

## Lecture 10: Hierarchical ORAM

Scribe: Qi Pang

March 12, 2024

In an earlier lecture, we learned oblivious sorting, including bitonic sort with  $\mathcal{O}(n \log^2 n)$  overhead [Bat68], bucket oblivious sort [ACN<sup>+</sup>] using oblivious random permutation to achieve  $\mathcal{O}(n \log n (\log \log n)^2)$  overhead, and we showed that we can future eliminate the  $(\log \log n)^2$  factor [RS21, GWC<sup>+</sup>23]. We will cover Hierarchical ORAM [GO96] in today's lecture, which consists of a hierarchy of oblivious hash tables. So, firstly, let's introduce oblivious hash table.

### 1 Oblivious Hash Table

**Definition 1** (Oblivious Hash Table). Consider a data array  $A = \{(k_i, v_i) | \perp\}_{i \in [N]}$ , where  $(k_i, v_i)$  is a real key-value pair element and  $\perp$  is a filler. And it's guaranteed that all real elements have distinct keys. An oblivious hash table contains the following operations:

- **Build( $A$ ):** Given a data array  $A$ , output a data structure.
- **Lookup( $k$ ):** Given a key  $k$ , output  $v$ . If  $k = \perp$  or  $k \notin A$ ,  $v = \perp$ , otherwise  $(k, v) \in A$ .

$\forall A_0, op_0, A_1, op_1$  s.t.  $|A_0| = |A_1|, |op_0| = |op_1|$  and both  $(A_0, op_0)$  and  $(A_1, op_1)$  satisfy the non-recurrent constraint, then the oblivious hash table requires:

$$\text{AccessPattern}(A_0, op_0) \approx \text{AccessPattern}(A_1, op_1)$$

That is, for any data array and operation sequences of the same length, the access patterns in oblivious hash tables are indistinguishable as long as the same key is never looked up twice.

Now let's introduce how to build an oblivious hash table. We use the balls and bins hashing with  $N$  elements (balls) and  $N/Z$  bins, where  $Z = \omega(\log N)$ . The expected number of elements in any fixed bin is  $Z$ . With Chernoff bound, the probability of any bin having more than  $2Z$  elements is negligible. Thus, we set the bin size to be  $2Z$ .

So now, we want to obviously throw the balls into the bins without revealing which bin each ball goes to. One way to randomly assign the element  $(k, v)$  to a bin is to use a pseudorandom function  $\text{PRF}_{sk}(k)$ , where the  $sk$  is the secret key held by the ORAM client and the PRF output range is  $[N/Z]$ , indicating the bin index. To do this obviously, we can use the butterfly network to route the elements to their respective bins, which is similar to the oblivious random permutation. So, everytime the clients lookup a fresh element, the ORAM servers will see they are accessing a random bin. In case if it's a filler element, the client will just lookup a random bin.

**Analysis.** Building an oblivious hash table involves routing elements to random bins. Recall that in oblivious random permutation, we can accomplish this with  $\mathcal{O}(N \log N)$  overhead [RS21, GWC<sup>+</sup>23]. For every lookup, the cost is the bin size, which is  $Z = \omega(\log N)$ . Different from static hashing that has constant cost for each lookup, oblivious hash tables have a bit higher overhead.

## 2 Hierarchical ORAM

So now with this oblivious hash table, we can construct a Hierarchical ORAM. In Hierarchical ORAM, we have a hierarchy of exponentially growing oblivious hash tables as shown in Figure 1. Each level of the hierarchy has a different capability, and the capability grows exponentially with the level. The  $i$ -th level's capability is  $2^i$ . The root level has the largest capability  $2^L$ , where  $L = \lceil \log N \rceil$ , and the leaf level has the smallest capability 1. Additionally, every level has a label, either **available** (**empty**) or **full**.

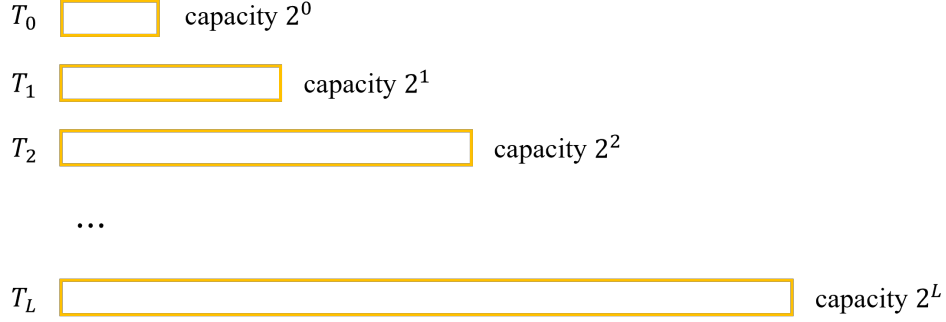


Figure 1: Hierarchical ORAM

Now let's consider two operations, **Read(addr)** and **Write(addr, data)**. Before we dive into the detailed algorithm, let's first introduce the high-level idea of how to perform these operations.

If we want to lookup some element, we will start from the smallest level. Since every level is either empty or full. If it's full, lookup in that hash table for the element. There are two outcomes: either we find the element, or we don't find the element. If we find the element, for all future levels, we just lookup filler and perform a fake operation. Similar to the tree-based ORAM, we will also need to relocate the retrieved element. So in Hierarchical ORAM, we will put the retrieved element to the smallest level if this level is empty. If this level is not empty, we will need to merge the retrieved element with the existing element in this level and put the merged element to the next level. Otherwise, if this level is still not empty, merge them again and put them to the next level, until we find the empty level to put the merged element.

**Algorithm.** The detailed algorithm is shown in Algorithm 1. The algorithm starts by initiating a binary variable **found** to **false**. Then, for each level, we check if the element is in the hash table. If we find the element, we set **found** to **true** and store the element in **data\***. If the element is already found, we just lookup filler in the hash table for the rest of the levels to prevent leakage in the access patterns. After we go through all levels, the fetched element is in **data\***.

In the following we will do the relocate operation. If the operation **op** is **read**, we will need to relocate the retrieved element  $T^\phi = \{(\mathbf{addr}, \mathbf{data}^*)\}$ . Otherwise, if the operation **op** is **write**, we will need to relocate the new element  $T^\phi = \{(\mathbf{addr}, \mathbf{data})\}$ . Let  $l$  be the smallest level such that  $T_l$  is empty, if all levels are full, let  $l = L$ . Let  $S$  be the set of elements to relocate, which is the union of the new element and all the elements in the hash tables from level 0 to level  $l - 1$ . If all levels are full, let  $S$  be the union of all the elements and the new element  $T^\phi$ . Then we build the hash table at level  $l$  with the set  $S$ . The build step is the same as the oblivious hash table, which is to throw the elements into the bins obliviously. After the build step, we mark the level  $l$  as full and the levels 0 to  $l - 1$  as empty.

Note that if the oblivious hash table doesn't remove the element we just looked up, there might be duplicate elements in the set  $S$  and we will need to suppress the duplicate elements. The elements in the smaller level is always fresher than the elements in the larger level, so we can just keep the fresher element and remove the older one. We use the oblivious sorting to

do the **SuppressDup**, which simply sorts the elements by key and by freshness and removes the duplicate elements using a linear scan.

---

**Algorithm 1:** HIERARCHICAL ORAM

---

**Input:** `addr`, an address to lookup in the hash tables; `op`, the operation `write` or `read`;  
`data`, the data to write.

**Output:** `data*`, the lookup result, if `op = read`.

```

1 found  $\leftarrow$  false;
2 for  $l \leftarrow 0$  to  $L$  do
3   if not found then
4     fetch  $\leftarrow T_l.\text{lookup}(\text{addr})$ ;
5     if fetch  $\neq \perp$  then
6       found  $\leftarrow$  true;
7       data*  $\leftarrow$  fetch;
8   else
9      $T_l.\text{lookup}(\perp)$ 
10 if op = read then
11    $\text{Let } T^\phi \leftarrow \{(\text{addr}, \text{data}^*)\}$ ;
12 else if op = write then
13    $\text{Let } T^\phi \leftarrow \{(\text{addr}, \text{data})\}$ ;
14  $\text{Let } l$  be the smallest level such that  $T_l$  is empty, if all levels are full, let  $l = L$ ;
15  $\text{Let } S \leftarrow T^\phi \cup T_0 \cup \dots \cup T_{l-1}$ ;
16 If all levels are full, let  $S \leftarrow S \cup T_L$ ;
17  $T_l.\text{Build}(\text{SuppressDup}(S))$ ;
18 Mark  $T_l$  as full;
19 Mark  $T_0, \dots, T_{l-1}$  as empty;
20 if op = read then
21   return data*;
```

---

**Analysis.** Now let's analyze the overhead of Hierarchical ORAM. There are two sources of the overhead: the lookup phase, and the rebuild phase. The lookup cost is  $L \times Z$ , the number of levels  $L$  times the lookup cost of at each level, which is at most  $Z$ . Recall  $Z = \log N \cdot \alpha(N)$ , where  $\alpha(N)$  is the superconstant function. Thus the lookup cost is  $\mathcal{O}(\log^2 N \cdot \alpha(N))$ .

For rebuild cost, a level of size  $2^i$  is rebuilt every  $2^i$  steps. So to rebuild a level of size  $2^i$ , the cost is  $2^i \cdot \log 2^i$  (see the analysis of the oblivious hash table). By amortizing the cost to each step, every level has a cost  $\leq \log N$ . Thus, the rebuild cost is  $\mathcal{O}(\log^2 N)$ . Note that the original Goldreich-Ostrovsky construction [GO96] has a cost of  $\mathcal{O}(\log^3 N)$ , because that they throw  $N$  balls into  $N$  bins, instead of  $N/Z$  bins.

## References

- [ACN<sup>+</sup>] Gilad Asharov, T-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. *Bucket Oblivious Sort: An Extremely Simple Oblivious Sort*, pages 8–14.
- [Bat68] K. E. Batchier. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring)*, page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.

- [GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, may 1996.
- [GWC<sup>+</sup>23] Tianyao Gu, Yilei Wang, Bingnan Chen, Afonso Tinoco, Elaine Shi, and Ke Yi. Efficient oblivious sorting and shuffling for hardware enclaves. Cryptology ePrint Archive, Paper 2023/1258, 2023. <https://eprint.iacr.org/2023/1258>.
- [RS21] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, pages 373–384. ACM, 2021.