Cryptography meets algorithms (15893) Lecture Notes

Lecture 12: Oblivious Data Structures

Scribe: Tanisha Saxena

March 23, 2024

1 Motivation

In previous lectures, we learned Oblivious RAM (ORAM), which allows us to compile *any* program into an oblivious counterpart. However, in practice, when you encounter a specific computational task, using generic ORAM to compile a non-oblivious algorithm may not be the best approach. For example, in a previous lecture, we showed how to design oblivious sorting algorithms that asymptotically outperform the naïve approach of directly using ORAM to compile Quick Sort. In this lecture, we will how to design customized oblivious data structures (e.g., dictionary, stack, queue, priority queue) that outperform generically using Path ORAM to compile a non-oblivious algorithm.

2 Background

Several of our algorithms will rely on a (non-recursive) ORAM tree which was used in Path ORAM [SDS+18]. Recall that in the ORAM tree, every element is assigned to some random path (from the root to some leaf node) — we call this path the element's *position identifier*. Suppose one already knows which path an element is assigned to, then to fetch the element, the client performs the following:

- read the corresponding path and remove the element from the path;
- assign the element to a random new path, and write it back to the root;
- perform an eviction operation on the path just read.

We shall assume that the client stores a superlogarithmically sized stash (which is required for storing the overflowing elements in Path ORAM), and we count cost in terms of the number of elements communicated to and from the server for each data structure request.

In Path ORAM, to find out which path the desired element resides on, we recursively store a position map in a smaller ORAM tree. The size of the position map is at least a constant factor smaller than the original ORAM. Therefore, after logarithmically many recursions, the position map becomes constant in size and can be stored directly by the ORAM client.

3 Oblivious Dictionary

3.1 Operations

We consider a standard dictionary data structure supporting the following operations:

- **Lookup(k):** Return the value with key k;

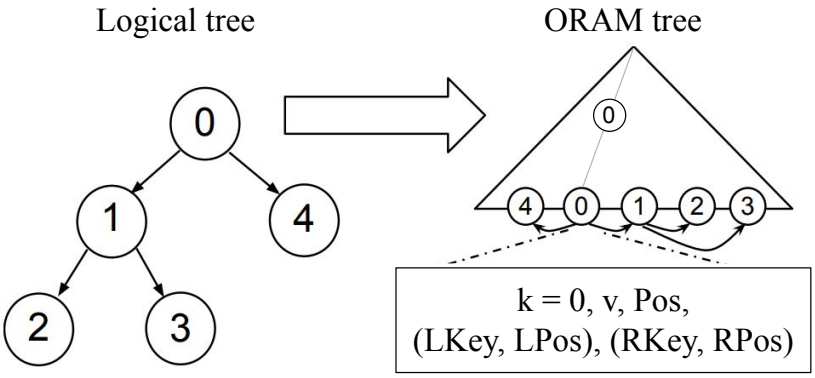

Figure 1: **Each element in the ORAM tree corresponds to a node in the logical tree. Each element stores the position identifiers of its two children. This eliminates the need to perform a recursive position map lookup for the next child node to read.** Note that the element is actually stored somewhere along the path leading to the leaf even though we draw it at the leaf. $(\mathsf{LKey}, \mathsf{LPos})$ and $(\mathsf{RKey}, \mathsf{RPos})$ denote the keys and the position identifiers of the two children. Pos denotes the position identifier of the current element, which is needed for performing eviction on the ORAM tree.

- **Insert**(k, v)**:** Insert the value v with key k into the dictionary;
- **Delete**(k)**:** Delete the value with key k from the dictionary.

Though this oblivious dictionary method can be generalized to include duplicate keys, we will only focus on the case where all keys are unique for the sake of simplicity. Note that ORAM itself realizes a memory array abstraction which is a special dictionary with contiguous key space $[1 \ldots N]$ where N is the maximum number of elements in the data structure. For a general dictionary, however, the key space is allowed to be much larger than the number of elements.

3.2 Construction for a Static Dictionary

For simplicity, let's first assume that we want to implement a static dictionary where we only need to support **Lookup** but no dynamic insertion and deletion.

The naïve approach is to use Path ORAM to directly compile a non-oblivious dictionary, e.g., an AVL tree — henceforth we shall call the AVL tree the *logical* tree to distinguish it from the ORAM tree. The cost per operation for a non-oblivious AVL tree is $O(\log N)$ where N is the maximum number of elements in the data structure, and the Path ORAM compilation incurs $O(\log^2 N)$ multiplicative overhead. So the naïve approach would incur $O(\log^3 N)$ cost per dictionary operation. Note that in an AVL tree, not all operations take the same amount of time. When we use ORAM to compile an AVL tree, we have to pad all operations to the same length to avoid leakage through the number of accesses.

If we directly use Path ORAM to compile the AVL logical tree, essentially we have to go through the entire ORAM recursion for reading every node in the logical tree. The following simple idea [GGH+13, WNL+14] allows us to avoid this recursion overhead.

As shown in Figure 1, each element in the ORAM tree corresponds to a node in the logical tree. We will have each element additionally store the position identifiers of its two children. Moreover, the client always stores the position identifier of the logical root. When the client has fetched some node in the logical tree, it immediately knows the position identifiers of its two children, and hence there is no more need to do a recursive position map lookup to find out the next child node it wants to read.

More formally, we can implement **Lookup**(k) as follows: Initially, let k' be the key of the logical root, let Pos denote the position identifier of the root. Let $D = 1.44 \log N$ which corresponds to the maximum depth of an AVL tree with N elements. Sample D new position identifiers denoted $\mathsf{Pos}'_0, \ldots, \mathsf{Pos}'_D$, and let $\mathsf{Pos}'_{D+1} = \perp$. Initialize $\mathsf{found} := \mathsf{false}$. Now, for $i = 1$ to $D + 1$, repeat the following:

- If not found, then read and remove the element with key k' from the path Pos. Let $(k', v, \mathsf{Pos}, (\mathsf{LKey}, \mathsf{LPos}), (\mathsf{RKey}, \mathsf{RPos}))$ be the element found:
  - if $k < k'$, then write back $(k', v, \mathsf{Pos}'_{i-1}, (\mathsf{LKey}, \mathsf{Pos}'_i), (\mathsf{RKey}, \mathsf{RPos}))$ to the root and let $k' := \mathsf{LKey}$;
  - else if $k > k'$, write back $(k', v, \mathsf{Pos}'_{i-1}, (\mathsf{LKey}, \mathsf{LPos}), (\mathsf{RKey}, \mathsf{Pos}'_i))$ to the root, and let $k' := \mathsf{RKey}$;
  - else if $k = k'$, let $\mathsf{found} := \mathsf{true}$, let $\mathsf{fetched} := (k', v, \mathsf{Pos}, (\mathsf{LKey}, \mathsf{LPos}), (\mathsf{RKey}, \mathsf{RPos}))$, and write back $(k', v, \mathsf{Pos}'_{i-1}, (\mathsf{LKey}, \mathsf{LPos}), (\mathsf{RKey}, \mathsf{RPos}))$ to the root;
- Else if found, then access a random path in the ORAM tree, and write back a filler element to the root.
- Perform eviction on the ORAM tree path just read.

Finally, the client updates the root's position identifier to be Pos'_0, and the result of the request is stored in $\mathsf{fetched}$.

Remark 1. In a non-oblivious AVL tree, the lookup algorithm stops whenever the desired key is found. However, in an oblivious dictionary, such early stopping would leak information about where the requested element is in the logical tree. Therefore, even after we have found the element, we still have to perform fake operations on the ORAM tree, such that all requests incur the same number of memory accesses.

Performance. The above algorithm involves visiting $O(\log N)$ ORAM tree paths for each lookup request. Therefore, the cost per **Lookup** is $O(\log^2 N)$.

Extending it to a dynamic dictionary. The above idea was first suggested by Gentry et al. [GGH+13] who considered a static logical tree that supports only **Lookup** but no dynamic insertion and deletion. Subsequently, Wang et al. [WNL+14] showed that it is not hard to extend the idea to support dynamic insertions and deletions. Specifically, during an insertion or deletion operation, the logical AVL tree may perform rotation operations to balance the logical tree. Each rotation operation involves modifying the pointers on constant number of nodes in the logical AVL tree. In our case, during a rotation, we just have to additionally update the children position identifiers for nodes involved in the rotation. Again, we have to pad all requests to the same length to avoid leakage through early stopping.

Remark 2. Using an optimal ORAM like OptORAMa [AKL+23] to directly compile an AVL tree also gives $O(\log^2 N)$ cost per operation. However, the resulting scheme is less practical and has only computational security, since the only known optimal ORAM constructions all rely on the existence of a pseudorandom function. The aforementioned constructions that makes non-blackbox usage of Path ORAM [SDS+18] or Circuit ORAM [WCS15]'s non-recursive ORAM tree are conceptually simpler, more efficient in hardware enclave or secure computation scenarios, and enjoy statistical security.

4 Oblivious Stack / Queue

An oblivious stack or queue can also be realized using a similar idea. For example, in an oblivious stack, each element can store the position identifier of the next element, and the client saves the position identifier of the top of the stack denoted Pos_0. For a push operation, the client assigns the new element to a random path Pos^*, and the have the new element record Pos_0 as the next element's position identifier. The client now writes the new element to the root bucket, and performs eviction on a random path. The client also updates $\mathsf{Pos}_0 := \mathsf{Pos}^*$. A pop operation can be implemented in a similar manner. In this way, each push or pop operation involves accessing only a single path in the ORAM tree, and the cost is $O(\log N)$.

A queue can also be realized in a similar manner, incurring $O(\log N)$ cost per PushBack or PopFront operation.

5 Oblivious Priority Queue

We first consider a basic priority queue with only the following two types of opearations — we will discuss how to support other operations such as DecreaseKey and Delete later:

- ExtractMin(): Return the element with the minimum key;
- Insert(k, v): Insert the value v with key k; or if $(k, v) = (\perp, \perp)$, then do nothing;

For simplicity, we shall assume that all elements have distinct keys, although it is not hard to extend the algorithm to support possibly duplicate keys.

We will describe Shi's construction of an oblivious priority queue [Shi20], also based on a non-recursive Path ORAM data structure.

Every element is assigned to a random path in the ORAM tree. We additionally augment each node v in the tree with an entry henceforth called the *subtree minimum*, which stores the minimum key in the subtree rooted at v, as well as the path this minimum element resides on.

- ExtractMin:
  - look up the root's subtree minimum; visit the corresponding path to fetch the element with the minimum key and remove this element from the path;
  - perform an evicition on the path just visited;
  - update the subtree minimum entries on the path just visited;
- Insert(k, v):
  - assign a random path to the element (k, v) and add it to the root, or if $(k, v) = (\perp, \perp)$, then pretend to add an element to the root;
  - perform eviction on a random path;
  - update the subtree minimum entries on the eviction path.

Observe that when one modifies a path, the subtree minimum entries of the path can be updated in logarithmic cost (i.e., proportional to the path length). Specifically, the leaf node on the path first computes its own subtree minimum from the elements contained in the leaf node. Next, starting from the level above the leaf back to the root, every node on the path can query the subtree minimum of its two children, and scan the elements contained in the node itself. This allows the node to correctly update its own subtree minimum.

Remark 3 (Hiding the type of operations)**.** The access pattern of the above algorithm are independent of the sequence of (key, value) pairs inserted, since every operation visits a random path whose choice has not been revealed before. However, the access patterns reveal whether the type of each operation is ExtractMin or Insert. It is not hard to modify the algorithm to hide the type of the operations too. In particular, we can have the ExtractMin operation perform a fake write to the root before the eviction step; and we can have the Insert operation begin with a fake look up along the eviction path.

Supporting additional operations. We now discuss how to support additional operations, such as FindMin, Delete and DecreaseKey . To define the syntax of Delete and DecreaseKey, we need to introduce the notion of a reference pointer (denoted ref) to reference the element being deleted or whose key is being decreased. Recall that the definition of Delete and DecreaseKey in the ordinary (non-oblivious) binary heap also used a reference pointer to refer to the element being deleted or whose key is being decreased. In our context, the reference pointer $\mathsf{ref} := (k, \mathsf{Pos})$ stores the key being deleted as well as the position identifier (denoted Pos) of the element.

Like in the ordinary (non-oblivious) binary heap, we shall assume that the $\mathsf{Insert}(k, v)$ operation returns a reference pointer to the element inserted, so we can later refer to them in Delete or DecreaseKey calls. It is not hard to modify our earlier insertion algorithm to return such a reference pointer. We now define the syntax of the additional operations:

- $(k, v) \leftarrow \mathsf{Delete}(\mathsf{ref})$: delete the element represented by ref and return the deleted element; or if $\mathsf{ref} = \perp$, then do nothing and return $\perp$;
- $\mathsf{DecreaseKey}(\mathsf{ref}, k)$: decrease the key of the element represented by ref to k.
- $(k, v) \leftarrow \mathsf{FindMin}$: return the element with the minimum key.

These operations can be supported as follows. $\mathsf{Delete}(\mathsf{ref})$ can be implemented by going to the corresponding path encoded in ref, reading and removing the element with the specified key, and performing an eviction on that path. Further, if $\mathsf{ref} = \perp$, then simply scan a random path and performing an eviction on that path.

$\mathsf{DecreaseKey}(\mathsf{ref}, k)$ can be implemented by calling $(k', v) \leftarrow \mathsf{Delete}(\mathsf{ref})$ first, and then calling $\mathsf{Insert}(k, v)$, or if $\mathsf{Delete}(\mathsf{ref})$ returned $\perp$, then call $\mathsf{Insert}(\perp, \perp)$ instead.

$\mathsf{FindMin}()$ can be implemented by first calling ExtractMin and then calling Insert to write back the extracted minimum element.

Finally, if one wishes to additionally hide the type of operations, then we can use a similar idea as in Remark 1 to make sure all requests exhibit the same access patterns.

Performance. It is not hard to see that all operations involve operating on $O(1)$ paths, and thus all operations require $O(\log N)$ cost.

6 Oblivious Turing Machine

The oblivious stack and queue constructions described earlier are randomized algorithms. It is actually possible to realize oblivious stacks and queues through a deterministic construction where each operation still takes $O(\log N)$ cost.

In fact, more generally, we can compile any Turing Machine to an oblivious Turing Machine (OTM) with $O(\log N)$ blowup in the running time, and the resulting OTM is deterministic. The main difference between a Turing Machine (TM) and a general Random Access Machine (RAM) is that in a Turing Machine, the current memory pointer can only move left or right by one position, or stay put — note that stacks and queues satisfy this so they have an efficient

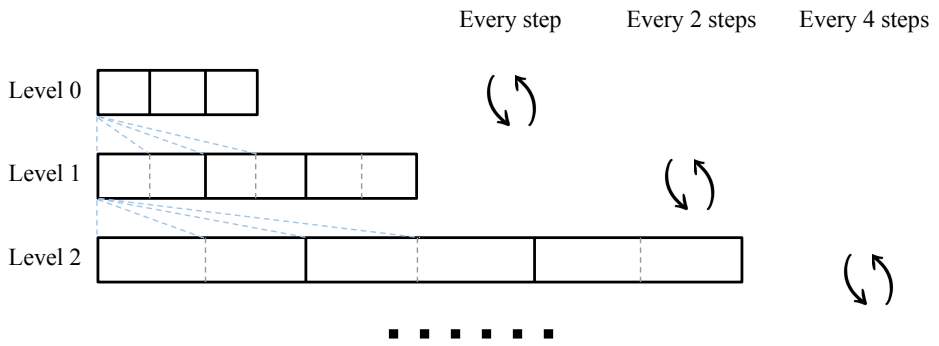

Figure 2: **Oblivious Turing Machine**: each level i contains 3 super-words of size 2^i.

implementation on a Turing Machine. By contrast, in a RAM, the current memory pointer can jump around to any location. For this reason, Turing Machines are easier to make oblivious than a RAM[1].

Oblivious Turing Machine was first shown by Pippenger and Fischer [PF79]. In this lecture, we will describe a simplified variant of their construction.

6.1 Construction

Henceforth we assume that the original Turing Machine has space $N = 3 \cdot 2^D$. As shown in Figure 2, in our OTM construction, we use a hierarchical data structure where the levels are indexed $0, 1, \ldots, D$. Each level $i \in \{0, 1, \ldots, D\}$ contains 3 superwords of size 2^i. A superword of size 2^i is simply an array of 2^i words.

Initialization. Initially, the largest level D contains the memory of the TM. For every $i = D \ldots 1$, level $i - 1$ *obliviously* copies three super-words of size 2^{i-1} from the next level i, such that the current memory pointer is in the middle superword in level $i - 1$.

Note that when level $i - 1$ copies from the next level i, the starting position of the copy can only be one of the four choices as shown in Figure 2. To hide which three superwords are copied from the next level, we have to pretend to make all four potential copies, such that only one of these copies is a real copy, and the rest are just fake operations.

Operations. In each time step $t = 1, 2, \ldots$, we will read and write the memory word at the current memory pointer, and then, the memory pointer either moves left/right, or stays put. To support this, our OTM performs the following:

- Perform a linear scan of level 0 to perform the read/write operation.
- Let 2^L be the maximum power of 2 that divides t. Now do the following:
  - For $i = 0$ to $\min(L, D-1)$, level i *obliviously* writes back its contents to the corresponding locations in level $i + 1$. To hide which locations are written in level $i + 1$, we pretend to make all four possible writebacks where only one of them is a true copy.
  - For $i = \min(L, D-1)$ downto 0, level i *obliviously* copies three superwords of size 2^i from level $i + 1$ such that the new memory pointer is centered in the middle superblock. To hide which locations are fetched from level $i + 1$, we pretend to make all four possible copies where only one of them is a true copy.

[1] One can show that to get an Oblivious RAM with non-trivial overhead, randomness is necessary. By contrast, OTM can be achieved with a deterministic construction.

The above OTM construction has deterministic access patterns regardless of the memory pointer movements in each time step. It is not hard to verify correctness of the scheme.

To analyze the cost, observe that every 2^i steps, level i and level $i+1$ need to communicate back and forth, and the cost of this communication is upper bounded by $C \cdot 2^i$ where C is some constant. Therefore, the total amortized cost per time step is:

$$C \cdot 2^0/2^0 + C \cdot 2^1/2^1 + \ldots C \cdot 2^{D-1}/2^{D-1} = O(\log N)$$

References

[AKL+23] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious RAM. *J. ACM*, 70(1):4:1–4:70, 2023.

[GGH+13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, apr 1979.

[SDS+18] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.

[Shi20] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 842–858. IEEE, 2020.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 850–861. ACM, 2015.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226, 2014.