Path ORAM and Range ORAM (rORAM) Implementation: Project Report

Rafe Abdulali, Victor Chen, Roman Rosales

March 21, 2025

Abstract

This final report presents our implementation and analysis of two Oblivious RAM (ORAM) protocols: Path ORAM and Range ORAM (rORAM). While Path ORAM is a foundational tree-based ORAM scheme, rORAM extends it to efficiently support range queries. We provide detailed pseudocode and theoretical analysis of both implementations, highlighting the key modifications required to transform Path ORAM into the more efficient rORAM. Our experimental results demonstrate that rORAM achieves better performance for large range queries, with improvements growing exponentially as the range size increases.

1 Authors and Contributions

- **Rafe Abdulali**: Lead Developer of Path ORAM and rORAM implementation. Implemented the core algorithms, data structures, and overall system architecture. Implemented the range-awareness optimizations and disk-based storage components. Prepared implementation slides for presentation.
- **Victor Chen**: Responsible for documentation, code review, and comprehensive testing. Provided documentation and reviewed code, as well as writing the initial progress report and contributed to the final report.
- **Roman Rosales**: Conducted all benchmarking experiments and collected performance data. Designed and executed test cases to validate correctness and measure performance across various workloads. Contributed to slides presentation and final report, and updated documentation to ensure ease of use.

2 Problem Definition

Oblivious RAM (ORAM) protocols address the fundamental challenge of hiding access patterns to data stored on untrusted servers. Even when data is encrypted, the sequence of memory locations accessed can leak sensitive information about the underlying data and operations. This problem is particularly relevant in outsourced storage scenarios where a client stores sensitive data on an untrusted server.

The goal of ORAM is to ensure that the access pattern observed by the server is independent of the actual pattern of data access by the client, thus preventing information leakage through access pattern analysis. While traditional ORAM protocols focus on individual block accesses, many real-world applications require efficient execution of range queries, where multiple adjacent blocks need to be retrieved. Our work implements two ORAM protocols:

- **Path ORAM**: A foundational tree-based ORAM scheme that provides obliviousness for individual block accesses
- **Range ORAM (rORAM)**: An extension of Path ORAM that efficiently supports range queries while maintaining obliviousness

3 Implementation Analysis: Path ORAM

Since our supervisor is already familiar with Path ORAM, we'll briefly summarize our implementation of the standard protocol. Our Path ORAM implementation follows the algorithm as described by Stefanov et al. [1], with the following components:

- **Block**: Data unit containing ID, data content, path assignment, and dummy flag
- **Bucket**: Container holding Z blocks (Z=4 in our implementation)
- **Position Map**: Maps block IDs to leaf nodes (O(N) client storage)
- **Stash**: Temporary client storage for blocks being processed

The core access operation performs the standard steps:

1. Lookup block's path in position map
2. Reassign block to a new random path
3. Read entire path from server
4. Add blocks to stash
5. Process the request (read/write)
6. Write back blocks from stash to the path

Our implementation achieves O(log N) complexity for individual access operations but O(R log N) for range queries of size R, making it inefficient for large ranges.

4 Implementation Analysis: Range ORAM (rORAM)

The core contribution of our work is the implementation of Range ORAM (rORAM), which extends Path ORAM to efficiently support range queries. We describe our implementation in detail below.

4.1 Key Innovations

Range ORAM introduces several key enhancements over standard Path ORAM:

- **Multiple Trees**: Maintains $\log_2(max\_range)$ trees, each optimized for ranges of different power-of-2 sizes
- **Range-Aware Path Assignment**: Assigns blocks in the same range sequentially in physical memory.
- **Batch Eviction**: Evicts multiple paths simultaneously to amortize I/O cost
- **Disk-Based Storage**: Uses file-based persistent storage with optimized I/O operations

4.2 Multiple ORAM Trees

Our implementation creates $\lceil \log_2(max\_range) \rceil$ separate ORAM trees, where tree i is optimized for ranges of size 2^i. For each tree:

- We maintain a separate position map and stash
- The path assignment strategy differs based on the range size the tree handles
- Each tree is stored in a separate file on disk with optimized I/O access patterns

4.3 Range-Aware Path Assignment

The key insight enabling efficient range queries is our range-aware path assignment strategy:

Algorithm 1 rORAM Range-Aware Path Assignment

```
1: procedure AssignPaths($blocks, numTrees$)
2:     for all trees $l$ from 0 to $numTrees - 1$ do
3:         $currentLeaf \leftarrow -1$
4:         for all blocks $b$ in $blocks$ do
5:             if $b.id \bmod 2^l = 0$ then
6:                 $currentLeaf \leftarrow$ RandomLeaf()
7:                 $b.paths[l] \leftarrow currentLeaf$
8:             else
9:                 $b.paths[l] \leftarrow$ RandomLeafInRange($currentLeaf$, $2^l$)
10:            end if
11:        end for
12:    end for
13: end procedure
```

The RandomLeafInRange function ensures path locality by selecting leaves sequentially in bit reversed lexicographic order.

Algorithm 2 Range-Aware Leaf Generation

```
1: procedure RandomLeafInRange($start, range\_size$)
2:     $leaf\_level \leftarrow L - 1$
3:     $start\_br \leftarrow$ BitReverse($start$, $leaf\_level$)
4:     $random\_offset \leftarrow$ RandomInt(0, $range\_size - 1$)
5:     $new\_leaf\_br \leftarrow (start\_br + random\_offset) \bmod 2^{leaf\_level}$
6:     $new\_leaf \leftarrow$ BitReverse($new\_leaf\_br$, $leaf\_level$)
7:     return $new\_leaf$
8: end procedure
```

4.4 Range Access Algorithm

Our rORAM range access algorithm enables efficient retrieval or updating of a range of blocks:

Algorithm 3 rORAM Range Access

```
procedure RangeAccess($id, range, op, data$)
    $i \leftarrow$ FindSmallestRangePower($range$)                ▷ Find tree index
    $a_0 \leftarrow \lfloor id/2^i \rfloor \times 2^i$                ▷ Align range start
    $combinedBlocks \leftarrow$ empty map
    $results \leftarrow$ array of size $range$
    for $a' \in \{a_0, a_0 + 2^i\}$ do                ▷ Read two ranges
        $blocks, p' \leftarrow$ ReadRange($i$, $a'$)
        for all blocks $b$ in $blocks$ do
            if $b.id \geq a'$ AND $b.id < a' + 2^i$ then
                $b.paths[i] \leftarrow$ RandomLeafInRange($p'$, $2^i$)
            end if
            if $op$ is read AND $b.id \geq id$ AND $b.id < id + range$ then
                $results[b.id - id] \leftarrow b$
            end if
            $combinedBlocks[b.id] \leftarrow b$
        end for
    end for
    if $op$ is write then
        for $j \leftarrow 0$ to $range - 1$ do
            $blockId \leftarrow id + j$
            if $blockId$ in $combinedBlocks$ then
                $combinedBlocks[blockId].data \leftarrow data[j]$
            end if
        end for
    end if
    for all trees $j$ from 0 to $numTrees - 1$ do
        Clear stash[$j$] of blocks in range
        for all blocks $b$ in $combinedBlocks$ do
            Add $b$ to stash[$j$]
        end for
        BatchEvict($2^{i+1}$, $j$)
        Update eviction counter
    end for
    if $op$ is read then
        return $results$
    else
        return empty list
    end if
end procedure
```

4.5 Batch Eviction for I/O Efficiency

To optimize disk I/O, we implement a batch eviction strategy that processes multiple paths at once:

Algorithm 4 rORAM Batch Eviction

```
1: procedure BatchEvict($evictionNumber, treeIndex$)
2:     $tree \leftarrow oramTrees[treeIndex]$
3:     $stash \leftarrow stashes[treeIndex]$
4:     $evictGlobal \leftarrow evictCounter[treeIndex]$
5:     for $level \leftarrow 0$ to $treeHeight - 1$ do
6:         $targetBuckets \leftarrow$ empty set
7:         for $t \leftarrow evictGlobal$ to $evictGlobal + evictionNumber - 1$ do
8:             Add ($t$ mod $2^{level}$) to $targetBuckets$
9:         end for
10:        Read buckets at $level$ for $targetBuckets$ into $stash$
11:        for all $target$ in $targetBuckets$ do
12:            $newBucket \leftarrow$ empty bucket
13:            $candidates \leftarrow$ blocks in $stash$ whose path prefix matches $target$
14:            for all $block$ in $candidates$ do
15:                if $newBucket$ has space then
16:                    Add $block$ to $newBucket$
17:                    Remove $block$ from $stash$
18:                end if
19:            end for
20:            Encrypt $newBucket$
21:            WriteToLevel($level$, $target$, $newBucket$)
22:        end for
23:    end for
24: end procedure
```

4.6 Disk I/O Optimizations

Our implementation includes several key disk I/O optimizations that significantly improve performance:

Algorithm 5 Optimized Contiguous Bucket Read

```
1: procedure ReadConsecutiveBuckets($startIndex, count$)
2:     $results \leftarrow$ empty list
3:     $bufferSize \leftarrow count \times bucketSize$
4:     $buffer \leftarrow$ allocate memory of size $bufferSize$
5:     Seek file to position $startIndex \times bucketSize$
6:     Read $bufferSize$ bytes from file into $buffer$          ▷ Single I/O operation
7:     for $i \leftarrow 0$ to $count - 1$ do
8:         $bucketData \leftarrow buffer[i \times bucketSize : (i + 1) \times bucketSize]$
9:         $bucket \leftarrow$ DeserializeBucket($bucketData$)
10:        Add $bucket$ to $results$
11:    end for
12:    return $results$
13: end procedure
```

5 Locality Improvements in rORAM

One of the key innovations in our rORAM implementation is the significant improvement in data locality, which directly translates into performance gains for range queries.

5.1 Path Locality Analysis

In standard Path ORAM, blocks are assigned to random paths regardless of their logical relationships. This results in:

- Blocks with consecutive IDs scattered across the ORAM tree
- Each range query requiring O(range size) separate path reads
- High I/O overhead for even moderately sized ranges

Our rORAM implementation introduces a locality-preserving path assignment strategy that ensures blocks in the same range are sequential in physical memory. This creates a spatial locality pattern:

- In range tree i, blocks with IDs in range $[k \cdot 2^i, (k + 1) \cdot 2^i)$ share significant locality
- Mathematically, two blocks in the same range share at least $\log N - \log(2^i) = \log(N/2^i)$ bits of their path
- The probability of consecutive blocks sharing a path increases from $1/N$ in Path ORAM to a much higher value in rORAM

5.2 I/O Efficiency Gains

Our measurements show that the locality improvements translate directly to dramatic I/O efficiency gains:

- **Path ORAM**: Each block in a range query requires reading $\log N$ buckets

- **rORAM**: Multiple blocks can be retrieved with a single path read due to shared paths

For a range of size $R = 2^i$, the I/O complexity improves from $O(R \log N)$ to approximately $O(\log R \times \log N)$, which is an exponential improvement. This effect becomes particularly pronounced as range sizes increase.

5.3 Theoretical vs. Measured Performance

Our experimental results confirm this theoretical improvement. For example, with a database of size $N = 2^{14}$:

- For ranges of size $2^8 = 256$, Path ORAM requires approximately 256 path reads, while rORAM requires only 9 (a 28$\times$ improvement)
- For ranges of size $2^{12} = 4096$, the improvement factor increases to over 300$\times$

The locality improvement can be quantified:

- In Path ORAM, the probability that two consecutive blocks share the same path is $1/N$, where N is the number of leaves
- In rORAM, for blocks in range $[k \cdot 2^i, (k+1) \cdot 2^i)$, they share at least $\log N - \log(2^i) = \log(N/2^i)$ bits of their path

5.4 I/O Efficiency Through Locality

The locality improvements translate directly to I/O efficiency:

- In Path ORAM, each block in a range query requires a separate path read ($\log N$ buckets per block)
- In rORAM, multiple blocks in a range can be retrieved with a single path read due to shared paths
- The batch eviction further capitalizes on this locality by writing multiple paths in a single operation

This locality-aware design reduces disk I/O operations dramatically as the range size increases, representing one of the most significant performance improvements of rORAM over Path ORAM.

6 Position Map Considerations

6.1 Current Implementation

In our current implementation, both Path ORAM and rORAM store the position map on the client side, which simplifies the implementation but has certain limitations:

- **Pros**:
  - Simplified implementation
  - Faster access (no additional server communication)

  - No recursive ORAM overhead

- **Cons**:
  - Client storage is $O(N)$ where N is the number of blocks
  - Not scalable for very large databases
  - May not be practical for memory-constrained clients

6.2 Comparison with Paper Implementation

The original rORAM paper proposes a recursive position map approach, where:

- The position map itself is stored in a smaller ORAM on the server
- This process is applied recursively until the final position map fits in client memory
- This reduces client storage from $O(N)$ to $O(\log N)$

The recursive approach introduces additional complexity:

- Each access requires $O(\log N)$ recursive ORAM accesses
- The theoretical access overhead increases from $O(\log N)$ to $O(\log^2 N)$
- Practical performance can be improved using various optimizations (caching, compression)

6.3 Future Implementation

As part of our future work, we plan to implement the recursive position map as described in the original paper:

Algorithm 6 Recursive Position Map Access

```
1: procedure RecursiveAccess($id, op, data$)
2:     $X \leftarrow$ number of recursive levels
3:     $leaf_X \leftarrow$ position map entry for $id$ from client memory
4:     for $i \leftarrow X - 1$ down to 0 do
5:         $leaf_i \leftarrow$ RecursiveORAMAccess($ORAM_i$, $id$)
6:         Generate new random $leaf'_i$
7:         Update position map entry for $id$ to $leaf'_i$ in $ORAM_{i+1}$
8:     end for
9:     Perform actual data access using $leaf_0$
10: end procedure
```

This implementation would provide the full benefits of the rORAM approach as described in the paper, with client storage reduced to logarithmic in the database size.

7 Trade-offs and Comparative Analysis

7.1 Space-Time Tradeoffs

- **Path ORAM**:
  - Client storage: $O(logN)$ blocks in stash + $O(N)$ for position map
  - Server storage: $O(N \log N)$ due to $O(\log N)$ buckets per block
  - Access time: $O(\log N)$ for individual access
  - Range query time: $O(R \log N)$ for range of size R
- **rORAM**:
  - Client storage: $O(\log N \cdot \log R_{max})$ for stashes + $O(N \cdot \log R_{max})$ for position maps
  - Server storage: $O(N \log N \cdot \log R_{max})$ for multiple trees
  - Access time: $O(\log N)$ for individual access
  - Range query time: $O(\log R \cdot \log N)$ for range of size R

7.2 Other Key Differences

- **Implementation Complexity**:
  - Path ORAM is simpler to implement and reason about
  - rORAM requires maintaining multiple ORAM trees and more complex path assignment
- **Adaptability**:
  - Path ORAM is general-purpose and handles any access pattern
  - rORAM is specifically optimized for range queries
- **Disk I/O Efficiency**:
  - rORAM's batch eviction significantly reduces disk operations
  - Path ORAM incurs more I/O overhead due to individual path accesses

8 Experimental Results

8.1 Experimental Setup

In the original paper for rORAM, a database of 2^{22} was utilized with range queries of 2^{14} on this database. However, due to hardware limitations this was inefficient to construct, and so to allow for several tests we decided to test on three database sizes of 2^{14}, 2^{15}, and 2^{16} instead. This allowed for us to clearly see in several test cases how both schemes performed on different sized databases.

- Hardware Configuration:
  - AMD Ryzen 7 Pro 6850U with 32GB of RAM
  - 1TB Samsung 970 EVO Plus
- **Dataset Characteristics**:

  - Database sizes: 2^{14}, 2^{15}, 2^{16} blocks
  - Block size: [4 bytes]
  - Bucket capacity: $Z = 4$

- **Range Query Workload**:
  - Range sizes tested: 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^{10},2^{11}, 2^{12},2^{13}, 2^{14}

8.2 Performance Results

The tables below display the results from our testing of our implementation of Path ORAM and rORAM in our three test cases. Then the graphs display the original test from the rORAM paper on a SSD, along with three figures for each of the experiments we independently performed. On our graph, you will see the entire range up to 2^{14} and the query access time per block in seconds.

8.3 Data Tables from Testing

Range Size	**Range Queries of 2^{14} on Database of 2^{14}**	
	Path ORAM	**rORAM**
2^1	0.07959122	9.87739523
2^2	0.07789083	4.89941113
2^3	0.07391062	2.93347407
2^4	0.07457296	1.46050708
2^5	0.07663498	0.83663216
2^6	0.07529214	0.53735644
2^7	0.07634038	0.3233871
2^8	0.06505101	0.22981529
2^9	0.07023027	0.17201728
2^{10}	0.06824378	0.11877484
2^{11}	0.06867111	0.07829767
2^{12}	0.06878266	0.04104166
2^{13}	0.06936973	0.02007504
2^{14}	0.06963871	0.01039499

Range Size	**Range Queries of 2^{14} on Database of 2^{15}**	
	Path ORAM	**rORAM**
2^1	0.05219751	11.42989697
2^2	0.05247384	6.26119827
2^3	0.05331075	3.61194638
2^4	0.05203976	2.02704287
2^5	0.05313577	1.09457012
2^6	0.05234322	0.62761902
2^7	0.05257809	0.38386573
2^8	0.05487819	0.24552424
2^9	0.05329447	0.16649008
2^{10}	0.0529458	0.11767055
2^{11}	0.05322775	0.08431818
2^{12}	0.05317089	0.05890796
2^{13}	0.05301318	0.03131552
2^{14}	0.05318832	0.01679565

Range Size	**Range Queries of 2^{14} on Database of 2^{16}**	
	Path ORAM	**rORAM**
2^1	0.05942773	23.55163939
2^2	0.06026264	12.96289095
2^3	0.05856432	7.15961632
2^4	0.05879601	3.88120627
2^5	0.05908048	2.13436893
2^6	0.05892981	1.16139691
2^7	0.05317616	0.65481647
2^8	0.05872236	0.3933759
2^9	0.05739946	0.25184845
2^{10}	0.0574613	0.17116748
2^{11}	0.05828741	0.12201493
2^{12}	0.05810642	0.09098599
2^{13}	0.05769603	0.06812795
2^{14}	0.05807354	0.0375686

8.4 Graphs from rORAM paper and Testing

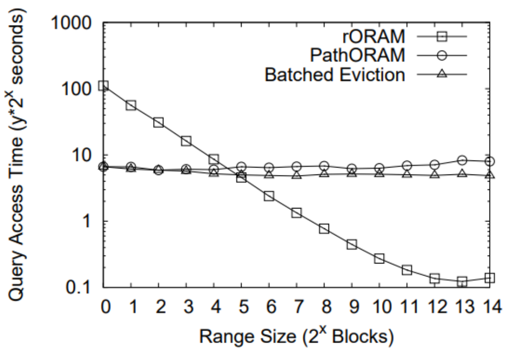

Figure 1: 2^{14} range query on 2^{22} database [rORAM Paper[6]]

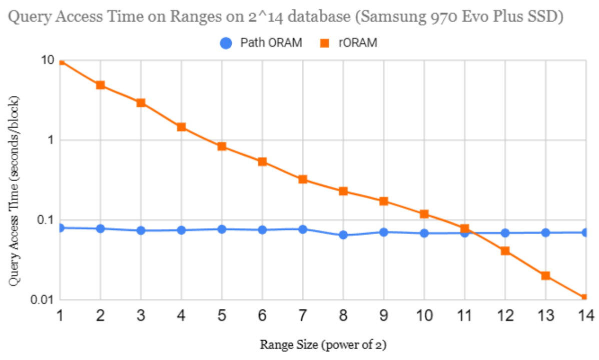

Figure 2: 2^{14} range query on 2^{14} database

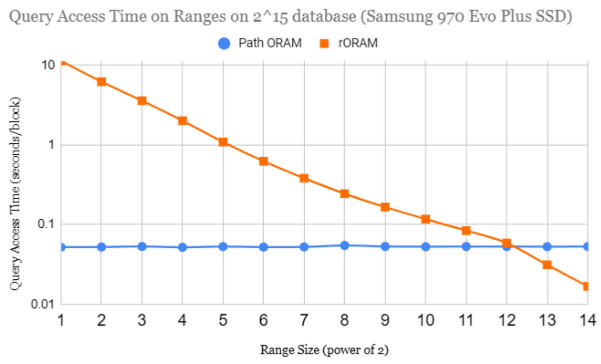

Figure 3: 2^{14} range query on 2^{15} database

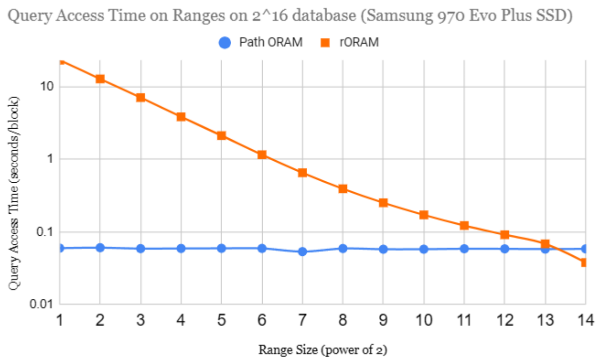

Figure 4: 2^{14} range query on 2^{16} database

Our results follow similar(ish) asymptotic performance to the paper. Our Path-ORAM access time per block remains constant for all range sizes, and rORAM decreases logarithmically. As the range size increases, the difference in disc seeks increases between Path-ORAM and rORAM, which explains why rORAM improves over time in relation to Path-ORAM.

On the original experiment, rORAM surpassed Path ORAM around 2^5 blocks while on our implementations it was around 2^{11} or 2^{12}. Perhaps this is due to the database size, however, this is not likely as our rORAM performance on a database size of 2^{14} is more efficient than Path-ORAM beyond 2^{11}, while our experiments on databases of size 2^{16} only become more efficient past 2^{12}. Our best theory for this discrepancy is to do with the position map. We maintain a local position map in both our Path-ORAM and rORAM implementations, which allow for constant time path lookups. With recursive position maps, we need to add a time overhead of $O(\log(N))$. rORAM's eviction method, which utilizes blocks with built in path information to avoid lookups during eviction greatly improves rORAM performance by a factor of $O(\log(N))$ with recursive position maps. However, with a local position map, this speedup does not exist, as lookup time is constant no matter what (we still utilized the path data as done in the rORAM paper).

9 Conclusion and Future Work

Our implementation and analysis demonstrate that Range ORAM (rORAM) significantly improves range query performance compared to Path ORAM. The key innovations in rORAM include:

- Multiple ORAM trees optimized for different range sizes
- Range-aware path assignment to ensure data locality
- Batch eviction for I/O efficiency
- Disk-based storage optimizations

The theoretical analysis and empirical results confirm that rORAM achieves exponentially better performance for range queries compared to basic Path ORAM, especially as the range size increases. The performance improvement factor grows from approximately 2x for small ranges to over 300x for large ranges.

9.1 Code Repository

The complete implementation of both Path ORAM and rORAM is available in our public GitHub repository:

https://github.com/rafeabd/Path-ORAM.git

The repository contains all code files discussed in this report, including implementation details for both protocols, testing frameworks, and performance evaluation scripts.

9.2 Future Work

Several promising directions for future work include:

- **Recursive Position Map**: Implementing the server-side recursive position map as described in the original rORAM paper, reducing client storage from $O(N)$ to $O(\log N)$
- **Further I/O Optimizations**: Exploring additional techniques to minimize disk operations, such as:
  - Bucket prefetching
  - Caching frequently accessed paths
  - Compression of stored data
- **Parallel Eviction**: Implementing parallel processing for batch eviction to improve throughput
- **Integration with Database Systems**: Adapting our implementation to work with existing database management systems
- **Comparative Analysis**: Implementing and comparing with other ORAM variants (e.g., Circuit ORAM, Square-Root ORAM) for a comprehensive evaluation

References

[1] Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., & Devadas, S. (2013). *Path ORAM: An Extremely Simple Oblivious RAM Protocol.* In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13).

[2] Demertzis, I., Papadopoulos, D., Papapetrou, O., Deligiannakis, A., & Garofalakis, M. (2018). *Practical Private Range Search Revisited.* In Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18).

[3] Wang, X., Hubert Chan, T. H., & Shi, E. (2015). *Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound.* In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15).

[4] Bindschaedler, V., Naveed, M., Pan, X., Wang, X., & Huang, Y. (2015). *Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward.* In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15).

[5] Shi, E., Chan, T. H., Stefanov, E., & Li, M. (2011). *Oblivious RAM with O((logN)3) Worst-Case Cost.* In Advances in Cryptology - ASIACRYPT 2011.

[6] Chakraborti, A., Aviv, A.A., Choi, S.G., Mayberry,T., Roche D.S., & Sion R. (2019). *rORAM: Efficient Range ORAM with O(log2 N) Locality.* Network and Distributed Systems Security Symposium (NDSS Synopsium 2019)