Applications of FHE to Privacy Preserving Protocols

Jeongeun Park

imec-COSIC, KU leuven

The 3rd ISC Winter School on Cryptography, March 2, 2023

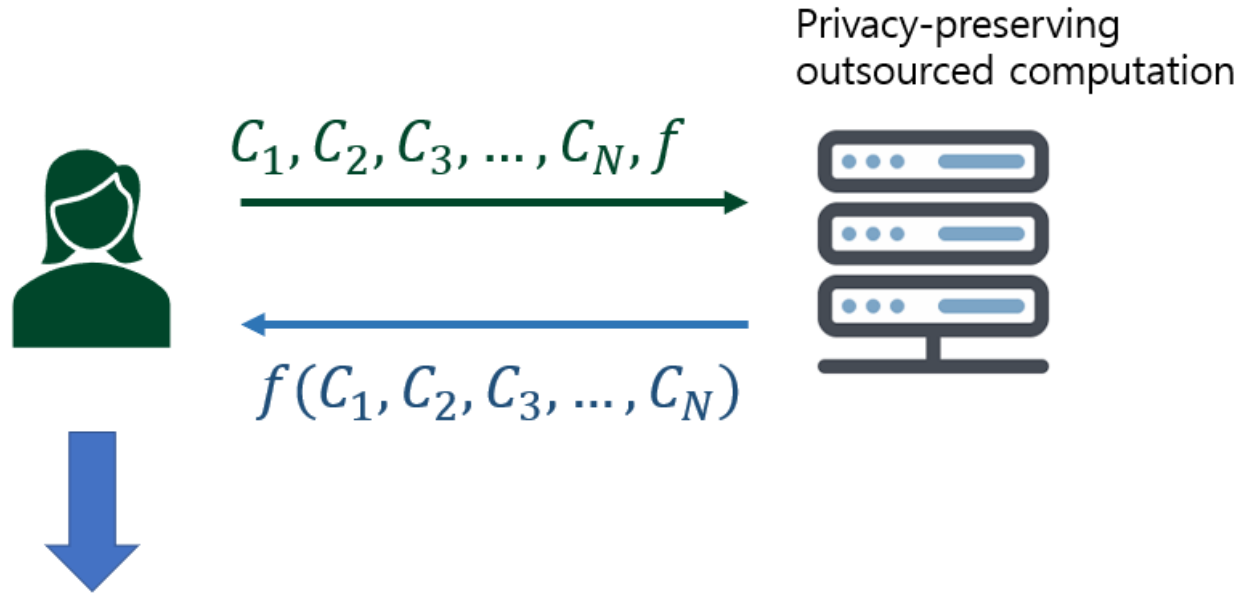
Outline

- Applications of FHE
- Private Information Retrieval
- Oblivious RAM
- Private Set Intersection
- Applications to Privacy Preserving Machine Learning

Applications of FHE

Outsourced Computation

Outsourced computation where a server does requested computation over encrypted data given from a client is a well known scenario for applications of FHE.



She decrypts the function value with her own secret key to get $f(m_1, m_2, \ldots, m_N)$, where m_i is the plaintext of C_i for $i \in [N]$.

Privacy preserving protocols based on FHE

There are advantages and disadvantages of privacy preserving protocols based on FHE:

- Single server scenario is possible
- asymptotically optimal communication/computation complexity is achievable
- non-interactive protocol is possible
- communication cost is high compared to other methods (like MPC)
- computation cost is very high!

We study few examples of such protocols especially focusing on how to solve the aforementioned negative points.

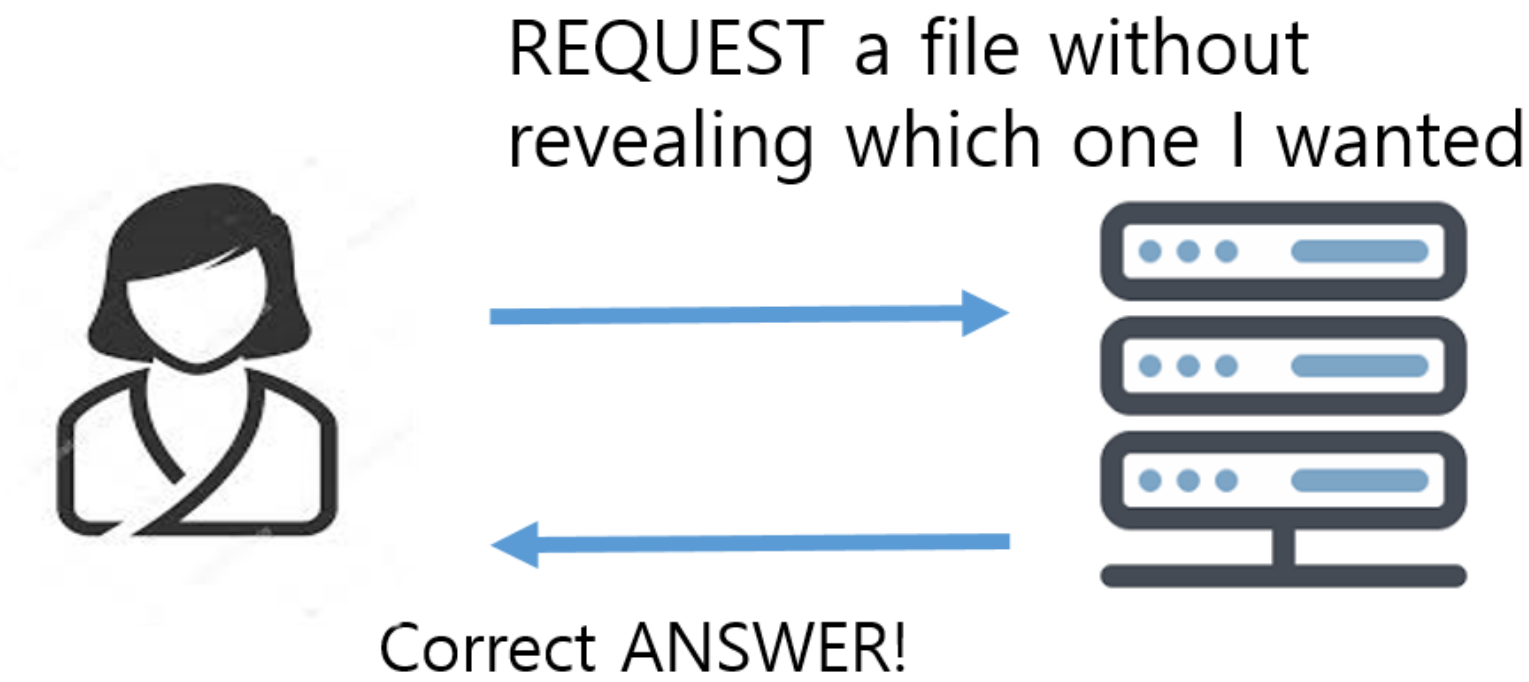
Threat model

Can we trust the computing party (cloud) (owned by companies) ? If so, how much should we trust him ?

- Semi-honest party : who follows a protocol honestly albeit it wants to know other parties' inputs.
  - But it cannot deviate from the protocol (changing the inputs and outputs, aborting the protocol, etc.)
- The security of FHE (IND-CPA) guarantees the client's security against semi-honest party (server)!
- Therefore, client's input data (can be confidential) can be securely preserved during computation with FHE if server is semi-honest.

Private Information Retrieval

Private Information Retrieval



Two basic types of PIR

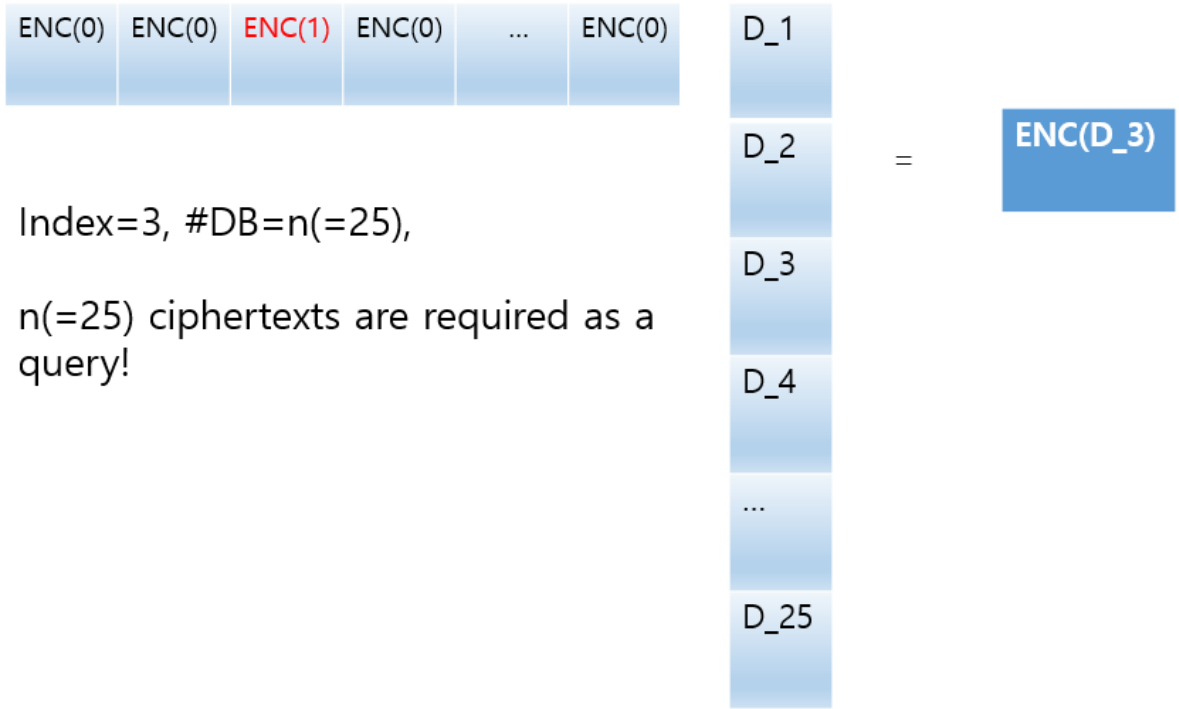
- Information-Theoretic PIR(IT-PIR)
  - \+ : Server's computation cost is relatively inexpensive(an XOR for each entry) & Information-theoretic privacy.
  - \- : non-collusion assumption between servers are required!
    (Chor et al. : transferring the entire database to the user has optimal communication complexity for IT-PIR with a single server.)

- Computational PIR (cPIR)
  - \+ : can be used with a single server under cryptographic hardness assumption & lower communication cost.
  - \- : more computational cost

We focus on cPIR based on homomorphic encryption schemes.

FHE based PIR protocols 1) XPIR[1]

Naive approach:



Too large network cost

[1]XPIR : Private Information Retrieval for Everyone

FHE based PIR protocols 1) XPIR

They use Stern's way to represent the query using $d\sqrt[d]{n}$ ciphertexts structuring DB as a d-dimensional hypercube to improve network cost.

ENC(0)	ENC(0)	ENC(1)	ENC(0)	ENC(0)

D(1,1)					ENC(0)
D(2,1)					ENC(1)
D(3,1)	D(3,2)				ENC(0)
D(4,1)					ENC(0)
D(5,1)				D(5,5)	ENC(0)

Let n=25, d=2.
Client wants to retrieve D(3,2).

= **ENC(D(3,2))**

$\therefore 2\sqrt{n} = 2\sqrt{25}$=10 ciphertexts are required as a query.

FHE based PIR protocols 2) SEAL-PIR[2]

Server's oblivious query expansion procedure

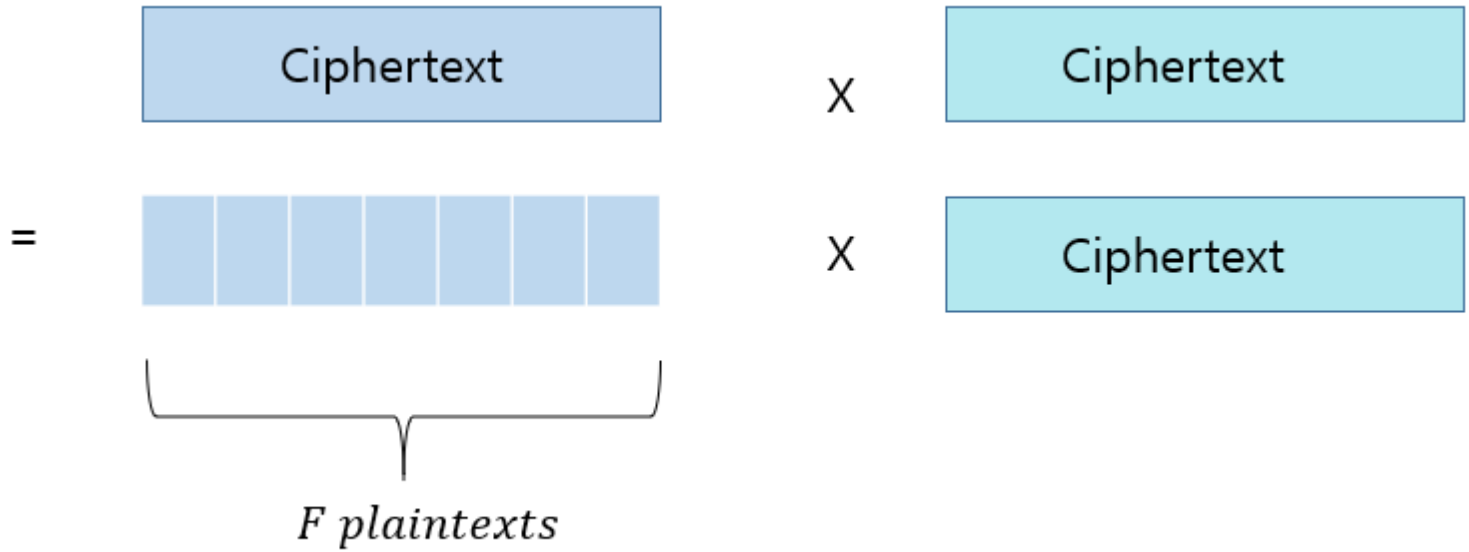
d(=2) ciphertexts are required as a query.
This procedure takes O(d$\sqrt[d]{n}$)

[2]PIR with compressed queries and amortized query processing

Server's reply

In order to avoid BFV's expensive (large noise growth, high computation cost) operation (multiplication over ciphertexts), both protocols use a multiplication between plaintext and ciphertext.

It results in larger response of server with the cryptosystem's expansion factor(F)=$|ciphertext|/|plaintext| \geq 4$:



Therefore, a client receives answers with F^{d-1} ciphertexts.

FHE based cPIR protocols 3) SHECS-PIR[3]

1) Query generation:

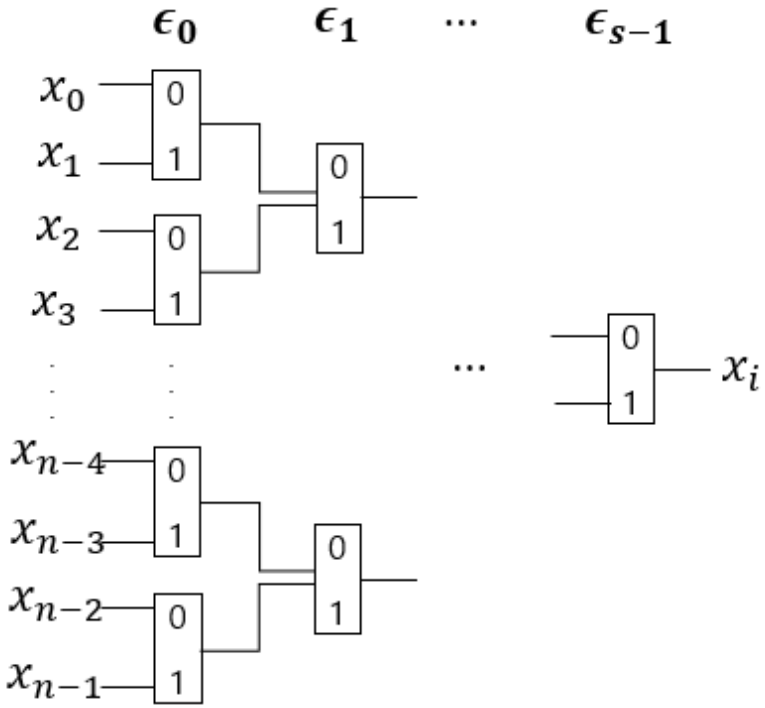
- Choose an index $i \in \{1, \ldots, n\}$
- Binarize the index : $i = \sum_{j=0}^{\log n - 1} \epsilon_j 2^j$ for $\epsilon_j \in \{0, 1\}$.
- Encrypt each $\epsilon_j \in \{0, 1\}$ for $j \in \{0, \ldots, \log n - 1\}$
- Send these ciphertexts to a server as a query.

[3]SHECS-PIR: Somewhat Homomorphic Encryption-Based Compact and Scalable Private Information Retrieval

2) Server's computation:

- Structure n data elements as a look up table (LUT) below.
- Run $n-1$ times homomorphic mux gate (called Cmux in TFHE) to select the desired ciphertext with the LUT and a binary tree (described below)
- Give the result to the client.

Figure 1 & Table 1: The Cmux binary decision tree (left figure) and Look Up Table (right table)



ϵ_{s-1}	...	ϵ_0	Data
0	...	0	x_0
0	...	1	x_1
⋮	⋮	⋮	⋮
1	...	1	x_{n-1}

Optimization for Query

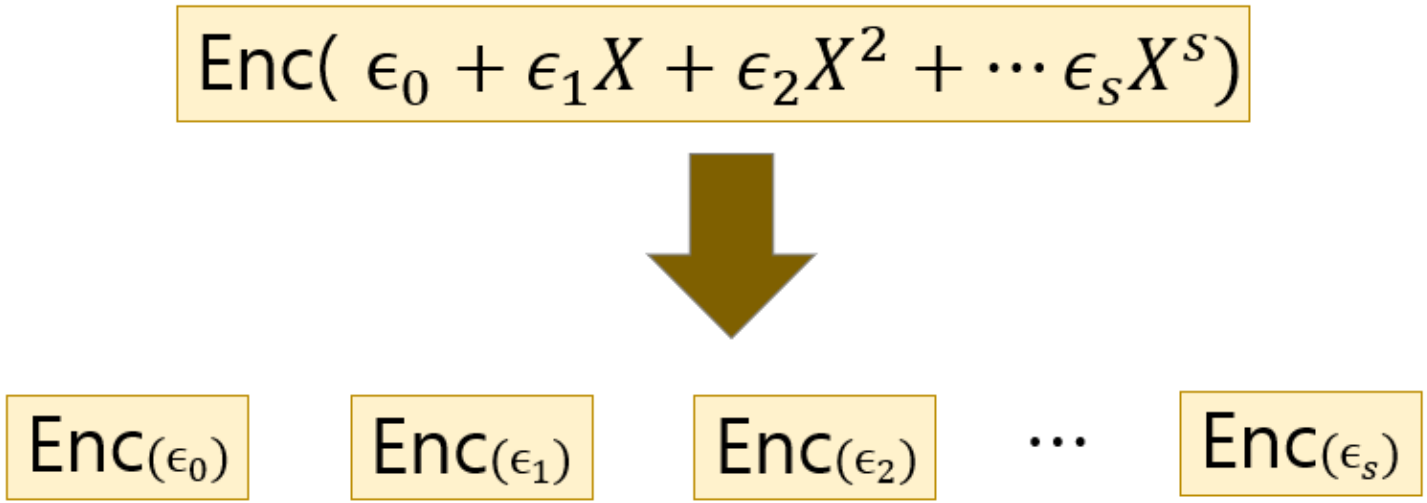
We reduce the size of query by packing query technique:

- Binarize the index i as $\sum_{j=0}^{s} \epsilon_j 2^j$, where $\epsilon_j \in \{0, 1\}, s = \log n - 1$.
- Set a plaintext polynomial $\epsilon_0 + \epsilon_1 X + \epsilon_2 X^2 + \cdots + \epsilon_s X^s$.
- Encrypt the polynomial.

Instead of encrypting each ϵ_j for $j \in \{0, \ldots, \log n - 1\}$, we pack each ϵ_j as a coefficient of a polynomial.

A server needs additional step called query unpacking to unpack the query as $\log n$ ciphertexts each of which encrypts ϵ_j.

Query Unpacking



This step takes additional time, however, the complexity is $O(\log n)$.

Complexity Comparison

Table: Communication and computation complexity, n = database size, q-u: query unpacking, ℓ= FHE parameter

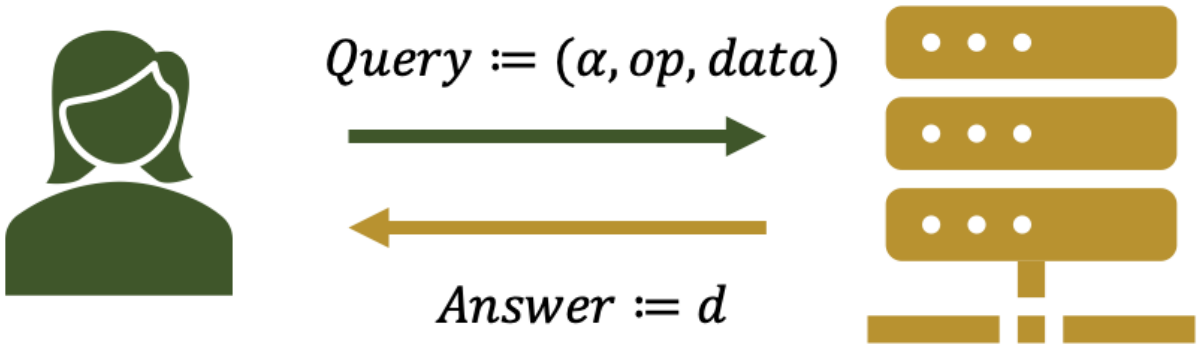
Protocol	Query	Answer	First-Step	Main
XPIR	$O(d\sqrt[d]{n})$	$O(F^{d-1})$	N/A	$\Omega(n + F\sqrt{n}$
SealPIR	$O(d\lceil\sqrt[d]{n}/N\rceil)$	$O(F^{d-1})$	$O(d\sqrt[d]{n})$	$\Omega(n + F\sqrt{n}$
SHECS-PIR	$O(\log n)$	$O(1)$	N/A	$O(\ell n)$
SHECS-PIR (q-u)	$O(\lceil\log n/N\rceil)$	$O(1)$	$O(\log n)$	$O(\ell n)$

First-Step is Query Unpacking in SHECS-PIR, Expand in SealPIR.

Oblivious RAM

Oblivious RAM (ORAM)

Oblivious RAM (ORAM) is a cryptographic primitive that allows a client to store its private data on an untrusted server without leaking any information to the server.



- If op = Read, data = $\perp$ and d = desired item.
- If op = Write, d = random value.

Note: ORAM cannot be used for PIR since the data base should be encrypted under client's key.

Existing ORAM designs

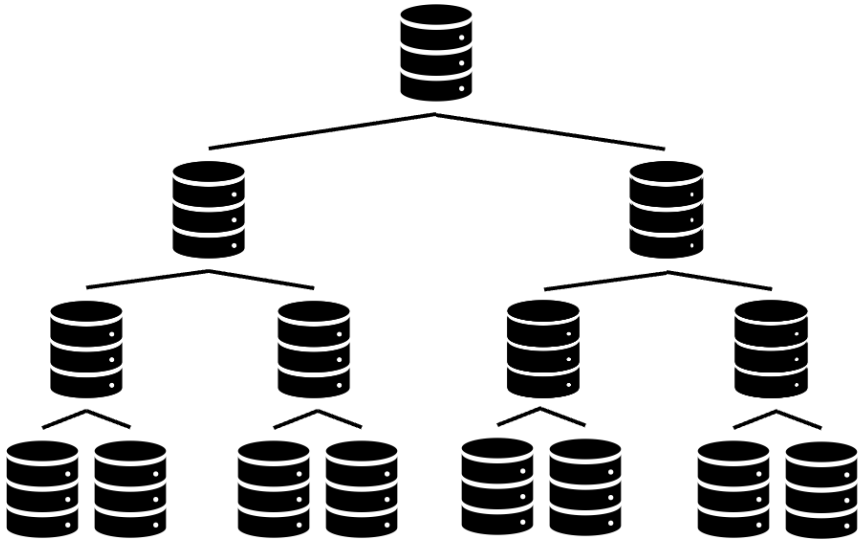
- ORAM was first introduced by Goldreich and Ostrovsky in 1987[4].
- Ideal ORAM design should achieve low computation overhead for client and low bandwidth blowup.
- Bandwidth blowup = the ratio between the communication cost of ORAM and the non-private case where the access pattern is not hidden
- Classical Model:
  - the server acts as a plain storage device to support only read and write operations.
  - Client does all the jobs to guarantee the desired security.
  - $\Omega(\log n)$ bandwidth blowup is inevitable in this model, where n= # data elements.
- Server computation model:
  - more realistic for the cloud storage scenario.
  - allows constant $O(1)$ bandwidth blowup via homomorphic encryption in 2016[5].

[4]Towards a theory of software protection and simulation by oblivious RAMs

[5]Onion ORAM: A constant bandwidth blowup oblivious RAM

ORAM based on (F)HE (Onion ORAM/Onion Ring ORAM[6])

Previous server aided efficient ORAM structure:



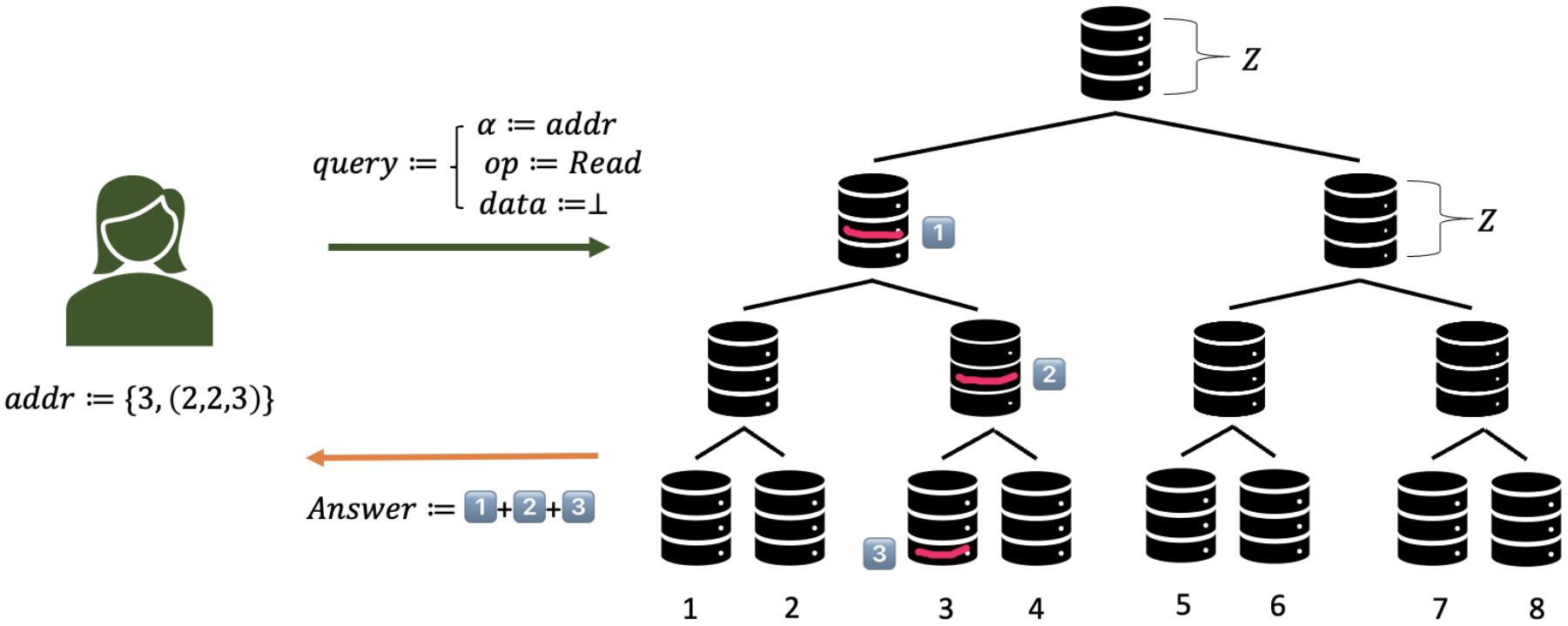
- n data elements are distributed in a tree of depth $\log n$.
- The client needs to maintain the state of the database locally.
- Server only touches a path which is indicated via client's query.
- logarithmic complexity in terms of both communication and computation

[6] Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE

ORAM based on (F)HE (Onion Ram/Onion Ring ORAM)

- Each node(bucket) consists of Z blocks.
- Each block is either an encrypted real data element or Enc(0) under an FHE scheme.
- Client knows where real elements are stored in the tree via setup phase.
- Client asks a path and a block per node on the path to be touched.
- Server homomorphically adds up all the requested $\log n$ blocks.

ORAM based on (F)HE (Onion Ram/Onion Ring ORAM)



When does "writing" happen?

- Server proceeds the same protocol as described in the previous slide.
- Server stores all the queries (specifically data part) at the root until it is fully filled (it can store up to Z queries)
- Once the root is full, server runs an interactive "eviction" protocol with the client to empty the root and move blocks towards the leaves via reshuffling all the blocks of buckets on a specified path.
- In other words, in every Z accesses, eviction should be run.
- During eviction, the requested update values are stored in the desired position.
- Eviction protocol is considered as "offline phase" since it does not depend on query.

Pros and cons of the previous approaches

- Pros:
  - Communication Complexity = $O(\log n)$, hence bandwidth blow-up = $O(1)$
  - (Online) computation complexity = $O(\log n)$; the actual running time is fast since it only consists of homomorphic addition (cheap).
- Cons:
  - Server's memory overhead: $O(Z \cdot n)$
  - Eviction is an interactive protocol, which requires multiple interactions between server and client (stateful client).
  - Eviction consists of heavy homomorphic operations to reshuffle blocks of $\log n$ buckets.
  - The homomorphic operation has hidden large constant factors $Z \log Z \approx 2^{13}$ in practice.
  - Client has to keep some parts of data to help Eviction, which requires somewhat large memory consumption.
  - Not suitable for fully outsourced storage.

Towards fully outsourced storage

What are the requirements for fully outsourced storage?

1. Non-interactive protocol.
2. Stateless client:
   - feel free to go offline at any point.
   - little client's effort in terms of memory and computation.
3. Low bandwidth blow-up.
4. Low latency of server.

Non-interactive and Stateless ORAM based on FHE[7]

$Query := (\alpha, op, data)$

$Answer := d$

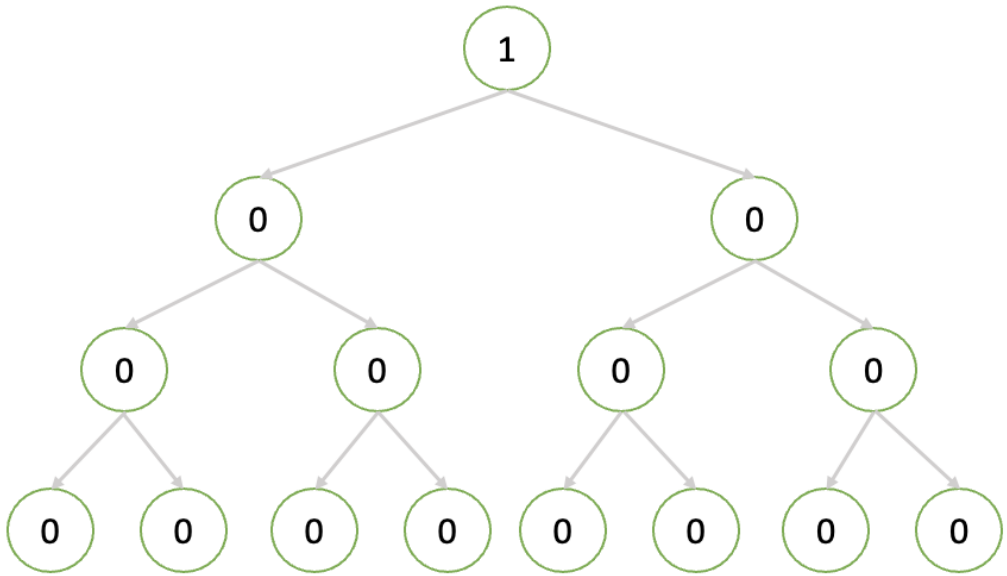
Index	data
1	D_1
2	D_2
⋮	⋮
n	D_n

- Query consists of the index of the data which a client wants to touch, an operation, and the update value.
- Since client does not give any help for computation of server (stateless), the server has to touch every element to give the correct answer.
- Communication = $O(\log n)$; asking bit representation of the index.
- Computation = $O(n)$.

[7] Panacea: Non-interactive and Stateless Oblivious RAM

Building Blocks of Panacea

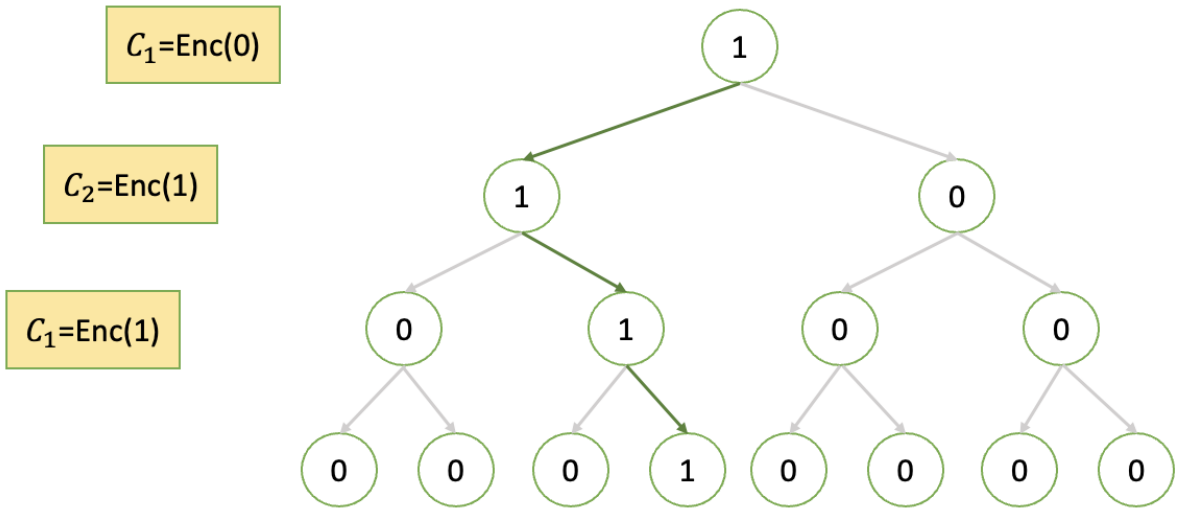
- $\mathrm{CMUX}(C, \mathrm{Enc}(a), \mathrm{Enc}(b)) := C \cdot (\mathrm{Enc}(b) - \mathrm{Enc}(a)) + \mathrm{Enc}(a)$
- Homomorphic De-multiplexer: It takes a (encrpyted) bit representation of a positive integer $a \in \{0, \ldots, n-1\}$, outputs an (encrypted) unit vector where the a-th component is Enc(1), Enc(0) elsewhere.
- We start with an empty tree except the root which has a value 1.
- After each computation on each level, the root value is homomorphically copied to the desired nodes.



Homomorphic De-multiplexer

Let the value of the parent node be b, and the value of child nodes is denoted by c. We need an extra information called a controller bit per level, denoted by C which is an encryption of a bit.

- Computation on the right child node:= $c := C \cdot b$
- Computation on the left child node:= $c := (1 - C) \cdot b$



- $a := 3$, encrypted as $(\text{Enc}(0), \text{Enc}(1), \text{Enc}(1))_2$; the 3rd element from the left (starting from 0-th).

Core design of Panacea

- Client:
  - sends a query $(\alpha, \text{op}, \text{data})$; α homomorphically indicates where he wants to touch, op is an encryption of bit, data is an encryption of an update value.
- Server:
  1. Response Phase
     - ★ runs homomorphic de-multiplexer to get an unit vector of dimension n which consists of encryption of 1 on the desired component.
     - ★ computes dot product between the unit vector and the data base vector.
     - ★ sends the result to the client.
  2. Update Phase
     for $j = 0, ..., n-1$
     - ★ runs CMUX$(\text{op}, d_j, data) =:$ temp, where d_j is the j-th original data
     - ★ updates $d_j :=$ CMUX(L_j, d_j, temp)

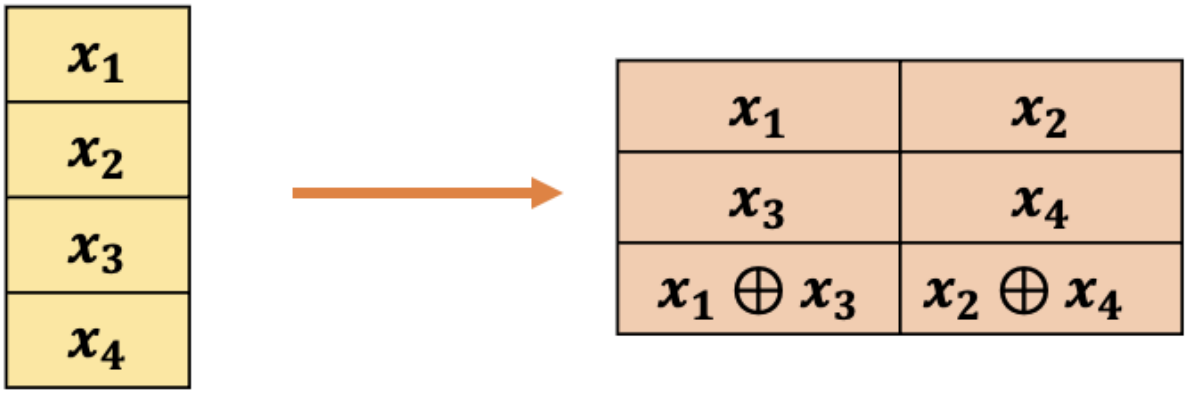
We note that the latency of the server only includes the response phase, update phase occurs after answering to the client.

Optimization for Latency

- Our computation complexity $O(n)$ is not desirable for real-world applications.
- Since our construction does not rely on any specific structure of the database, there are more ways to optimize our design than tree-based approaches.
- We can take advantage of parallel computation!
- We can also make use of batch code method to amortize the computation cost.

Probabilistic Batch Code (PBC)

- A probabilistic batch code, parameterized by the 5-tuple (n, m, k, b, p), takes as input a database with n elements, and outputs a set of m (possibly distinct) elements distributed among b buckets, such that any k elements from the original database can be retrieved by fetching at most one element from each of the b buckets with probability p.
- We want $m < kn$ to have any performance improvement for the server.
- We see the example (4,6,2,3,1) below[8]



[8] the example is from "SealPIR"

Database allocation

- To distribute n elements in b buckets, server uses ω hash functions: $H_1, \ldots, H_\omega$.
- Server structures a database as a simple hash table like below:
- group its items by hashed value, resulting in a $b \times B$ matrix like the below on the right (Note that $b \cdot B \approx m \leq k \cdot n$):

H_i \ DB	x_1	x_2	...	...	...	x_n
H_1	3	1	...	...	...	b
H_2	2	5	...	...	...	3
⋮			...	...	...	
H_ω	4	2	...	...	...	1

1	x_2	x_n	...	...	...
2	x_1	x_2	...	...	...
⋮			...	...	...
b	x_n	...		...	...

Client's batch query

- Client wants to fetch k elements by touching b buckets, each of which should be touched at most once.
- The problem can be looked at as a balls-and-bins problem. The client has to place k balls (the intended read/write operations) into b bins (the buckets in the database), where each bin can have at most one ball.
- Client uses Cuckoo Hashing technique to solve the problem.

Cuckoo Hashing

There are ω number of hash functions $\{H_i\}_{i\in[\omega]}$, where $H_i(\cdot) \in [b]$, for $b \in \mathbb{Z}^+$.

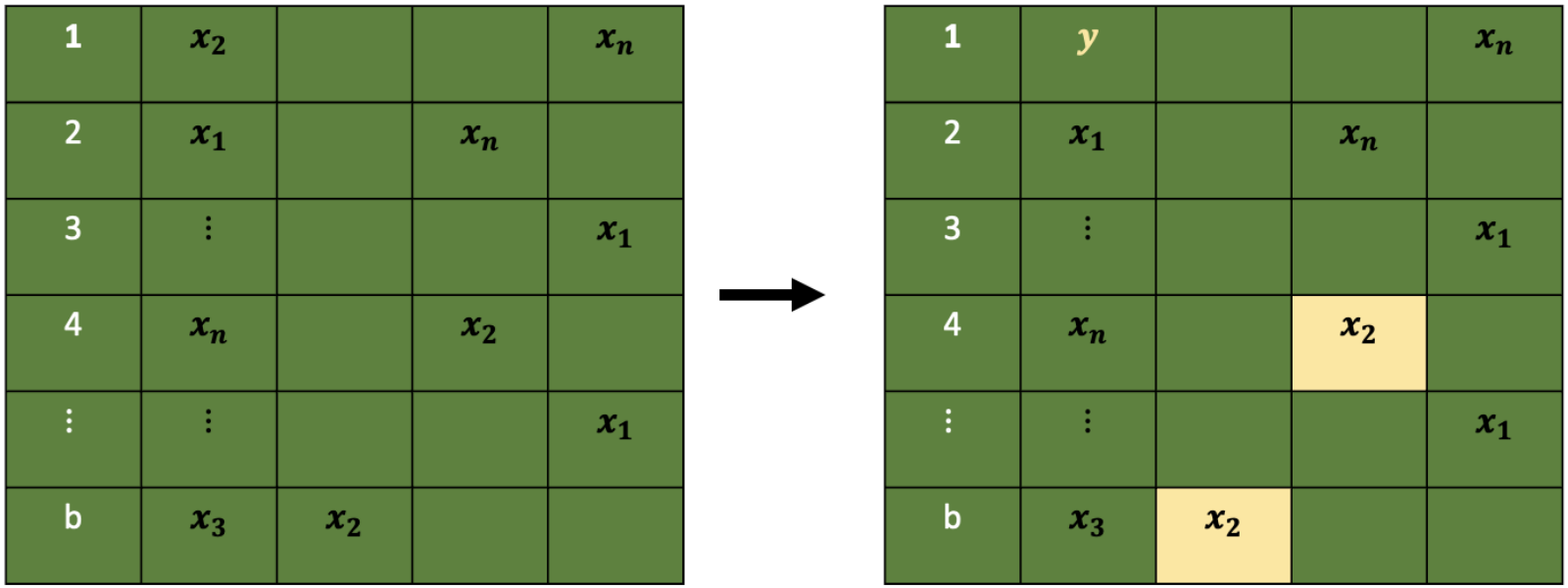
- There is a table of $b = \omega \cdot k$ slots.
- To insert an item x into the table, sample $i \leftarrow [\omega]$ at random, and insert (x, i) at location $H_i(x)$ if the slot is empty.
- If the slot is already occupied by (y, j), we replace (y, j) with (x, i), then choose $j' \leftarrow [h]\backslash\{j\}$, and recursively re-insert (y, j') into the table.

By doing so, you can insert all k items into b slots without collision.

Consistency Correction

- However, we cannot just use the same technique of PIR, directly since client can overwrite the existing file in ORAM.
- In other words, since the batching technique requires redundancy, all ω copies of data elements need to be updated with the same requested value for each write access.
- In order to avoid data inconsistency, there is another solution introduced, called consistency correction.

Consistency Correction



- The other two x_2 in 4th bucket and b-th bucket should be also updated to y.
- Note that everything is encrypted in server's database.

Consistency Correction

- Server knows where the redundancy per item occurs in the data table via his hash table, being oblivious to which items are encrypted.
- For simplicity, we set $\omega = 3$. Server knows the three tuples of indices $(1,1), (4,3), (b,2)$ which encrypts the same item.
- Let's make the three value on (1,1), $(4,3)$ and $(b,2)$ same with the value on those positions after update phase.
- Requirements:
  - the operation bits B_1, B_4, B_b per the three buckets
  - Unit vector L_1, L_4, L_b computed via homomorphic de-multiplexer.

$$V^{new} := V_{1,1} \cdot (1 - L_{1,1} \cdot B_1 - L_{4,3} \cdot B_4 - L_{b,2} \cdot B_b) + V_{1,1} \cdot L_{1,1} \cdot B_1 + V_{4,3} \cdot L_{4,3} \cdot B_4 + V_{b,2} \cdot L_{b,2} \cdot B_b.$$

- Since all the queries are distinct, no more or equal to two of $L_{i,j}$'s are encryptions of 1.
- $B_i = \mathrm{Enc}(0)$ if $\mathrm{op}_i = \mathrm{Read}$.

Overall complexity

Table: Comparison between our design and previous ORAM approach in terms of memory consumption (in bits) and computation of both server and client. We denote Z is the block size of each node in stateful design, n is the number of data elements.

	Panacea	Stateful ORAM
Memory (Server)	$O(w \cdot n \cdot N \cdot \log q)$	$O(Z \cdot n \cdot N \cdot \log q)$
Memory (Client)	$O(w \cdot n \cdot \log n)$	$O(n \cdot \log n + Z \cdot N \cdot \log q)$
Stateful	$\times$	$\checkmark$
Comp (server)	$O(n)$	$O(Z \cdot \log Z \cdot \log n)$
Comp (Query)	$O(k \cdot \log n)$	$O(k \cdot \log n)$
Comp (Eviction)	N/A	$O(Z \cdot \log Z \cdot \log n)$

Panacea: Non-interactive and Stateless ORAM

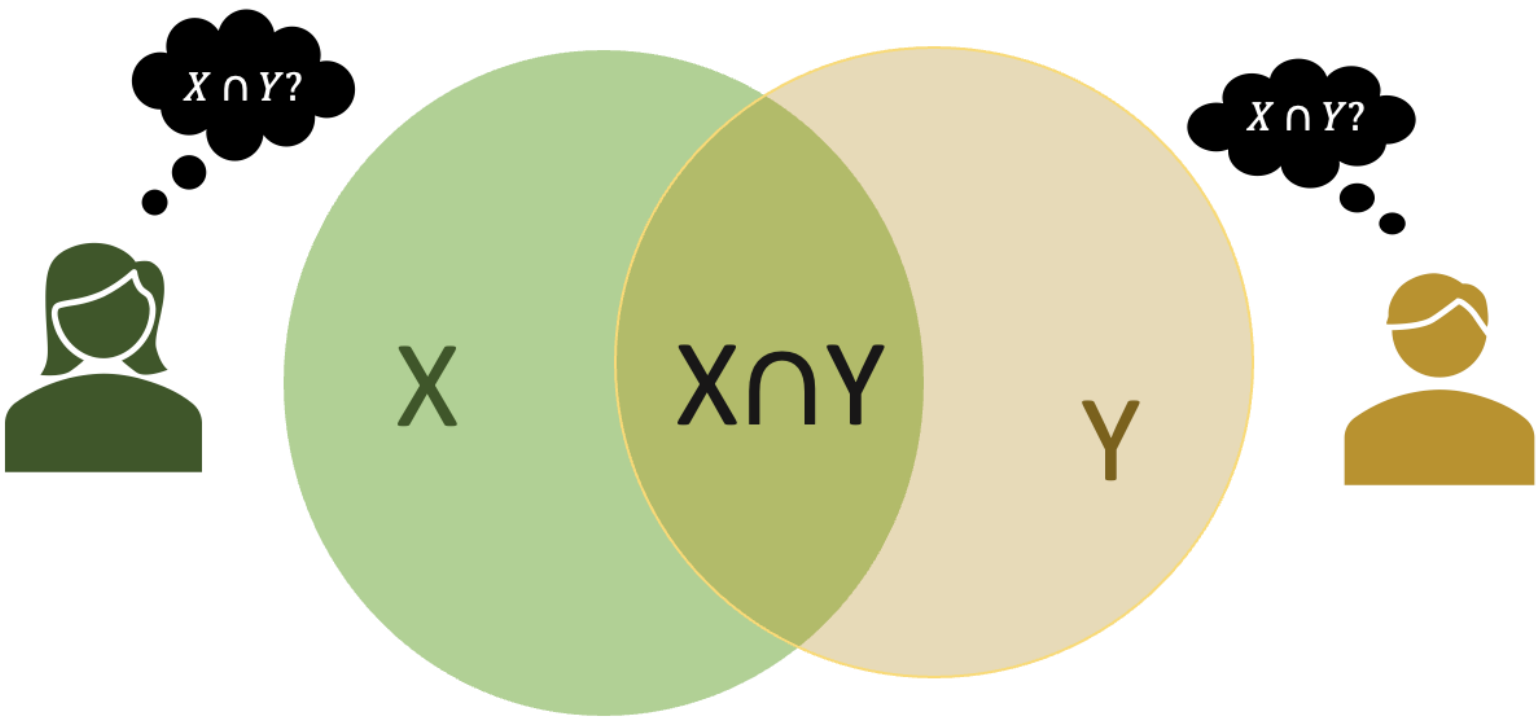
Table: Computation time required by the server for database size n, with n from 2^{12} to 2^{19}, for the size of the batch $k = 256$ with PBC. Numbers in brackets are amortized cost. All times are in seconds. # threads= 36

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)

Private Set Intersection

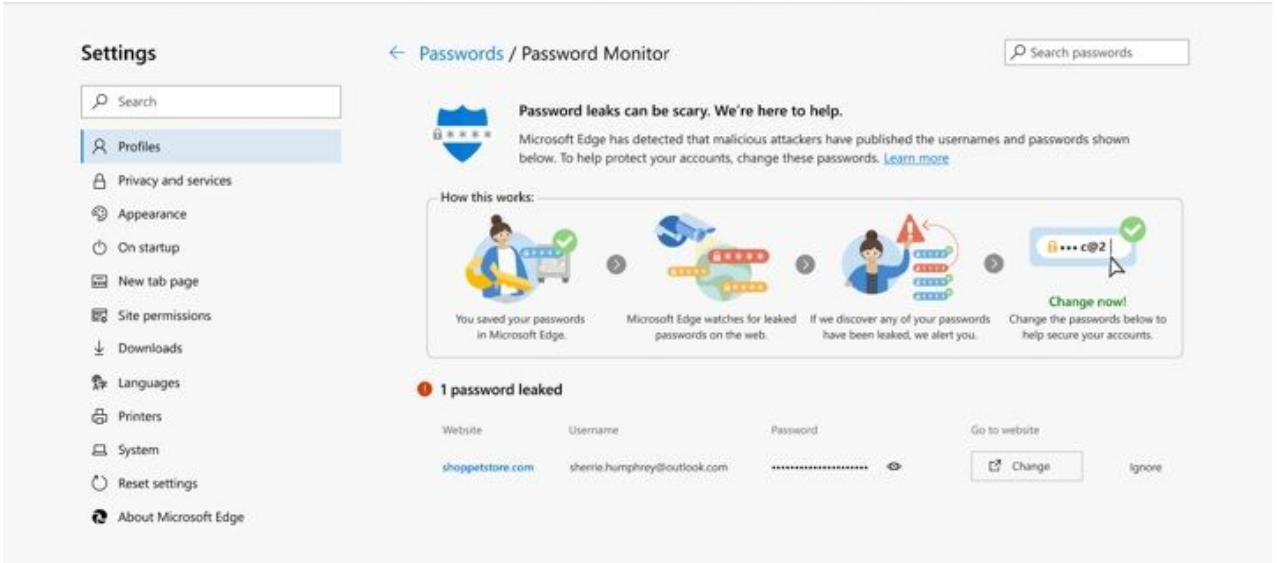
Private Set Intersection (PSI)

Private Set Intersection (PSI) is a two party protocol where both parties holding their own sets wish to learn the intersection of both sets keeping oblivious to the rest.



Applications of PSI

- **Private Contact Discovery**: Given a user's address book, find out which of their contacts also use the same messenger (such as WhatsApp) without revealing its information to the messenger's server.
- **Password Monitor**: Microsoft offers a service called Password Monitor for clients based on PSI via homomorphic encryption to check if their password stored in Microsoft have been found in a third-party breach.



PSI for unbalanced Sets

- The most common example is Private Contact Discovery.
- There are extremely fast protocols based on oblivious transfer (OT), which requires high communication complexity and multiple interactions between two parties.
- Communication is linear in both set sizes.
- Undesirable complexity for unbalanced sets
- We need something else for such scenario.

FHE based PSI[9]

- Non-interactive protocol.
- Asymptotically almost optimal complexity.
- Better performance when one set is much smaller than the other set.
- Computational overhead is still a bottleneck, but comparable to non-FHE based protocols due to many optimization techniques.

[9] Fast Private Set Intersection from Homomorphic Encryption

Naive approach

- Server's set $Y := (y_1, \ldots, y_{N_y})$, $|Y| = N_y$
- Client's set $X := (x_1, \ldots, x_{N_x})$, $|X| = N_x \ll N_y$
- Client sends all $\mathrm{Enc}(x_i)$'s to the server.
- Server computes the following for each x_i:

$$r_i \prod_{j=1}^{N_y} (\mathrm{Enc}_{\mathrm{sk}}(x_i) - y_j) = r_i \prod_{j=1}^{N_y} (\mathrm{Enc}_{\mathrm{sk}}(x_i - y_j)),$$

sk : client's secret key, and r_i : a random element sampled from a proper set.

- Communication: $O(N_x) \rightarrow$ it looks optimal!
- Computation: $O(N_x \cdot N_y) \rightarrow$ too much!

Optimization 1 : Batching

- There is a ring isomorphism between $\mathbb{Z}_t^N$ and $Z_t[X]/(X^N+1)$ for a prime $t \equiv 1 \mod 2N$ and N(a power of 2),

$$\begin{array}{ccc} \iota: & \mathbb{Z}_t^N & \longrightarrow & Z_t[X]/(X^N+1) \\ & \Psi & & \Psi \\ & (m_1, \ldots, m_N) & \longmapsto & T(X) \end{array}$$

- One ring addition/multiplication over $Z_t[X]/(X^N+1)$ gives us N parallel pointwise additions/multiplications over $\mathbb{Z}_t$.
- If a client splits its set elements into N_x/N chucks, and each chunk which contains N elements is encoded via the map ι and encrypt them,
  - the number of ciphertexts reduces down to $O(N_x/N)$
  - the computation complexity is improved to $O(N_x \cdot N_y/N)$.
  - if $N \approx N_x$, complexity becomes linear in N_y.

Optimization 2 : Hashing

- With only batching technique of client, server has to encode each y_j into individual polynomial, resulting in preparing N_y polynomials.
- Homomorphic operations between $\mathrm{Enc}(\mathrm{Encode}((x_1, x_2, \ldots, x_{N-1}, x_N)))$ and $\mathrm{Enc}(\mathrm{Encode}((y_j, y_j \ldots, y_j)))$
- In order to maximize the effect of batching technique, server also needs to structure data in an amenable format, letting him make use of batching as well.

Optimization 2 : Hashing

Server does the following:

1. prepare k hash functions $H_1, \ldots, H_k$, where $H_i(\cdot) \in \{1, \ldots, m\}$
2. generate a simple hash table like the figure below (on the left):
3. group its items by hashed value, resulting in a $m \times B$ matrix like the below on the right (Note that $k \cdot N_y \leq m \cdot B$):

H_i \ Y	y_1	y_2	...	...	...	y_{N_y}
H_1	1	3	...	...	...	m
H_2	2	5	...	...	...	3
⋮			...	...	...	
H_k	4	2	...	...	...	1

1	y_2	y_{N_y}	...	...	...
2	y_1	y_2	...	...	...
⋮			...	...	...
m	y_{N_y}	...	...	...	...

Optimization 2 : Hashing

Client does the following:

1. prepare the same k hash functions as the server's
2. use Cuckoo Hashing technique to distribute all N_x items in different buckets (See figure below on the left).

1	x_5
2	x_3
⋮	
m	x_1

1	y_2	y_{N_y}	⋯	⋯	⋯
2	y_1	y_2	⋯	⋯	⋯
⋮			⋯	⋯	⋯
m	y_{N_y}	⋯	⋯	⋯	⋯

3. Note that each item is assigned to its own (unique) bucket ($N_x \leq m$).

Optimization 2 : Hashing & Batching

- Client:
  1. parses the hash table into m/N vectors of dimension N
  2. encodes each vector individually.
  3. encrypts all, resulting in m/N ciphertexts.
- Server:
  1. parses each of B columns into m/N vectors of dimension N.
  2. encodes each vector individually, resulting in $B \cdot m/N$ polynomials.
  3. perform PSI.
- Complexity:
  - Communication: from $O(N_x/N)$ to $O(m/N)$, where $N_x \leq m$.
  - Computation: from $(N_x \cdot N_y/N)$ to $O(k \cdot N_y/N)$, where $k \ll N_x$ in practice $(k = 3)$.

Optimization 3: Windowing

Recall server's computation for each query $c := \mathsf{Enc}_{sk}(x)$, where $x \in X$:

$$r \prod_{j=1}^{N_y} (c - y_j)$$
$$= r \cdot c^{N_y} + r \cdot a_{N_y-1} \cdot c^{N_y-1} + \cdots + r \cdot a_1 \cdot c + r \cdot a_0$$

- Given only c, the multiplication depth is $\log(N_y + 1)$ to compute $r \cdot c^{N_y}$.
- If client sends all c^j for $j \in [N_y]$ to the server, then the multiplication depth is reduces to 1;
  computing dot product between $(c^{N_y}, c^{N_y-1}, \ldots, 1)$ and $(r, r \cdot a_{N_y}, \ldots, r \cdot a_0)$
- This results in huge communication blow-up; $O(N_y)$.

Optimization 3: Windowing

Let's strike a balance between computation and communication by introducing another parameter ℓ.

- Client sends $c_{i,j} := c^{i \cdot 2^{\ell \cdot j}}$, for $1 \leq i \leq 2^{\ell} - 1$ and $0 \leq j \leq \lfloor \log_2 N_y / \ell \rfloor$

i \ j	0	1	2	3	⋯
1	c	$c^{2^{\ell}}$	$c^{2^{2 \cdot \ell}}$	$c^{2^{3 \cdot \ell}}$	⋯
2	c^2	$c^{2 \cdot 2^{\ell}}$	$c^{2 \cdot 2^{2 \cdot \ell}}$	$c^{2 \cdot 2^{3 \cdot \ell}}$	⋯
3	c^3	$c^{3 \cdot 2^{\ell}}$	$c^{3 \cdot 2^{2 \cdot \ell}}$	⋱	
4	c^4	$c^{4 \cdot 2^{\ell}}$	⋱		⋱

Optimization 3: Windowing

Let's strike a balance between computation and communication by introducing another parameter ℓ.

- Client sends $c_{i,j} := c^{i \cdot 2^{\ell \cdot j}}$, for $1 \leq i \leq 2^{\ell} - 1$ and $0 \leq j \leq \lfloor \log_2 N_y / \ell \rfloor$

i \ j	0	1	2	3	⋯
1	c	$c^{2^{\ell}}$	$c^{2^{2 \cdot \ell}}$	$c^{2^{3 \cdot \ell}}$	⋯
2	c^2	$c^{2 \cdot 2^{\ell}}$	$c^{2 \cdot 2^{2 \cdot \ell}}$	$c^{2 \cdot 2^{3 \cdot \ell}}$	⋯
3	c^3	$c^{3 \cdot 2^{\ell}}$	$c^{3 \cdot 2^{2 \cdot \ell}}$	⋱	
4	c^4	$c^{4 \cdot 2^{\ell}}$	⋱		⋱

Optimization 3: Windowing

- Communication increases; $N_x \cdot (2^\ell - 1) \cdot \lfloor \log_2 N_y/\ell \rfloor$ from N_x
- The maximum number of multiplication is $\lfloor \log_2 N_y/\ell \rfloor + 1$.
- Hence, the multiplication depth is reduced down to $O(\log_2(\lfloor \log_2 N_y/\ell \rfloor + 1))$ from $O(\log_2 N_y + 1)$

Putting it all together (with windowing, batching and hashing)

- Communication:
  - $m/N \cdot (2^{\ell} - 1) \cdot \lfloor \log_2 B/\ell \rfloor$, where $N_x \leq m$ and $B \geq \frac{N_y \cdot k}{m}$
- Multiplication depth:
  - $O(\log_2(\lfloor \log_2 B/\ell \rfloor + 1))$
- Complexity:
  - Communication: $O(N_x \cdot \log N_y)$
  - Computation: $O(N_y)$

Applications to Privacy Preserving Machine Learning

Machine learning as a cloud-based service

- Stock-price prediction
- Spam-filtering
- Recommender system

Can we design a privacy preserving machine learning algorithm which satisfies:

- optimal communication/computation complexity
- less communication cost
- less computation cost
- practical implementation result?

There are already many attempts..

How? Improving the efficiency of their bulding blocks;

- Privacy preserving convolution neural network[10]
  - evaluting sigmoid functions homomorphically
  - evaluating max function homomorphically
  - optimal way of matrix encoding
- Privacy preserving k-nn algorithm[11]
  - evaluating Euclidean distance between two vectors homomorphically
- Private decision tree evaluation (PDTE)[12]
  - evaluating comparison function homomorphically
  - applying transciphering to reduce communication cost.

[10] Applying Neural Networks to Encrypted Data with High Throughput and Accuracy,Secure human action recognition by encrypted neural network inference

[11] Efficient homomorphic evaluation of k-NN classifiers

[12] SortingHat: Efficient Private Decision Tree Evaluation via Homomorphic Encryption and Transciphering

Are they practical enough?

- Their communication overhead is still high (due to ciphertext expansion).
- Inference phase: evaluation is done in few seconds[13].
- SIMD operation, hardware acceleration boosts the efficiency.
- Even though their building blocks themselves are optimized, there are still some ways to be "more" optimized, depending on application scenarios.

[13] Secure human action recognition by encrypted neural network inference

Machine learning as a service

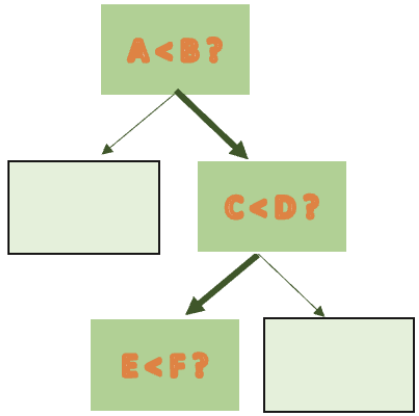
- Scenario: a server holds a machine learning model and a client wants to evaluate the prediction/inference using the server's model without revealing its data to the server.
- Server knows the model !
- Do we really need ciphertext-ciphertext operation then?
- We study the most recent technique which takes all the advantage of server which knows "machine learning model" in clear-text.

SortingHat: efficient Private Decision Tree Evaluation via Homomorphic Encryption and Transciphering

- Computationally efficient Private Decision Tree Evaluation (PDTE) called SortingHat via Homomorphic Encryption
  - Designed an efficient homomorphic comparison algorithm where one operand is a plaintext.
  - Designed a homomorphic traversal algorithm.
- Bandwidth efficient PDTE called t-SortingHat via Homomorphic Encryption and Transciphering
  - Thanks to transciphering, communication cost is 20,000 times less than previous approach.
  - Improved an existing transciphering technique (x400 faster)

Private Decision Tree Evaluation

- An important class of classifiers in machine learning.
- Each decision node : comparison between a threshold value owned by a server and its assigned attributes given by a client
- The output of the comparison function (either 0 or 1) denotes which child node to travel.
- The traversal outputs a vector where the only one component corresponding to the classification label is an encryption of 1, the rest are encryptions of 0.



Building blocks: i) Homomorphic Comparison

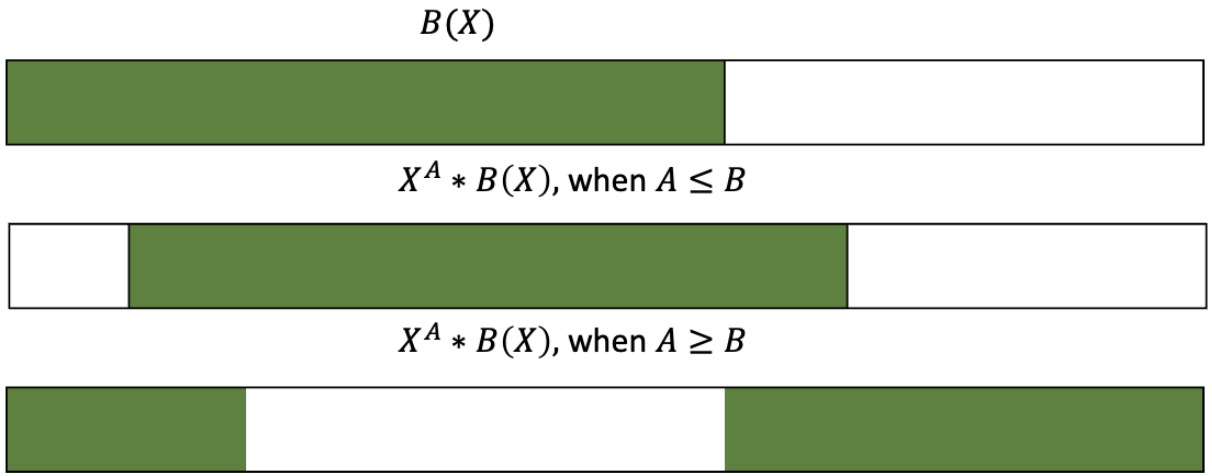
- Hom.Comp(Enc(A), Enc(B))=Enc(1) if $A \geq B$, $Enc(0)$ otherwise.
  - Threshold values are parts of classification model which a server knows.
  - Ciphertext-plaintext operation is possible (much cheaper than ciphertext-ciphertext operation used in previous works)
- In other words,
  we need Hom.Comp(Enc(A), B), where B is a plaintext.

Hom.Comp(Enc(A), B)

- Define $R_q := \mathbb{Z}_q[x]/(X^N + 1)$ and $X^N = -1$.
- If A is encoded as $A(X) := X^A$, and $B(X) = X^{2N-B}$, where $0 \leq A = B \leq N$.
  $A(X) \cdot B(X) = X^A \cdot X^{2N-B} = X^{A+2N-B} = X^{2N} = (X^N)^2 = (-1)^2 = 1$.
- Likewise, we encode B as a polynomial like the following:

$$X^{2N-B} + X^{2N-(B+1)} + \ldots X^{2N-N}$$

- The constant term of $A(X) \cdot B(X)$ will be 1 if $A \geq B$, 0 otherwise.
- $\text{Enc}(A(X)) \cdot B(X) = \text{Enc}(A(X) \cdot B(X))$; it only needs two polynomial multiplications to compare A and B!



Hom.Comp(Enc(A), B)

- If $A = \sum_i \epsilon_i \cdot 2^i$, $\forall \epsilon_i \in \{0, 1\}$ is encrypted as bit-by-bit like $\{\mathrm{Enc}(\epsilon_i)\}$, we tweak the bitwise comparison method.
- Bitwise comparison starts from the most significant bit, gradually proceeding towards lower significant bits until an inequality is found.
- $A = (A_3 A_2 A_1 A_0)_2$, and $B = (B_3 B_2 B_1 B_0)_2$
- $x_i := XNOR(A_i, B_i) = 1$ when $A_i = B_i$, 0 otherwise.
- $\bar{B}_i := NOT(B_i)$.
- $\mathrm{Comp}(A, B) := A_3\bar{B}_3 + x_3 A_2 \bar{B}_2 + x_3 x_2 A_1 \bar{B}_1 + x_3 x_2 x_1 A_0 \bar{B}_0$

Hom.Comp(Enc(A), B)

- Hom.XNOR(Enc(A_i), B_i) := Enc(A_i) if $B_i = 1$,
  $1 -$ Enc(A_i)($=$ Enc($1 - A_i$)) otherwise.
- Now we have three values A, B, C, and compute

  1) Hom.Comp(Enc(A), B)

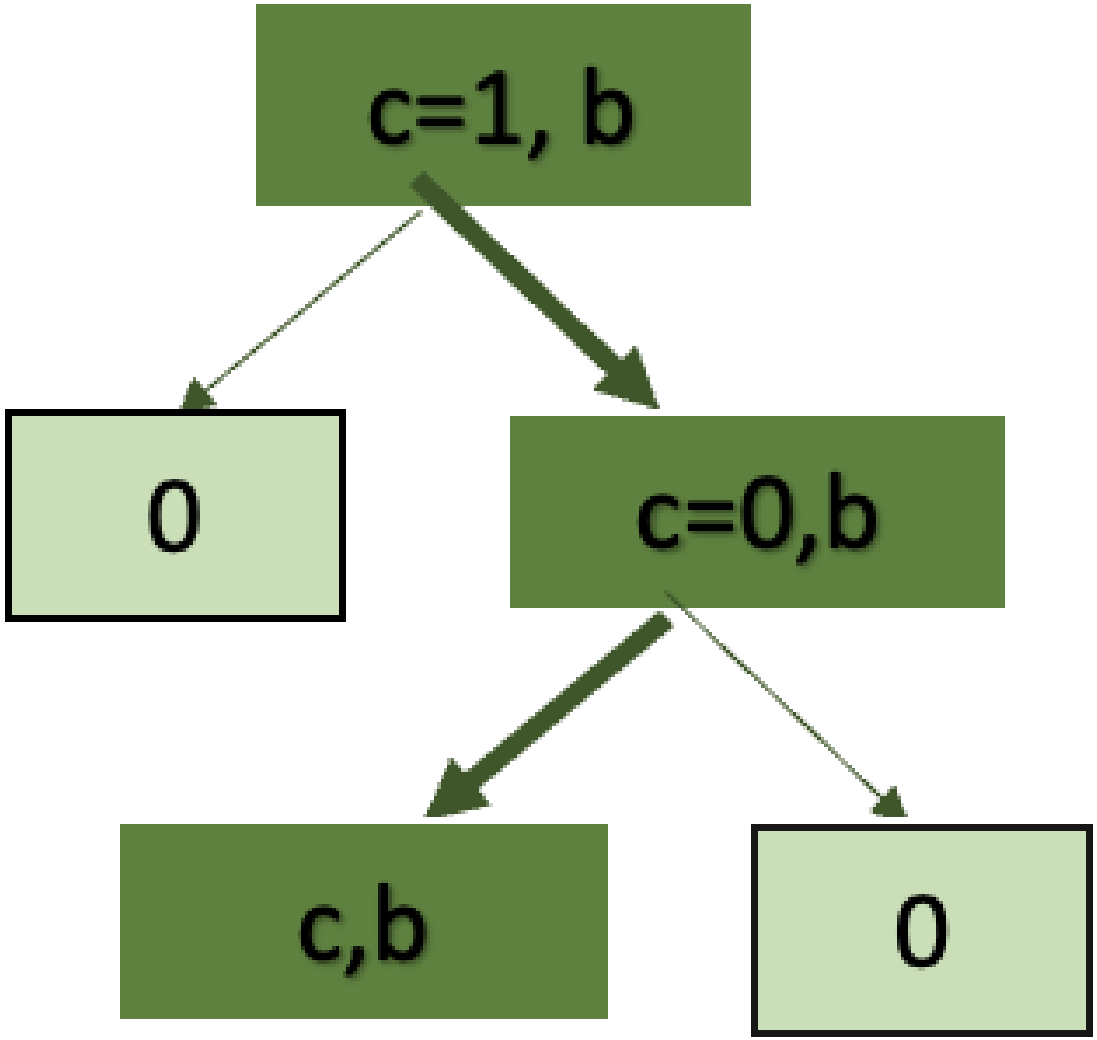
  2) Hom.Comp(Enc(A), C), where $B = 1001_2$ and $C = 1010_2$.

- After 1), we already know x_3, x_2, $A_3\bar{B}_3$, and $A_2\bar{B}_2$.
- We can reuse them for 2)
- This is possible because B and C are plaintext.

Building blocks: ii) Homomorphic Traversal

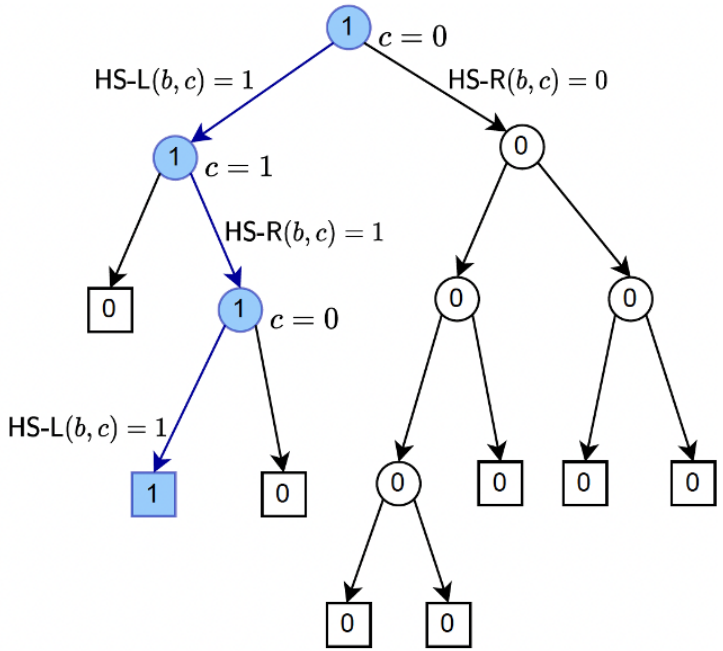
- The goal of this algorithm to copy the value of the root to the desired leaf of a tree.
- We already studied similar one in ORAM section, called homomorphic de-multiplexer.
- Each decision node contains a precomputed function value $c \in \{0, 1\}$ called controller bit, and an empty slot.
- The root value (b) is copied to the empty slot of the right child if root's $c = 1$, otherwise the slot remains empty.
- We start at the root and repeat the process untill the leaf nodes are reached.
- The only one leaf is encryption of b and the rest are encryptions of 0

Building blocks: ii) Homomorphic Traversal



SortingHat

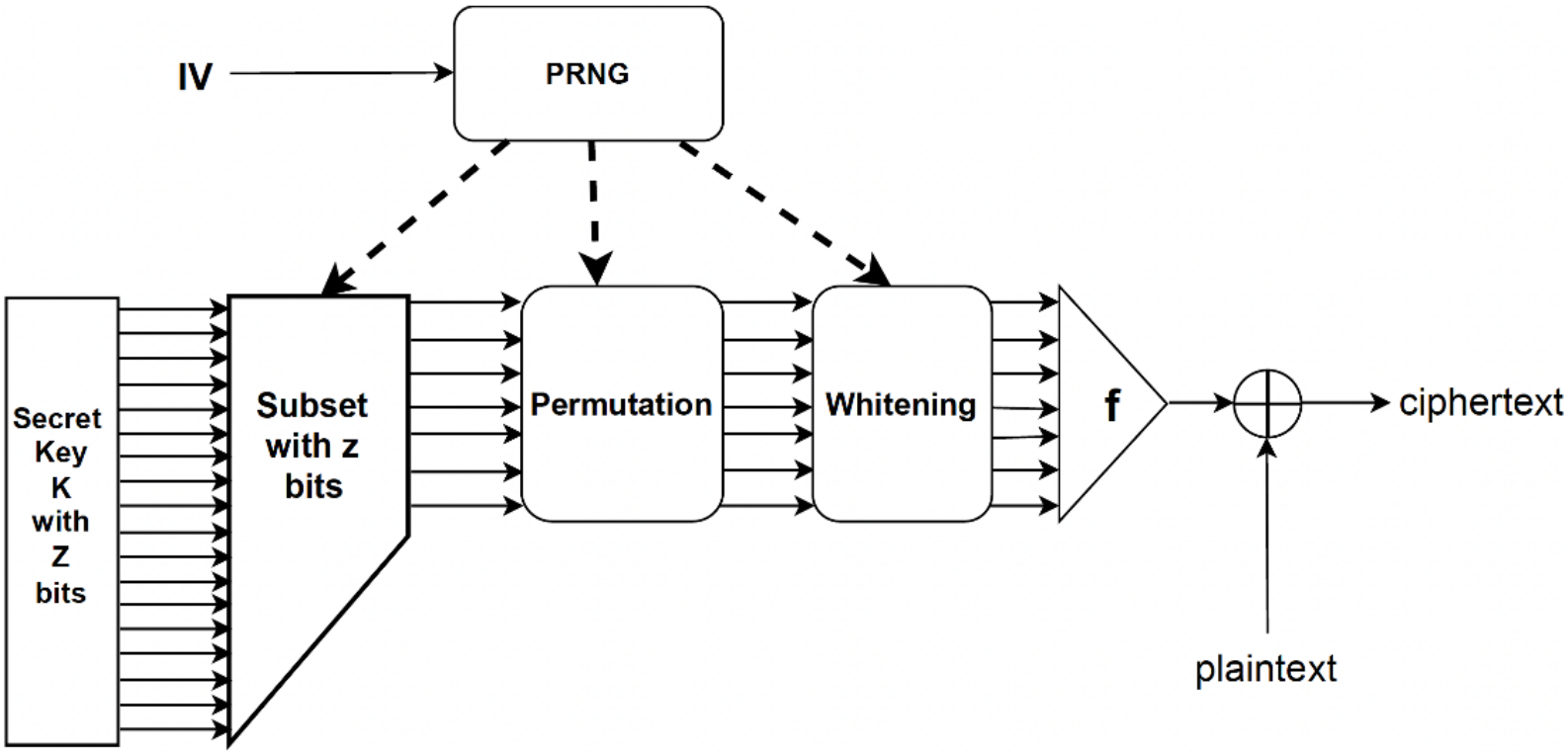
- Client: sends encryptions of its attribute vector $x := (x_1, x_2 \ldots, x_n)$
- Server:
  - Initialize all the node values to 0 except the root which is 1.
  - Run Hom.Comp$(x_i, d_i) := c_i$ at each decision node.
  - Run HomTrav$(c_i, 1) \rightarrow \vec{L}$.
  - Compute $V := \langle \vec{L}, \vec{\tau} \rangle$, where $\vec{\tau}$ is a vector consisting of the corresponding classification labels.
  - Send V to the client.



Transciphering

- FHE ciphertexts are too big !
- Transciphering:
  - symmetric key ciphertexts $\longrightarrow$ FHE ciphertexts via homomorphic decryption of the symmetric key scheme.
- Computational overhead of transciphering?
- We consider FiLIP cipher, a stream cipher specifically designed to be evaluated with the underlying FHE scheme.

FiLIP cipher



FiLIP cipher

- function $f := \mathrm{XTHR}_{k,d,s}$ is defined as

$$\mathrm{XTHR}_{k,d,s}(z) = \mathrm{XOR}_k(x_1, \ldots, x_k) + \mathrm{T}_{d,s}(y_1, \ldots, y_s) \in \mathbb{F}_2,$$

where $\mathrm{XOR}_k(x_1, \ldots, x_k) = x_1 + \cdots + x_k \in \mathbb{F}_2$.

  - $\mathrm{T}_{d,s}$ is defined as:

$$\forall y = (y_1, \ldots, y_s) \in \mathbb{F}_2^s,\ \mathrm{T}_{d,s}(y) = \begin{cases} 1 & \text{if } \mathrm{W_H}(y) \geq d, \\ 0, & \text{otherwise} \end{cases}$$

where $\mathrm{W_H}(y)$ is the Hamming weight of a binary vector y.

- Once we have $\mathrm{Enc}(\mathrm{W_H}(y))$, we can run $\mathrm{Hom.Comp}(\mathrm{Enc}(\mathrm{W_H}(y)), d)$, since d is already known to server, as a public protocol value.

Improvement on Transciphering

Table: Comparison between the communication cost of the setup phase and the running times of the homomorphic FiLIP decryption.

	[HMR20][14]	This approach	Improvement
Setup	800 MB	200 MB	$\times 4$
Timing	1018 ms	2.62 ms	$\times 388$

[14] Transciphering, Using FiLIP and TFHE for an Efficient Delegation of Computation

Implementation Result of SortingHat

Table: PDTE results for various datasets. The time is given in milliseconds. τ is the number of threads.

					[TBK20][15]	[LZS18][16]	SortingHat	
dataset	ID	d	m	n	$\tau = 16$	$\tau = 16$	$\tau = 1$	$\tau = 6$
heart	1565	3	5	13	940	590	42.3	10.5
breast	1510	7	17	30	-	-	154	34.8
steel	1504	5	6	33	-	-	51.9	12.3
housing*	N/A	13	92	13	6300	10270	892	190
spam	44	16	58	57	3660	6880	553	115
artificial*	N/A	10	500	16	22390	56370	4787	1045

- d = the depth of a tree, m = # decision node, n= # attributes.
- The input of client is encoded in the exponent of X like X^A. so that we can use the fastest comparison algorithm.

[15] Non-interactive private decision tree evaluation

[16] Non-interactive and output expressive private comparison from homomorphic encryption

t-SortingHat

- The communication cost is about $2 \cdot 10^4$ times lower via transciphering

Table: The timing is listed in seconds.

dataset	[TBK20] ($\tau = 16$)	Naive ($\tau = 1$)	Recursive ($\tau = 1$)
heart	0.94	1.51	1.52
housing*	6.3	30.18	28.60
spam	3.66	20.3	21.49
artificial*	22.39	145.9	92.44

- Transciphering outputs $\{Enc(\epsilon_i)\}$ where $A := \sum_i \epsilon_i \cdot 2^i$.
- Then we use plaintext-ciphertext version of bitwise-comparison.
- When the number of nodes are much higher than the number of features (the artificial dataset), then the comparison technique outperforms the naive approach.

Thank you for your attention