KU LEUVEN

Panacea: Non-interactive and Stateless Oblivious RAM

Kelong Cong, Debajyoti Das, **Georgio Nicolas** and Jeongeun Park

KU LEUVEN

Recap on Oblivious Random Access Memory (ORAM)

Setting the Stage

- Client-Server Scenario
- The client stores a database encrypted by a key they control on a remote untrusted server.
- The client would like to request the server to perform a **read** or a **write** operation on any data element.

ORAM Security Properties

- The server performs the requested operation and returns the targeted data element without knowing any of the following:
    - Whether the client requested a **Read** or a **Write** operation
    - **Which** data element was targeted by the client
    - **What** is the content of any stored data element

Simple Solution

- The client encrypts each element in the database with a randomized symmetric-key encryption scheme.
- For each access, the client does the following:
  1. Requests the whole database
  2. Decrypts the db
  3. Does whatever they want with the data
  4. Re-encrypts the db and sends it back to the server.

- Since encryption is randomized, the Server would not know which element was targeted by the client, nor whether the client performed a read or a write operation.

Problems with the Simple Solution

- The client might not have enough storage available to download the entire database locally.
- It is not efficient (bandwidth-wise) to download the entire database if the client needed to access only a small number of elements.
- It could take some time to decrypt then re-encrypt the entire database on a client without much computational resources.
- The client has to maintain a dynamic state because of the randomized encryption scheme in order to be able to decrypt.
- ORAM schemes try to solve these problems in a more efficient manner.

Some Important Properties 1/3

Bandwidth Blowup

- The ratio of communication cost required by ORAM to that of performing the same operation in plaintext

Some Important Properties 2/3

Stateful ORAM

- The client maintains a dynamic state which is updated after every number of queries to the server

Some Important Properties 2/3

Stateful ORAM

- The client maintains a dynamic state which is updated after every number of queries to the server

Stateless ORAM

- The client only needs to maintain static data (keys, or hashes of indices) that are computed once during the setup
- No need to synchronise state, or for writeable client storage

Some Important Properties 3/3

Interactive ORAM

- The client and the server perform multiple rounds of interaction before the server can return the final result or update the database

Some Important Properties 3/3

Interactive ORAM

- The client and the server perform multiple rounds of interaction before the server can return the final result or update the database

Non-Interactive ORAM

- The client can send their query and then do nothing
- The query would contain everything that the server needs to respond and update the database

ORAM Models

Classical Model

- The client performs all the computation
- The server only stores data and sends it back on demand
- Lower bound of $\Omega(log(n))$ proven in 1996 by Goldreich and Ostrovsky

ORAM Models

Classical Model

- The client performs all the computation
- The server only stores data and sends it back on demand
- Bandwidth blowup lower bound of $\Omega(log(n))$ proven in 1996 by Goldreich and Ostrovsky

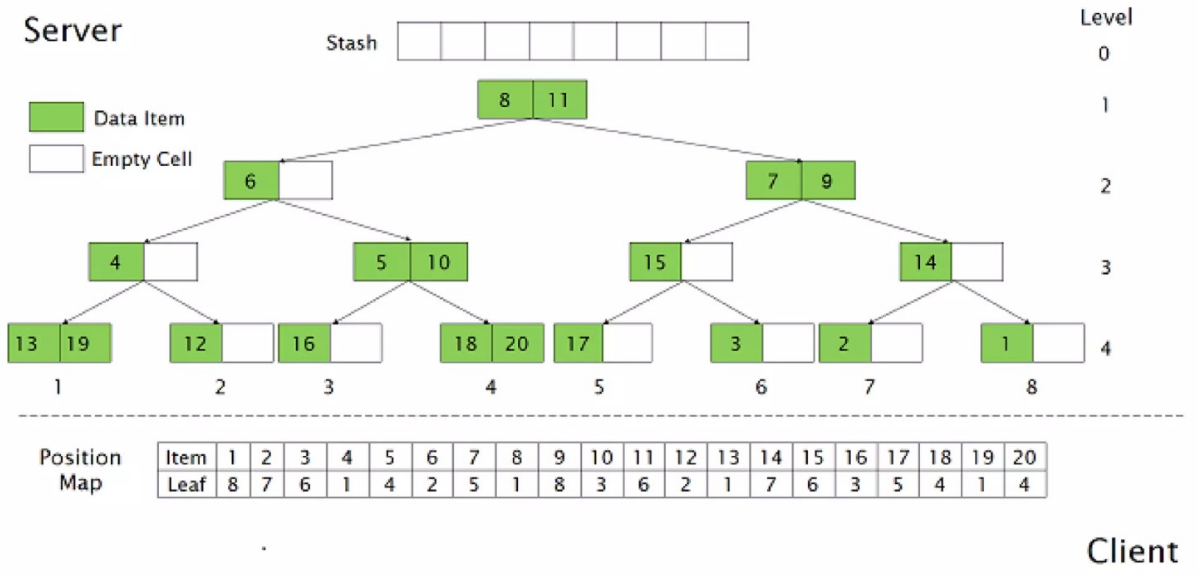
Server-Computation Model

- The server contributes to the computation which guarantees security
- Less load on the client
- Constant bandwidth overhead first achieved by Circuit ORAM (with tight assumptions)
- State of the art was Onion ORAM (theoretical result) leveraging HE for a smaller overhead with looser assumptions

Previous ORAM Schemes

Stateful and Interactive

- Tree-based database structures
  - Circuit ORAM (Wang et. al, 2014)
  - Onion ORAM (Devadas et al.)
  - PathORAM (Stefanov et al., 2013)…
- Hierarchical database structures
  - (Ostrovsky, 1992)



Position Map

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Leaf	8	7	6	1	4	2	5	1	8	3	6	2	1	7	6	3	5	4	1	4

Circuit ORAM

Ostrovsky’s Hierarchical ORAM

Image credit: https://www.slideshare.net/AshutoshSatapathy4/oram

PathORAM (Stefanov et al., 2013)

- Tree-based data structure
- Client maintains a position map which maps every element to a path to a leaf node from the tree
- Client requests all blocks in a given path ($O(log^2(N))$), decrypts it and operates on the data
- There is a shuffling procedure involved which updates the position map
- Client returns updated nodes to the server

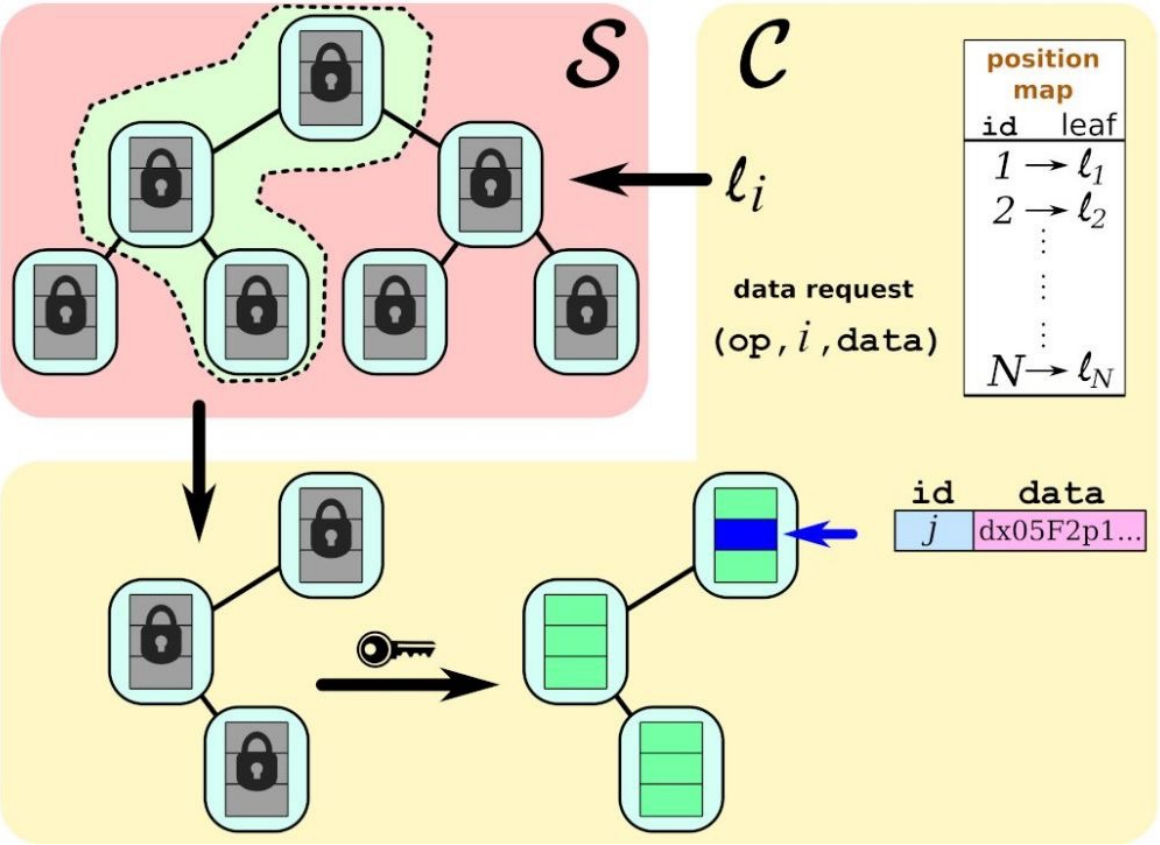


Image credit: https://research.kudelskisecurity.com/2020/04/22/an-introduction-to-oblivious-ram-oram/

Onion Ring ORAM (Chen et. al, 2019)

- FHE-based
- Tree-based structure
- Stateful and Non-Interactive* (*: technically still interactive)
- Non-Interactive and the first to provide non-theoretical constant bandwidth overhead in the “Online” phase
- However requires:
  - The client to maintain a dynamic state
  - An expensive interactive “offline” eviction phase after a fixed number of queries with non-constant bandwidth blowup
  - Writes to occur only after eviction

Some Caveats

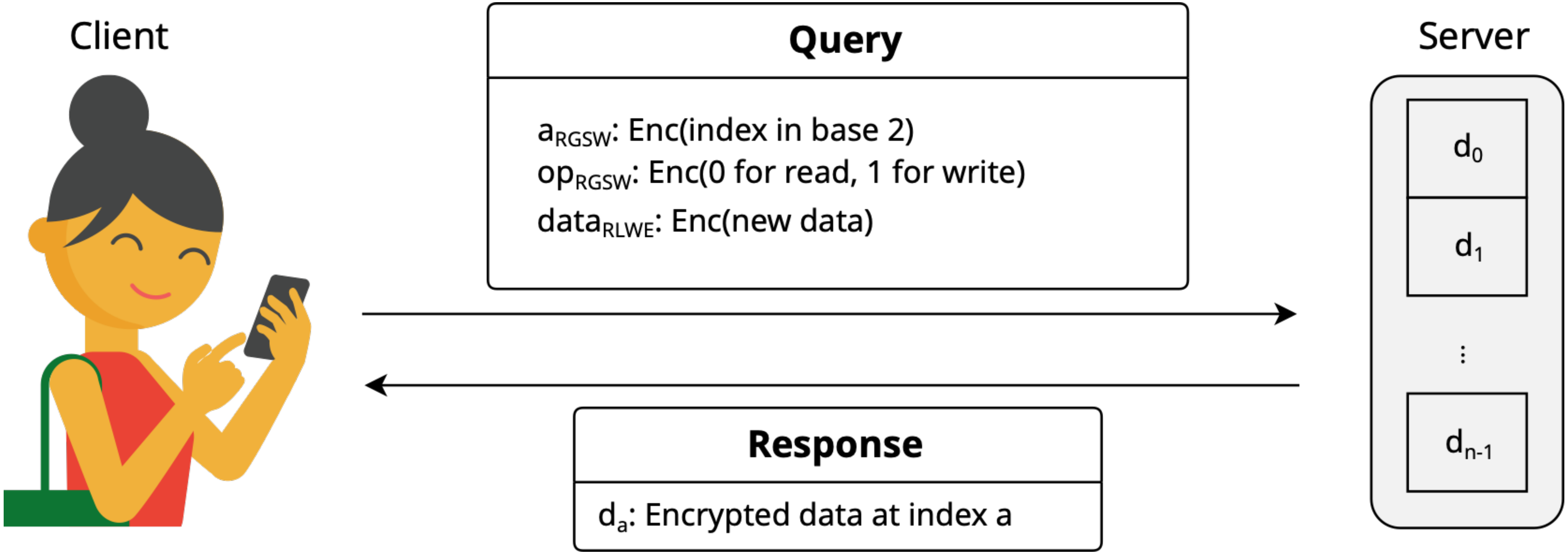
- We think it’s interesting to have an ORAM scheme that is completely **Stateless** and **Non-Interactive**, with **Constant Bandwidth Blowup**
- Previous schemes demand that the client:
  - Performs intensive computations (ex. During the eviction phase)
  - Maintains a dynamic state (ex. Positions of elements after shufffling)
  - Interacts with the server before being able to write to the db
  - Transfers a lot of data
- A cloud-based password manager wouldn’t be quite nice to use if it requires such properties

Our Proposed ORAM Scheme

Our ORAM Scheme: Panacea

- Pros
  - We can leverage FHE for a much simpler design
  - Constant bandwidth overhead
  - Stateless and non-interactive
  - No offline phase
  - Does not require a powerful client (can run on a phone)
  - Open source implementation: https://github.com/KULeuven-COSIC/Panacea
- Cons
  - Linear computational complexity

Simple Stateless and Non-Interactive ORAM



Client

Query

a_{RGSW}: Enc(index in base 2)
op_{RGSW}: Enc(0 for read, 1 for write)
$data_{RLWE}$: Enc(new data)

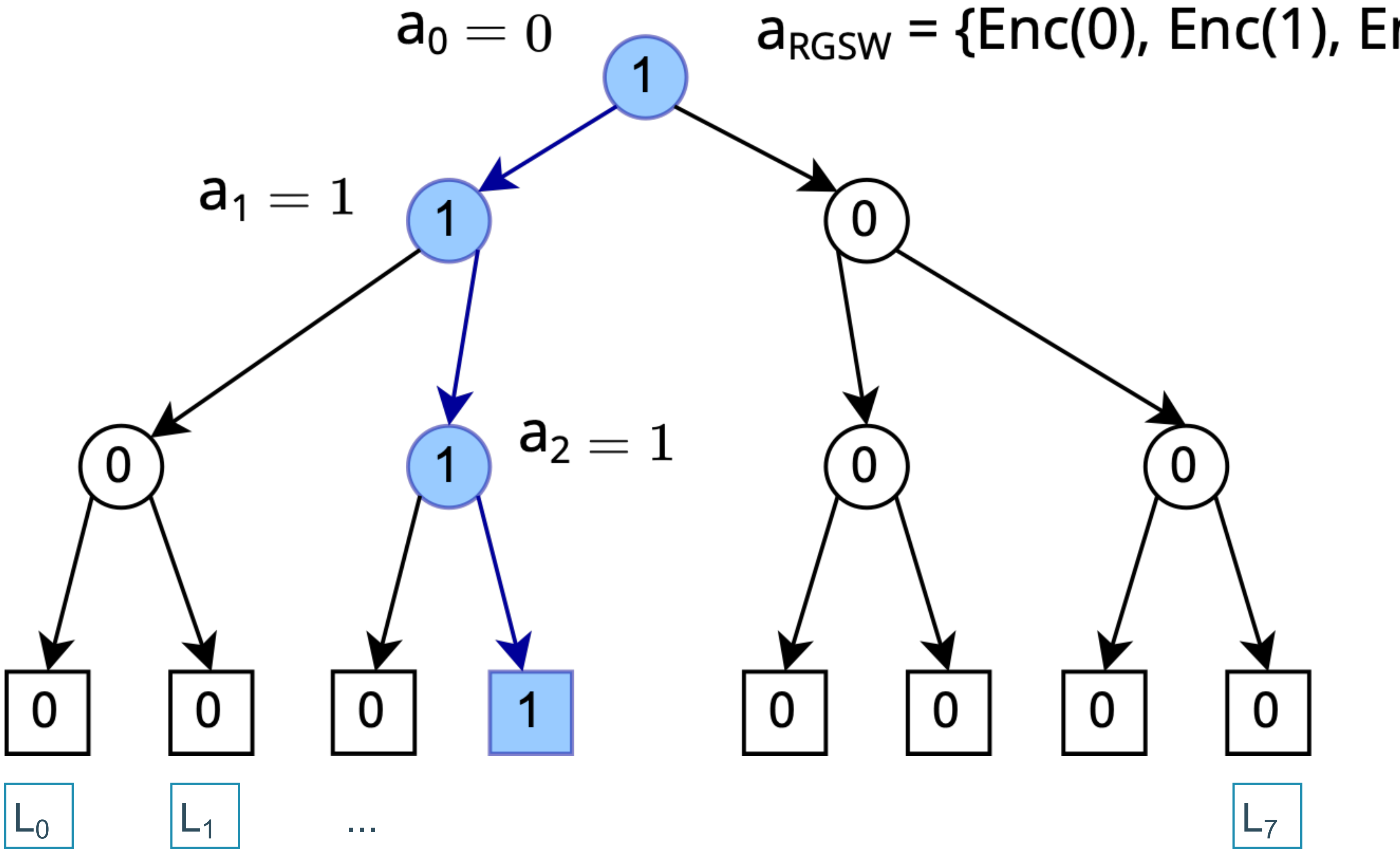
Response

d_a: Encrypted data at index a

Server

Homomorphic Demultiplexing

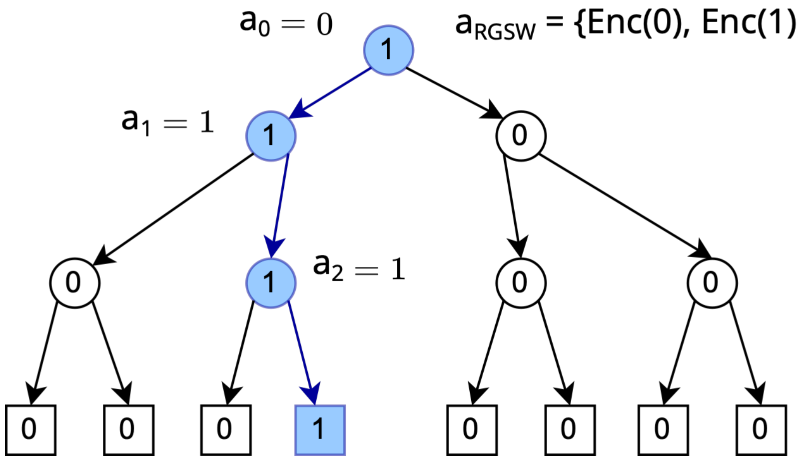
a_{RGSW} = {Enc(0), Enc(1), Enc(1)}



n leaves

$(n = 8)$

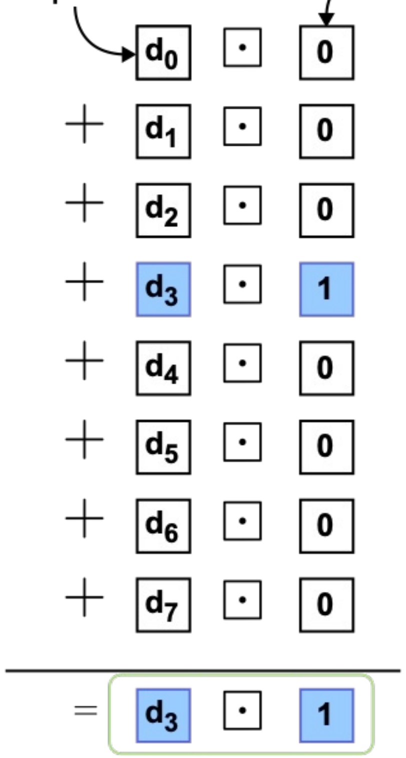
Response Calculation



a_{RGSW} = {Enc(0), Enc(1), Enc(1)}

n leaves

$(n = 8)$



RLWE ciphertexts

Run RLWEtoRGSW on the leaf nodes from the result of the traversal. Denote the resulting RGSW ciphertext by L_j

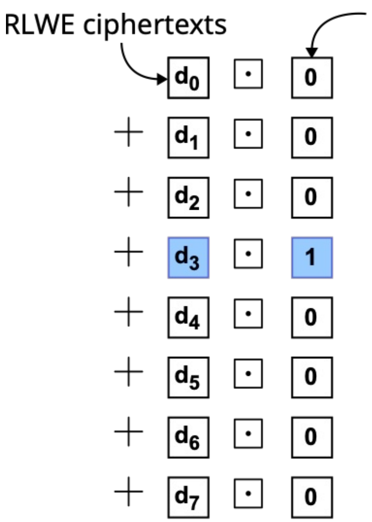
$$d_0 \boxdot 0 + d_1 \boxdot 0 + d_2 \boxdot 0 + d_3 \boxdot 1 + d_4 \boxdot 0 + d_5 \boxdot 0 + d_6 \boxdot 0 + d_7 \boxdot 0 = d_3 \boxdot 1$$

Send back to the client (prior to updating the DB)

Update Step

Query

a_{RGSW}: Enc(index in base 2)
op_{RGSW}: Enc(0 for read, 1 for write)
$data_{RLWE}$: Enc(new data)



Run RLWEtoRGSW on the leaf nodes from the result of the traversal. Denote the resulting RGSW ciphertext by L_j

- We make use CMUX(c, a, b)
  - Define as CMUX(Enc(0), a, b) = b, CMUX(Enc(1), a, b) = a
- For j in $0..n$
  1. $temp$ = CMUX($op, data, d_j$); // output $data$ if it's a write
  2. d_j = CMUX($L_j, temp, d_j$); // output $temp$ when at index a

Summary So Far

1. Client encrypts a query and sends it
2. Server performs:
    1. Homomorphic Demultiplexing (compute the unit vector)
    2. Response Calculation (the inner product)
    3. Responding to the client
3. Client receives the response and decrypts it
4. Server performs the Update Step (to write to the database)

Summary So Far

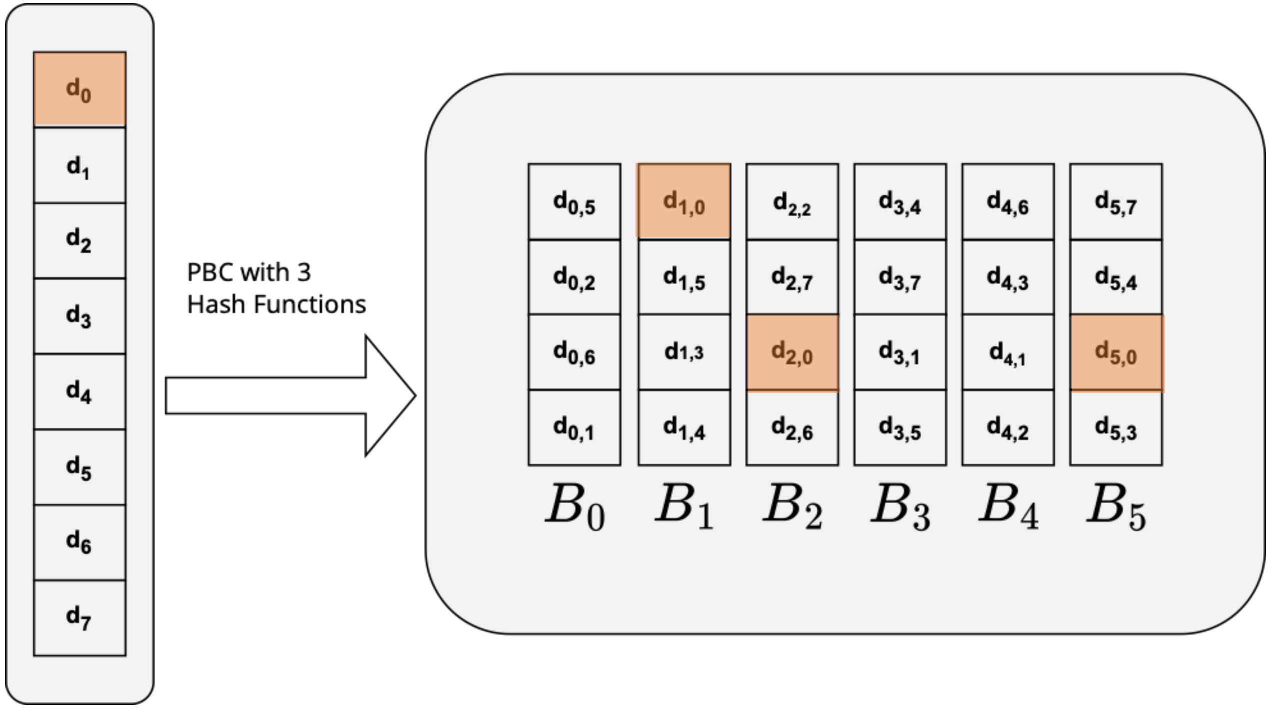
1. Client encrypts a query and sends it: **O(log n) comm.**
2. Server performs:
    1. Homomorphic Demultiplexing (compute the unit vector): **O(n)**
    2. Response Calculation (the inner product): **O(n)**
    3. Responding to the client: **O(1) comm.**
3. Client receives the response and decrypts it: **O(1)**
4. Server performs the Update Step (to write to the database): **O(n)**

But We Can Do Better

- Some applications may wish to query multiple elements at a time
    - e.g., retrieving a lot of data to run analysis
- Running the protocol above many times is not ideal
- Our batched design uses probabilistic batch codes
    - This was never done in ORAM due to consistency issues, which we fix!

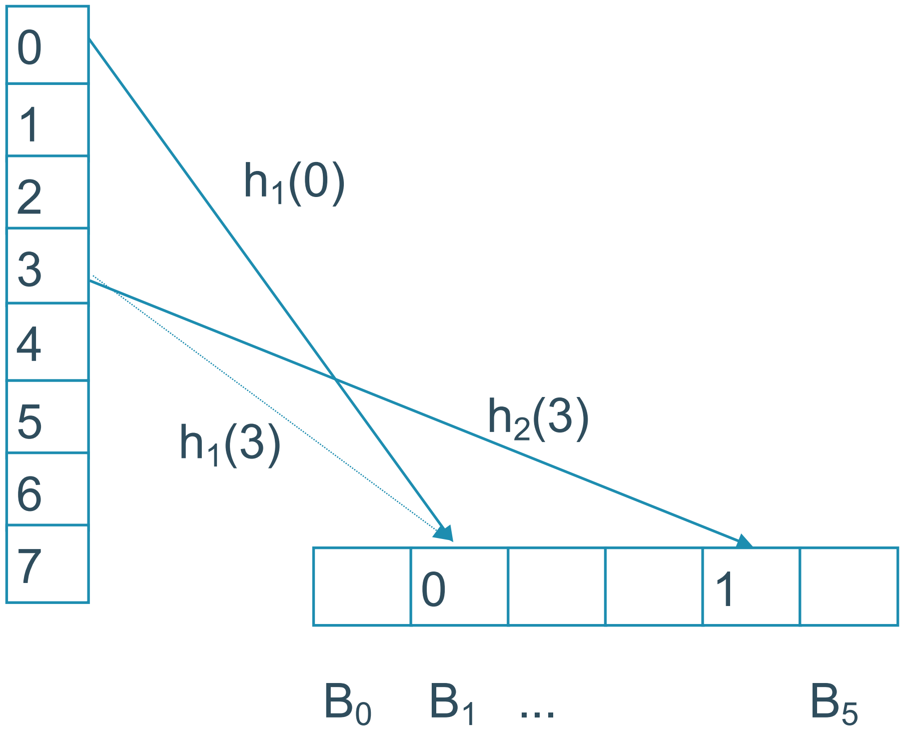
Batching Queries - Setup

- We use a probabilistic batch code (PBC) where every element is mapped to 3 locations
- Let *k* be the batch size
- Let *B* = *1.5k* as the number of columns in the encoded table
- Ideally, the same item should not be mapped to the same column more than once
- The goal of the client is to perform one access for each column



B_0	B_1	B_2	B_3	B_4	B_5
$d_{0,5}$	$d_{1,0}$	$d_{2,2}$	$d_{3,4}$	$d_{4,6}$	$d_{5,7}$
$d_{0,2}$	$d_{1,5}$	$d_{2,7}$	$d_{3,7}$	$d_{4,3}$	$d_{5,4}$
$d_{0,6}$	$d_{1,3}$	$d_{2,0}$	$d_{3,1}$	$d_{4,1}$	$d_{5,0}$
$d_{0,1}$	$d_{1,4}$	$d_{2,6}$	$d_{3,5}$	$d_{4,2}$	$d_{5,3}$

Batching Queries - Query creation



= *a* (index to query)

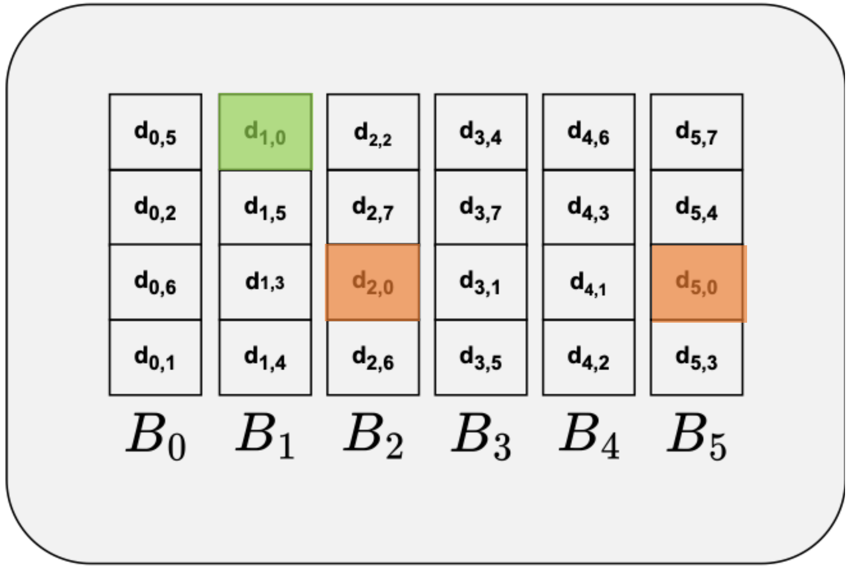
- Client makes $B = 1.5k$ queries to the server using cuckoo hashing
- We use $k=256$, $B=384$, $h=3$ to achieve 2^{-40} failure probability

Batching Queries - Server computation

- Server performs the basic Panacea protocol for every column (in parallel!)
    - Demultiplexing, response calculation, response
- Client decrypts the response and finds the *k* responses that it is interested in
- Server performs the update
- But there will be consistency issues!

Consistency Issue

- Suppose client wants to update d_0
- It would need to update three locations
- So it is no longer a batched query!
- Goal: if client updates $d_{1,0}$, the server should figure out how to also update $d_{2,0}$ and $d_{5,0}$



B_0	B_1	B_2	B_3	B_4	B_5
$d_{0,5}$	$d_{1,0}$	$d_{2,2}$	$d_{3,4}$	$d_{4,6}$	$d_{5,7}$
$d_{0,2}$	$d_{1,5}$	$d_{2,7}$	$d_{3,7}$	$d_{4,3}$	$d_{5,4}$
$d_{0,6}$	$d_{1,3}$	$d_{2,0}$	$d_{3,1}$	$d_{4,1}$	$d_{5,0}$
$d_{0,1}$	$d_{1,4}$	$d_{2,6}$	$d_{3,5}$	$d_{4,2}$	$d_{5,3}$

Consistency Correction Algorithm

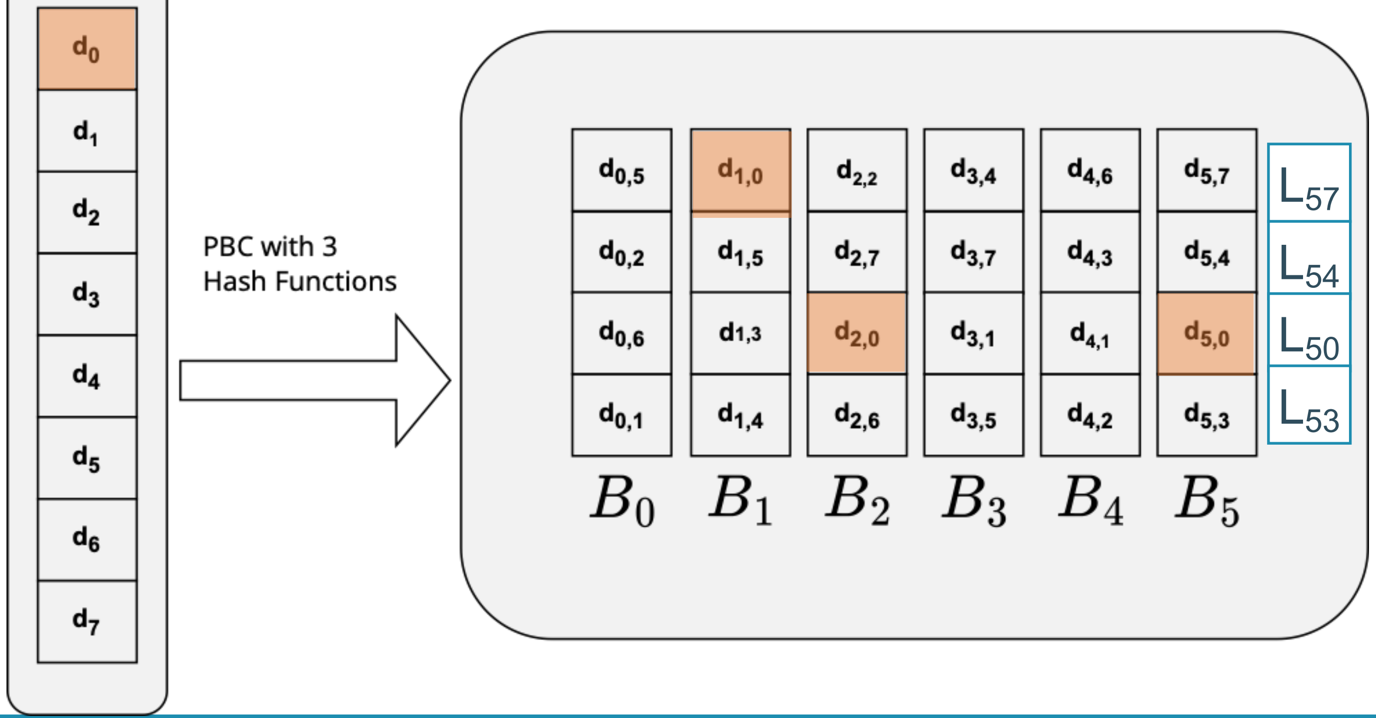
$$d_{1,0} \bullet (1 - L_{1,0} \bullet op_1 - L_{2,0} \bullet op_2 - L_{5,0} \bullet op_5) +$$

0 if we're writing, else $d_{1,0}$

$$(d_{1,0} \bullet L_{1,0} \bullet op_1 + d_{2,0} \bullet L_{2,0} \bullet op_2 + d_{5,0} \bullet L_{5,0} \bullet op_5)$$

The latest data if we're writing

at most one L is Enc(1)

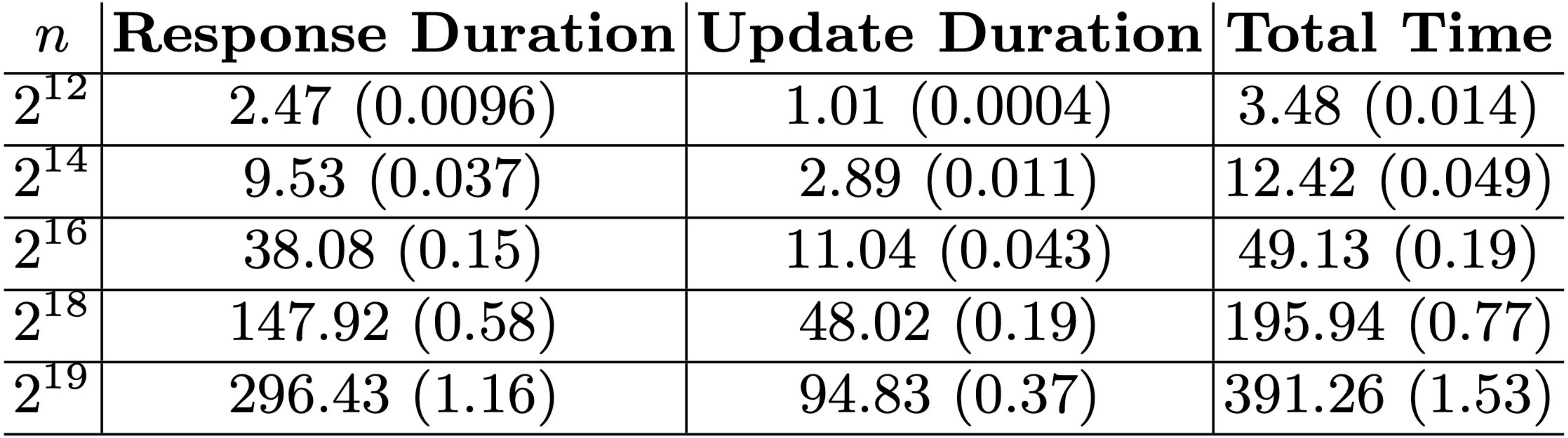


B_0	B_1	B_2	B_3	B_4	B_5	
$d_{0,5}$	$d_{1,0}$	$d_{2,2}$	$d_{3,4}$	$d_{4,6}$	$d_{5,7}$	L_{57}
$d_{0,2}$	$d_{1,5}$	$d_{2,7}$	$d_{3,7}$	$d_{4,3}$	$d_{5,4}$	L_{54}
$d_{0,6}$	$d_{1,3}$	$d_{2,0}$	$d_{3,1}$	$d_{4,1}$	$d_{5,0}$	L_{50}
$d_{0,1}$	$d_{1,4}$	$d_{2,6}$	$d_{3,5}$	$d_{4,2}$	$d_{5,3}$	L_{53}

The expression outputs the latest data if there is a write in any of the three positions and then it is copied to $d_{1,0}$, $d_{2,0}$ and $d_{5,0}$

Some Numbers

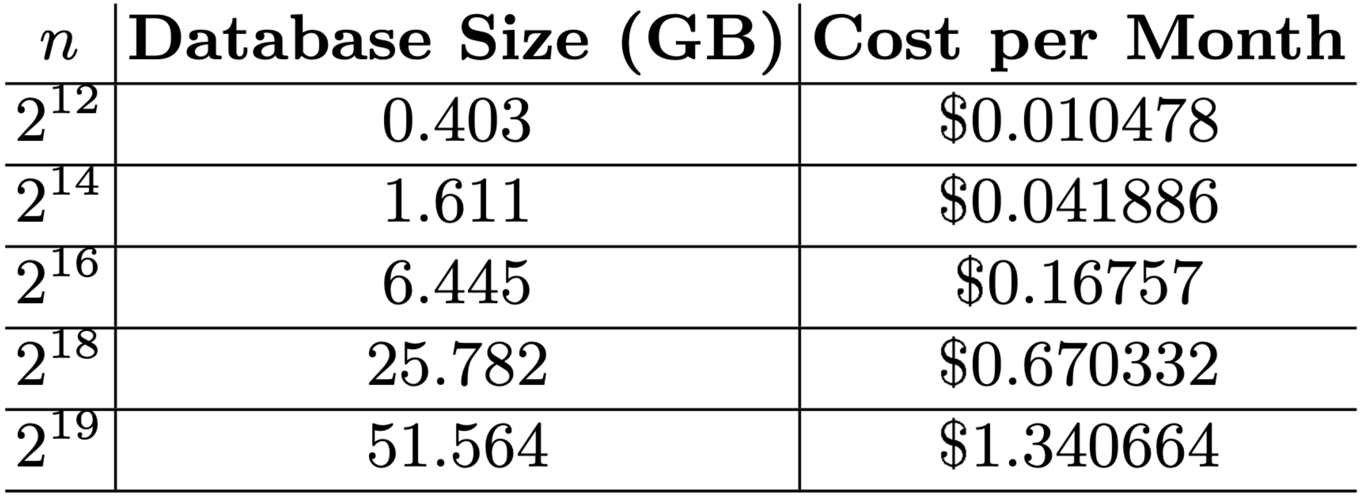
- Implementation in Rust based on the concrete-core library (TFHE)
- Computation time in seconds required by the server for database of size n, batch size $k = 256$ (ie 384 queries with 128 dummies)



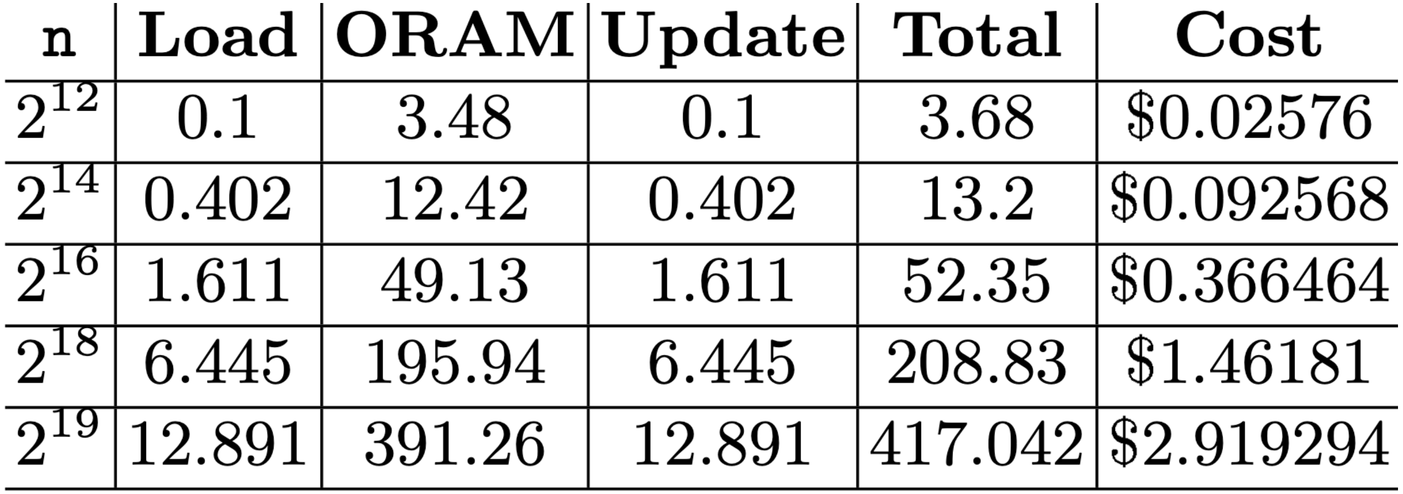
n	**Response Duration**	**Update Duration**	**Total Time**
2^{12}	2.47 (0.0096)	1.01 (0.0004)	3.48 (0.014)
2^{14}	9.53 (0.037)	2.89 (0.011)	12.42 (0.049)
2^{16}	38.08 (0.15)	11.04 (0.043)	49.13 (0.19)
2^{18}	147.92 (0.58)	48.02 (0.19)	195.94 (0.77)
2^{19}	296.43 (1.16)	94.83 (0.37)	391.26 (1.53)

More Numbers

- Top table:
  Cost of storing the database
- Bottom table:
  Server compute time and cost for processing one batch of queries
- Real numbers from Google Cloud
- Batched Scenario (256 elements)



n	**Database Size (GB)**	**Cost per Month**
2^{12}	0.403	\$0.010478
2^{14}	1.611	\$0.041886
2^{16}	6.445	\$0.16757
2^{18}	25.782	\$0.670332
2^{19}	51.564	\$1.340664



n	**Load**	**ORAM**	**Update**	**Total**	**Cost**
2^{12}	0.1	3.48	0.1	3.68	\$0.02576
2^{14}	0.402	12.42	0.402	13.2	\$0.092568
2^{16}	1.611	49.13	1.611	52.35	\$0.366464
2^{18}	6.445	195.94	6.445	208.83	\$1.46181
2^{19}	12.891	391.26	12.891	417.042	\$2.919294

KU LEUVEN

Thank you :-)

[ia.cr/2023/274](ia.cr/2023/274)