Cryptography meets algorithms (15893) Lecture Notes

Lecture 12: Oblivious Data Structures

Scribe: Tanisha Saxena

March 12, 2024

1 Motivation

In previous lectures, we learned how ORAM allows up to compile any program into an oblivious counterpart by simply using ORAM instead of RAM. However, this sort of generic application can cause unnecessarily high overhead. For example, in previous lectures we saw how ORAM could be used on sorting which caused $O(\log n)$ overhead. Instead, we can actually use the non-recursive tree idea from Path ORAM to create oblivious data structures that allow us to more easily create oblivious protocols with lower overhead [SDS+18].

2 Background

Recall the recursive tree data structure used in Path ORAM [SDS+18]. In this protocol, the client stores a small local stash. The server-side storage is treated as a tree of buckets where each block is stored in a random leaf and the unique path from the root to said leaf contains all the blocks not in the client stash. An access to Path ORAM can be described by the simple pseudocode shown in Figure 1.

3 Oblivious Dictionary

3.1 Operations

For an oblivious dictionary, we want to support the following operations [WNL+14]:

- **Lookup(k):** Return the value with key k
- **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 can be viewed as a special case of the dictionary (lookup is a "read" and insert is a "write") where the key space is contiguous.

3.2 Construction

A naive way to implement dictionaries would be to compile a classic logical tree with ORAM. This would involve giving each element a key alongside its value to generate a logical tree as depicted in Figure 2. Even if we assume, for simplicity, that the tree is balanced and static, it's still an inefficient way to make an oblivious dictionary. Instead, we can encode a logical tree *within* the recursion used by Path ORAM itself.

Access(op, a, data*):

1: $x \leftarrow$ position[a]
2: position[a] $\leftarrow x^* \leftarrow$ UniformRandom$(0 \dots 2^L - 1)$
3: **for** $\ell \in \{0, 1, \dots, L\}$ **do**
4: $\quad S \leftarrow S \cup$ ReadBucket$(\mathcal{P}(x, \ell))$
5: **end for**
6: data $\leftarrow$ Read block a from S
7: **if** op $=$ write **then**
8: $\quad S \leftarrow (S - \{(\mathsf{a}, x, \mathsf{data})\}) \cup \{(\mathsf{a}, x^*, \mathsf{data}^*)\}$
9: **end if**
10: **for** $\ell \in \{L, L-1, \dots, 0\}$ **do**
11: $\quad S' \leftarrow \{(\mathsf{a}', x', \mathsf{data}') \in S : \mathcal{P}(x, \ell) = \mathcal{P}(x', \ell)\}$
12: $\quad S' \leftarrow$ Select $\min(|S'|, Z)$ blocks from S'.
13: $\quad S \leftarrow S - S'$
14: $\quad$ WriteBucket$(\mathcal{P}(x, \ell), S')$
15: **end for**
16: **return** data

Figure 1: Pseudocode taken from [SDS+18] showing the logic necessary to make an oblivious access in the Path ORAM model

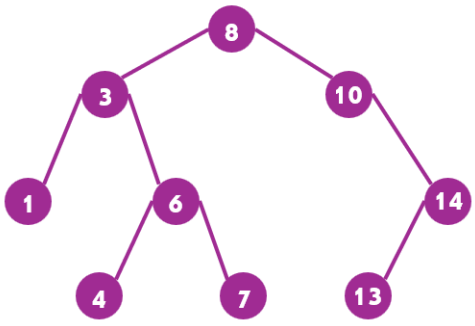

Figure 2: Naive logical tree compiled with ORAM

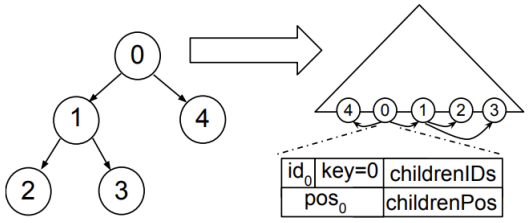

Figure 3: A diagram representation of a block within the oblivious binary tree [SDS+18]

The implementation of a dictionary with binary search trees is simple and intuitive. Each block can be described as containing the following: a key, a value, a pointer to the current block's path, the left key, a pointer to the left child's path, the right key, and a pointer to the right child's path as can be seen in Figure 3.

3.3 Access Protocol

The only part of the access protocol that differs significantly from the regular setup of Path ORAM is the existence of parents and children. Thus, we must simply maintain the invariant that the parent's child pointers get updated whenever the child's own path is changed on access. This idea can be extended for insertion and deletion in dynamic trees but it remains within the same complexity class for overhead. Beyond that, the access protocol is very similar to the pseudocode shown in Figure 1.

3.4 Overhead

The improved oblivious data structure has an overhead of $O(\log^2 n)$ with $O(\log n)$ from the recursion level and $O(\log n)$ from the size of the tree. This is an improvement in contrast to the naive approach that compiles binary searching–which automatically incurs a $O(\log n)$ search cost [GGH+13]– with Path ORAM–that already has a $O(\log^2 n)$ lookup cost– giving an overall $O(\log^3 n)$ overhead.

It is possible to achieve $O(\log n)$ overhead as shown by Asharov et. al. [AKM23]. With this low overhead, this implementation of an oblivious dictionary finally becomes practical and a viable choice for real-world security. However, there are many restrictions and assumptions that must be met before FuturORAMa can be used and thus some people still prefer to use Path ORAM or Circuit ORAM despite the larger overhead.

4 Oblivious Stacks / Queues

4.1 Operations

Zahur et. al. also proposes a circuit-based construction of a stack/queue but, for the proof of concept, we will only describe the Path ORAM implementation for stacks [ZWR+16]. Note that oblivious queues can be made in a very similar way. The oblivious stack supports the following operations [Tof11]:

- **Push(v):** Push the value v on the stack
- **Pop():** Pop the top element off the stack and return it

4.2 Construction

You only need one tree to store a stack. Then, the head pointer holds the path on which the head of the stack resides. Each item holds a value, v, a pointer to itself, and a pointer to the next block's path.

4.3 Access Protocol

Push simply requires us to choose a path for the new element and then put it onto the stack by setting it to the new head. Pop is very similar and just pops off and removes the current head of the stack.

4.4 Overhead

Because there is only one tree and we only need to worry about modifying one element in the tree at a time, each operation takes $O(\log n)$ time. Note that this is a randomized algorithm because there is only one tree and thus this scheme uses non-recursive Path ORAM. We'll learn about a deterministic oblivious algorithm in a later section on oblivious Turing machines.

5 Oblivious Heaps / Priority Queues

This oblivious data structure constructed using non-recursive Path ORAM is somewhat similar to that of oblivious stacks and queues but has a few notable differences. For simplicity, we are again assuming that all keys are distinct.

5.1 Operations

- **Extract_min():** Returns the item with the minimum priority
- **Insert(k, v):** Inserts the value v with priority k

5.2 Construction

Using the hierarchical ORAM proposed by Kasper and Larsen, we can store the elements in order with a block containing the the value, a self-referential pointer, a pointer to this subtree minimum's path, and the subtree's minimum itself [LN18] [AKM23]. [WNL$^+$14].

5.3 Access Protocol

5.3.1 Extract_min()

In extract-min(), we first get the minimum value of this tree, aka the subtree minimum at the root. Let ptr^* denote the path the subtree minimum value resides on. We want to remove the subtree minimum value's record from the path stored in ptr^*. To do this, we evict on the path ptr^* and, to maintain correctness, we recalculate the subtree minimum values stored in each node along ptr^*'s path. Intuitively, since we're removing the subtree minimum from the path, the subtree minimum of all blocks along that path can change. Note that the subtree minimum of a parent is equal to the minimum between the subtree minimums of its two children, so as long as we update from the leaf node upwards, it only requires $O(\log n)$ overhead.

5.3.2 Insert(k, v)

First, we choose a path at random and assign that to the block we want to insert. Then, we add our new block to the root of the tree. Lastly, we perform an eviction on two random paths and recalculate the subtree minimum for each node on the two evicted paths to maintain the correctness property. To maintain the oblivious property, we simply pad either operation such that both operations read the same number of paths. This makes the operations indistinguishable from each other to an outside perspective.

5.3.3 Other operations

We can also implement "$delete(ref)$" where ref is a reference to the pointer we want to delete, and "$decrease\_key(ref, k')$" where we decrease the key of the value ref is referencing to k'. In both these cases, we use references to access the path and ensure that correctness is maintained over the tree.

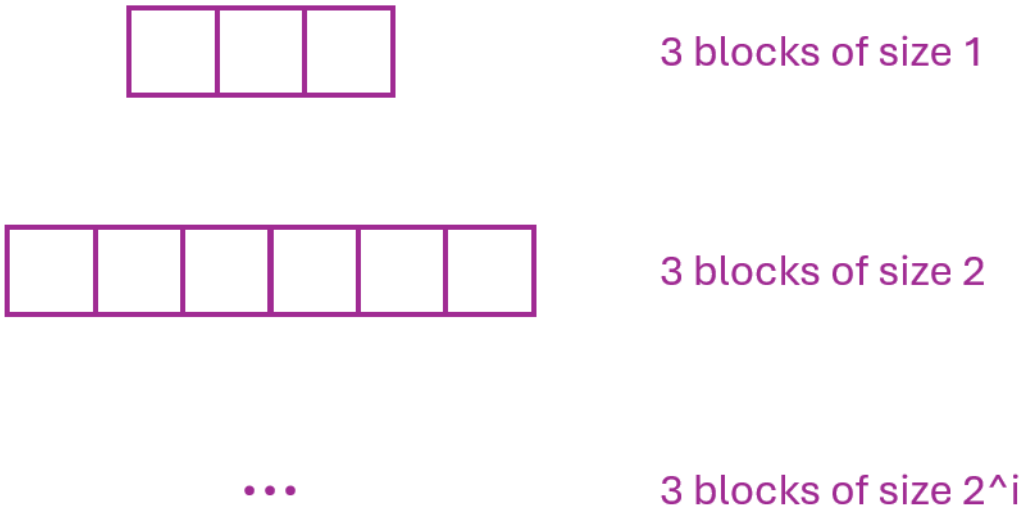

Figure 4: A diagram representation of the levels within an oblivious Turing machine

- **Delete(ref):** Access the path pointer denoted by ref, removes the item with key k, evicts along the path pointed to by ref, and recalculates the subtree minimum for each node along the path
- **Decrease_key(ref, k):** Delete(ref), insert(k', v)

5.4 Overhead

As expected, since insertion and extract-min() only require us to update the path the element resides on in linear time, they only incur a $O(\log n)$ overhead.

6 Oblivious Turing Machines

6.1 Operations

As a reminder, in the oblivious Turing machine model we want to hide whether the Turning machine is going left, right, or staying still on the tape.

6.2 Construction

Each level of the structure consists of three "superblocks" increasing exponentially in size by a power of two (ex: the first level has three superblocks of size 1, the second has three superblocks of size 2, the ith level has three superblocks of size 2^i, and so on) as seen in Figure 4. We denote $HEAD$ to be the head pointer of the Turing machine. We then maintain the following invariant throughout the protocol: When a level is rebuilt–for all except the last nth level–HEAD lies in the middle superblock.

We also want an invariant that indicates how, if level i is updated, all levels 1 to $(i-1)$ should be notified that i has been updated. The reason for that is each element has multiple copies across levels, thus we must ensure that each copy is updated when the element is updated. This propagation can be done in linear time with respect to the level size.

6.3 Access Protocol

The access protocols are easy to implement because a Turing machine can only "read" or "write" which can be done with the same access protocol as ORAM as shown in Figure 1.

6.4 Overhead

The ith level is rebuilt every 2^i steps with a rebuilding cost in $O(2^i)$. Thus, over the protocol the costs to rebuild are:

Level 0: $c \cdot 2^0 = c$

Level 1: $c \cdot 2^1 \cdot$ rebuild every 2 steps $\Rightarrow c \cdot 2^0$(amortized) (Rebuild cost is linear)

etc ...

At each level we get c work and thus we incur $O(\log n)$ cost per operation. This result is actually quite old and has been proven since Fischer and Pippenger in 1979 [PF79].

References

[AKM23] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. Futorama: A concretely efficient hierarchical oblivious ram. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 3313–3327, 2023.

[GGH+13] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, 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 13*, pages 1–18. Springer, 2013.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *Annual International Cryptology Conference*, pages 523–542. Springer, 2018.

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

[Tof11] Tomas Toft. Secure data structures based on multi-party computation. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 291–292, 2011.

[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.

[ZWR+16] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root oram: efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 218–234. IEEE, 2016.