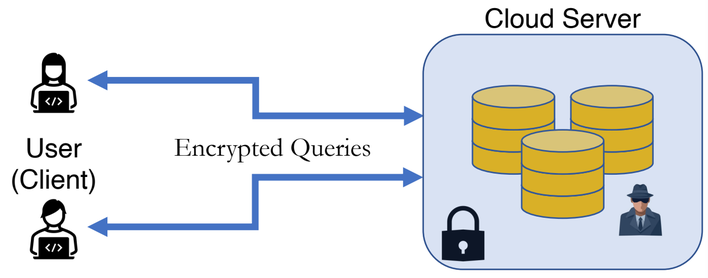
Implementing Oblivious Data Structures using Red-Black trees

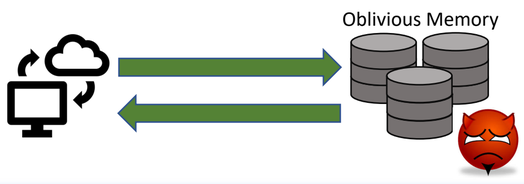
Prathamesh Pradeep Khole
University of California, Santa Cruz
pkhole@ucsc.edu
CSE-290X Course Project

Introduction

- Using cloud storage has become widespread nowadays.
- The server or the cloud stores and protects the client's data.
- The server is honest but also curious.



- Since the data is encrypted, the only thing the server can learn about the data is the search and access pattern of the user.
- Search and access patterns can reveal a lot about the data; as a result, data privacy may be compromised even in the presence of encryption.
- Oblivious memory can help mitigate this problem.
- Oblivious memory works by performing a certain number of dummy accesses with the actual ones to trick or confuse the observer.
- Extra operations act like padding so that all operations appear to have accessed the same number of elements.
- As a result, any two user operations are indistinguishable from one another for the observer.

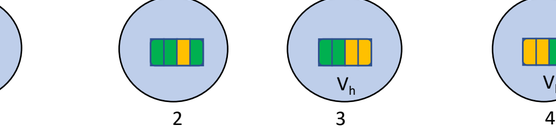


Problem Definition

- This Project aims to implement efficient oblivious data structures using Red-Black trees with existing searchable encryption schemes.
- To do this, we implement an Oblivious Random Access Machine (ORAM) using a Red-Black tree as the underlying data structure.
- Current ORAM implementations also use balanced binary search trees, the most popular being the AVL trees.
- AVL Trees are widely used in ORAMs since they are strictly balanced and thus provide fast lookups.
- The reason behind using Red-Black trees for oblivious RAM is that Red-Black trees as data structures are faster at insertions and deletions than AVL trees.
- Also, although Red-Black trees are less strictly balanced binary search trees, they have logarithmic search speeds.

Observations and results:

- Implemented ORAM with Red-Black tree as the underlying data structure which supports insertions, search and delete operations on encrypted data.
- Experimentally verified the number of nodes accessed by Red-Black trees during insertion and delete operations is less than AVL trees for large number of elements.



Tree Based Oblivious RAM

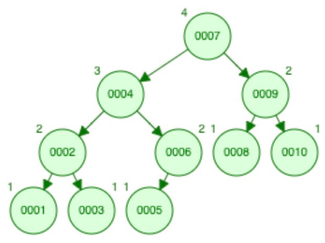
- In case of tree based ORAMs the server stores the ORAM as a complete binary tree.
- Complete binary tree is a type of binary tree where every level, excluding possibly last one is filled completely.
- In case of ORAM, this binary tree has N leaves in the last level and a height of at least $\log_2(N)$.

- After each ORAM operation (e.g., read), the system logically maps the data block to a new random root->leaf path.
- The eviction operation slowly pushes the re-mapped block from the root towards the selected leaf.

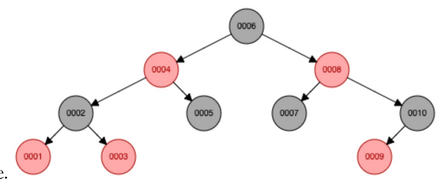
AVL Tree vs Red-Black Tree

- AVL and Red-Black trees are self-balancing binary search trees but differ in maintaining the height balance.
- Due to this they get appear different for same inputs.

AVL Tree and Red-Black tree for input: 10, 9, 8, 7, 6, 1, 2, 3, 4, 5

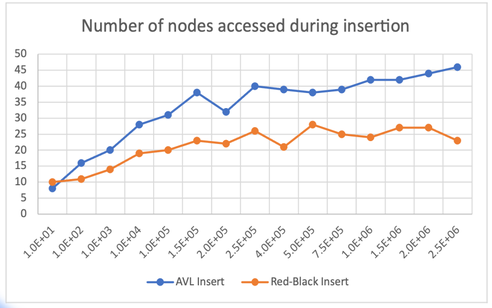


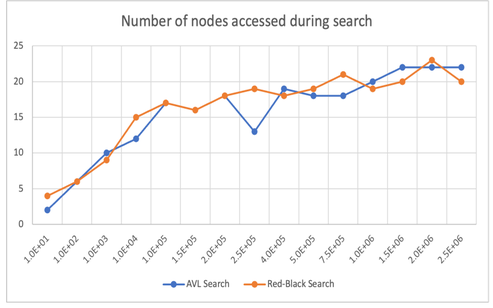
- AVL trees are strictly height balanced BSTs.
- They have minimal height for any BST of same size.
- They provide very fast lookups.
- Slow insertions and deletions due to more rotations needed to restore balance.
- In worst case $\log_2(n)$ rotations are needed to restore balance.

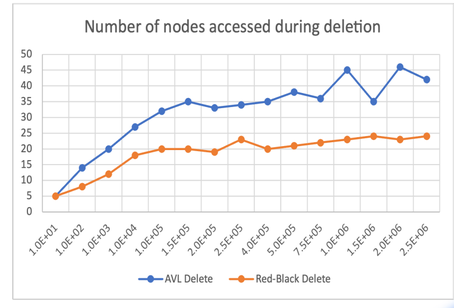


- Red-Black trees are less strictly balanced BSTs.
- Each node is colored either red or black based on certain predefined rules.
- They do not use height as a strict criteria for balancing.
- Insertions and deletions are fast as fewer operations (re-color or rotations) are needed to restore the tree properties.
- Constant number of operations are needed to restore tree properties.

Comparison between AVL tree and Red-Black tree for insert, search and delete operations based on number of nodes accessed (y-axis) for n-elements (x-axis).



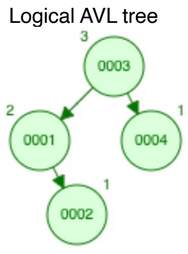




Related Work

- The paper, Path ORAM first introduced Path ORAM as a practical and simple Oblivious RAM with a small amount of client storage.
- Then Paper, *Oblivious Data Structures* discusses using logical Binary search trees on top of ORAM trees to speed up search process on the ORAM data.
- AVL Trees are the most prevalent underlying logical data structures with ORAMs because of their fast lookups.
- The paper, *Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions* uses a Path ORAM with AVL trees as the underlying data structure.
- This scheme makes use of a logical AVL tree built on top of an ORAM represented as a full binary tree having N leaf nodes.

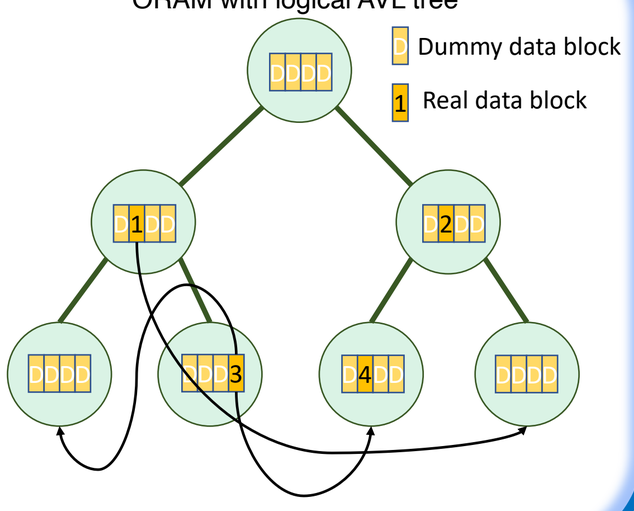
Logical AVL tree



Structure of data block

id_0	key_0	right_id	left_id
right_pos		height	
left_pos			

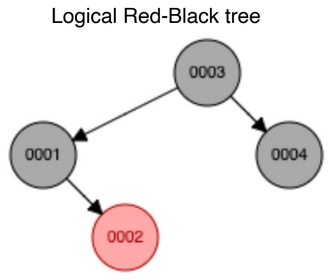
ORAM with logical AVL tree



Project's Approach

- Our project aimed to improve the performance of *Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions* by replacing the AVL tree data structure with a red-black tree, which is also a balanced binary search tree.
- We leveraged the ORAM structure and updated the node and data block structure to support Red-Black trees, modifying the rotation and balance functions to handle the necessary re-coloring.
- It is possible to implement these changes while still using the encryption schemes of the previous ORAM.
- These changes allowed us to compare the performance of Red-Black trees versus AVL trees and demonstrate their potential to enhance the efficiency of this encryption technique.

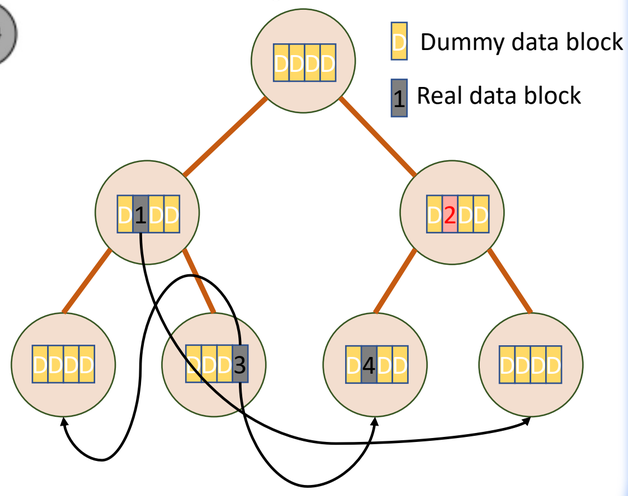
Logical Red-Black tree



Structure of data block

key_0	id_0	right_id	left_id
$color_0$			
right_pos		redHeight	
left_pos		blackHeight	

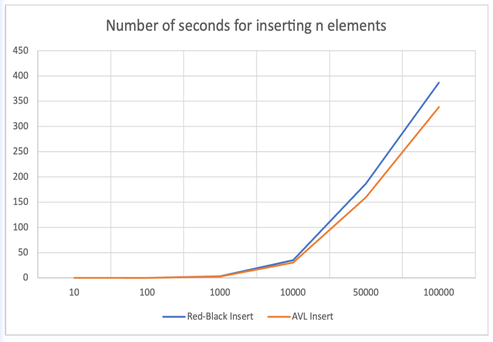
ORAM with logical Red-Black tree

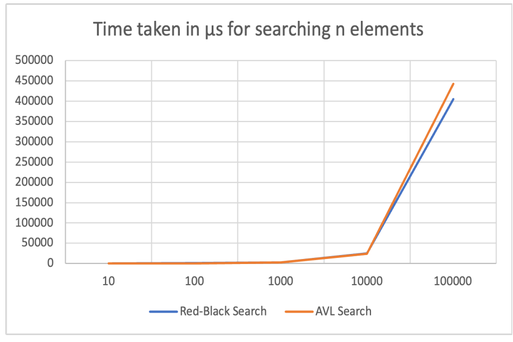


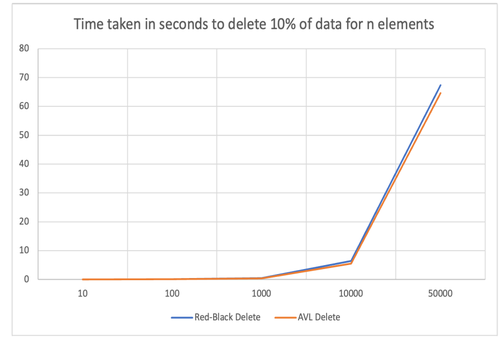
Although the ORAM tree is like the previous implementations, the logical tree is different and thus uses different rules to maintain its properties.

Experimental Results

- The AVL tree based ORAM performs slightly better than the new Red-Black tree-based approach.
- We defined alpha (α) as an empirical parameter to find the worst-case number of nodes accessed during any operation in Red-Black trees based on the equation: worst-case nodes accessed = $2*\alpha*\log_2(n)$, since worst case height of a Red-Black tree is $2*\log_2(n+1)$ where n is the number of elements.
- Experimentally alpha (α) lies in the range [0.455, 1.55].







Resources:
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html, https://www.cs.usfca.edu/~galles/visualization/AVLtree.html, https://eprint.iacr.org/2022/648.pdf, https://eprint.iacr.org/2013/280.pdf, https://eprint.iacr.org/2014/185.pdf, https://github.com/jgharehchamani/OS-SSE, https://www.programiz.com/dsa/red-black-tree, https://www.programiz.com/dsa/avl-tree, lecture slides of Professor Ioannis Demertzis, ODS slides by Amin Karbas.