Ex1- OBLIVIOUS RAM

(path ORAM)

Alex Kovtun
316229798

Architecture

Block Class
Represents a data unit stored in the ORAM tree. Each block can either be real or a dummy (used to ensure indistinguishability). Real blocks hold encrypted content (nonce, ciphertext, tag), ensuring confidentiality using AES-GCM.

Note that the AES-GCM, includes both symmetric encryption and both the Authentication mechanism(using the tag) that verifies that the message wasn't corrupted.

Attributes:

- _id: The unique identifier for the block (int or None for dummy).
- _nonce, _ciphertext, _tag: Encrypted data components (AES-GCM).
- _is_dummy: Boolean indicating whether this is a dummy block.

Key Methods:

- data: Getter and setter for encrypted payload tuple.
- id: Property getter/setter to manage block IDs.
- is_dummy: Returns whether the block is a dummy.

Bucket Class
Encapsulates a fixed-size container of Block objects. Buckets are the nodes in the ORAM tree. Each bucket can store up to a predefined number of blocks, preserving structural constraints of ORAM.

Attributes:

- _capacity: Maximum number of blocks per bucket.
- _blocks: List of current Block instances stored in the bucket.

Key Methods:

- add_block(block): Adds a block if capacity allows.
- get_blocks(): Returns a copy of the block list.

- reset_content(): Clears the bucket's contents.
- capacity and blocks properties for safe access/modification

Utilities (TreeUtils)
Includes helper functions like create_perfect_tree and find_path_indices to build the tree and locate all nodes along a path, respectively. These functions abstract the tree's internal index representation, allowing efficient operations without revealing structure to the server.

- **create_perfect_tree(depth, bucket_capacity)**:
  Returns a flat list of Bucket objects forming a complete binary tree of specified depth and capacity.
- **find_path_indices(leaf_pos, depth)**:
  Calculates the sequence of node indices from the root to a given leaf in the flattened tree structure.

Server Class
Models the **untrusted storage server** that holds the ORAM tree. Internally, the tree is represented as a flat list of Bucket objects built using create_perfect_tree. The server provides methods to read and write entire root-to-leaf paths—consistent with Path ORAM's principle of full-path access.

Attributes:

- _depth: Depth of the binary tree.
- _bucket_capacity: Number of blocks each bucket can store.
- _tree: Flat list of Bucket objects representing the full tree.

Key Methods:

- read_path(leaf_index): Reads and returns all blocks from the root to a specified leaf.
- write_path(leaf_index, blocks): Writes a list of blocks top-down along a path.

Client Class
Acts as the **trusted party** responsible for managing encryption, block placement (position map), and temporary storage (stash). It encrypts all data before sending it to the server, maintains obliviousness by:

- Randomly remapping each block to a new leaf after access
- Fetching and rewriting entire paths
- Hiding access patterns via dummy blocks and constant-time path operations
- Using a stash to hold blocks temporarily for reshuffling without leaking patterns

How this is satisfies the requirements of ORAM:

Obliviousness (Access Pattern Hiding)
The architecture achieves obliviousness by ensuring that every read, write, or delete operation follows a uniform access pattern. Regardless of the specific operation or data being accessed, the client always reads and writes an entire path from the root of the ORAM tree to a randomly assigned leaf. This makes it impossible for the server to distinguish between different types of operations or infer which specific data is being accessed.

All accessed paths are of equal length due to the fixed tree depth, and data within each block is encrypted using AES-GCM, making the actual content indistinguishable. Furthermore, dummy blocks are used to fill unused slots in buckets, maintaining consistent bucket sizes.

After each access, the block's position is re-randomized using the _assign_new_leaf() method, ensuring that repeated accesses to the same block appear unrelated to an observer.

Data Confidentiality and Integrity
To guarantee data confidentiality and integrity, all content stored in the ORAM is encrypted on the client side using AES in GCM mode. This ensures not only confidentiality of the block data but also its authenticity, as GCM provides integrity verification through cryptographic tags.

The untrusted server never sees plaintext data at any point. If any tampering occurs during storage or retrieval, the AES-GCM decryption process will fail due to tag mismatch, allowing the client to detect and reject corrupted data.

Benchmark discussion:

Setup:

To evaluate the performance of our ORAM implementation, we conducted controlled experiments measuring throughput (requests per second) and latency (time per request) across a range of database sizes.

The experiments were performed using the following setup:

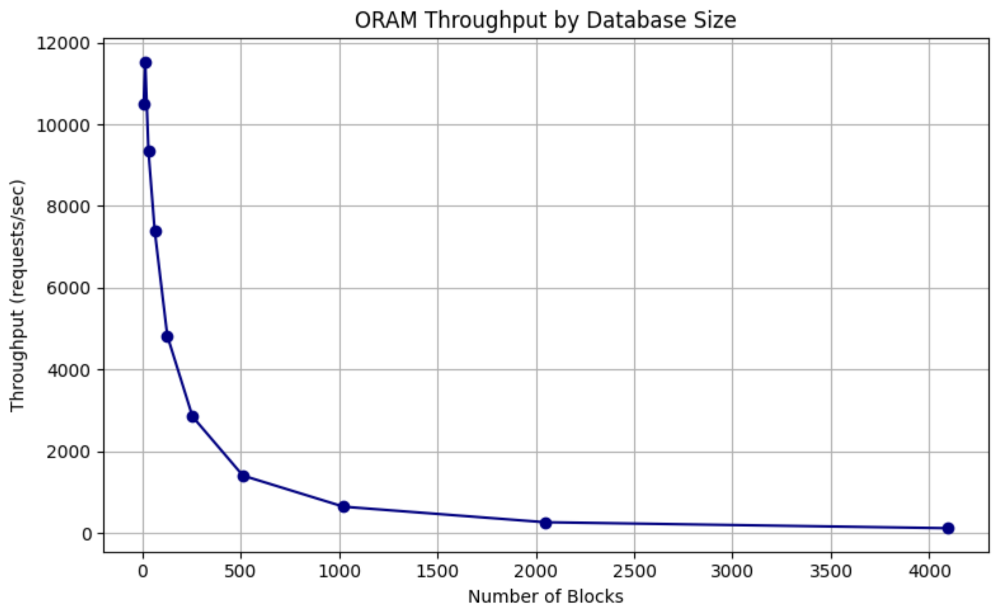
- Requests per trial: 50
- Number of trials per configuration: 5
- Bucket capacity: 2
- Database sizes tested: 8 to 4096 blocks (powers of 2)

Each trial involved randomly accessing blocks within the ORAM to simulate realistic workloads. Results were averaged over 5 independent runs to ensure statistical reliability.

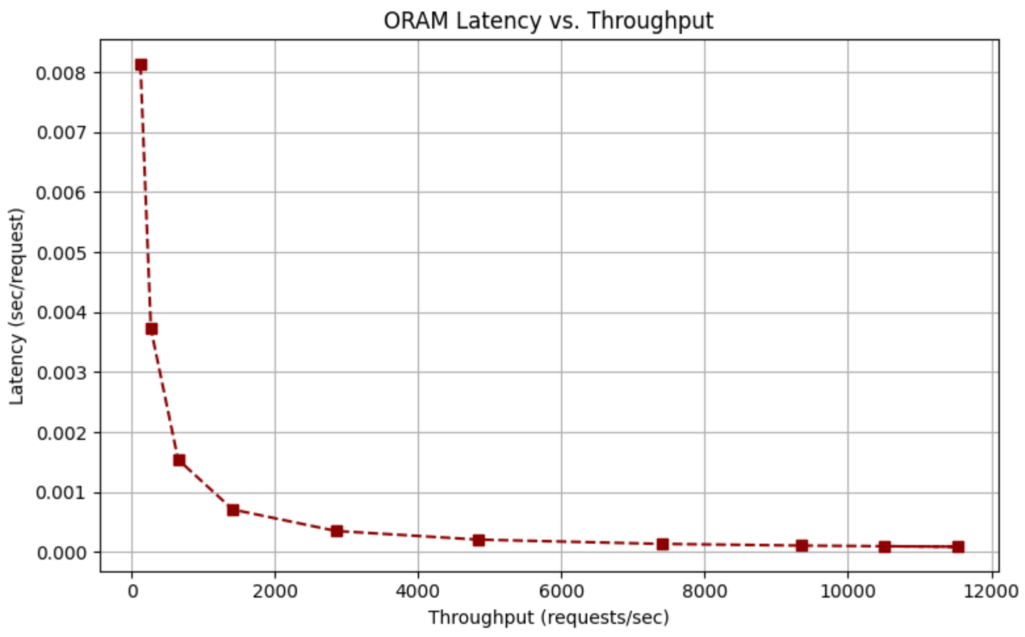
All measurements were taken in seconds for latency and requests per second for throughput.

Results:

Database Size	Avg Throughput (req/s)	Avg Latency (s/req)
8	10,512.95	0.000096
16	11,530.66	0.000088
32	9,358.48	0.000108
64	7,402.14	0.000136
128	4,831.33	0.000207
256	2,854.20	0.000351
512	1,410.80	0.000709
1024	649.18	0.001542
2048	269.62	0.003722
4096	123.18	0.008142



This graph clearly illustrates that throughput decreases as the database size increases. This is expected behaviour in Path ORAM due to the logarithmic increase in tree depth and corresponding linear growth in path access time. Smaller trees allow faster path operations, while larger trees require traversing and rewriting longer paths.



This plot reveals the inverse relationship between throughput and latency. As the throughput drops for larger datasets, latency per request increases accordingly. The curve shows an exponential decay, confirming the trade-off inherent in ORAM schemes between access privacy and efficiency.

The performance results highlight a clear trade-off in Path ORAM between privacy and efficiency. As the database size increases, throughput decreases significantly while latency per request rises.

This behaviour is expected due to the logarithmic growth in tree depth, which leads to longer path traversals and increased processing time. The data also shows an inverse relationship between throughput and latency, with the latency curve following an exponential trend as dataset size grows.

These results emphasize the scalability limitations of Path ORAM and point to the need for optimizations—such as parallelism and batch processing—to improve performance for larger datasets.

Can we benefit from multicore?

The current ORAM implementation operates in a single-threaded manner, with both client operations and server interactions executed sequentially. As a result, it does not take advantage of multicore systems.

However, performance could be significantly improved in the future by introducing parallelism in path reads and writes, enabling asynchronous processing of the stash, and implementing batch encryption and decryption techniques.