Assignment 3

Jiyanshu Dhaka

1.1 Square-Root ORAM Construction

Construction

Initialization

1. Begin with an array of blocks, including both real data blocks and dummy blocks.
2. Append $\sqrt{n}$ dummy blocks to the input array, where n is the size of the array.
3. Generate a pseudorandom permutation π of indices from 1 to $(n + \sqrt{n})$.
4. Sort the blocks according to the permutation π, effectively shuffling them.

Access

To access a block with logical index i:

1. Check if the block is in the Stash (a temporary cache).
2. If not found in the Stash, compute the physical index p based on $\pi(i)$.
3. Retrieve the block from the Shuffle (main memory structure) at the physical index p.
4. Move the retrieved block to the Stash.
5. If the accessed block is a dummy block, replace it with a randomly chosen unused block.

Periodic Reshuffling

1. After a certain number of accesses (T accesses), perform a reshuffling of the blocks.
2. During reshuffling, remove all dummy blocks from both the Shuffle and the Stash.
3. Reinitialize the ORAM structure using the remaining real blocks, repeating the initialization process.

Role of Dummy Items and Stash

1. **Dummy Items**: Dummy blocks are added to the array during initialization to obscure the true size of the array and prevent inference of the access pattern.Dummy items are like fake placeholders added to a list or array when it's first set up. They're used to hide how big the list really is and to make it harder for someone to figure out how the list is being used. They also help maintain a consistent access time by ensuring that each access requires the same amount of effort, whether it's a real data block or a dummy block.

2. **Stash (Cache)**: The Stash acts as a temporary cache for recently accessed blocks. It improves access efficiency by reducing the need to repeatedly access blocks from the main memory structure (Shuffle). Stash management ensures that only the most recently accessed blocks are retained, while older blocks are replaced using a least recently used (LRU) policy.

Example: Consider an ORAM with an initial array size of $n = 16$. After appending $\sqrt{16} = 4$ dummy blocks, the total array size becomes $n + \sqrt{n} = 20$. A pseudorandom permutation π is generated for indices from 1 to 20, and the blocks are sorted accordingly. Accesses to logical indices are then mapped to physical indices based on π, ensuring privacy and security.

1.2 Amortized Cost Calculation

To calculate the amortized cost of one read operation in the scenario where the stash size is changed to $n^{1/3}$, let's first break down the costs involved:

1. **Cost of accessing blocks in the stash**: Since the stash size is $n^{1/3}$, the cost of accessing a block in the stash is $O(n^{1/3})$.

2. **Cost of computing PRF $\pi(x)$ for each access**: In the initialization algorithm, $n + \sqrt{n}$ evaluations of $\pi(i)$ are needed. Since evaluating each PRF requires tens of thousands of gates, we'll approximate this cost as $O(n)$.

3. **Cost of shuffling**: The original scheme requires a $\Theta(n \log^2 n)$ oblivious sort on the data blocks every T accesses. Let's denote this cost as $O(S)$, where S represents the size of the stash.

Now, let's denote T as the number of accesses before shuffling, and A as the amortized cost per access.

We can express the amortized cost per access A as:

$$A = \frac{\text{Total cost}}{\text{Total number of accesses}}$$

The total cost consists of the cost of accessing blocks in the stash, computing PRF, and shuffling. So,

Total cost = $T\times$Cost of accessing a block in the stash+Cost of computing PRF+Shuffle cost

$$\text{Total cost} = T \times O(n^{1/3}) + O(n) + O(n \log^2 n)$$

Since each access triggers a shuffle after T accesses, the total number of accesses is T.

Thus,

$$A = \frac{T \times O(n^{1/3}) + O(n) + O(n \log^2 n)}{T}$$

Now, we need to find an expression for T in terms of n.Set $T = n^{1/3}$.

$$A = \frac{n^{1/3} \times O(n^{1/3}) + O(n) + O(n \log^2 n)}{n^{1/3}}$$

$$A = O(n^{1/3}) + O(n^{2/3}) + O(n^{2/3} \log^2 n)$$

$$A = O(n^{2/3} \log^2 n)$$

So, the amortized cost per read operation with a stash size of $n^{1/3}$ is $O(n^{2/3} \log^2 n)$.

Example: Let's consider a memory with size $n = 64$. If we change the stash size to $n^{1/3}$, it becomes $n^{1/3} = 4$. Therefore, the amortized cost per read operation would be $O(64^{2/3} \log^2 64) = O(16 \log^2 64) = O(16 \times 6^2) = O(16 \times 36) = O(576)$.

2.1.(a) Insert Operation in Oblivious Min Heap

To maintain the heap property while inserting a new element into the shared array A, parties $P0$ and $P1$ proceed as follows:

1. Both parties $P0$ and $P1$ increment the size of their respective arrays $A0$ and $A1$ by 1 to accommodate the new element.
2. Collaboratively, they generate a new element M, where $M = M0 + M1$, with $M0$ and $M1$ being the shares held by $P0$ and $P1$ respectively.
3. Each party inserts their corresponding share $M0$ and $M1$ into their arrays $A0$ and $A1$.
4. Ensuring the heap property, $P0$ inserts $M0$ into $A0[n+2]$, and $P1$ inserts $M1$ into $A1[n+2]$.

After the insertion, the modified shared array $A' = A0' + A1'$ must satisfy the heap property:

$$A0'[i] + A1'[i] \leq A0'[2i] + A1'[2i] \quad \text{and} \quad A0'[i] + A1'[i] \leq A0'[2i+1] + A1'[2i+1]$$

where $A0'$ and $A1'$ represent the updated arrays after insertion.

- For a general example, suppose the initial shared arrays for parties $P0$ and $P1$ are $A0 = [a_0, b_0, c_0, d_0]$ and $A1 = [a_1, b_1, c_1, d_1]$ respectively. If $P0$ inserts $M0 = e_0$ and $P1$ inserts $M1 = e_1$, then the modified arrays become $A0' = [a_0, b_0, c_0, d_0, e_0]$ and $A1' = [a_1, b_1, c_1, d_1, e_1]$. The updated array $A' = A0' + A1' = [a_0 + a_1, b_0 + b_1, c_0 + c_1, d_0 + d_1, e_0 + e_1]$ maintains the heap property, with the last element M appended with respect to the original array A.

2.1.(b) Violation of Heap Property after Insert Operation

1. **Observation**: After increasing the array size and placing the new element at $A[n+2]$, the heap property may not be satisfied.
2. **Heap Property**: For any node i, the value of i must be less than or equal to the values of its children $2i$ and $2i+1$ for even and odd cases, respectively.
3. **Explanation**:
   - When $n+2$ is even, M_0 and M_1 are not compared with the elements at $A_0[\frac{n+2}{2}]$ and $A_1[\frac{n+2}{2}]$ before insertion as they may not satisfy the heap property.
   - When $n+2$ is odd, M_0 and M_1 are not compared with the elements at $A_0[\frac{n+1}{2}]$ and $A_1[\frac{n+1}{2}]$ before insertion as they may not satisfy the heap property.

   Here's an example demonstrating the violation of the heap property after an insert operation:

   Suppose we have a min-heap represented as an array:

   $$A = [5, 7, 10, 15, 20]$$

   Now, we want to insert the element 3 into this heap. We increase the array size and place the new element at $A[6]$, making it:

   $$A = [5, 7, 10, 15, 20, 3]$$

At this point, the heap property is violated because 3 is less than its parent node (7). In the heap, the parent of a node at index i is $\lfloor \frac{i}{2} \rfloor$.

So, after the insert operation, the heap property is violated since $A[6] = 3$ is less than its parent $A[3] = 15$. This violates the condition where every node must be greater than or equal to its parent in a min-heap.

2.1.(c) Employing Discreet Comparisons to Rectify the Heap Characteristic

To fix the heap property in the shared array A after adding data using additive shares, let's presume, without losing generality, that $n+2$ is even. P_0 and P_1 can pursue these measures using the discreet comparisons secret box.

Discreet Box: This secret box compares pairs of elements in the array and decides whether a switch is necessary to sustain the heap property.

Presumption: I hypothesize $n + 2$ is even without compromising generality. If at any step of the recurring process, $\frac{n+2}{2}$ is not even, then we'll compare it with $\left[\frac{n+2}{2} + 1\right]$ for discreet comparison.

- P_0 transfers $(A_0[\frac{n+2}{2}], A_0[\frac{n}{2}])$ to the discreet comparisons secret box.
- P_1 transfers $(A_1[\frac{n+2}{2}], A_1[\frac{n}{2}])$ to the same discreet comparisons secret box.
- Comparisons occur without revealing the actual values. The secret box dispatches c_0 to P_0 and c_1 to P_1. We then confirm the heap property: if $c_0 + c_1 = 1$, then the heap property holds and no switch is needed. Otherwise, if $c_0 + c_1 = 0$, then P_0 plans to switch $A_0[\frac{n+2}{2}]$ and $A_0[\frac{n}{2}]$, and P_1 intends to switch $A_1[\frac{n+2}{2}]$ and $A_1[\frac{n}{2}]$. These switches need to be executed in secret to keep P_0 and P_1 unaware of whether the actual elements of the array are switched or not.

Plan:

In the plan, we remove everything that is useless and add:

- If $(A < B)$, $c = 0$:
- $A'_0 + A'_1 = A_0 + A_1$
- $B'_0 + B'_1 = B_0 + B_1$
- Else if $(A > B)$, $c = 1$:
- $A'_0 + A'_1 = B_0 + B_1$

- $B'_0 + B'_1 = A_0 + A_1$
- Create the first share of the pair as:
- $A'_0 + A'_1 = c(B_0 + B_1) + (1 - c)(A_0 + A_1) = (c_0 + c_1)(B_0 + B_1) + (A_0 + A_1) - (c_0 + c_1)(A_0 + A_1) = (c_0 + c_1)[(B_0 - A_0) + (B_1 - A_1)] + (A_0 + A_1)$

We will use Du Atallah protocol to compute these shares with the assistance of a trusted third party. Similarly, we can derive the other shares for the below equations from following the above procedure:

- $B'_0 + B'_1 = c(A_0 + A_1) + (1 - c)(B_0 + B_1) = (c_0 + c_1)(A_0 + A_1) + (1 - (c_0 + c_1))(B_0 + B_1)$

However, a potential issue arises after switching elements $A_0\left[\frac{n+2}{2}\right]$ and $A_1\left[\frac{n+2}{2}\right]$ to fix the heap property at level $\frac{n+2}{2}$.

There's now a possibility that the following inequality might not hold:

$A_0\left[\frac{n+2}{4}\right] + A_1\left[\frac{n+2}{4}\right] \leq A_0\left[\frac{n+2}{2}\right] + A_1\left[\frac{n+2}{2}\right]$

Therefore, to tackle the potential violation of the heap property at lower levels, we repeat the same procedure recursively until $1 \leq \frac{n+2}{2^n}$. However, this time, we use the following element pairs for discreet comparisons:

- P_0 transfers $(A_0\left[\frac{n+2}{4}\right], A_0\left[\frac{n+2}{2}\right])$ to the discreet comparisons secret box.
- P_1 transfers $(A_1\left[\frac{n+2}{4}\right], A_1\left[\frac{n+2}{2}\right])$ to the same discreet comparisons secret box. Hence, in this way, we can restore the heap property.

In summary, using the oblivious comparisons black box allowed us to compare the new element with its parent without revealing their actual values. This enabled us to maintain the confidentiality of the heap structure while ensuring its correctness.

By using the oblivious comparisons black box, we can perform the necessary comparisons without revealing the actual values of the elements being compared. This allows us to maintain the confidentiality of the heap structure while ensuring its correctness.

2.2. (a) Extract Min

The protocol begins with both P0 and P1 removing the element at index $A_0[1]$ and putting $A_0[n]$ and removing $A_1[1]$ and putting $A_1[n]$ respectively. To maintain the min-heap property, the replacements for these elements are chosen based on selecting the smallest elements among the children nodes.

Selecting Replacement for $A_0[1]$:

- After removing the element at index $A_0[1]$, P0 needs to select a replacement from its children nodes: $A_0[2]$ and $A_0[3]$.
- P0 compares the elements $A_0[2]$ and $A_1[2]$.
- If $A_0[2]$ is smaller than or equal to $A_1[2]$, then P0 selects $A_0[2]$ as the replacement for $A_0[1]$.
- If $A_1[2]$ is smaller than $A_0[2]$, then P0 selects $A_1[2]$ as the replacement for $A_0[1]$.
- This ensures that the heap property is maintained, as the parent ($A_0[1]$) will be less than or equal to its left child ($A_0[2]$) and right child ($A_0[3]$).

Selecting Replacement for $A_1[1]$:

- After removing the element at index $A_1[1]$, P1 needs to select a replacement from its children nodes: $A_1[2]$ and $A_1[3]$.
- P1 compares the elements $A_0[3]$ and $A_1[3]$.
- If $A_0[3]$ is smaller than or equal to $A_1[3]$, then P1 selects $A_0[3]$ as the replacement for $A_1[1]$.
- If $A_1[3]$ is smaller than $A_0[3]$, then P1 selects $A_1[3]$ as the replacement for $A_1[1]$.
- This ensures that the heap property is maintained, as the parent ($A_1[1]$) will be less than or equal to its left child ($A_1[2]$) and right child ($A_1[3]$).

Extract Min:

- The "Extract Min" operation involves removing the minimum element from the heap. In this case, it is $A_0[1]$ or $A_1[1]$, whichever is smaller.
- After the replacements are made for $A_0[1]$ and $A_1[1]$, the minimum element among them will be at $A_0[1]$ or $A_1[1]$.
- Therefore, after step (a), the minimum element can be extracted simply by removing $A_0[1]$ or $A_1[1]$, depending on which one is smaller.

2.2.(b) Heap property

After the first step, where P0 and P1 remove elements from $A_0[1]$ and $A_1[1]$ respectively, the replacement elements chosen for these positions might not necessarily satisfy the heap property.

The reason is that the chosen replacements are based only on the immediate children of the removed elements. While selecting the replacements, we ensure that each parent is smaller than or equal to its immediate children, which satisfies a local heap property. However, this does not guarantee that the entire heap satisfies the min-heap property, where every parent node is smaller than or equal to its children across the entire heap.

Therefore, additional steps might be needed to restore the heap property throughout the entire heap after the initial removal and replacement steps. These steps could involve recursively adjusting the elements to ensure that the min-heap property is maintained at every level of the heap.

In summary, the replacements chosen after the first step ensure a local heap property within the immediate subtree rooted at the removed elements, but further adjustments may be required to satisfy the min-heap property across the entire heap.

2.2.(c) Restoring Heap Property Using Oblivious Comparisons

Introduction

In the realm of secure computation, maintaining the heap property while ensuring data privacy poses a unique challenge. Heap operations often involve comparing elements and potentially swapping them to maintain the heap's integrity. However, in scenarios where data privacy is paramount, such as distributed environments or secure multi-party computation, traditional methods fall short. In this section, we explore a technique to restore the heap property while preserving privacy using an oblivious comparisons black box.

Heapify Algorithm

To update the heap, we need to run the heapify algorithm, starting from the root. For a normal Min Heap, two children of the parent are compared, and then the smaller child is compared to the parent. This procedure is then repeated for the new child as the next parent, continuing until a leaf is reached or at a point where no swapping has taken place.

Using Distributed ORAM to Update the Heap

(a) **Obtaining Shares of the Smaller Child Index:** Using the oblivious comparisons block (same as above), we can get shares of the index of the smaller child.

(b) **Comparing Children:** Suppose the children of the current parent have the indices $2i$ and $2i + 1$. They can use the oblivious comparisons block to compare $A[2i]$ and $A[2i + 1]$ and get shares of c such that $c = 1$ if $A[2i] < A[2i + 1]$ otherwise $c = 0$.

(c) **Multi-Party Computation:** Following this, they can use multi-party computation to evaluate the following function:

$$c \cdot (2i) + (1 - c) \cdot (2i + 1)$$

Clearly, this function records the index of the smaller element of heap A.

(d) **Updating Shares:** Therefore, parties involved have the shares of the index of the smaller child.

(e) **Reading Shares:** Now using the Distributed ORAM protocol, parties can proceed to read shares of the smaller child.

(f) **Conditional Swap:** Then they use these shares to compare it with the parent and conditionally swap it to obtain the shares of the new parent and the new child (which will be conditionally swapped, that is swapped if and only if the child is less than the parent).

(g) **Write Operation:** Now parties can perform a write operation of the Distributed ORAM to update the shares of the new child and the parent.

(h) **Recursive Procedure:** Following this, they already have the shares of the index of the new child, and now considering this as the next parent, this procedure can be repeated until the leaf of the heap is reached.

Note

Even though this procedure does not terminate when the parent is already lesser than or equal to both its children, it still works fine because the heap property is already satisfied down the heap. Therefore, effectively, conditional swapping won't make any difference.

Conclusion

By leveraging techniques like oblivious comparisons and Distributed ORAM, we can update the heap while preserving data privacy. This approach ensures that sensitive information remains secure even during fundamental operations like heap maintenance.