Cryptography Meets Algorithms (15893) Lecture Notes

Lecture 10: Hierarchical ORAM

Scribe: Qi Pang

March 18, 2024

In an earlier lecture, we learned how to construct oblivious sorting, including bitonic sort which has $\mathcal{O}(n \log^2 n)$ cost [Bat68], bucket oblivious sort [ACN$^+$] which has $\mathcal{O}(n \log n (\log \log n)^2)$ cost, and we showed that we can future eliminate the $(\log \log n)^2$ factor using techniques from [RS21, GWC$^+$23].

We will cover Hierarchical ORAM [GO96] in today's lecture. Chronologically, Hierarchical ORAM was actually the original ORAM construction first proposed by Goldreich and Ostrovsky [GO96]. The version we will describe in today's class is *a simplified and optimized variant* of Goldreich and Ostrovsky's original construction [GO96] introduced in Chan et al. [CGLS17]. Since hierarchical ORAM consists of a hierarchy of oblivious hash tables, let's first introduce oblivious hash table.

1 Oblivious Hash Table

We consider a standard static hash table abstraction.

Definition 1 (Static Hash Table)**.** Consider a data array $A = \{(k_i, v_i)|\perp\}_{i\in[N]}$, i.e., each element is either a *real* key-value pair denoted (k_i, v_i) or a *filler* denoted $\perp$. It is guaranteed that all real elements have distinct keys. An oblivious hash table contains the following operations:

- $\mathsf{Build}(A)$: Given a data array A, output a data structure.
- $\mathsf{Lookup}(k)$: Given a key k, output v. If $k = \perp$ or $k \notin A$, then output $v = \perp$; otherwise the output v satisfies $(k, v) \in A$.

The hash table we will construct satifies obliviousnes only if the same real key is never looked up twice. It turns out that such a hash table is sufficient for constructing hierarchical ORAM. We now formally define this non-recurrent requirement. A sequence of Lookup operations denoted *op* is said to be *non-recurrent* if every real key requested is distinct. However, the sequence *op* is allowed to make filler requests $k = \perp$ multiple times.

Obliviousness. Obliviousness requires the following: $\forall A_0, op_0, A_1, op_1$ s.t. $|A_0| = |A_1|, |op_0| = |op_1|$ and both op_0 and op_1 satisfy the non-recurrent constraint, then it holds that

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

where $\text{AccessPattern}(A, op)$ denotes the access patterns emitted by the oblivious hash table when making a single call to $\mathsf{Build}(A)$ followed by a sequence of Lookup operations denoted *op*, and $\approx$ means computational indistinguishability. More intuitively, we want that for any two data arrays 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.

Construction. 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[1], 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 in N. Thus, we set the bin size to be $2Z$. One way to randomly assign the element (k, v) to a bin is to use a persudorandom function $PRF_{sk}(k)$, where the sk is the secret key held by the ORAM client and the PRF output range is $[N/Z]$, indicating the desired bin index.

So now, we want to *obliviously* throw the balls into the bins without revealing which bin each ball goes to. At the end, each bin is padded with filler elements to the maximum capacity. To achieve this, we will use the "oblivious random bin assignment" construction from the last lecture — recall from the last lecture that oblivious random bin assignment was a key building block for constructing an oblivious random permutation, which in turn served as a key building block in the construction of bucket oblivious sort. More specifically, the initial labels of an element with key k is computed by $PRF_{sk}(k)$, and then we use a butterfly network (where all intermediate bins are of size $2Z$) to route all elements to their desired destination bins[2].

To implement $\mathsf{Lookup}(k)$, if k is a real key, the client simply reads the entire bin indexded $PRF_{sk}(k)$. If $k = \perp$, then the client reads a random bin.

Analysis. Building an oblivious hash table calls the "oblivious random bin assignment" algorithm of the last lecture. Recall that in the last lecture, we mentioned that using techniques from [RS21, GWC+23], we can accomplish this with $\mathcal{O}(N \log N)$ cost. For every lookup, the cost is the bin size, which is $Z = \omega(\log N)$. Different from (non-oblivious) static hashing that has constant cost for each lookup, our oblivious (static) hash table has a bit higher overhead.

2 Hierarchical ORAM

So now with this oblivious hash table, we can construct a Hierarchical ORAM. The hierachical ORAM essentially uses Bentley and Saxe's hierarchical data structure idea [BS80] that converts a static data structure (in our case, an oblivious hash table) into a dynamic one.

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 capacity, and the capacity grows exponentially with the level. The levels are indexed $0, 1, \ldots, L$, and the i-th level's capacity is 2^i. Additionally, every level has a label, either **empty** or **full**. "Empty" means that this level current is not being used, and "full" means that this level is currently in use. Whether each level is empty or full only depends on the time step (i.e., how many operations have been performed so far), so this information is public and is known by the adversary.

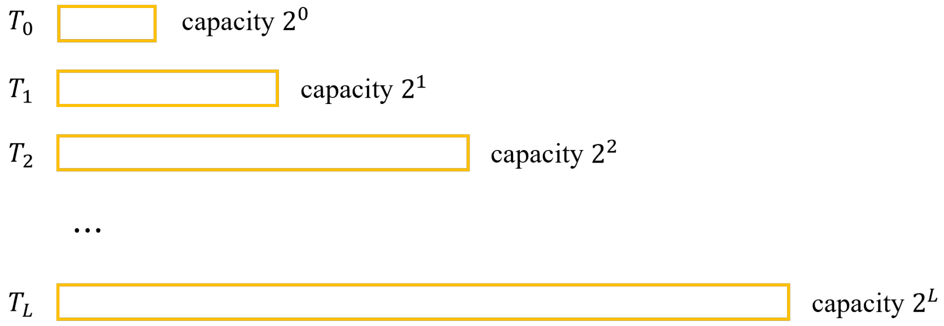

Figure 1: Hierarchical ORAM

[1] For simplicity, we assume that N is divisible by Z.

[2] Note that it is also possible to route the elements to bins using oblivious sorting itself as a building block.

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.
found ← $false$;
for $l \leftarrow 0$ to $L$ do
    if not found then
        fetched ← $T_l$.Lookup(addr);
        if fetched $\neq \perp$ then
            found ← $true$;
            data* ← fetched;
    else
        $T_l.lookup(\perp)$
if op = read then
    Let $T^{\phi} \leftarrow \{(\texttt{addr}, \texttt{data}^*)\}$;
else if op = write then
    Let $T^{\phi} \leftarrow \{(\texttt{addr}, \texttt{data})\}$;
Let $l$ be the smallest level such that $T_l$ is empty, if all levels are full, let $l = L$;
Let $S \leftarrow T^{\phi} \cup T_0 \cup \cdots \cup T_{l-1}$;
If all levels are full, let $S \leftarrow S \cup T_L$;
$T_l$.Build(SuppressDup($S$));
Mark $T_l$ as full, mark $T_0, \cdots, T_{l-1}$ as empty;
if op = read then
    return $data^*$;
```

Recall that we want to implement a memory abstraction with the address space $1 \dots N$. Initially, we may assume that all memory cells are empty and contain the symbol $\perp$. Therefore, initially all the ORAM levels are marked empty. However, later, as we perform read and write operations, at any point of time, some levels might be full and others are empty.

Now suppose a sequence of operations arrive. There are two types of operations, `Read(addr)` and `Write(addr, data)`.

Intuition. Before diving into the detailed algorithm, let's first introduce the high-level idea of how to perform these operations.

If we want to look up some element, we will start from the smallest level. Since every level is either empty or full. If it's full, look up that hash table for the desired element. There are two outcomes: either we find the element, or we don't find the element. If we have found the element, for all future levels, we just perform a fake lookup by requesting a filler element $\perp$. 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 and merged hash table of size 2 is copied to the next level if it is empty. However, if the next level is not empty, then it causes a cascading merge until we find a level that is empty or we reach the largest level.

Algorithm. The detailed algorithm is shown in Algorithm 1. We use $T_0, \dots T_L$ to denote the oblivious hash table of the $L + 1$ levels. 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 look up a filler for all future levels — importantly, *this guarantees that the non-recurrent constraint is satisfied for every oblivious hash table.* After we go through all levels, the fetched element is in data*.

If the operation op is read, we will need to write back the retrieved element $T^\phi = \{(\texttt{addr}, \texttt{data}^*)\}$. Otherwise, if the operation op is write, we will need to write back the new element $T^\phi = \{(\texttt{addr}, \texttt{data})\}$. Let l be the smallest level such that T_l is empty, if all levels are full, let $l = L$. We want to merge all levels $0 \ldots l-1$ along with the new element into level l (or merge all levels $0 \ldots L$ along with the new element into the largest level L). Then we obliviously build the hash table at level l with the input set S. 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 can use the oblivious sorting to do the duplicate suppression, denoted SuppressDup in Algorithm 1. Specifically, we obliviously sort the elements by key and by freshness and removes the duplicate elements using a linear scan.

Analysis. Now let's analyze the overhead of Hierarchical ORAM. For each request, 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 the bin size Z. Recall we can set $Z = \log N \cdot \alpha(N)$, where $\alpha(N)$ is an arbitrarily small super-constant 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 $\mathcal{O}(\log^2 N)$.

Summarizing the above, each memory request has cost $\mathcal{O}(\log^2 N \cdot \alpha(N))$ where $\alpha(N)$ may be an arbitrarily small super-constant function. 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.

Remark 1. Bentley and Saxe's hierarchical data structure idea [BS80] is a well-known and widely used technique for converting static data structures into dynamic ones, for example, see 15451 lecture notes (Section 2.2 in https://www.cs.cmu.edu/~15451-s23/lectures/lec03-amortized.pdf). There has also been various applications in the cryptography literature besides ORAM, e.g., to obtain dynamic versions of proofs of retrievability [SSP13], searchable encryption [SPS14], and private information retrieval [KC21].

References

[ACN+] 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. Batcher. 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.

[BS80] Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[CGLS17] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In

Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I, volume 10624 of *Lecture Notes in Computer Science*, pages 660–690. Springer, 2017.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, may 1996.

[GWC$^+$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`.

[KC21] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 875–892. USENIX Association, 2021.

[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.

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[SSP13] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 325–336. ACM, 2013.