CS670 Assignment-3

Chayan Kumawat - 220309

Program:	B.Tech
Branch:	CSE
Batch:	2026
Date of Sub:	21.04.2024 (DD-MM-YYYY)

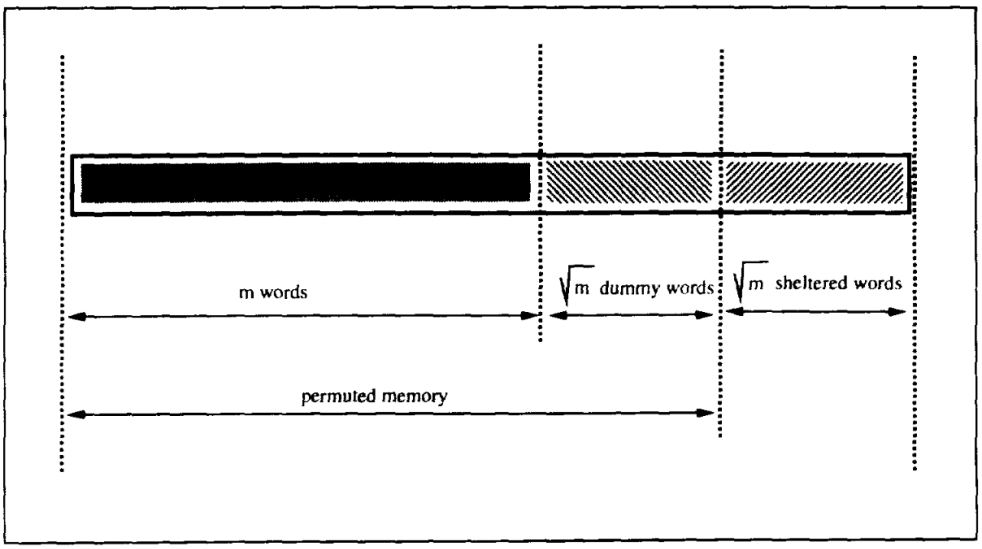

Figure 1: Data Structure for "square root ORAM" solution

Question 1

1. Square Root ORAM

The Square Root ORAM construction here simulate such a RAM by an RAM of size $N+2\sqrt{N}$, such that t steps of the original RAM are simulated by $O(t \cdot \sqrt{N})$ steps of the oblivious RAM.The Square Root ORAM proceeds in rounds(the number of access), and simulates $\sqrt{N}$ blocks operations of the client in a round. (Further, we will discuss what we mean by rounds)

Basic Square Root ORAM construction:

The first N blocks are the encrypted database blocks. The extra storage of $2\sqrt{N}$ contains two parts: the first $\sqrt{N}$ blocks are called *dummy block locations*, and the second $\sqrt{N}$ blocks are called *stash/cache*. These extra blocks have their particular purposes, essentially playing an important role in hiding the access pattern, in the case if the client is reading the same block of memory more than once.

- In the initialization phase, the first N blocks of cloud storage store the encrypted data blocks, with identifiers ranging from 1 to N, corresponding to the client database. The additional $2\sqrt{N}$ blocks (located in the dummy and stash locations) are initialized with randomly generated blocks by the client, each with identifiers from $(N+1)$ to $(N+2\sqrt{N})$.
- Just after initialization, the block with identifier i is located at the ith position. (Here, the block ID is encrypted and hidden from the cloud server).
- First, at the beginning of each $\sqrt{N}$ query access by the client, the client ReEncrypt the whole cloud storage and then performs an *oblivious random permutation* with the cloud as described in the Appendix.
- The blocks in locations 1 through $(N+\sqrt{N})$ are randomly permuted according to a permutation π. The Basic-SR ORAM assumes that the client has only O(1) storage; hence, this permutation is performed by interactive oblivious sorting with the cloud. In this process, the client uses a hash function $h : \text{id} \rightarrow Z^+$ to map a block identifier to a random hash value (to be attached to a block and used for oblivious sorting to achieve an oblivious random permutation).

- Note that using a standard cryptographic hash function such as SHA-256, h uses only constant space, and the client can remember h locally. Since the block ID is encrypted inside a block, after the oblivious random permutation, the cloud loses track of where these $N + \sqrt{N}$ blocks (data + dummy blocks) are located.

- The one set of $\sqrt{N}$ after the "re-shuffling" that we stated above is proceeded as follows: in this set, the jth operation (id_j) is simulated as follows. The client retrieves the shelter locations and checks whether the block with identifier id_i is in one of these locations.

- If the block with id_j is found in location x where $x \in [N + \sqrt{N} + 1, N + 2\sqrt{N}]$, the client accesses a dummy block in location $\pi(N + \text{count})$ (reads and writes the re-encrypted, same block back), where *count* is an index to keep track of the number of operations performed so far in the current round. Otherwise, the client retrieves the block with identifier id_j in location $\pi(id_j)$. Since the permutation function π is not stored in the client (which would require $O(N + \sqrt{N})$ space), to access location $\pi(N + \text{count})$ or $\pi(id_j)$, an interactive binary search is performed on blocks in locations 1 through $N + \sqrt{N}$ between client and cloud, according to the hash values produced by h. Lastly, the client scans through the $\sqrt{N}$ shelter locations again, reads and writes each shelter block in this process . In particular, if the block id_j has been found in a shelter location and the operation is write, the client writes block j into the same location. If the block id_j has not been found in any shelter location, and has been retrieved among $N + \sqrt{N}$ data and dummy locations, the client writes either that block (re-encrypted) or block j into the shelter location $N + \sqrt{N} + \text{count}$ for read and write respectively. Otherwise, the client simply writes the same block (re-encrypted) back to the same shelter location.

- In a nutshell, for each operation, from the perspective of the cloud, the client needs to scan all shelter locations, reads a seemingly random location from the first $N + \sqrt{N}$ data and dummy locations, and finally reads and writes every shelter location again. Clearly, the access to any block id_j is oblivious to the cloud. Finally, at the end of a round (after $\sqrt{N}$ operations), we relocate all $N + 2\sqrt{N}$ blocks back to their locations in the initialization phase, using an oblivious sort by their identifiers .

The role of dummy items and stash in Square Root ORAM

- The role of the stash is to keep track of the previously accessed blocks (which we call *used blocks* in one set of $\sqrt{N}$ accesses), so that depending on whether we have accessed it previously or not, we operate based on that as *we have to go to an unused item and also satisfy finding the Block$_j$*.

- The role of the dummy items is to provide memory blocks on which we can make a fake query and also maintain that the fake query on the *unused block* $\pi(N + j)$ on the jth operation, if the block id_j is found in the stash.

- The combination of these two used and unblock block storage help us to make the proptocol such that for a query of same logical index, each time we make a different query to the memory.

2. When stash size is $n^{1/3}$

The amortized cost to perform one reading operation is the ratio of the number of accesses done by the oblivious RAM over the number of original accesses.

- With a stash size of $n^{1/3}$, we will also consider keeping the size of the dummy block to be $n^{1/3}$, because there will be a maximum of $n^{1/3}$ dummy accesses in the algorithm since we have changed the stash size to $n^{1/3}$ in one set of epochs. Thus, there is no point in keeping more than $n^{1/3}$ dummy blocks in memory. Now the size of the permuted memory will be $n + n^{1/3}$.
- Since the new stash size is $n^{1/3}$, now we have to "re-shuffle" the memory blocks (blocks from 1 to $n + n^{1/3}$) in sets of $n^{1/3}$ accesses because the stash will be full after $n^{1/3}$ accesses.
- The following calculation will be based on considering one epoch of actual access because things will be uniform in terms of each epoch (i.e., $n^{1/3}$ virtual accesses).
  - The number of actual accesses in the step of sorting the memory of $n + n^{1/3}$ will be based on Batcher's Sorting Network, namely $O((n + n^{1/3}) \cdot \log^2(n + n^{1/3})) \Rightarrow O(n \cdot \log^2(n))$ for one epoch.
  - In the algorithm:
    - Scanning through stash for virtual address i takes $n^{1/3}$ actual accesses.
    - If found in the stash, it takes $1 + \log(n + n^{1/3})$ actual accesses to perform a binary search for the dummy element $\pi(n + count)$ over the permuted memory.
    - If not found in the stash, it will again take $1+\log(n+n^{1/3})$ to perform a binary search for block $\pi(i)$ over the permuted memory.
    - Now we again go over the stash and write the (possibly) updated value of the i-th virtual word to the stash. So, it takes $n^{1/3}$ actual accesses.

    Thus, in total, it takes $O(n \cdot \log^2(n))/n^{1/3} + n^{1/3} + n^{1/3} + 1 + \log(n + n^{1/3})$ actual accesses.

  Thus, the total number of actual accesses will be of order $O(n^{2/3} \cdot \log^2(n)) + 2n^{1/3} + \log(n + n^{1/3})$.

$$\Rightarrow O(n^{2/3} \cdot \log^2(n)) + 2n^{1/3} + \log(n)$$
$$\Rightarrow O(n^{2/3} \cdot \log^2(n)) + \log(n)$$
$$\Rightarrow O(n^{2/3} \cdot \log^2(n))$$

  Thus, the amortized cost to perform one read operation is $O(n^{2/3} \cdot \log^2(n))$.

NOTE :

- If there is Oblivious Sorting Algorithm with Time complexity $O(n \cdot \log(n))$ then the amortized expression can be calculated as follows
- $O(n \cdot \log(n))/n^{1/3} + n^{1/3} + n^{1/3} + 1 + \log(n + n^{1/3})$ as above with minor change in the order of Oblivious Sorting Algorithm

$$\Rightarrow (O(n \cdot \log(n))/n^{1/3} + n^{1/3} + n^{1/3} + 1 + \log(n + n^{1/3}))$$
$$\Rightarrow (O(n^{2/3} \cdot \log(n)) + 2n^{1/3} + \log(n + n^{1/3}))$$
$$\Rightarrow O(n^{2/3} \cdot \log(n)) + 2n^{1/3} + \log(n)$$
$$\Rightarrow O(n^{2/3} \cdot \log(n)) + \log(n)$$
$$\Rightarrow O(n^{2/3} \cdot \log(n))$$

Question 2 : Oblivious DataStructures

1(a)

- The Servers P0 and P1 increase the size of the array by 1.
- To insert M into the shared array A:
  - Server P0 puts M0 in A0[n + 2].
  - Server P1 puts M1 in A1[n + 2].

1(b)

The heap property may be violated as the newly added element can be less than its parent. This could happen if:

$$\Rightarrow A_0\left[\left\lfloor\frac{n+1}{2}\right\rfloor\right] + A_1\left[\left\lfloor\frac{n+1}{2}\right\rfloor\right] > M$$
$$\Rightarrow A_0\left[\left\lfloor\frac{n+1}{2}\right\rfloor\right] + A_1\left[\left\lfloor\frac{n+1}{2}\right\rfloor\right] > M_0 + M_1$$
$$\Rightarrow A_0\left[\left\lfloor\frac{n+1}{2}\right\rfloor\right] + A_1\left[\left\lfloor\frac{n+1}{2}\right\rfloor\right] > A_0[n+2] + A_1[n+2]$$

$$\Rightarrow A\left[\left\lfloor\frac{n+1}{2}\right\rfloor\right] > A[n+2]$$

This above equation violates the heap property, which states that children must be greater than or equal to their parents.

1(c)

Since, as per the question, we have an oblivious comparison protocol, in order to restore the heap property, we need an oblivious swap operation to swap the elements.

Let's define an oblivious conditional swap protocol for X and Y, where c_0 is held by server P_0 and c_1 is held by server P_1:

$$\begin{aligned}&\text{Assign } Y = Y + X\\&\text{Assign } X = X + (c_0 + c_1) \times (Y - X)\\&\text{Assign } Y = Y - X\end{aligned}$$

The above protocol ensures that X and Y will be swapped if $c_0 + c_1 = 1$, otherwise X and Y remain unchanged. The addition and subtraction of shares are trivial. To distribute the shares of $X + (c_0 + c_1) \times (Y - X)$ to server P_0 and P_1, we can use the Du Atallah protocol.

The following is the Du Atallah protocol for the computation of shares of $X + (c_0 + c_1) \times (Y - X)$:

- Initially, P_0 has $(x_0, y_0), c_0$ and P_1 has $(x_1, y_1), c_1$. The honest server P_2 sends $(f_0, g_0, f_0 \cdot g_0 + t)$ to P_0 and sends $(f_1, g_1, f_1 \cdot g_1 - t)$ to P_1.
- Server P_0 will send $(y_0 - x_0 + f_0, c_0 + g_0)$ to server P_1 and P_1 will send $(y_1 - x_1 + f_1, c_1 + g_1)$ to server P_0.

- Then, server P_0 will compute the share as

$$z_0 = x_0 + c_0(y_0 - x_0 + (y_1 - x_1 + f_1)) - f_0(c_1 + g_1) + f_0 \cdot g_1 + t$$

- Similarly, P_1 will compute the other share as

$$z_1 = x_1 + c_1(y_1 - x_1 + (y_0 - x_0 + f_0)) - f_1(c_0 + g_0) + f_1 \cdot g_0 + t$$

Thus, the shares of the new X value will be z_0 and z_1.

Restore Heap

$i \leftarrow n + 2$
while $i > 1$ **do**
  Run oblivious comparison protocol for $(A_0[\lfloor \frac{i-1}{2} \rfloor] + A_1[\lfloor \frac{i-1}{2} \rfloor])$ and $(A_0[i] + A_1[i])$
  Such that $c_0 + c_1 = 1$ if $A[\lfloor \frac{i-1}{2} \rfloor] > A[i]$ and $c_0 + c_1 = 0$ othewise.
  if $c_0 + c_1 = 1$ **then**
    Swap $A[\lfloor \frac{i-1}{2} \rfloor]$ and $A[i]$ using the above protocol
  end if
  $i \leftarrow \lfloor \frac{i-1}{2} \rfloor$
end while

2(a)

The last element of A_0, A_1 should be place in $A_0[1]$ and $A_1[1]$ so that the binary tree like structure of heap will be maintained.

2(b)

The heap property is not necessarily satisfied at the new root $A[1] = A_0[1] + A_1[1]$ as both of its children may not be greater than or equal to it.

2(c)

To restore the heap property, we have to start with the index i = 1, now we will compare the value of heap at index i i.e A[i] with its child which is smaller among the two, if the value of that child is less then the parent we will swap this child and the parent i.e the A[i]. Then we will change the index i, to the index of the child was smaller (index of it before it was swaped).Do this till we have reached at the leaf of the tree. In this one operation of swaping with the parent with the smaller child obliviously , we will require 3 conditional swaps with oblivious comparisions to make them possible. The algorithm looks like this

Restore Heap

1: $i \leftarrow 1$
2: **while** $i \neq$ idex of the leaf node **do**
3: Calculate the c_0,c_1 such that $c_0 + c_1 = 1$ if the left child is greater then the right child and $c_0 + c_1 = 0$ otherwise, using the oblivious comparision black box we have .
4: **if** $c_0 + c_1 = 1$ **then**
5: Swap left and right child using the protocol defined in at the end.
6: **end if**
7: Calculate new x_0,x_1 such that $x_0 + x_1 = 1$ if the parent is greater then the left child and $x_0 + x_1 = 0$ otherwise, using the oblivious comparision black box we have .
8: **if** $x_0 + x_1 = 1$ **then**
9: Swap parent and left child using the protocol defined in at the end.
10: **end if**
11: Using the c_0,c_1 that we computed earlier we will again do a conditional swap
12: **if** $c_0 + c_1 = 1$ **then**
13: Swap left and right child using the protocol defined in at the end.
14: **end if**
15: These above 3 conditional swaps have changed swaped the parent with the smaller child, if the smaller child was smaller then the parent.
16: Now we have to calculate the index of the smaller child.
17: Let i_0,i_1 be the shares of the index of the parent that server P_0,P_1 holds.
18: The shares of the index for the server P_0,P_1 that we have to change it is $2*i_0+c_0$ and $2*i_1+c_1$ respectivly.
19: $i \leftarrow (2 * i_0 + c_0) + (2 * i_1 + c_1)$
20: **end while**

Conditional Swap Protocol

Let's define an oblivious conditional swap protocol for X and Y, where c_0 is held by server P_0 and c_1 is held by server P_1:

$$\begin{aligned}&\text{Assign } Y = Y + X\\&\text{Assign } X = X + (c_0 + c_1) \times (Y - X)\\&\text{Assign } Y = Y - X\end{aligned}$$

The above protocol ensures that X and Y will be swapped if $c_0 + c_1 = 1$, otherwise X and Y remain unchanged. The addition and subtraction of shares are trivial. To distribute the shares of $X + (c_0 + c_1) \times (Y - X)$ to server P_0 and P_1, we can use the Du Atallah protocol.

The following is the Du Atallah protocol for the computation of shares of $X + (c_0 + c_1) \times (Y - X)$:

- Initially, P_0 has $(x_0, y_0), c_0$ and P_1 has $(x_1, y_1), c_1$. The honest server P_2 sends $(f_0, g_0, f_0 \cdot g_0 + t)$ to P_0 and sends $(f_1, g_1, f_1 \cdot g_1 - t)$ to P_1.
- Server P_0 will send $(y_0 - x_0 + f_0, c_0 + g_0)$ to server P_1 and P_1 will send $(y_1 - x_1 + f_1, c_1 + g_1)$ to server P_0.
- Then, server P_0 will compute the share as

$$z_0 = x_0 + c_0(y_0 - x_0 + (y_1 - x_1 + f_1)) - f_0(c_1 + g_1) + f_0 \cdot g_1 + t$$

- Similarly, P_1 will compute the other share as

$$z_1 = x_1 + c_1(y_1 - x_1 + (y_0 - x_0 + f_0)) - f_1(c_0 + g_0) + f_1 \cdot g_0 + t$$

Thus, the shares of the new X value will be z_0 and z_1.