊔ leaf
    // Read i from server by loading the path to leaf; See [23, 24]

def Evict :
    // Store stash elements to server by evicting paths; See [23, 24]
```

Figure 1: Our OSAM removes the position map from tree-based ORAM. In particular, procedures *ReadAndRm* and *Evict* can be taken from the Path ORAM construction [23] or the Circuit ORAM construction [24].

ReadAndRm and *Evict* are sub-procedures typical in tree-based ORAM. *ReadAndRm* fetches the value (if any) written at a specified address by reading a path from the root to the specified leaf. Note that this *deletes* the value from server memory, and this space in the ORAM tree can be used later. Hence, the space required on the server scales only with the maximum number of written-but-not-read addresses. If no value is written to the specified address, then *ReadAndRm* returns *None* (recall, returning *None* is important for allowing read-without-write). *Evict* moves values, including those in the stash, towards their assigned leaves and is used to write values back to the server. Thus, *ReadAndRm* can be used to implement *Read* and *Evict* can be used to implement *Write.* Note that *Write* also calls *ReadAndRm* on a dummy address to ensure obliviousness: regardless of whether the memory request is a *Read* or a *Write*, the server observes the client read a uniformly random path, followed by an eviction.

Figure 1 can be instantiated with different underlying tree-based ORAMs. The two most natural choices are Path ORAM [23] and Circuit ORAM [24]. Path ORAM bounds the stash size (client memory) to $O(\lambda)$ words, and each read/write consumes $O(\log n)$ words of communication [23]. Circuit ORAM can additionally outsource the stash to server memory to achieve $O(1)$ client memory, at the expense of $O(\lambda)$ read/write cost [24]. These immediately give the following main results of this paper.

Theorem 4.3 (Oblivious SAM Construction). *Let* Π *denote the compiler formalized in Figure 1.* Π *is an oblivious SAM (Definition 4.2). Let* $\mathcal{R}$ *denote a length-m sequence of SAM requests, and let the memory store n words of size* $w = \Theta(\log n)$*. If Figure 1 is instantiated using Path ORAM [23], then* Π *achieves the following performance characteristics:*

- $\Pi(1^\lambda, \mathcal{R})$ *outputs* $O(m \cdot \log n)$ *random access memory requests,*
- $\Pi(1^\lambda, \mathcal{R})$ *runs in* $O(w \cdot \lambda)$ *bits of space where* λ *is a statistical security parameter,*
- $\Pi(1^\lambda, \mathcal{R})$ *incurs exactly m roundtrips.*

If Figure 1 is instantiated using Circuit ORAM [24], then Π *achieves the following performance characteristics:*

- $\Pi(1^\lambda, \mathcal{R})$ *outputs* $O(m \cdot \lambda)$ *random access memory requests,*
- $\Pi(1^\lambda, \mathcal{R})$ *runs in* $O(w)$ *bits of space,*
- $\Pi(1^\lambda, \mathcal{R})$ *incurs* $O(m)$ *roundtrips.*

Proof. We first prove that the construction is correct, i.e., a *Read* operation to an address returns the written value to that address, or *None* if the address has not been written. Observe that the address contains the ORAM leaf (path) assigned to that address. If an address has been written before, the tree-based ORAM invariant ensures that a block containing the address and its written value will be found on the path by the *ReadAndRm* of the *Read* (there can be only one such *Read* in a SAM program). If the address has not been written before, no block with that address exists in the ORAM, and the *ReadAndRm* of the *Read* will return *None* as required.

We prove our construction is oblivious by constructing a simulator $\mathcal{S}$. The simulator, and the corresponding indistinguishability argument, are essentially the same as those of tree-based ORAM [23, 24]. Let m be the number of *Read* or *Write* operations in the sequence $\mathcal{R}$ (i.e., not counting *Alloc*). $\mathcal{S}$ does the following m times: calls *ReadAndRm*(*Alloc*()) and then *Evict*(). In the simulated view, the adversarial server $\mathcal{A}$ sees a sequence of physical memory requests due to *ReadAndRm* being called on randomly generated paths, followed by those due to *Evict*. In the real world, by inspection of Figure 1, for each *Read* or *Write* operation, the server also sees physical memory requests resulting from *ReadAndRm* and *Evict*. For each *Read*, the path for *ReadAndRm* corresponds to a fresh random address, i.e., it has not been used in another *ReadAndRm*, due to the validity of the SAM program. For each *Write*, the path for *ReadAndRm* is randomly generated on the fly. The underlying tree-based ORAM thus ensures the simulated view and the real execution are statistically close.

The cost of *Read*/ *Write* operations, client space, and roundtrips required are straightforward from the respective underlying ORAM construction. □

Augmenting SAM with Priority Queue Operations. We leverage prior work [19] to extend the SAM model with the following operations of a priority queue (1) *Insert*(*val*, *p*) : inserts value $val \in \{0,1\}^w$ into a priority queue with priority p (2) $val \leftarrow$ *Pop*() : reads and removes the element with the highest priority from the queue (3) *IsQueueEmpty* : checks if the queue is empty. In this extended model, the number of *Read*/ *Write*/*Insert*/*Pop* requests are leaked. Appendix D presents a formal definition, as well as our construction and related theorems.

5 SMART POINTERS

In subsequent sections, we use the SAM model to construct specific data structures and algorithms. Here, we develop *smart pointers*, which abstract detailed handling needed to allow two nodes to share the same SAM address. We begin by describing the interface of our smart pointers; our implementation on top of the basic SAM operations (*Alloc*/*Read*/ *Write*) follows.

A smart pointer is conceptually a pointer that can be dereferenced to obtain a value of some user specified type, which we henceforth refer to as *userT*. A user specified type is permitted to hold a constant number of smart pointers. This allows us to build up complex data structures. Operations on pointers, which are of type *ptr*, include the following:

- *new*(*userT*) $\rightarrow$ *ptr* : Save an instance of the user datatype to memory, and return a smart pointer to the allocated address.
- *get*(*ptr*) $\rightarrow$ *userT* : Dereference a smart pointer. A pointer can be dereferenced multiple times.
- *put*(*ptr*, *userT*) : Overwrite content of the pointee. A pointer can be used to overwrite its pointee multiple times.
- *operator* $\leftarrow$ (*ptr*, *ptr*) : Assign one smart pointer to another by creating a copy, thereby creating *multiple* pointers that point to the same content.
- *delete*(*ptr*) : Delete a smart pointer.
- *isnull*(*ptr*) $\rightarrow \{0,1\}$: Check if a given smart pointer is null.

There are two points worth exploring. First, we have *overloaded* the syntax $x \leftarrow y$. In particular, if x and y are smart pointers (are of type *ptr*), then the statement $x \leftarrow y$ *does not* mean that x becomes a verbatim, bitwise copy of y. Instead, an algorithm runs to set up queues between nodes (see discussion in Section 3). As a result, y becomes a "smart copy" of x, and it is safe to dereference both x and y.

Second, when a variable falls out of lexical scope, we automatically call *delete* on that variable. Calling *delete* is important to ensure that the cost of dereferencing a pointer depend solely on the number of pointers *currently* referencing an object.

Our final two operations extend our assignment and delete operators to user specified types in the natural manner:

- *operator* $\leftarrow$ (*userT*, *userT*) : Assign one piece of user data to another by smart-copying any contained pointers.
- *delete*(*userT*) : Delete the specified content by deleting any contained pointers.

5.1 Implementing Smart Pointers

```
def initQueue() → addr,addr :
    head ← SAM.Alloc()
    tail ← head
    return head, tail

def enqueue(tail: addr, a: addr) → addr :
    tail' ← SAM.Alloc()
    SAM.Write(tail, { a, tail' })
    return tail'

def dequeue(head) → addr, addr :
    switch SAM.Read(head) do
        case None do
            return null, null
        case { c, head' } do
            return c, head'
```

Figure 2: SAM program fragment for an address queue.

Address queues. Recall from Section 3 that we enable multiple pointers to share a pointee via *address queues*. The pointee uses such queues to update pointers to it, alerting each pointer of its latest SAM address. We start by constructing this simple address queue data structure in the SAM model; see Figure 2.

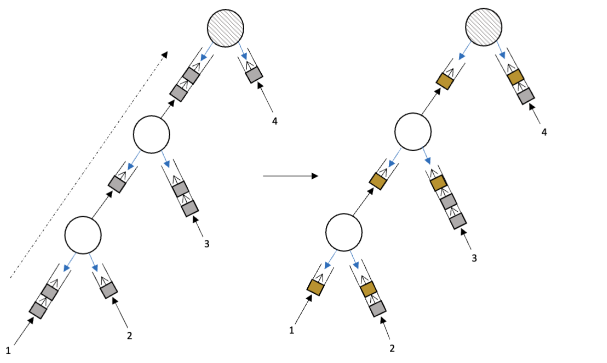


Figure 3: De-referencing pointer 1 in an inverted tree of 4 pointers. Black and blue arrows indicate queue heads and tails, respectively. Yellow blocks indicate newly enqueued addresses.

An address queue is a sequence of addresses sent from a *sender* to a *receiver*. Each node in the queue stores (1) an address from the sender and (2) the address of the next node in the queue. The SAM program manipulates the queue via a pair of addresses – *head* points to the queue's first node, and *tail* is a *pre-allocated* address that the next node will live at. Addresses are dequeued by reading *head*, and they are enqueued by writing to the queue's *tail* and allocating a fresh tail. Note that calling *dequeue* when the *head* has not been written to returns *None*, which indicates that the queue is empty. It is clear from inspection that each of our address queue operations uses only $O(1)$ SAM operations.

Overview of implementation of smart pointers. We first describe how address queues can be used to allow two pointers to point to the same pointee. We later extend this idea to allow an arbitrary number of pointers to the same pointee.

A connection between a pointer and its pointee is established by allowing them to share a queue, with the pointer as the receiver and the pointee as the sender. Each pointer holds the head of an address queue, and the pointee holds the two tails (as there is one queue per pointer). When one of the two pointers is de-referenced, the pointee is re-written to a new address and alerts both pointers of this fact by calling *enqueue* to write the new address into both queues using their tails. If the other pointer is de-referenced later, the *dequeue* procedure can be used to chase addresses through the queue until reaching the tail. The last address in the queue can then be read to fetch the pointee. We can determine if the queue's tail has been reached due to SAM's support for read-without-write.

To allow an arbitrary number of pointers to point to the same pointee, we construct an "inverted" binary tree. The pointee is at the root of this tree. A non-root node has a directed edge to its parent, and has at most two address queues "leading to" it. Just like the case with two pointers sharing the same pointee, after fetching the node using one of these queues, we can still fetch the node using the other queue. Thus, when a pointer is de-referenced, we can fetch the pointee by fetching the parent until we reach the root. Figure 3 provides an illustration.

Figure 4 implements helper procedures for our smart pointer operations, building on basic SAM operations and address queues.

```
struct ptr :
    head : addr

struct userT :
    ...   // user-specified fields

// node in inverted tree
struct node :
    tailL : addr
    tailR : addr
    isRoot : bool

// root holding the pointee
struct rootNode extends node
:
    content : userT
    isRoot : true

// non-root node
struct branchNode extends
node :
    headP : addr
    isRoot : false
```

```
def chase(head : addr) →
  node :
    target ← null
    latest ← null
    tail ← null
    while head ≠ null do
        latest ← target
        tail ← head
        target, head ←
          dequeue(head)
    n ← OSAM.Read(latest)
    if n.tailL = tail then
        n.tailL ← null
    else n.tailR ← null
    return n

def saveNode(n: node) :
    a ← OSAM.Alloc()
    if n.tailL then
        n.tailL ←
          enqueue(tailL, a)
    if n.tailR then
        n.tailR ←
          enqueue(tailR, a)
    OSAM.Write(a, v)

def addTail(n: node) → addr :
    head, tail ← initQueue()
    if n.tailL= null then
      n.tailL ← tail
    else n.tailR ← tail
    return head
```

Figure 4: Smart pointers helper procedures.

At the top we declare our data types which include the type of smart pointers (*ptr*), a user-specified datatype (*userT*) for the pointee, and a type *node* for each node in the inverted tree. Figure 5 implements smart pointer operations using the helper procedures.

Smart pointer helper procedures. The procedure *chase* is used to fetch a node in the inverted tree. Once a node is fetched, it must be saved back to SAM memory so that it can be dereferenced again later. This involves allocating and writing the node to a new address, and we enqueue the newly allocated address to each queue leading to the node. The helper procedure *saveNode* handles these.

When we wish to create or copy a pointer, we must connect that pointer to a node, which involves creating a new address queue that is shared between them. *addTail* establishes such a connection by initializing an address queue and storing the tail in the node.

Smart pointer operations. Each of our smart pointer operations is primarily a delegation to the above three helper procedures.

get dereferences a pointer by repeatedly calling *chase* to fetch the parent node to eventually fetch the root of the inverted tree where the pointee resides. Note that *chase* chases down an address queue, and then removes the tail of the chased queue from the dereferenced element. This is because after an address queue is chased down, it is destroyed. *get* ensures that a dereferenced pointer can be dereferenced again by re-establishing the connection between a node and its parent (via *addTail*) before saving it back to memory.

get contains one subtle but important detail: *get* returns a smart *copy* of the user data type i.e., any pointers within the user type are

```
def get(p: ptr) → userT:
    n ← chase(p.head)
    p.head ← addTail(n)
    while ¬n.isRoot do
        n' ← chase(n.headP)
        n.headP ← addTail(n')
        saveNode(n)
        n ← n'
    // out is a smart copy of n.content
    out ← n.content
    saveNode(n)
    return out
def put(p: ptr, c: userT):
    n ← chase(p.head)
    p.head ← addTail(n)
    while ¬n.isRoot do
        n' ← chase(n.headP)
        n.headP ← addTail(n')
        saveNode(n)
        n ← n'
    // n.content is a smart copy of c
    n.content ← c
    saveNode(n)
```

```
def delete(p: ptr):
    if p.head ≠ null then
        n ← chase(p.head)
        if n.isRoot then
            if ¬(n.tailL ∨ n.tailR) then
                delete(n.content)
            else saveNode(n)
        else
            if n.tailL then tail ← n.tailL
            else tail ← n.tailR
            n ← chase(n.headP)
            if ¬n.tailL then
                n.tailL ← tail
            else n.tailR ← tail
            saveNode(n)
def delete(c: userT):
    // Delete user type by deleting
    // its constituent pointers
    ...
def isnull(p: ptr) → bool:
    return p.head = null
```

```
def operator ←($p_0$ : ptr, $p_1$ : ptr):
    n ← chase($p_1$.head)
    if n.tailL ∨ n.tailR then
        nNew ← branchNode{
            .headP ← addTail(n)}
        saveNode(nNew)
        n ← nNew
    $p_0$.head ← addTail(n)
    $p_1$.head ← addTail(n)
    saveNode(n)
def operator ←($c_0$ : userT, $c_1$ : userT):
    // Copy user type by smart copying
    // its constituent pointers
    ...
def new(c: userT) → ptr:
    // .content is a smart copy of c
    v ← val{
        .tailL ← null,
        .tailR ← null,
        .content ← c}
    p ← ptr{.head ← addTail(v)}
    saveNode(v)
    return p
```

Figure 5: Smart pointers abstract the underlying SAM model, making SAM operations easier to work with. A smart pointer can be created (new), deleted (delete), copied (←), dereferenced (get), or updated (put). When dereferenced, a smart pointer returns a user-specified data type, which might hold other smart pointers. If a (smart) copy of that same pointer is also dereferenced, it will yield a (smart) copy of the same content.

"smart copied". This is crucial, because it ensures that the version of the element stored in machine memory and the version stored in the SAM program's local memory do not hold two copies of the same SAM address. This avoids a possible error where one could (1) dereference an element stored in SAM memory, (2) read a SAM address within that element, (3) dereference the element from SAM memory a second time, and (4) read the *exact same SAM address within that element a second time*. Such a sequence would yield an invalid SAM program, and we avoid it by making a smart copy when dereferencing.

put is similar to *get*: we repeatedly use *chase* to fetch the root, make a smart copy of the value to be stored in memory, and save the root back to memory. While doing this, we make sure to re-establish queues between nodes and their parents. *put* makes a smart copy for the same reasons as *get*.

new saves a user datatype (possibly some default initial value) to memory and returns a pointer to it. This creates the root of the inverted tree with the pointer directly pointing to this. To do so, we initialize a single address queue (via *addTail*) and save the resulting *rootNode* to memory (via *saveNode*). *new* makes a smart copy of the saved value for the same reasons as discussed above for *get*.

delete deletes a pointer by by deleting the node that the pointer points to. This is done by copying the tail of the other queue leading to the node to the node's parent, and saving the parent back to memory. Special care is taken when the pointer directly points to the root. In this case, if the root does not have another pointer pointing to it, we recursively delete the content of the pointee.

The overloaded ← operator for pointers creates a smart copy of a pointer by using the pointer's address queue to fetch the node, say *n*, being pointed to. If *n* already has a second queue leading to it, a new pointer cannot be made to point to it. Instead, a new node *nNew* is created, and the new pointer and the pointer being copied are made to point to *nNew*, which is made to point to *n*.

5.2 Validity and Efficiency

In the subsequent sections, we will use smart pointers to implement data structures and algorithms. To properly analyze such programs, we must argue two points:

- A smart-pointer-based program is a valid SAM program.
- Smart-pointer operations have good efficiency.

Both of these points rely on the properties of the smart pointer interface itself. Thus, we formalize the rules for using smart pointers:

Definition 5.1 (Smart-Pointer-Based Program). A SAM program $\mathcal{P}$ makes legal use of the smart pointer interface if it satisfies the following criteria:

- $\mathcal{P}$ issues no calls to *Alloc*/*Read*/*Write*, except those implied by the implementation of smart pointers.
- $\mathcal{P}$ does not call *get*/*put* on null smart pointers. That is, $\mathcal{P}$ does not dereference null smart pointers.

If it satisfies the above criteria, we say that $\mathcal{P}$ is a smart-pointer-based program.

We argue that any smart-pointer-based program is a valid SAM program. The $\leftarrow$ operator for pointers is overloaded to create only smart pointer copies. This ensures that, under the hood, every address queue used by the program is *unique*. We provide a proof sketch of the below Lemma in Appendix A.

LEMMA 5.2. *Let $\mathcal{P}$ be a smart-pointer-based SAM program (Definition 5.1). $\mathcal{P}$ is a valid SAM program (Definition 4.1).*

We also argue the *efficiency* of smart-pointer-based programs.

LEMMA 5.3. *Consider a smart-pointer-based SAM program (Definition 5.1). Each call to a smart pointer operation (Figure 5) issues amortized $O(d)$ SAM memory requests, where d pointers point to the associated pointee.*

PROOF. It suffices to show that *addTail*, *chase*, and *saveNode* each issue amortized $O(1)$ SAM requests. Since the height of the inverted tree is d in the worst case, each smart pointer operation makes $O(d)$ calls to these sub-procedures, and the lemma is proved.

It is clear from inspection of Figures 2 and 4 that *addTail* and *saveNode* each issue $O(1)$ SAM requests. *chase* is more nuanced: a call to *chase* can cause the program to chase down a queue of arbitrary length, incurring an arbitrary number of calls to *dequeue*. However, we discharge this cost by charging in advance - for every block that is read, we charge this cost at the time when block was *written* to the queue during *saveNode*. □

We can reduce this cost to $O(\log d)$ by maintaining the invariant that the tree of pointers pointing to a pointee is always a *complete* binary tree (which is balanced). This can be done with the following changes to the implementations of $\leftarrow$ (copy) and *delete*. Our new implementation of $\leftarrow$ creates a new node at the location expected for a complete binary tree with one more node, and the new pointer is made to point to this new node (irrespective of which source pointer is being copied). Our new implementation of *delete* also needs to keep the tree complete. This is done by swapping the to-be-deleted pointer with the *last* pointer in the complete binary tree, i.e., the one that points to the rightmost node in the last level. We ensure that given the root, this rightmost node can be fetched by 1) storing the value of d in the root and 2) making each node also store edges to its children. These edges are implemented as address queues. We formally provide the changes required in Appendix B.

6 OBLIVIOUS DATA STRUCTURES

In this section, we apply the SAM model to construct oblivious data structures. Table 2 summarizes the asymptotic performance of our constructions. Our constructions themselves are formalized using our smart pointer interface (Section 5); formally, each of our constructions is a smart-pointer-based program (Definition 5.1) that is almost identical to the equivalent RAM implementation. As we present our constructions, we use them to prove interesting properties of OSAM.

6.1 Doubly Linked Lists; OSAM Lower Bound

Doubly Linked Lists. We start with a doubly-linked list (DLL) to showcase the capabilities of smart pointers. A DLL is a list of nodes

```
// The type userT is set to node
struct node :
    prev : ptr
    next : ptr
    data : int

first : ptr ← null
last : ptr ← null

def next(p: ptr) → ptr :
    n ← get(p)
    return n.next

def prev(p: ptr) → ptr :
    n ← get(p)
    return n.prev

def insertAfter(p: ptr, d: int)
  → ptr :
    n ← get(p)
    q ← new(node {
        .prev ← p,
        .next ← n.next,
        .data ← d })
    if isnull(n.next) then
        last ← q
    else
        nnext ← get(n.next)
        nnext.prev ← q
        put(n.next, nnext)
    n.next ← q
    put(p, n)
    return q

def insertBefore(last : ptr, d:
  int) → ptr :
    // Analogous to insertAfter

def insertBeg(d: int) → ptr :
    if isnull(first) then
        p ← new(node{
            .prev ← null,
            .next ← null,
            .data ← d })
        first ← p
        last ← p
    else insertBefore(first, d)

def insertEnd(d: int) → ptr :
    // Analogous to insertBeg

def remove(p: ptr) :
    n ← get(p)
    if isnull(n.prev) then
        first ← n.next
    else
        nprev ← get(n.prev)
        nprev.next ← n.next
        put(n.prev, nprev)
    if isnull(n.next) then
        last ← n.prev
    else
        nnext ← get(n.next)
        nnext.prev ← n.prev
        put(n.next, nnext)
```

Figure 6: Our SAM-based doubly-linked list.

where each node stores some data, as well as pointers to the next and previous nodes in the list. The user can access the first and last elements of the list, and if holding a pointer to an element in the middle of the list, can move to the left/right, and access/insert/delete elements. Figure 6 lists our smart-pointer-based implementation. Note that two nodes of the DLL can point to one another, and this non-tree-like structure was out of scope for prior work.

Each of our DLL procedures uses a constant number of smart pointer operations. Since each node has at most two pointers pointing to it, each procedure uses amortized $O(1)$ SAM operations. Thus, when we compile our data structure with our OSAM, our DLL uses amortized $O(\log n)$ words of communication per procedure call. We remark that [25] also describes an oblivious doubly-linked list, but theirs requires packing $\Theta(\log n)$ elements in each ORAM block, requiring a block size of $\Omega(\log^2 n)$.

OSAM Lower Bound. Using the same smart-pointer-based style as Figure 6, it is trivial to construct stacks supporting push/pop and deques/queues supporting enqueue/dequeue. Each such procedure uses $O(1)$ smart pointer operations, and hence, in the oblivious setting, incurs amortized $O(\log n)$ words of communication.

Interestingly, these straightforward constructions immediately imply a lower bound on the bandwidth cost of any OSAM. [13] proved that *any* oblivious stack/queue/deque *must* have expected amortized cost $\Omega(\log n)$, given that the client runs in sublinear space and the data structure stores elements that match the server's word size of $\Theta(\log n)$ bits.

Theorem 6.1 (*OSAM* Lower Bound). *Let* Π *be an OSAM compiler that runs in space* $n^{1-\epsilon}$*, where* $\epsilon > 0$ *and where the word size is* $w = \Theta(\log n)$*. Given a length-m sequence of SAM requests* $\mathcal{R}$, $\Pi(\mathcal{R})$ *in expectation outputs a sequence of RAM requests of length* $\Omega(m \cdot \log n)$*.*

This implies that our tree-based OSAM construction (Figure 1) is essentially optimal, as it issues sequences of length $O(m \cdot \log n)$.

6.2 Trees

By again applying our smart-pointer-based methodology, we can implement arbitrary tree data structures, so long as each tree node has a constant number of children. We emphasize our ability to handle arbitrarily *unbalanced* trees, i.e., trees with depth $\omega(\log n)$. Our implementations are almost identical to their non-oblivious versions and, for reference, we present some of them in Appendix C. In particular, we highlight our ability to handle *tries* and *splay trees*, and use these to connect OSAM with ORAM.

Tries and connections to RAM. A *trie* (or *prefix-tree*) is a search tree where each key is a string over some alphabet. The tree is structured such that each subtree contains all strings that start with the same prefix, and each node has one child per character in the alphabet. Thus, a given search string determines a path through the tree, and we store the value associated with that string at the end of that path. Because the height of a trie is determined by the longest string in its key set, it may be unbalanced.

Appendix C formalizes our smart-pointer-based trie. This code is standard and included for completeness. We limit our handling to alphabets of constant size. When searching for a string of length ℓ, our trie issues $O(\ell)$ SAM memory requests. By compiling with OSAM, we obtain an oblivious trie structure where each lookup incurs $O(\ell \cdot \log n)$ bandwidth blow-up and $O(1)$ roundtrips.

Our oblivious trie's lookup operation issues a number of memory requests that depends on the search string length ℓ, and this may raise concern about security. However, the server's view is determined by the *aggregate* of all requests issued by an entire SAM program. A particular SAM program might look up elements in a trie multiple times, perhaps interleaved with operations to other SAM-based data structures; the server learns only the *total number* of SAM memory requests.

A trie on the alphabet $\{0, 1\}$ can instantiate a random access memory: each logical address is treated as a string, and by searching for a logical address, we access the content of that logical access. For a memory with n elements, each logical address is a string of length $\log n$, so the trie has $\log n$ depth. Since each node has a single pointer pointing to it, searching for a logical address can be done using $O(1)$ SAM operations. By implementing a trie in the SAM model, we establish a connection between RAM and SAM:

Theorem 6.2 (RAM from SAM). *Let* $\mathcal{P}$ *be a random access machine program with memory size n and word-size* $w = \Theta(\log n)$ *that halts in time* T*. There exists a SAM program that on the same input incurs while issuing* $T \cdot O(\log n)$ *SAM memory requests.*

As a corollary, this means that we can plug SAM-based RAM in our OSAM construction (Figure 1) and achieve an ORAM with $O(\log^2 n)$ bandwidth blow-up and $O(\log n)$ roundtrips. This is not surprising: the SAM program that emulates RAM via a trie embeds the $O(\log n)$ position maps of a tree-based ORAM into a single OSAM. While not surprising, it is reassuring that moving from the RAM model to the SAM model does not lose asymptotic performance: we can always compile RAM operations to SAM operations without asymptotic overhead as compared to using RAM directly.

Splay trees and caching ORAM. A splay tree [22] is a *self-adjusting* binary tree where each time a node is accessed, a *splay operation* rotates that node to the tree's root. Splay trees are known to have good locality properties. For instance, performing an in-order traversal of the leaves of a size-n splay tree only takes time $O(n)$; see further discussion in Appendix C. The data structure also has good amortized performance: its lookup procedure incurs amortized $O(\log n)$ cost, regardless of the access pattern. As with any binary tree structure, it is easy to embed splay trees in our smart pointer framework. Appendix C presents the code, which is standard and included for completeness.

Splay trees are rightfully the focus of some theoretical attention. Since their introduction [22], they have been conjectured to be the "asymptotically best possible binary tree". The long-standing Dynamic Optimality Conjecture [22] roughly states that for any sequence of lookups, the tree will perform within a constant factor of any binary tree algorithm that is *custom designed* for that sequence; see Appendix C for the formal conjecture.

It is easy to implement random access memory with a splay tree by using logical memory addresses as keys. Thus, by plugging a splay tree into our OSAM, we immediately obtain an interesting object that we refer to as *caching ORAM*:

Theorem 6.3 (Caching ORAM). *Assume the Dynamic Optimality Conjecture holds. There exists a statistically-secure ORAM* Π *with the following properties:*[1]

- *The RAM has n addressable memory cells.*
- *The client runs in* $O(w \cdot \lambda)$ *bits of space.*
- *Let* $\mathcal{R}$ *be a length-*$\Omega(n)$ *sequence of memory requests issued by the client, and suppose there exists some binary tree algorithm that could satisfy the requests in* $\mathcal{R}$ *in time* T*. Then* $\Pi(\mathcal{R})$ *issues* $O(T \cdot \log n)$ *memory requests to the server.*

This caching ORAM has amortized cost at most $O(\log^2 n)$ per access, but it can have cost as low as $O(\log n)$, depending on the access pattern. Sequences that tend to repeatedly access a relatively small number of elements, or that scan elements that are close together, will be accelerated. Even if the Dynamic Optimality Conjecture proves false, this splay-tree-based statistical ORAM will still have interesting properties, as splay trees are *known* to satisfy certain weaker properties, such as the static optimality; see [22].

[1]The stated efficiency is based on an instantiation with Path ORAM. If we instead instantiate caching ORAM via Circuit ORAM, we achieve $O(w)$ bits of client space and $O(T \cdot \lambda)$ memory requests.

7 OBLIVIOUS GRAPH ALGORITHMS

In this section, we use the SAM model to implement oblivious graph algorithms for the breadth-first search (BFS), depth-first search (DFS), single-source shortest path (SSSP), and minimum spanning tree (MST) problems. We refer to these as our target algorithms. All our algorithms run at cost $O(|E| \log |E|)$ memory requests, where $|E|$ is the number of edges. We remark that we consider *directed graphs.* For SSSP and MST, we equip our OSAM with an oblivious priority queue using the techniques of [19]. [19] adds no asymptotic overhead, and allows us to efficiently solve SSSP and MST.

Smart pointers can be directly used to implement textbook versions for these problems (after some natural modifications). These algorithms require dereferencing each pointer to a vertex to *visit* it. Since smart pointer operations incur $O(\log d)$ SAM memory requests when the dereferenced pointee is shared by d pointers, oblivious versions of these textbook algorithms are only efficient if d is a small constant. But for graphs of arbitrary degree, the total cost can be $O(|E| \log^2 |E|)$ in the worst case.

We can reduce the cost to $O(|E| \log |E|)$ even for arbitrary degree graphs by ensuring that each vertex in the graph is visited only once. We emulate the arbitrary degree graph, which we call the *original* graph, by a larger graph of constant degree that stores information about whether a vertex has been visited. If the original graph has $|V|$ vertices and $|E|$ edges, then the emulating graph has $O(|E|)$ vertices and $O(|E|)$ edges. Our approach is to specify a *template* algorithm that traverses each edge in the emulating graph at most twice. Being a graph of constant degree, this incurs only $O(|E|)$ SAM memory requests – and hence the compiled OSAM program makes $O(|E| \log |E|)$ requests to the server. Each target algorithm is achieved by plugging in appropriate details to the template. More precisely, our emulation proceeds as follows:

- For each vertex in the original graph, create an *original* vertex in the emulating graph, denoted by u.
- Consider an original vertex u. For each of u's incoming edges (v, u) in the original graph, we add a vertex to the emulating graph encoding that edge. Each such vertex is called an *incoming edge vertex*, denoted ${}_v u$.
- For each edge (v, u) in the original graph, we create a smart pointer to ${}_v u$. This pointer is called an *original edge*, denoted by $v \longrightarrow u$.
- For each edge (u, v) in the original graph, we create a vertex in the emulating graph. We call this vertex an *outgoing edge vertex*, denoted u_v. We store the original edge $u \longrightarrow v$ (recall, the original edge is a pointer) in u_v.
- Consider all outgoing edge vertices originating from u. We use smart pointers to create a binary tree where the original vertex u is the root and each outgoing edge node u_v is a leaf. This tree is called u's *outgoing edge tree.*
- Consider all incoming edge vertices incident on u. We use smart pointers to create a binary tree where the original node u is the root and each incoming edge node ${}_v u$ is a leaf. We augment this tree with parent pointers. Namely, from a tree node, we can traverse to its two children or its parent. This tree is called u's *incoming edge tree.*

Figure 12 in Appendix E depicts an example of an arbitrary degree graph emulated by a constant degree graph. Note that the number of edges in the emulating graph is only a constant factor higher than the number of edges in the original graph.

7.1 Implementing Oblivious Graph Algorithms

We solve SSSP using Dijkstra's algorithm and solve MST using Prim's algorithm. A common structure shared by these algorithms is to traverse the graph and generate a labeling for the original vertices. In the case of SSSP, each label is that vertex's distance from the source; in the other algorithms, each label is a pointer to the parent in a tree that describes the traversal. Each algorithm's traversal is guided by a data structure that dictates the order in which vertices should be visited. The particular *traversal structure* is specific to the algorithm:

Problem	**Labels**	**Traversal Structure**
DFS	pointer to parent in tree	stack
BFS	pointer to parent in tree	queue
MST	pointer to parent in tree	priority queue
SSSP	distance from source vertex	priority queue

Typically, graph algorithms are written in a style where metadata corresponding to each vertex (e.g., latest distance from the source node in Dijkstra's algorithm) is stored in an external array. For us, it is more efficient to store such metadata in the vertices themselves. In particular, we store whether an original vertex has been visited or not in its incoming edge vertices, we store edge weights in outgoing edge vertices, and the label in the original vertex itself.

The core loop of each of our algorithms follows the following template formally given in Figure 13 in Appendix E .

- Pop a pointer to a vertex u from the traversal structure. More precisely, pop a pointer to some incoming edge vertex ${}_v u$, along with information needed to update u's label.
- Check whether or not u has been visited. We store whether u has been visited in each incoming edge vertex ${}_v u$. If u has been visited, proceed to the next iteration of the loop.
- Otherwise, traverse the incoming edge tree to find the original vertex u and update u's label.
- Add all neighbors of u to the traversal structure. More precisely, we walk u's outgoing edge tree, and for each leaf u_v, we add u_v to the structure, along with data (the output of *getL*) needed to update that neighbor's label.
- We mark u as visited so that it will not be visited again. More precisely, we walk u's incoming edge tree, and for each leaf ${}_v u$, we update ${}_v u$ to denote that u has already been visited.

Instantiating our graph algorithms thus amounts to plugging into the above template: (1) the correct traversal structure and (2) algorithm-specific handling for labels. We remark that we tweak Dijkstra's algorithm in order to fit into the template. Appendix E gives a side-by-side comparison of the original Dijkstra's algorithm and the tweaked version and presents formalized SAM programs for BFS, DFS, SSSP, and MST .

Crucially, each of our algorithms dereferences each emulating graph vertex no more than twice. Indeed, we dereference each original vertex, as well as each of its outgoing edge vertices, exactly once. We dereference each incoming edge vertex once to set an original vertex as visited, and some incoming edge vertices will be dereferenced a second time to perform a visit. Since there are

$O(|E|)$ vertices in the emulating graph, our algorithms perform a total of $O(|E|)$ SAM memory requests and, when compiled with OSAM, our oblivious algorithms incur $O(|E| \cdot \log |E|)$ bandwidth blow-up and $O(|E|)$ roundtrips.

REFERENCES

[1] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: Optimal Oblivious RAM. In *EUROCRYPT 2020, Part II (LNCS, Vol. 12106)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 403–432. https://doi.org/10.1007/978-3-030-45724-2_14

[2] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. 2023. FutORAMa: A Concretely Efficient Hierarchical Oblivious RAM. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3313–3327.

[3] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. 2023. FutORAMa: A Concretely Efficient Hierarchical Oblivious RAM. *CCS* (2023).

[4] Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. 2013. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS 13*, Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng (Eds.). ACM Press, 207–218.

[5] David Cash, Andrew Drucker, and Alexander Hoover. 2020. A lower bound for one-round oblivious RAM. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18*. Springer, 457–485.

[6] Kai-Min Chung, Zhenming Liu, and Rafael Pass. 2014. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead. In *ASIACRYPT 2014, Part II (LNCS, Vol. 8874)*, Palash Sarkar and Tetsu Iwata (Eds.). Springer, Heidelberg, 62–81. https://doi.org/10.1007/978-3-662-45608-8_4

[7] Samuel Dittmer and Rafail Ostrovsky. 2020. Oblivious Tight Compaction In O(n) Time with Smaller Constant. In *SCN 20 (LNCS, Vol. 12238)*, Clemente Galdi and Vladimir Kolesnikov (Eds.). Springer, Heidelberg, 253–274. https://doi.org/10.1007/978-3-030-57990-6_13

[8] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. 2015. Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM. *Cryptology ePrint Archive* (2015).

[9] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*. Springer, 563–592.

[10] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473. https://doi.org/10.1145/233551.233553

[11] Michael T Goodrich and Michael Mitzenmacher. 2011. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages, and Programming*. Springer, 576–587.

[12] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 157–167.

[13] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. 2019. Lower Bounds for Oblivious Data Structures. In *30th SODA*, Timothy M. Chan (Ed.). ACM-SIAM, 2439–2447. https://doi.org/10.1137/1.9781611975482.149

[14] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2012. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 143–156.

[15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 359–376. https://doi.org/10.1109/SP.2015.29

[16] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 377–394. https://doi.org/10.1109/SP.2015.30

[17] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 415–430.

[18] Zihao Shan, Kui Ren, Marina Blanton, and Cong Wang. 2018. Practical secure computation outsourcing: A survey. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–40.

[19] Elaine Shi. 2020. Path Oblivious Heap: Optimal and Practical Oblivious Priority Queue. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 842–858. https://doi.org/10.1109/SP40000.2020.00037

[20] Elaine Shi, T H Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O ((log N) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings 17*. Springer, 197–214.

[21] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT 2011 (LNCS, Vol. 7073)*, Dong Hoon Lee and Xiaoyun Wang (Eds.). Springer, Heidelberg, 197–214. https://doi.org/10.1007/978-3-642-25385-0_11

[22] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. *Journal of the ACM (JACM)* 32, 3 (1985), 652–686.

[23] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, 299–310. https://doi.org/10.1145/2508859.2516660

[24] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, 850–861. https://doi.org/10.1145/2810103.2813634

[25] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *ACM CCS 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, 215–226. https://doi.org/10.1145/2660267.2660314

[26] Peter Williams and Radu Sion. 2012. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 293–304.

[27] Samee Zahur and David Evans. 2013. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 493–507. https://doi.org/10.1109/SP.2013.40

Appendices

A PROOFS OF SECURITY

Smart Pointer Validity. Recall that a SAM program is valid if an address is allocated before use, and each address is read from or written to only once. It is essential that a SAM program is valid for obliviousness since the security of our OSAM compiler holds only for valid SAM programs. We provide a proof sketch that any smart pointer-based smart program (Definition 5.1) is a valid SAM program.

Lemma 5.2 *Let $\mathcal{P}$ be a smart-pointer-based SAM program (Definition 5.1). $\mathcal{P}$ is a valid SAM program (Definition 4.1).*

Proof Sketch. SAM program validity requires that each SAM memory address is allocated before being written to and read from at most once each. We argue that smart-pointer-based programs automatically satisfy this constraint. Note that since dereferencing null pointers is disallowed by Definition 5.1, we only need to consider potential invalid operations that occur as the result of interacting with non-null pointers.

We argue that operations on a given smart pointer result in a valid SAM program. First, note that the SAM memory requests internal to *addTail*, *saveNode* and *chase* are valid. This is because each directed edge is implemented as a separate address queue. Calls made to *get* and *put* are valid since a different address is read each time a node in the inverted tree is fetched by calling *chase*. This is because each call to *chase* fetches a node that was earlier saved to memory by calling *saveNode* during which a node is written to a *newly allocated* address that is enqueued to both its dependent queues. Calls made to *chase* while deleting a pointer with *delete* or copying a pointer using $\longleftarrow$ are also valid due to the same reasons. Recall that while deleting a pointer, we copy the tail of the other dependent queue to the parent and save the parent back. This is valid since the copied tail was pre-allocated and has not been written to.

We now show argue that $\mathcal{P}$ is valid since $\mathcal{P}$ cannot copy a smart pointer, except by using the overloaded assignment operator provided in Figure 5. In other words, $\mathcal{P}$ cannot generate two copies of the same address queue. Note that if two exact address of the same pointer *were* created, then reading the *head* of both copies would break validity. It is relatively straightforward from inspection that $\mathcal{P}$ can only copy pointers via $\longleftarrow$. The only interesting corner cases are in *get*, *put*, and *new*, where our operations copy a user-specified datatype which might itself contain pointers. However, we overload assignment for user-specified types as well, ensuring that copying a user type applies our pointer assignment operator as well. Thus, there simply is no syntactic mechanism by which $\mathcal{P}$ can copy a pointer, except by calling $\longleftarrow$.

Now, notice that the only way to construct a non-null pointer is by calling *new* or by copying with $\longleftarrow$. Both of these procedures use *addTail* to establish address queues between each pointer and its pointee. This – combined with the fact that *saveNode* alerts all incoming pointers via *enqueue* – ensures that each pointer has its own address queue, and by calling *chase*, the pointer will obtain a single address which has been written, but not read. Thus, dereferencing a pointer leads to a valid SAM memory request. □

B EFFICIENT SMART POINTER OPERATIONS

In Section 5, we implemented smart pointer operations such that each operation incurred $O(d)$ SAM memory requests, where d is the number of shared pointers pointing to the pointee. The main reason for this cost is that the tree may be unbalanced, with a worst-case height of d. In this section, we describe how to reduce this amortized cost to $O(\log d)$. At a high level, we do this by maintaining the invariant that, after every smart pointer operation, the resulting tree of pointers is always a *complete* binary tree that is balanced. In a complete binary tree, each level except the last is completely filled. Nodes in the last level are as much to the left as possible.

Recall our implementation of the $\longleftarrow$ operator in Section 5: we copy a pointer by chasing it to obtain the node *n* that it points to, create a new node *nNew* that points to *n*, and make the pointer and its copy point to *nNew*. However, repeatedly copying the same pointer results in many such new nodes being created and forming a long path from the pointer to the root. This is precisely what causes the tree to be unbalanced.

We present a new implementation for the $\longleftarrow$ operation that ensures that the tree is balanced. In our new implementation, irrespective of the pointer being copied, a new node is created as the rightmost node in the last level that is not yet filled. The new pointer is then made to point to this new node. To determine how this node must be created, we make two changes. First, the root additionally stores the number of pointers pointing to the pointee. Given the root, if nodes in the tree also had edges to their children, this count could be used to determine the path of child nodes to be traversed in order to determine where the new node must be created. Thus, our second change is to use address queues to implement edges from nodes to their children as well. Now, given a pointer to be copied, we can first fetch the root and then create the rightmost node in the tree.

To maintain our invariant, we additionally need to update the implementation of the *delete* operation. In our new implementation, we replace the deleted pointer with the pointer that was most recently added. To do this, we first fetch the root. Then, we can fetch the rightmost node in the last level, and copy the tail of the most recently added pointer to replace the deleted pointer. We finally delete the rightmost node and copy the tail of the other pointer pointing to it to the rightmost node's parent. As a result, our new implementation of the *delete* operation has the effect of "undoing" what is done when a pointer is copied.

Figure 7 presents our new implementation of the *copy* ($\longleftarrow$) and *delete* operations. Our new implementations require making changes to the structure of the nodes in the tree. All nodes additionally store heads of address queues that lead to their children, the root additionally stores a count of the pointers pointing to the pointee, and non-root nodes additionally store the tail of the queue held by their parent. The helper procedures *saveNode* and *addTail* are updated to account for a node having three queues possibly leading to it. We introduce two additional helper procedures: *ascend* - to fetch the root given a pointer - and *descend* - to fetch the rightmost node in the last level. We stress that our implementations of *new*, *get* and *put* remain the same.

```
// node in inverted tree
struct node :
    tailL : addr
    tailR : addr
    headL : addr
    headR : addr
    isRoot : bool

// root holding the pointee
struct rootNode extends node :
    content : userT
    count : int
    isRoot : true

// non-root node
struct branchNode extends node :
    headP : addr
    tailP : addr
    isRoot : false
```

```
def saveNode(n: node) :
    a ← OSAM.Alloc()
    if n.tailL then
        n.tailL ← enqueue(tailL, a)
    if n.tailR then
        n.tailR ← enqueue(tailR, a)
    if n.tailP then
        n.tailP ← enqueue(tailP, a)
    OSAM.Write(a, v)

def addTail(n: node) → addr :
    head, tail ← initQueue()
    if n.tailP= null then
        n.tailP ← tail
    if n.tailL= null then
        n.tailL ← tail
    else n.tailR ← tail
    return head
```

```
def descend(root: node) :
    pow ← $\lfloor \log_2(root.count)) \rfloor$
    rightmost ← $\lfloor \frac{root.count - 2^{pow} - 1}{2} \rfloor$
    n ← root
    for bit in rightmost from MSB to LSB
     do
        if bit = 0 then
            if n.headL then
                n' ← chase(n.headL)
            else
                n' ← newNode()
                n'.tailL ← n.tailL
                n.tailL ← null
                n'.headP ← addTail(n)
            n.headL ← addTail(n')
        else
            if n.headR then
                n' ← chase(n.headR)
            else
                n' ← newNode()
                n'.tailR ← n.tailR
                n.tailR ← null
                n'.headP ← addTail(n)
            n.headR ← addTail(n')
        saveNode(n)
        n ← n'
    return n

def ascend(p: ptr) → rootNode :
    n ← chase(p.head)
    p.head ← addTail(n)
    while ¬n.isRoot do
        n' ← chase(n.headP)
        n.headP ← addTail(n')
        saveNode(n)
        n ← n'
    return n

def newNode() → branchNode :
    return branchNode{
        .tailL ← null,
        .tailR ← null,
        .headL ← null,
        .headR ← null}

def operator ←($p_0$ : ptr, $p_1$ : ptr) :
    root ← ascend($p_1$)
    root.count++
    n ← descend(root)
    $p_0$.head ← addTail(n)
    saveNode(n)

def delete(p: ptr) :
    root ← ascend(p)
    root.count-=1
    // get rightmost node and latest tail.
     Delete rightmost node by copying
     remaining tail to parent
    n ← descend(root)
    tailLatest ← n.tailR
    parent ← chase(n.head)
    if ¬parent.tailL then
        parent.tailL ← n.tailL
    else
        parent.tailR ← n.tailL
    saveNode(parent)
    // copy latest tail to node that deleted
     pointer points to
    n' ← chase(p)
    if ¬n'.tailR then
        n'.tailR ← tailLatest
    else
        n'.tailL ← tailLatest
    saveNode(n')
```

Figure 7: Updated implementation of smart pointer operations for copying a pointer (←) and deleting a pointer (*delete*). As a result, all our smart pointer operations incur a cost of $O(\log d)$ SAM memory requests, where d is the number of pointers pointing to the associated pointee.

Lemma B.1. *Consider a smart-pointer-based SAM program (Definition 5.1). Each call to a smart pointer operation (Figure 5, Figure 7) issues amortized $O(\log d)$ SAM memory requests, where d pointers point to the associated pointee.*

Proof. Since the tree is now always balanced, the height of the tree is $O(\log d)$. The rest of the proof follows from the proof of Lemma 5.3. □

C UNBALANCED SEARCH TREES

In this section, we discuss tries and splay trees. Tries and splay trees are search trees that allow values associated with keys to be efficiently retrieved, inserted, and deleted. We provide implementations of oblivious tries and splay trees using our smart pointer interface. We also review the fascinating properties of splay trees.

C.1 Tries

Our trie implementation is given in Figure 8. Recall that a trie is a prefix tree where each key is a string over some alphabet, and each node represents the *prefix* of a key. Each node stores (1) a list of smart pointers to its children, (2) a boolean value *eow* indicating whether the associated prefix is a key, (3) whether the value associated with it is a key. The root of the trie is initialized to represent an empty string. We describe the procedures that we implement for our trie.

search returns the value associated with a key by using each letter in the key to traverse a path. If a *null* pointer is encountered on this path, then the key is *not* found, and *None* is returned.

Insert is used to insert a key. It is similar to search. The difference is that each time a *null* pointer is encountered, it is replaced with a pointer to a new node created to represent the missing prefix.

delete deletes a string in the dictionary. It is also similar to *search*, with the difference being that if the string is found, its boolean value *eow* is set to false.

```
struct node :
    eow : {0,1}
    children :
      [ptr](AlphabetSize)
    value : int

root: ptr ← new(node{
    .eow ← 0,
    .children ← [null] })

def Insert(s : string, v : int) :
    p ← root
    n ← get(p)
    for letter in s do
        label = letter - 'a'
        if
          isnull(n.children[label])
          then
            n.children[label]
              ← new(node{
                      .eow ← 0,
                      .children
                        ← [null],

                      .value ←
                        v})
            put(p, n)
        p ← n.children[label]
        n ← get(p)
    n.eow ← true
    put(p, n)

def search(s : string) → int :
    n ← get(root)
    for letter in s do
        label = letter - 'a'
        if n.children[label]
          then
            n ←
              get(n.children[label])
        else return None
    if n.eow then
        return n.value
    return None

def delete(s : string) :
    p ← root
    n ← get(p)
    for letter in s do
        label = letter - 'a'
        if n.children[label]
          then
            p ←
              n.children[label]
            n ← get(p)
        else return
    n.eow ← false
    put(p, n)
```

Figure 8: Implementation of an oblivious trie using smart pointers.

C.2 Splay Trees

Implementing an Oblivious Splay Tree. We implement an oblivious splay tree in Figure 10. Each time a key is accessed, a *splay* operation is performed that restructures the tree by repeatedly performing rotations on the associated node's ancestors to bring the node to the tree's *root*. These rotations are similar to those required for regular balanced binary search trees. We give the splay operation, along with helper procedures to implement it, in Figure 9.

Our implementation of the splay operation follows the standard implementation. We provide helper procedures to rotate a node to the left or right to replace a node with its right or left child, respectively. How the node is structured with respect to its parent and grand-parent decide the direction in which it must be rotated, and whether the parent or the grand-parent is rotated first. Thus, each node stores pointers not only to its children, but also to its parent. Note that this is possible only because of our support for sharing. To help with performing rotations and splaying, we provide procedures to fetch the node's left child, right child, parent, and grand-parent.

Operations to search for and insert a node are similar to those used in balanced binary search trees, except that the node holding the key just accessed is immediately splayed to bring it to the root. Deleting a key is, however, slightly different. To delete a key, we first search for it to bring it to the root. We then "join" the left and right subtrees of the root to delete it. This is done by splaying the maximum value in the left sub-tree, thus ensuring that the root of the left subtree has no right child, after which the root of the right subtree is made its right child.

Properties of Splay Trees. Splay trees have been proven to have good locality optimizing properties. On splaying a node, not only is it brought to the root, but the average heights of *all* nodes on the path to it are approximately halved. For a long enough sequence of accesses, splay trees have been proven to have an amortized cost that either matches that of balanced binary search trees, or *outperforms* them. Consider the following theorems.

- *Balance Theorem:* The amortized cost of accessing a key k is $O(\log n)$. This shows that, irrespective of the access pattern, splay trees are as efficient as balanced binary search trees.
- *Static Optimality Theorem:* The amortized cost of accessing a key k is $O(\log(n/q_k))$ where q_k is the frequency with which k is accessed in the sequence. The more frequently a key is accessed, the cost of accessing it decreases.

```
struct node:
    key: int
    value: int
    left: ptr
    right: ptr
    parent: ptr

def l(n: node) → node:
    return get(n.left)

def r(n: node) → node:
    return get(n.right)

def leftRot(yPtr: ptr):
    y ← get(yPtr)
    xPtr ← y.right
    x ← get(xPtr)
    pPtr ← y.parent
    p ← get(pPtr)
    if y.key == l(p).key then
        p.left ← xPtr
    else p.right ← xPtr
    x.parent ← pPtr
    put(pPtr, p)
    zPtr ← x.left
    if zPtr then
        z ← get(zPtr)
        y.right ← zPtr
        z.parent ← yPtr
        put(zPtr, z)
    x.left ← yPtr
    y.parent ← xPtr
    put(xPtr, x)
    put(yPtr, y)

def rightRot(yPtr: ptr):
    // Similar to leftRot

def p(n: node) → node:
    return get(n.parent)

def gp(n: node) → node:
    return p(p(n))

def splay(xPtr: ptr):
    x ← get(xPtr)
    while p(x) do
        if !gp(x) then
            if x.key == l(p(x)).key then
                rightRot(x.parent)
            else
                leftRot(x.parent)
        else if x.key == l(p(x)).key and p(x).key == l(gp(x)).key then
            rightRot(p(x).parent)
            rightRot(x.parent)
        else if x.key == r(p(x)).key and p(x).key == r(gp(x)).key then
            leftRot(p(x).parent)
            leftRot(x.parent)
        else if x.key == l(p(x)).key and p(x).key == r(gp(x)).key then
            rightRot(x.parent)
            leftRot(x.parent)
        else
            leftRot(x.parent)
            rightRot(x.parent)
        x ← get(xPtr)
```

Figure 9: Helper procedures to implement an oblivious splay tree.

- *Static Finger Theorem:* For any fixed key f, the amortized cost of accessing a key k is $O(\log(|k - f| + 1))$. Thus, accesses within the vicinity of a "finger" are faster.
- *Working Set Theorem:* The amortized cost of accessing a key k is $O(\log(t(k) + 1))$ where $t(k)$ is the number of keys accessed since the last time k was accessed. This means the most recently accessed keys, which form a working set, are easier to access.

While unproven, splay trees are conjectured to perform as well as a binary search tree *tailored* to answer a sequence of queries.

Conjecture C.1 (Dynamic Optimality [22]). *Let A be* any *binary search tree algorithm where we charge unit cost to (1) perform a tree rotation or (2) access a child from a parent. Let S be a sequence of queries to a search tree of size n, and let $A(S)$ be the total cost to satisfy S with algorithm A. A splay tree handles S at cost $O(n + A(S))$.*

```
struct node:
    key: int
    value: int
    left: ptr
    right: ptr
    parent: ptr

root: ptr ← null

def search(k: int) → value:
    p ← root
    while p do
        n ← get(p)
        if k== n.key then
            break
        else if k > n.key then
            n.right ? p ← n.right: break
        else
            n.left ? p ← n.left: break
    splay(p)
    root ← p

def delete(key: int):
    search(key)
    n ← get(root)
    if !isnull(n.left) then
        root ← n.left
        search(key)
        nl ← get(root)
        nl.right ← n.right
        put(root, nl)
    else root ← n.right

def Insert(k : int, v : int):
    p ← root
    q ← null        // Pointer to parent
    while p do
        n ← get(p)
        q ← p
        if k > n.key then
            p ← n.right
        else
            p ← n.left
    m ← n // Leaf node acting as parent
    n ← node{
        .key ← k, .value ← v,
        .left ← null, .right ← null,
        .parent ← null}
    p ← new(n)
    if isnull(q) then
        root ← p
    else
        if k > m.key then
            m.right ← p
        else
            m.left ← p
        n.parent ← q
        put(q, m)
    put(p, n)
    splay(p)
    root ← p
```

Figure 10: Implementation of an oblivious splay tree using smart pointers.

D OBLIVIOUS PRIORITY-QUEUE-AUGMENTED SAM

Recall that some of our graph algorithms require that we extend the SAM model with operations of a priority queue. In this section, we define this extended SAM model and modify our OSAM compiler to additionally handle requests to a priority queue.

The Augmented SAM-PQ model. The extension of the SAM model to handle priority queue operations is straightforward, and involves simply adding new operations to the interface of our single access machine:

Definition D.1 (Priority-Queue-Augmented SAM (SAM-PQ)). A **Priority-Queue-Augmented SAM** (SAM-PQ) is a memory storing a polynomial number of addressable memory cells, each of some specified bit-width w. The machine also stores a priority queue holding cells of width w. The machine responds to SAM memory requests, as well as the following:

```
counter ← 0
stash ← empty-list
addrMin ← null
queueSize ← 0

def Alloc() → addr :
    // Same as before

def Read(i : addr) → val | None :
    // Same as before

def Write(i : addr, v : val, p : T ← ∞) :
    ReadAndRm(Alloc())              // Read a dummy address
    stash.append({ p, i, v })
    Evict()

def Pop() → val | ⊥ :
    if queueSize then
        queueSize -= 1
        return Read(addrMin)
    return ⊥

def Insert(v : val, p : T) :
    queueSize += 1
    Write(Alloc(), v, p)

def IsQueueEmpty() → {0, 1} :
    return queueSize== 0

def ReadAndRm(i : addr) → val | None :
    ...
    // additionally calls P.updateMin(), where P is the path
     identified by i

def Evict() :
    ...
    // additionally calls P.updateMin(), for each path P
     evicted along
```

Figure 11: OSAM client extended to include an oblivious priority queue.

- *Insert*(*val*, p): The machine inserts value $val \in \{0,1\}^w$ into a priority queue with priority p.
- $val \leftarrow$ *Pop*(): The machine responds with the element of highest priority in the priority queue and removes that value from the queue. If the queue is empty, the machine instead halts and outputs $\perp$.
- $\{0,1\} \leftarrow$ *IsQueueEmpty*: The machine responds with 1 if the priority queue is empty and 0 otherwise.

A **SAM-PQ program** is an interactive, randomized algorithm that issues memory requests to the machine. A SAM-PQ program is valid if it does not cause the machine to output $\perp$.

The OSAM-PQ compiler. By leveraging prior work [19], it is straightforward to construct tree-based OSAM-PQ. We instantiate our priority queue using a min heap, with smaller elements having higher priorities. We give a brief overview of the ideas presented in [19].

Recall that in tree-based ORAM, the server's memory is arranged as a binary tree, and each logical block is mapped to a particular path through that tree. [19]'s elegant observation is that this structure is naturally compatible with priority queue operations. Specifically, [19] augments each node in the tree with the minimum element in its subtree, along with the leaf assigned to it. Thus, the minimum element in the root stores the minimum element in the priority queue, and the client locally stores the address of this element. Each time an element is inserted or popped, this augmented data is updated for all nodes on the path to the element's assigned leaf using a procedure called *updateMin*.

Figure 11 presents our OSAM-PQ construction. Since each node in the tree additionally stores a priority, the *Write* procedure is updated to take a priority as a third input, with a default value of ∞. To update priorities on insert and pop, the *ReadAndRm* and *Evict* procedures additionally call *updateMin*. The client stores two additional values - the number of elements in the queue, and the address of the minimum element. *Insert* inserts an element to the priority queue by making a call to *Write*. *Pop* calls *Read* to read the address of the minimum element. The number of elements in the queue is updated accordingly.

Definition D.2 (Oblivious SAM-PQ (OSAM-PQ)). An SAM-PQ compiler Π is a poly-time, online algorithm that implements the SAM-PQ interface and issues random access memory requests. We say that Π is an **oblivious SAM-PQ** (OSAM-PQ) if there exists a poly-time simulator $\mathcal{S}$ such that for any polynomial-length sequence of SAM-PQ requests $\mathcal{R}$, the following ensembles are statistically close (in λ):

$$\Pi(1^\lambda, \mathcal{R}) \overset{s}{=} \mathcal{S}(1^\lambda, \mathcal{L}(\mathcal{R}))$$

Above, $\mathcal{L}(\mathcal{R})$ denotes the number of *Read*/*Write*/*Insert*/*Pop* requests.

The following holds by the same reasoning as Theorem 4.3:

Theorem D.3 (Oblivious SAM-PQ Construction). *Let Π denote the SAM-PQ compiler formalized in Figure 11. Π is an oblivious SAM-PQ (Definition D.2). Let $\mathcal{R}$ denote a length-m sequence of SAM-PQ requests, and let the memory store n words of size $w = \Theta(\log n)$. If Π is instantiated using Path ORAM [23], then it achieves the following performance characteristics:*

- *$\Pi(1^\lambda, \mathcal{R})$ outputs a sequence of random access memory requests of length $O(m \cdot \log n)$.*
- *$\Pi(1^\lambda, \mathcal{R})$ runs in $O(w \cdot \lambda)$ bits of space where λ is a statistical security parameter.*
- *$\Pi(1^\lambda, \mathcal{R})$ incurs exactly m roundtrips.*

If Π is instantiated using Circuit ORAM [24], then it achieves the following performance characteristics:

- *$\Pi(1^\lambda, \mathcal{R})$ outputs a sequence of random access memory requests of length $O(m \cdot \lambda)$.*
- *$\Pi(1^\lambda, \mathcal{R})$ runs in $O(w)$ bits of space.*
- *$\Pi(1^\lambda, \mathcal{R})$ incurs $O(m)$ roundtrips.*

E OBLIVIOUS ALGORITHMS FOR ARBITRARY GRAPHS

In this section we formally provide implementations of our graph algorithms. We also explain how we modify the textbook Dijkstra's algorithm to fit into a common template that we use for our graph algorithms.

E.1 Implementing our Target Algorithms

Recall that we obtain our algorithms by emulating an arbitrary degree graph as a constant degree graph (see Section 7). Figure 12 gives an example of an emulating graph. Vertices in the emulating graph are of three types. *Original vertices* correspond to the vertices in the original graph, and store the label generated by the target algorithm. *Incoming edge vertices* can be used to reach an original vertex, and store whether the original vertex has already been visited. *Outgoing edge vertices* store pointers to incoming edge vertices. Since incoming edge vertices can be used to reach an original vertex, these pointers effectively serve as directed edges and are thus called *original edges.*

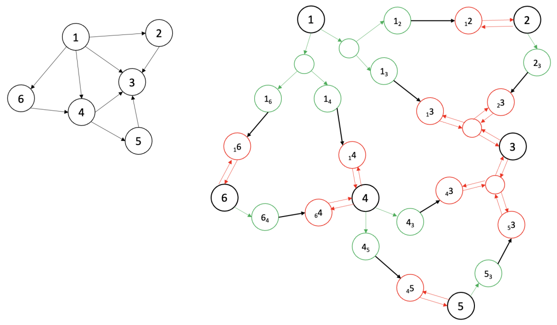


Figure 12: Emulating an arbitrary degree graph (left) by a constant degree graph (right). Green vertices encode outgoing edge trees, and red nodes encode incoming edge trees.

Helper procedures. Recall that our target algorithms follow a common template (Figure 13). All our target algorithms perform a graph traversal by visiting vertices and adding pointers to neighboring vertices in a data structure that determines the vertex to be visited next. Figure 14 presents the helper procedures using which we implement our target algorithms. At the top we provide data types implementing the types of vertices in the emulating graph. Each vertex contains fields to (1) help distinguish what type it is (2) point to its neighbors in the incoming and outgoing edge trees (3) indicate that it is a leaf node of a tree. We now describe our helper procedures *visit* and *addNgbrs.*

- *visit:* Given a pointer to a vertex in an incoming edge tree, returns the root of the incoming edge tree after marking it as visited. Recall that this is an *original vertex.* This is done by traversing a path in the tree from leaf to root. From the root, a breadth-first-search traversal of the incoming edge tree is performed to update all incoming edge vertices to indicate that original vertex has been visited.

```
def traverse(p : ptr) :
    // Visit the source node
    p ← visit(p, $\phi$)        // $\phi$ is a pre-determined result for the source
    addNgbrs(p, dsPtr, getL)   // dsPtr is a pointer to the data structure
    setVisited(p)
    while dsPtr do
        // Visit a node if not visited
        dsPtr, {label, ogE} ← remove(dsPtr)
        n ← get(ogE)
        if !n.visited then
            p ← visit(ogE, label)
            addNgbrs(p, dsPtr, getL)
            setVisited(p)
```

Figure 13: Our template used to define our SAM-based oblivious algorithms for the graph DFS, BFS, MST, and SSSP problems. *getRes* **is a function, defined differently for each target algorithm, that computes a value needed to properly update the vertex's neighboring labels.**

- *addNgbrs:* Given an original vertex, performs a breadth-first-search traversal of its outgoing edge tree. As the traversal is performed, original edges contained in outgoing edge vertices, along with an updated label for the vertex being pointed to, are added to a data structure.

Target algorithms. Instantiating our graph algorithms amounts to plugging into Figure 13: (1) the correct traversal structure and (2) algorithm-specific handling for labels.

Figure 15 presents algorithms for BFS, DFS, SSSP, and MST. Note that our algorithms for SSSP and MST additionally define a function *getP* that computes the priority with which a vertex and its label should be inserted into the priority queue. Recall that whether a vertex has been visited or not is stored in incoming edge vertices. This visited bit *toggles* between 0 and 1, and checking if a vertex has been visited amounts to comparing this bit with a *global* bit for the graph that is also toggled before each traversal.

E.2 Modifying Dijkstra's Algorithm

Our target algorithms broadly follow the template provided in Algorithm 13. However, we slightly tweak Dijkstra's algorithm for SSSP to fit this template. Figure 16 gives a side-by-side comparison of the tweaked algorithm with the original. We highlight key changes made below.

(1) The array d that stores the distance of a vertex is written to only once for each vertex - when the distance for the vertex is finalized. This array is never read.
(2) Each time a vertex is encountered through *any* path, an entry for it is added to the priority queue, even if the path is *not* shorter. This entry not only contains the identity of the vertex, but also the distance of this path.

```
struct vertex :
    id : int
    type : enum
    inLeft : ptr
    inRight : ptr
    outLeft : ptr
    outRight : ptr
    label : int

struct outEdge :
    type : enum
    isLeaf : {0,1}
    left : ptr
    right : ptr
    ogE : ptr
    weight : int

struct inEdge :
    type : enum
    isLeaf : {0,1}
    left : ptr
    right : ptr
    parent : ptr
    visited : {0,1}
```

```
def visit(ogE : ptr, label : int) → vertex :
    p ← ogE
    v ← get(p)
    while v.type ≠ VERTEX do
        p ← v.parent
        v ← get(p)
    v.label ← label
    put(p, v)
    // Updating incoming edge vertices to mark the vertex as visited
    head, tail ← initQueue()
    ogV ← get(p)
    if ogV.inLeft then
        tail ← enqueue(tail, ogV.inLeft)
    if ogV.inRight then
        tail ← enqueue(tail, ogV.inRight)
    while head do
        head, p ← dequeue(head)
        v ← get(p)
        if !v.isLeaf then
            if v.left then
                tail ← enqueue(tail, v.left)
            if v.right then
                tail ← enqueue(tail, v.right)
        else
            v.visited ← globalVisited
            put(p, v)
    return ogV
```

```
def addNgbrs(ogV: vertex, ds: enum, pDS: ptr, getL: fn, getP: fn) :
    head, tail ← initQueue()
    if ogV.outLeft then
        tail ← enqueue(tail, ogV.outLeft)
    if ogV.outRight then
        tail ← enqueue(tail, ogV.outRight)
    while head do
        head, p ← dequeue(head)
        v ← get(p)
        if !v.isLeaf then
            if v.left then
                tail ← enqueue(tail, v.left)
            if v.right then
                tail ← enqueue(tail, v.right)
        else
            if getP then
                pty ← getP(ogV, v)
            label ← getL(ogV, v)
            switch ds do
                case QUEUE do
                    pDS ← enqueue(pDS, {label, v.ogE})
                case STACK do
                    pDS ← push(pDS, {label, v.ogE})
                case PQ do
                    OSAM.Insert({label, v.ogE}, pty)
```

Figure 14: Helper procedures to implement our oblivious graph algorithms.

(3) An additional *visited* array is maintained. This array is updated to indicate that a vertex has been visited when it is popped for the first time from the priority queue. A vertex is visited only if it has not been visited earlier.

Note that adding multiple entries for a vertex does not change the order in which vertices are traversed, since it still remains that a vertex is visited only when it is the closest to the source among all vertices that have not been visited yet. Thus, our modified algorithm is still correct, while ensuring that the distance array need not be updated multiple times.

```
struct vertex :
    id : int, label : int
    visited : {0,1}, edges : [{ptr, int}]

def addDS(ds : enum, pDS : ptr, getP : fn, data : {int, ptr}) :
    switch ds do
        case QUEUE do
            pDS ← enqueue(pDS, data)
        case STACK do
            pDS ← push(pDS, data)
        case PQ do
            pty ← getP(v, w)
            OSAM.Insert(data, pty)

globalVisited ← 0

def visit(p : ptr, l : int) → vertex :
    v ← get(p)
    v.label ← l
    v.visited ← ¬v.visited
    put(p, v)
    return v

def addNgbrs(v : vertex, ds : enum, pDS : ptr, getL : fn, getP : fn) :
    for {e, w} in v.edges do
        label ← getL(v, w)
        addDS(ds, pDS, getP, {w, e})
```

```
def BFS(pSrc : ptr) :
    head, tail ← initQueue()
    globalVisited ← ¬globalVisited
    src ← visit(pSrc, null)
    addNgbrs(src, QUEUE, tail, getL, null)
    while head do
        head, {label, p} ← dequeue(head)
        v ← get(p)
        if v.visited ≠ globalVisited then
            v ← visit(p, label)
            addNgbrs(v, QUEUE, tail, getL, null)

    def getL(v: vertex, w : int) → int :
        return v.id

def DFS(pSrc : ptr) :
    top ← initStack()
    globalVisited ← ¬globalVisited
    src ← visit(pSrc, null)
    addNgbrs(src, STACK, top, getL, null)
    while top do
        top, {label, p} ← pop(top)
        v ← get(p)
        if v.visited ≠ globalVisited then
            v ← visit(p, label)
            addNgbrs(v, STACK, top, getL, null)

    def getL(v: vertex, w : int) → int :
        return v.id

def Dijkstra(pSrc : ptr) :
    globalVisited ← ¬globalVisited
    src ← visit(pSrc, 0)
    addNgbrs(src, PQ, null, getL, getP)
    while !OSAM.isQueueEmpty() do
        key, {label, p} ← OSAM.Pop()
        v ← get(p)
        if v.visited ≠ globalVisited then
            v ← visit(p, label)
            addNgbrs(v, PQ, null, getL, getP)

    def getL(v: vertex, w : int) → int :
        return v.label + w

    def getP(v: vertex, w : int) → int :
        return v.label + w

def Prims(pSrc : ptr) :
    globalVisited ← ¬globalVisited
    src ← visit(pSrc, null)
    addNgbrs(src, PQ, null, getL, getP)
    while !OSAM.isQueueEmpty() do
        key, {label, p} ← OSAM.Pop()
        v ← get(p)
        if v.visited ≠ globalVisited then
            v ← visit(p, label)
            addNgbrs(v, PQ, null, getL, getP)

    def getL(v: vertex, w : int) → int :
        return v.id

    def getP(v: vertex, w : int) → int :
        return w
```

Figure 15: Oblivious algorithms for our target algorithms. These are obtained by plugging in to Figure 13 the appropriate data structure for performing the traversal and function for generating labels. For algorithms that use a priority queue, we additionally provide a function for computing the priority

```
for $v \in V$ do
    if $v \neq src$ then
        $d[v] \leftarrow \infty$
        $pq.Insert(v, d[v])$
$d[src] \leftarrow 0$
$pq.Insert(src, 0)$
while $!pq.Empty()$ do
    $v = pq.Pop()$
    for $u \in N(v)$ do
        if $d[v] + w(u, v) < d[u]$ then
            $d[u] \leftarrow d[v] + w(u, v)$
            $pq.decPriority(u, d[u])$
```

```
for $v \in V$ do
    $visited[v] \leftarrow false$
$visited[v] \leftarrow true$
$pq.Insert(src, 0)$
while $!pq.Empty()$ do
    $\{d, v\} = pq.Pop()$
    if $!visited[v]$ then
        $d[v] \leftarrow d$
        $visited[v] \leftarrow true$
        for $u \in N(v)$ do
            $d \leftarrow d[v] + w(u, v)$
            $pq.Insert(\{d, u\}, d)$
```

Figure 16: Dijkstra's algorithm for SSSP (left) and a modified version of Dijkstra's (right), where each node is visited only once. Array accesses are required only to maintain the *visited* array, and to write the final result.