CS670: A3 Solutions

Prashant Kumar, 231110036

April 21, 2024

Square Root ORAM

Basic Idea: Square Root ORAM (SR-ORAM) aims to hide access patterns to a remote database by creating a structured system of data blocks, dummy blocks, and a client-side stash.

1. Construction of SR-ORAM:

Database: Consists of N actual data blocks.
Dummy Blocks: $\sqrt{N}$ blocks containing random data, used to obfuscate access patterns.
Stash (Cache): $\sqrt{N}$ block storage used to temporarily hold the blocks.

Operations:

Initialization:

- Data blocks are encrypted and placed at random positions in the remote database.
- Dummy blocks are also placed randomly next to data blocks creating an block array of size $(N+\sqrt{N})$.
- Permute these $(N+\sqrt{N})$ values obliviously and the mapping of permutation function π is stored at the client side, π gives a random mapping for each index of the data block.
- Stash is empty initially.

Access (Read/Write):

- **Search Stash:** For each operation client starts looking for the desired block within the stash, and in both cases (found or not found) client scans whole stash.
- **Read:** if the block has been found in a stash location then it reads next dummy element (keeps track by a count variable) and if element is not found in the stash then it directly goes to the index mapped by permutation function π to read the required block and also write it into the stash.
- **Write:** If the block is found in a stash location and the operation is a write, the client directly writes the block into that stash location. If the block is not found in any stash location but is retrieved among $(N + \sqrt{N})$ data and dummy locations, the client writes the block (re-encrypted) into the stash location $(N + \sqrt{N} + \text{count})$. Otherwise, the client simply writes the same block (re-encrypted) back to the same shelter location.
- **Eviction:** When the stash is full, an oblivious permutation is performed again (with new mapping function/hash function) with the remote database to reposition blocks (data and dummy) and clear the stash, this increases the computational cost of SR-ORAM by limiting only $\sqrt{N}$ operations per permutation/sort.

Role of dummy element and cache: Dummy elements are primarily there to access unique items if an item is found on stash, For an alternate way to do the same we have to keep the track of accessed/non-accessed items that creates an overhead (that is why there are only $\sqrt{N}$ dummy items because of only $\sqrt{N}$ operations can be done in a round) and stash items are there so that we don't have the same access pattern while accessing the same item again (that's why we scan through all stash even if i found the item).

2. Amortized read Cost

Modified Stash Size: $n^{1/3}$
Assumptions: Database size is n, and oblivious sort has a typical cost of $O(n \log n)$ as basic implementation of SR-ORAM considers O(1) space complexity.

Operations per Eviction: With a stash size of $n^{1/3}$, we perform $n^{1/3}$ read/write operations before a costly oblivious sort.

Cost of Oblivious Sort: $O(n \log n)$
Amortized Cost:

$$\text{Divide the cost of oblivious sort across the operations performed: } \quad \frac{O(n \log n)}{n^{1/3}} = O(n^{2/3} \log n)$$
$$\text{and, cost of individual read operation } = O(n^{1/3})$$
$$\textbf{Overall Amortized Cost: } O(n^{2/3} \log n(\text{ dominating term})$$

Therefore, with a stash size of $n^{1/3}$, the amortized cost per read operation becomes $O(n^{2/3} \log n)$.
Note: there is one other implementation of oblivious sort of complexity $O(n \log^2 n)$ considering this implementation the amortized cost per read operation becomes $O(n^{2/3} \log^2 n)$.

Solution-2: Oblivious Data structures

References: The list of predefined functions that I will further use in the assignment.

Oblivious comparisons black box: A protocol having the following functionality: If P_0 gives (x_0, y_0) and P_1 gives (x_1, y_1). After the end of the protocol, P_0 gets c_0 and P_1 gets c_1. They have the property that, if $(x_0 + x_1 < y_0 + y_1)$, then $c_0 + c_1 = 1$. Otherwise, $c_0 + c_1 = 0$.

Du-Attalah MPC for $(c_0 + c_1).(A0 + A1) + (1 - c_0 + c_1)(B0 + B1)$: This MPC is basically Du-Attalah MPC multiple times, mentioned as one just for clarity in the answer and not repeating things multiple times. Here's how this will break into steps:

- First, 2 parties P_0 and P_1 do a standard Du-Attalah MPC for $(c_0 + c_1).(A0 + A1)$, At the end, parties receive shares α_0 and α_1 respectively such that $\alpha_0 + \alpha_1 = (c_0 + c_1).(A0 + A1)$. Note that this will be equal to $(A0 + A1)$ if $(c_0 + c_1) = 1$.
- Again, 2 parties P_0 and P_1 do a standard Du-Attalah MPC for $(1 - c_0 + c_1).(B0 + B1)$ (can take $1 - c_0$ as one value). At the end, parties receive shares β_0 and β_1 respectively such that $\beta_0 + \beta_1 = (1 - c_0 + c_1).(B0 + B1)$. Note that this will be equal to 0 if $(c_0 + c_1) = 1$.
- Similarly, if $c_0 + c_1 = 0$ then $\alpha_0 + \alpha_1 = 0$ and $\beta_0 + \beta_1 = (B0 + B1)$.
- Now, by observation, I can say that if I add shares $\alpha_0+\beta_0$ and shares $\alpha_1+\beta_1$, then the property that will hold will be $(\alpha_0+\beta_0)+(\alpha_1+\beta_1) = (A0+A1)$ if $(c_0+c_1) = 1$ and $(\alpha_0+\beta_0)+(\alpha_1+\beta_1) = (B0+B1)$ if $(c_0 + c_1) = 0$.
- At the end of this protocol (i.e., Du-Attalah MPC for $(c_0+c_1).(A0+A1)+(1-c_0+c_1)(B0+B1)$), return the share $(\alpha_0 + \beta_0)$ to P_0 and the share $(\alpha_1 + \beta_1)$ to P_1.

ORAM Read and Write: ORAM read using shares of index i_* $(i.e. i_0 + i_1)$, in arrays holding shares of a database, gives the shares m_0 and m_1 such that $m = m_0 + m_1$, and m is the value in the database at index i_*. Similarly, ORAM write writes the shares of m, m_0 and m_1 at index i_* in arrays holding shares of the database such that $m_0 + m_1 = m$. (We can construct DPFs using shares of the index for ORAM read, and shares of index, shares of values for ORAM write)

Part 1: Insert Operation

Considering the size of the array is increased (as mentioned in assignment) the last empty index is $n+2$:

(a) **Inserting a New Element:**

P0: Sets $A0[n + 2]$ to $M0$ (its share of the new element).
P1: Sets $A1[n + 2]$ to $M1$ (its share of the new element).

(b) **Potential Heap Violation**

The heap property, $(A_0[i] + A_1[i] \leq A_0[2i] + A_1[2i]$ and $A_0[i] + A_1[i] \leq A_0[2i + 1] + A_1[2i + 1])$ which essentially means parent is smaller than or equal to the child may be broken because the new element $(M0 + M1)$ could be smaller than its potential parent in the heap.

(c) **Restoring the Heap Property (Oblivious Heapify)**

Starting Point: New element is at index $n + 2$, i.e., $M0$ in $A_0[n + 2]$ and $M1$ in $A_1[n + 2]$

current_index: $n + 2$

Loop: While $current\_index > 1$

step-1: **Parent_Index:** $\text{parent\_index} = \left\lfloor \frac{current_index}{2} \right\rfloor$

step-2: **Oblivious Compare:** P0 and P1 use the Oblivious blackbox with inputs:

P0: ($A0$[current_index] as x_0, $A0$[parent_index] as y_0)

P1: ($A1$[current_index] as x_1, $A1$[parent_index] as y_1)

Outcome:

P_0 receives c_0 and $P1$ receives c_1 such that $c_0 + c_1 = 1$ iff ($A0$[current_index] + $A1$[current_index]) <

Note that $(c_0 + c_1) = 1$ if current_index's value is smaller than its parent in oblivious heap.

step-3: **Du-Attalah MPC:** Now parties P0 and P1 do Du-Attalah MPC for

$[(c_0 + c_1).(A0[\text{current\_index}] + A1[\text{current\_index}]) + (1 - c_0 + c_1).(A0[\text{parent\_index}] + A1[\text{parent\_ind}$

At the end of the MPC, parties P0 and P1 will get shares $z0$ and $z1$ such that

$$z0 + z1 = \begin{cases} A0[\text{current\_index}] + A1[\text{current\_index}] & \text{if } (c_0 + c_1) = 1 \\ A0[\text{parent\_index}] + A1[\text{parent\_index}] & \text{otherwise} \end{cases}$$

Store the shares $z0$, and $z1$ somewhere (can't update the array values now for obvious reasons).

step-4: **Du-Attalah MPC:** Again, parties P0 and P1 do Du-Attalah MPC for

$[(c_0 + c_1).(A0[\text{parent\_index}] + A1[\text{parent\_index}]) + (1 - c_0 + c_1).(A0[\text{current\_index}] + A1[\text{current\_ind}$

At the end of the MPC, parties P0 and P1 will get shares $z'0$ and $z'1$ such that

$$z'0 + z'1 = \begin{cases} A0[\text{parent\_index}] + A1[\text{parent\_index}] & \text{if } (c_0 + c_1) = 1 \\ A0[\text{current\_index}] + A1[\text{current\_index}] & \text{otherwise} \end{cases}$$

step-5: **Updating Array:** Overwrite the shares $z0$ and $z1$ in array at index=parent_index to both P0's and P

Then overwrite the shares $z'0$ and $z'1$ at index=current_index to both P0's and P1's side respective

(Alternatively, we can just add $A'i$[j] $-$ Ai[j] in array Ai for i'th party at index j as required).

step-6: **Updating Index:** Set $current\_index = parent\_index$

step-7: goto loop condition

Proof:

- The protocol mentioned above returns the shares of child value (as $z0$ and $z1$) and shares of parent value (as $z'0$ and $z'1$) if the child is smaller than the parent; otherwise, it returns the shares of parent value (as $z0$ and $z1$) and shares of child value (as $z'0$ and $z'1$).
- Now observe that after having all the above shares if we overwrite the shares $z0$, $z1$ at parent index and $z'0$, $z'1$ at child index, it basically swaps the value, and the interesting thing is that even if the swap is not happening then also it changes the values of shares for the same final value (not necessarily).

Part 2: Extract Min Operation

(a) **Removing the Minimum:**

P0: Removes $A0[1]$

P1: Removes $A1[1]$

Promoting last element to maintain heap:

last_index $= n$ (index of the last element in the heap)

P0 moves $A0$[last_index] to $A0[1]$

P1 moves $A1$[last_index] to $A1[1]$

P0 and P1 decrease the size of their arrays by 1 or the last index is empty

(b) **Heap Property Violation:**

The new root element i.e. sum of both shares ($A0[1] + A1[1]$) might be larger than one or both of its children, violating the heap property ($A_0[i] + A_1[i] \leq A_0[2i] + A_1[2i]$ and $A_0[i] + A_1[i] \leq A_0[2i+1] + A_1[2i+1]$)

(c) **Restoring the Heap Property:**

Starting point:
current_index = i (where i =1 at the start)
The shares of the current index value are r_0 and r_1, like $r_0 = A_0[i]$ and $r_1 = A_1[i]$.
Let shares of the index of the current index be I_0 and I_1 (shares of 1 initially).

Loop: until the left child of current_index is not out of bound to the array
(in each step, i denotes the current_index however value of current_index is as per the last assignment to the current_index)

- **Step 1**: P_0 and P_1 have the shares of the left child (of current_index) as $2 \times I_0[i]$ and $2 \times I_1[i]$. To get the shares of the left child, they do an ORAM read. Note that this protocol works because $(2 \times I_0 + 2 \times I_1) = 2 \times (I_0 + I_1) = 2 \times i$ (where i is the current index). Similarly, they also obtain the shares of the right child.

  Note: This thing is not actually needed in the first iteration when they know the shares of the left child and right child clearly, but this is useful from the next iterations when they don't exactly know the current_index, rather both parties have the shares of the index of the current node.

  Initializing x_0, x_1, y_0, and y_1 for black-box computation:
  x_0, $x_1 \longleftarrow P_0$ and P_1's shares of the left child of current_index i
  y_0, $y_1 \longleftarrow P_0$ and P_1's shares of the right child of current_index i

- **Step 2**: Both parties give their respective values (in x_0, x_1, y_0, and y_1) to Black-box. At the end, P_0 gets c_0 and P_1 gets c_1 such that $c_0+c_1 = 1$ if $(A_0[2i]+A_1[2i]) < (A_0[2i+1]+A_1[2i+1])$. Note that if $c_0 + c_1 = 1$, it means the left child is smaller than the right child.

- **Step 3**: **Du-attalah MPC for shares of smaller value:**
  Do an Du-attalah MPC for $(c_0 + c_1) \cdot (x_0 + x_1) + (1 - c_0 + c_1) \cdot (y_0 + y_1)$. This MPC will give shares of the smaller value. Let at the end of this MPC, P_0 get the share v_0 and P_1 get the share v_1.

- **Step 4: Du-attalah MPC for shares of smaller value's index:**
  Do an Du-attalah MPC for $(c_0 + c_1) \cdot (2I_0 + 2I_1) + (1 - c_0 + c_1) \cdot (2I_0 + 2I_1 + 1)$. This MPC will give shares of the index of the smaller value (between both the child's of current_index i), and let P_0 receives the share s_0 and P_1 receives the share s_1 at the end of protocol.

- **Step 5:** Reassigning x_0, x_1, y_0, and y_1 for black-box computation:

  $x_0 = P_0$'s share of current_index's value (r_0)
  $x_1 = P_1$'s share of current_index's value (r_1)
  $y_0 = P_0$'s share of the smaller value (v_0)
  $y_1 = P_1$'s share of the smaller value (v_1)
  Both parties give their respective values to Black-box. At the end, P_0 gets c'_0 and P_1 gets c'_1 such that $c'_0 + c'_1 = 1$ if current_index's value is smaller than the smaller child.

- **Step 6: Obtaining shares of the smallest value and the second smallest value:**
  Do a Du-attalah MPC for $(c'_0 + c'_1) \cdot (r_0 + r_1) + (1 - c'_0 + c'_1) \cdot (v_0 + v_1)$. This MPC will give shares of the smallest value (between the smaller value and value at current_index). Suppose P_0 gets the share z_0 and P_1 gets the share z_1.

z_0 , z_1 $\longleftarrow$(c'$_0 + c'_1) \cdot (r_0 + r_1) + (1 - c'_0 + c'_1) \cdot (v_0 + v_1)$ (such that $z = z_0 + z_1$ is the smallest value).
Similarly, z'_0 , z'_1 $\longleftarrow$(c'$_0 + c'_1) \cdot (v_0 + v_1) + (1 - c'_0 + c'_1) \cdot (r_0 + r_1)$ (such that $z' = z'_0 + z'_1$ is the other value).

- **Step 7: Using the same analogy to obtain shares of the smallest index and share of the second smallest index:**

  q_0 and q_1 $\longleftarrow$(c'$_0 + c'_1) \cdot (I_0 + I_1) + (1 - c'_0 + c'_1) \cdot (s_0 + s_1)$ (shares of the smallest value's index)
  Similarly,
  q'_0 and q'_1 $\longleftarrow$(c'$_0 + c'_1) \cdot (s_0 + s_1) + (1 - c'_0 + c'_1) \cdot (I_0 + I_1)$ (shares of the second smallest value's index)

- **Step 8:** Write shares of smallest value z at current_index using ORAM write scheme as both parties have shares of current_index and shares of smallest value z (index shares q_0, q_1 and message shares z_0, and z_1).

- **Step 9:** Write shares of other value z' at second smallest value's index using ORAM write scheme as both parties have shares of other value (the value smaller than current_index's value) and shares of the second smallest index (index shares q'_0, q'_1 and message shares z'_0, and z'_1).

  Now, note that the smallest value is now at current_index and the second smallest value is at either of the child and we have the shares of the index of the second smallest value.

- **Step 10:** Update shares as:
  $r_0 = z'_0$ (P_0's share of second smallest element, becomes new current index)
  $r_1 = z'_1$ (P_1's share of second smallest element, becomes new current index)
  $I_0[i] = q'_0$ (P_0's share of index of second smallest element, becomes new current index)
  $I_1[i] = q'_1$ (P_1's share of index of second smallest element, becomes new current index)
- **Step 11:** Goto loop condition