Course: 67515
My Name: Yiftach Sabag

Exercise 1

1 Implementation Details

1.1 Functionality: Client API

Client Write. Client's `write` methods works as follows: Client reads all buckets on the root-leaf path, flush the ORAM tree in the server. Finally, he writes the new added data to the root node. In the write process, all buckets along the path are encrypted, including the root node.

Client Read. client's `read` is very similar to the `write` operation. Client reads all buckets along the corresponding root-leaf path. If the requested file is found, client re-encrypt the data, delete if from the current bucket, and write it to the root node. Finally, client flushes the ORAM tree in the server. all buckets along the root-leaf path are re-encrypted.

Client Flush. client's `flush` works as follows: The client chooses randomly 2 nodes from each level in the tree, then 2 files are chosen randomly from each bucket. These 2 files are then pushed down into the lower level in the tree: they are written to the correct children, in relation to the root-leaf path. all randomly chosen buckets are re-encrypted. If chosen node is a leaf node, then the chosen data is deleted from the server, and returned back to the client.

Client Delete. The `delete` operation of the client for a file `file` is used by reading all the buckets along the corresponding root-leaf path. If the file is found, "0" is written into the root node. Thus, all `read, write` and `delete` operations look the same to the server or to a potential attacker.

1.2 Functionality: Server API

Server Owrite. `owrite` request from the server asks it to write the given `file`, assigned to `leaf_id`, into the bucket inside the root node of the ORAM tree. All written data is encrypted by the Client.

Server Oread. `oread` request from the server, given a node key of some node in the tree, returned the corresponding bucket inside the node with key `key`.

1.3 Security

End to end encryption. Encryption was implementing with asymmetric–encryption approach using `rsa` module. This allows the client to re-encrypt the data with the same public key, and still decrypt it with the same private key.

ORAM. Was implemented with perfect binary tree, with n leaves. The tree is represented by the class `BinaryTree`. Each `Node` inside the tree contain `Bucket` instance of size $\log n$.

Authentication. was implemented using a signature over the hash of the data (using the private key of the client). We used `sha256` as hash function using `hashlib` module.

1.4 Limitations

1. A client can only read a full bucket, not a specific file.
2. Number of leaves inside the tree can only be 2^k for some $k \in \mathbb{N}$. This way, the tree can be a perfect binary tree.
3. `Server` is designed to serve only one `client` instance
4. A `client` can be registered to many `Server` instances

2 Results

All the results were measured as a function of number of leaves N, for $N = 1, 2, 4, 8, ..., 4096$.

2.1 Throughput

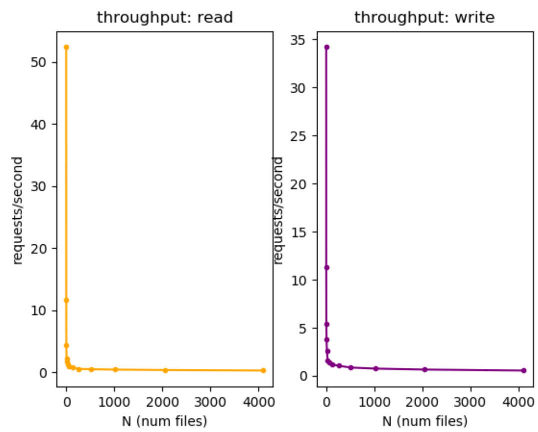


Figure 1: Results of throughput of `read, write` client calls.

2.2 Latency

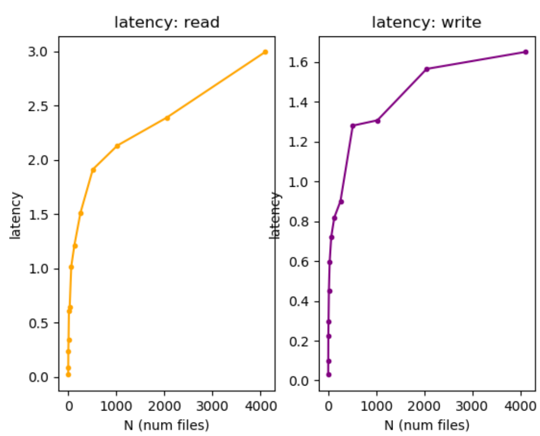


Figure 2: Results of latency of `read`, `write` client calls.

2.3 Throughput versus Latency

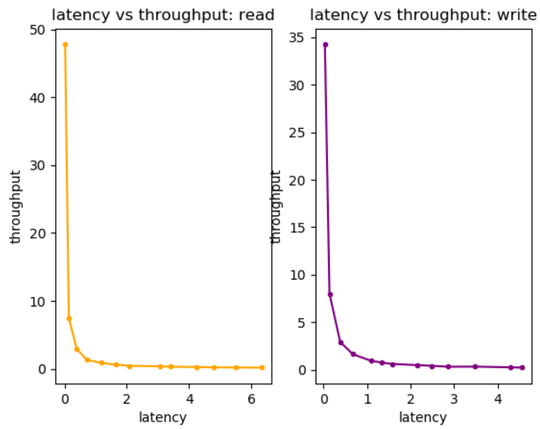
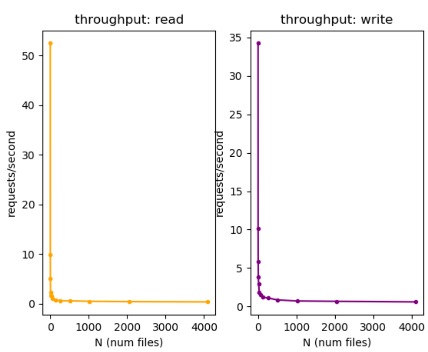
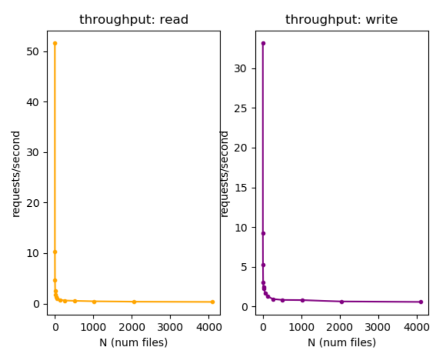


Figure 3: Results of latency vs throughput of `read`, `write` client calls.

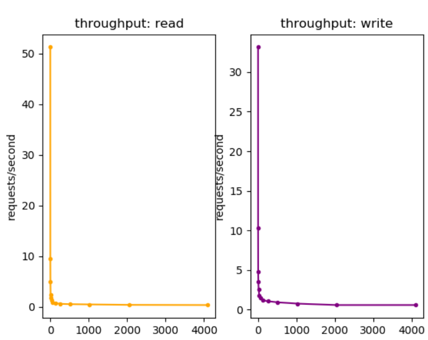
2.4 Multi-core Comparison



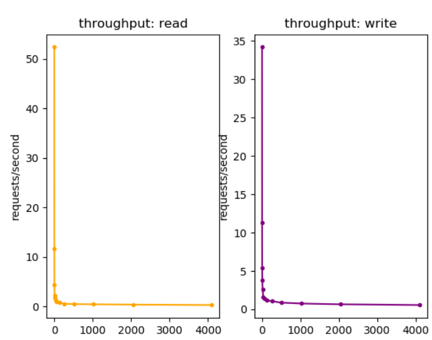
(a) Using 1/6 of CPU cores



(b) Using 3/6 of CPU cores

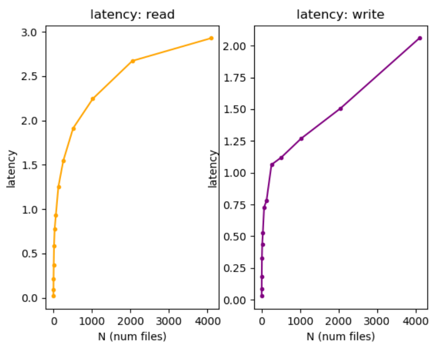


(c) Using 4/6 of CPU cores

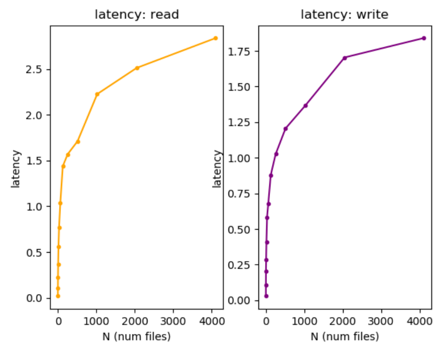


(d) Using 6/6 of CPU cores

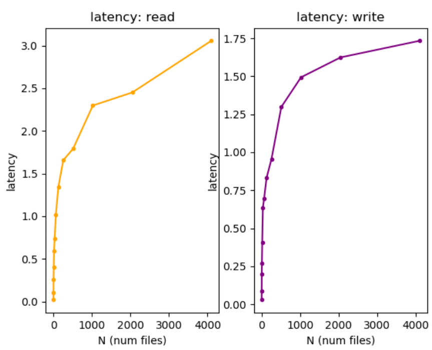
Figure 4: Comparing results using different number of cores when running the program (Throughput)



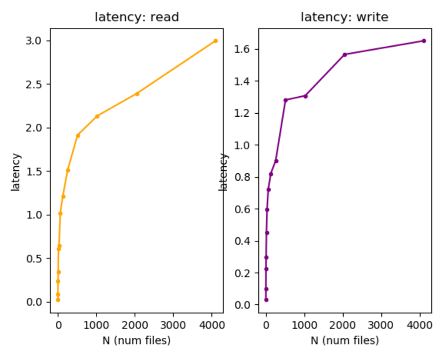
(a) Using 1/6 of CPU cores



(b) Using 3/6 of CPU cores



(c) Using 4/6 of CPU cores



(d) Using 6/6 of CPU cores

Figure 5: Comparing results using different number of cores when running the program (Latency)

Conclusion. Improvement of throughput and latency performances by using more cores is very subtle (if any).