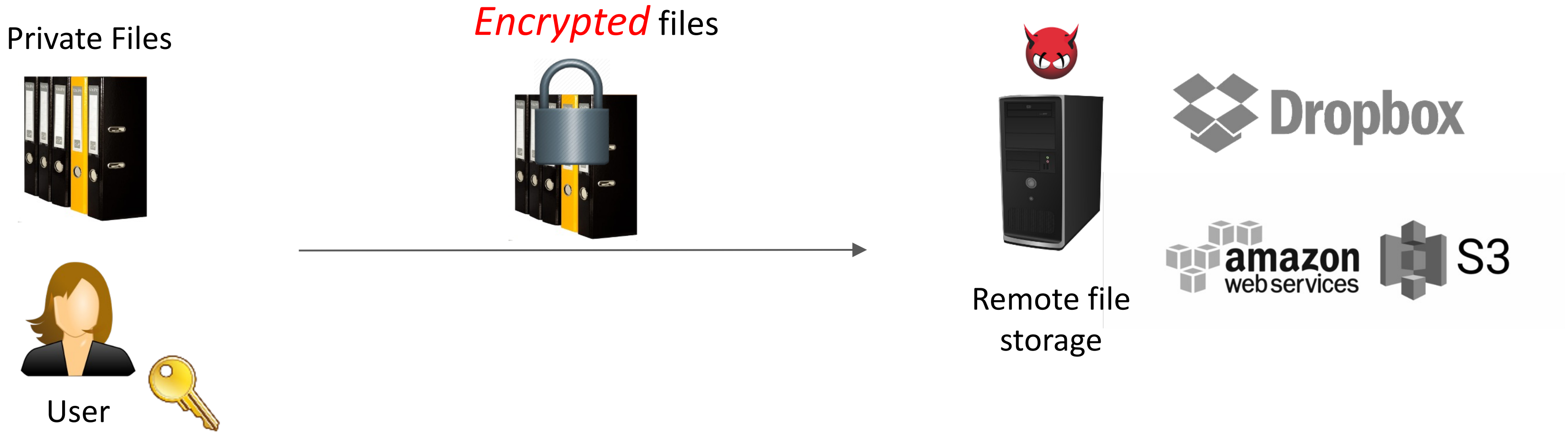
Private Information Retrieval

Sujaya Maiyya
Slides partially acquired from Ishtiyaque Ahmed

The problem of protecting *private data repositories* stored remotely is well-studied



Encryption hides file contents from an attacker.

ORAM (STOC ‘87) hides data access patterns for private files

Private Files

User

Oblivious RAM

(Goldreich STOC ‘87, Path ORAM JACM ‘18, SCORAM CCS ‘14, …)

Encryption +
randomized data accesses

Remote file storage

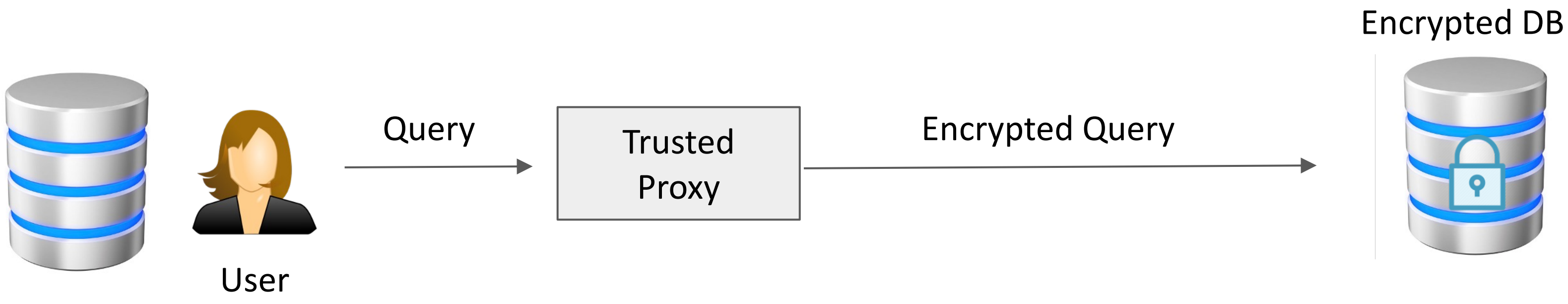
Hidden:

- ➔ Which file is being accessed?
- ➔ Whether the access is a read or write
- ➔ When was the file accessed last

…

We can extend protection to *private relational databases* stored remotely

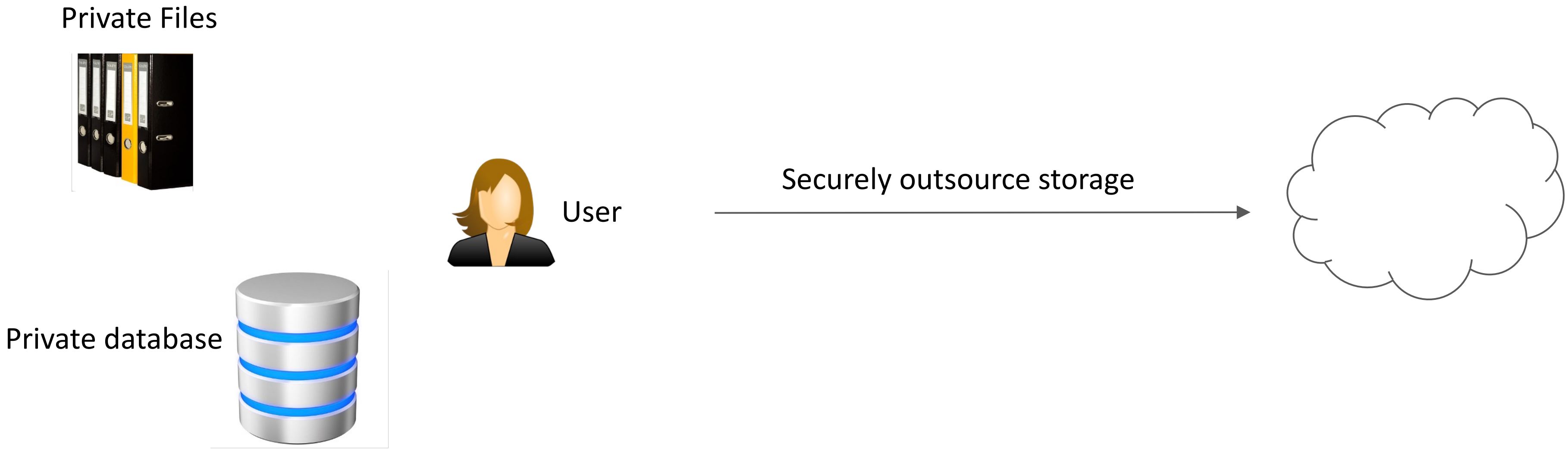
CryptDB, Arx, ObliDB, SMCQL …



Hidden:

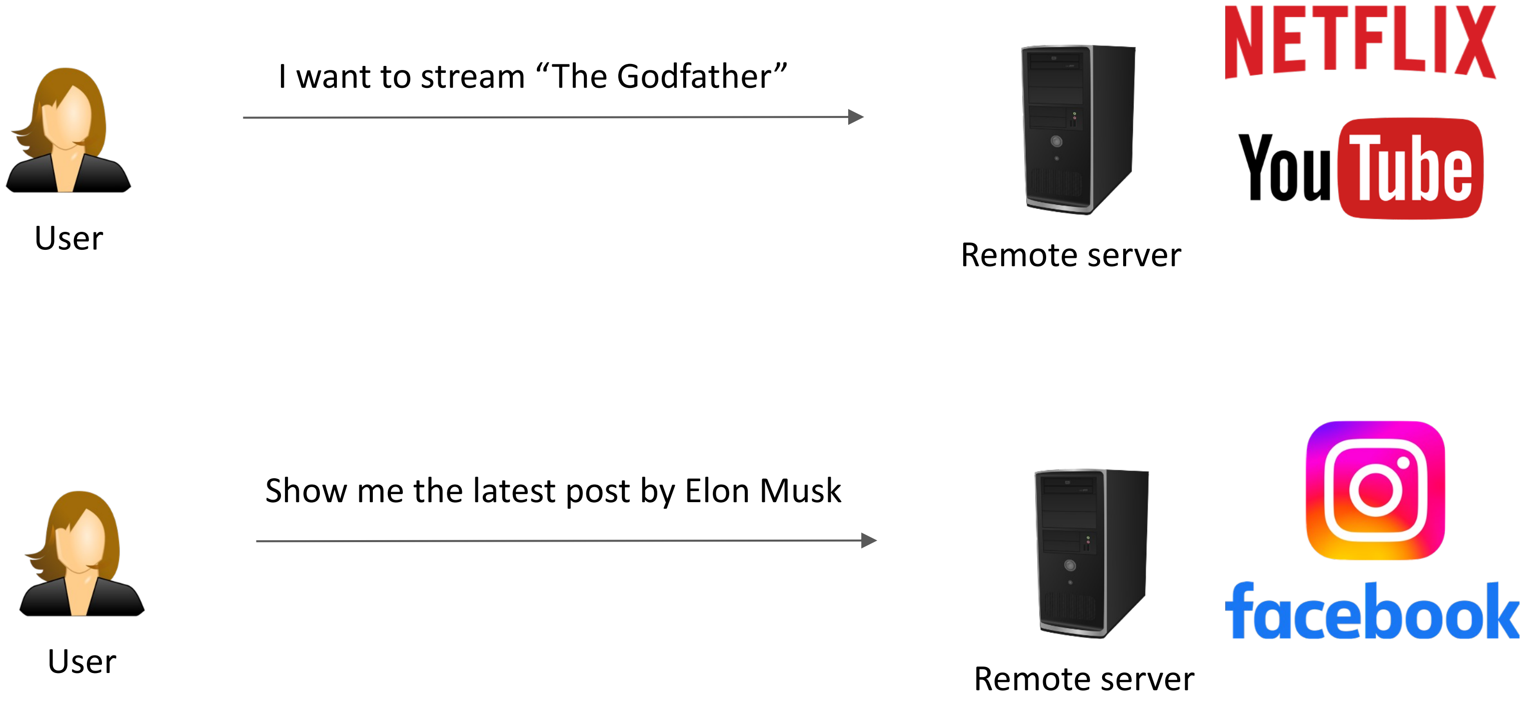
- ➔ Database content
- ➔ Query parameters

What is common to all of these cases?

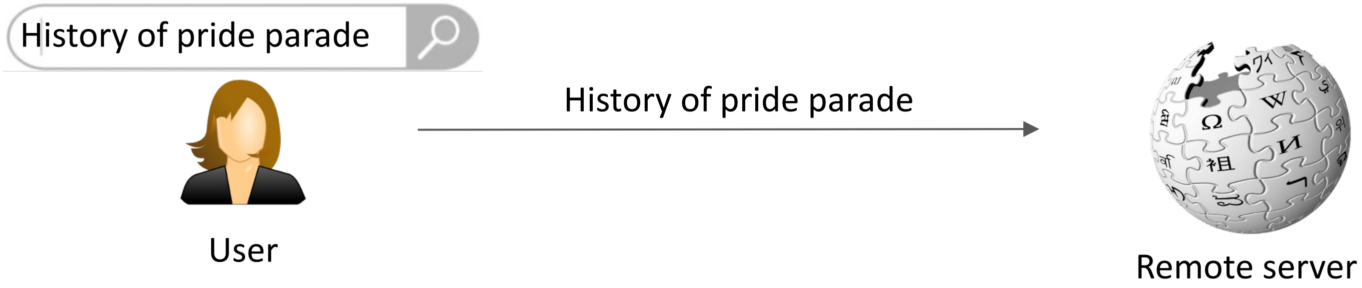


The user owns the data!

But, much of the content on the Internet is in ***public*** *data repositories*



But, much of the content on the Internet is in ***public data repositories***

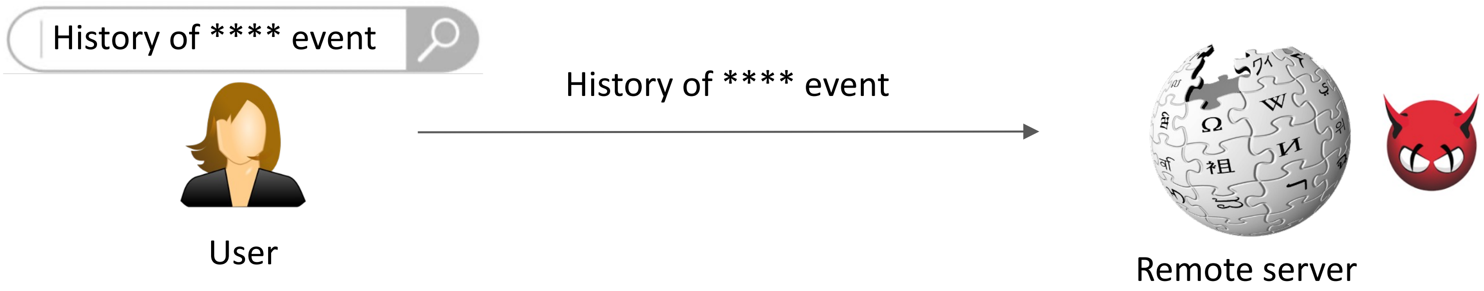


Cannot use:

- Encryption
- ORAM
- CryptDB-like solution

How can we hide access patterns (queries) over public data repositories?

Both users and service providers want to hide access patterns over public repositories



User may:

- Consider queries private
- Belong to a vulnerable population or a minority group

Server can be:

- Hacked by an outsider
- Compromised by an insider
- Coerced by a nation state [1, 2]

1. Brian Fung. *Analysis: There is now some public evidence that China viewed TikTok data.* CNN, 2023.
2. Sapna Maheshwari and Ryan Mac. *Driver's Licenses, Addresses, Photos: Inside How TikTok Shares User Data.* New York Times, 2023

This lecture: Private information retrieval (PIR)

Discuss a cryptographic method to privately retrieve data from public data repositories, thus making server *opaque* to data access patterns

Private retrieval from public databases can be abstracted into the key-value store model

k

Client retrieves:

- v, if (k,v) at Server
- ∅, otherwise

k_0	v_0
k_1	v_1
k_2	v_2
…	…
k_{n-1}	v_{n-1}

Untrusted Server

Two types of PIR

- Computationally secure – CPIR
- Information-theoretically secure – IT-PIR

We will discuss two types of CPIR

Part 1: Retrieval by location

key	location
k_0	0
k_1	1
…	…
k_{n-1}	n-1

Has (key, location) mapping

Give me the *i*-th value →

0	v_0
1	v_1
2	v_2
…	…
n-1	v_{n-1}

Untrusted Server

Part 1: How can the client privately retrieve the value corresponding to a given location?

We will discuss two types of CPIR

Part 2: Retrieval by key

key	location
k_0	0
k_1	1
…	…
k_{n-1}	n-1

k

Client retrieves:

- v, if (k,v) at Server
- Ø, otherwise

Give me value for key k

k_0	v_0
k_1	v_1
k_2	v_2
…	…
k_{n-1}	v_{n-1}

Untrusted Server

Part 2: How can the client privately retrieve the value corresponding to a given key?

Part 1: Retrieval by location

key	location
k_0	0
k_1	1
…	…
k_{n-1}	n-1

Has (key, location) mapping

Give me the *i*-th value →

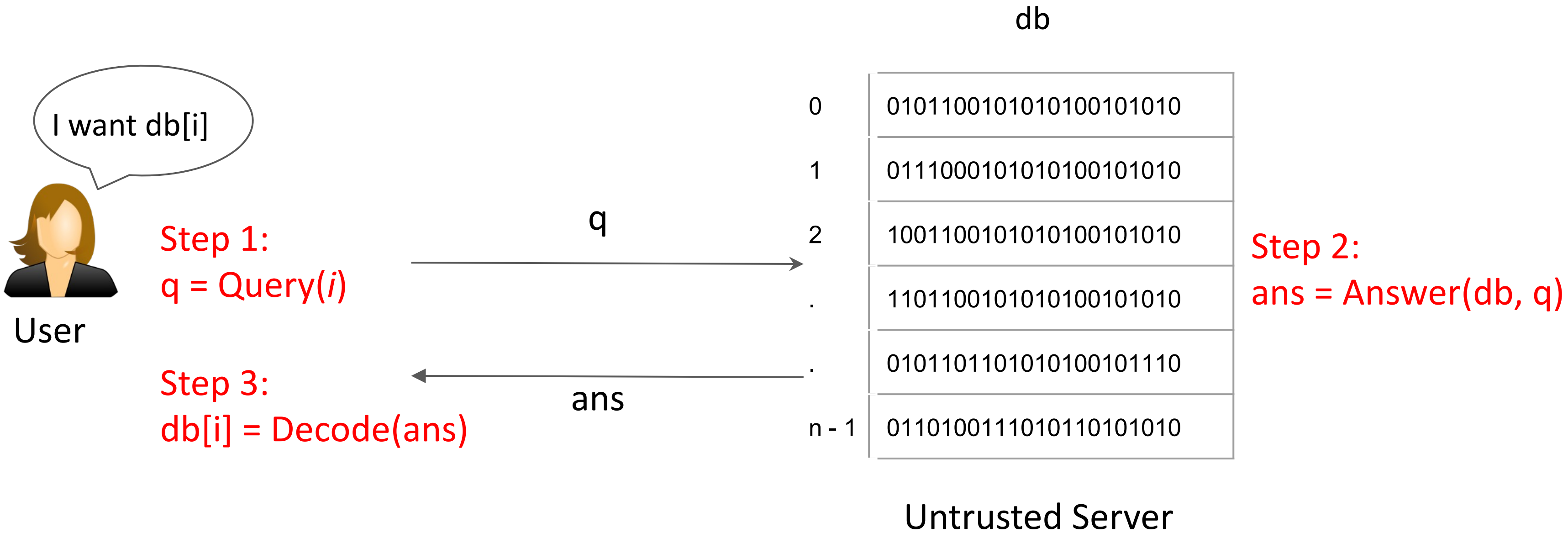
0	v_0
1	v_1
2	v_2
…	…
n-1	v_{n-1}

Untrusted Server

Part 1: How can the client privately retrieve the value corresponding to a given location?

This problem can be solved using **Private Information Retrieval** (PIR) (Chor et al. FOCS '95)

PIR: Query, Answer, Decode



PIR has two key requirements

Correctness

Query for db[i] returns db[i] to the user

Decode(Answer(db, Query(*i*))) = db[i]

Privacy

Server learns “nothing” about the location i

For all locations i, j,

{View of the server in answering Query(i)} ≈

{View of the server in answering Query(j)}

One solution to private information retrieval in *Trivial PIR*

q = Give me the entire db

ans = db

db

0	0101100101010100101010
1	0111000101010100101010
2	1001100101010100101010
.	1101100101010100101010
.	0101101101010100101110
n - 1	0110100111010110101010

Untrusted Server

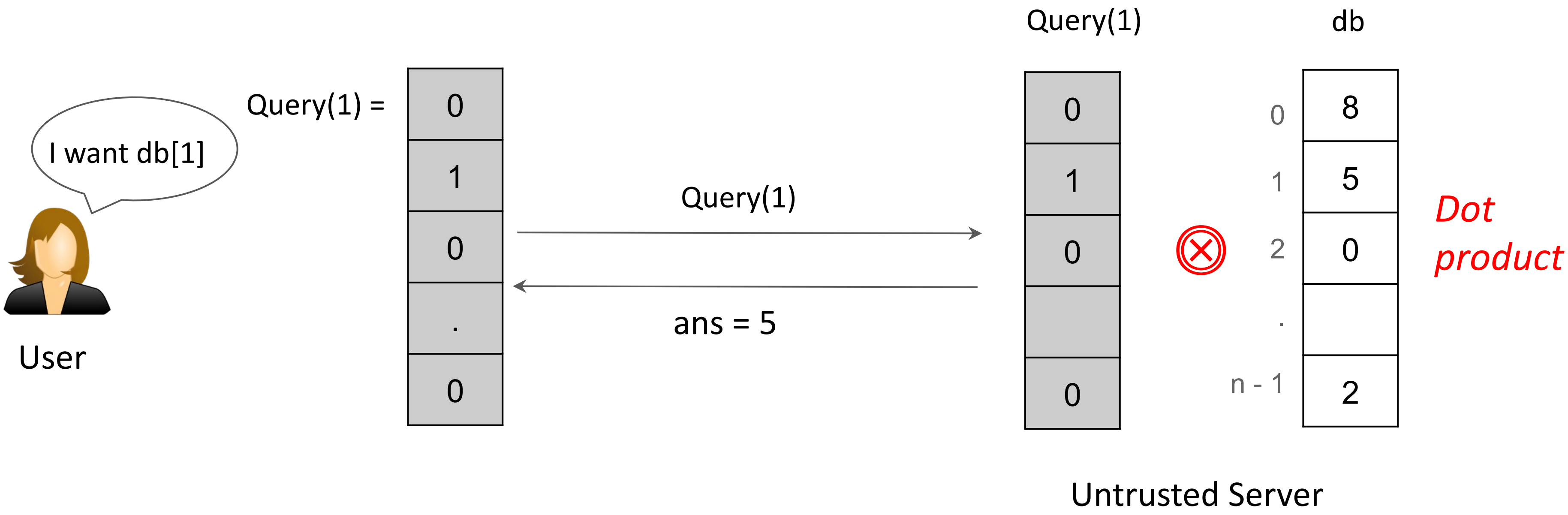
Query(i): A single bit

Answer(db, q): db

Decode(i, ans): select the i-th item from ans

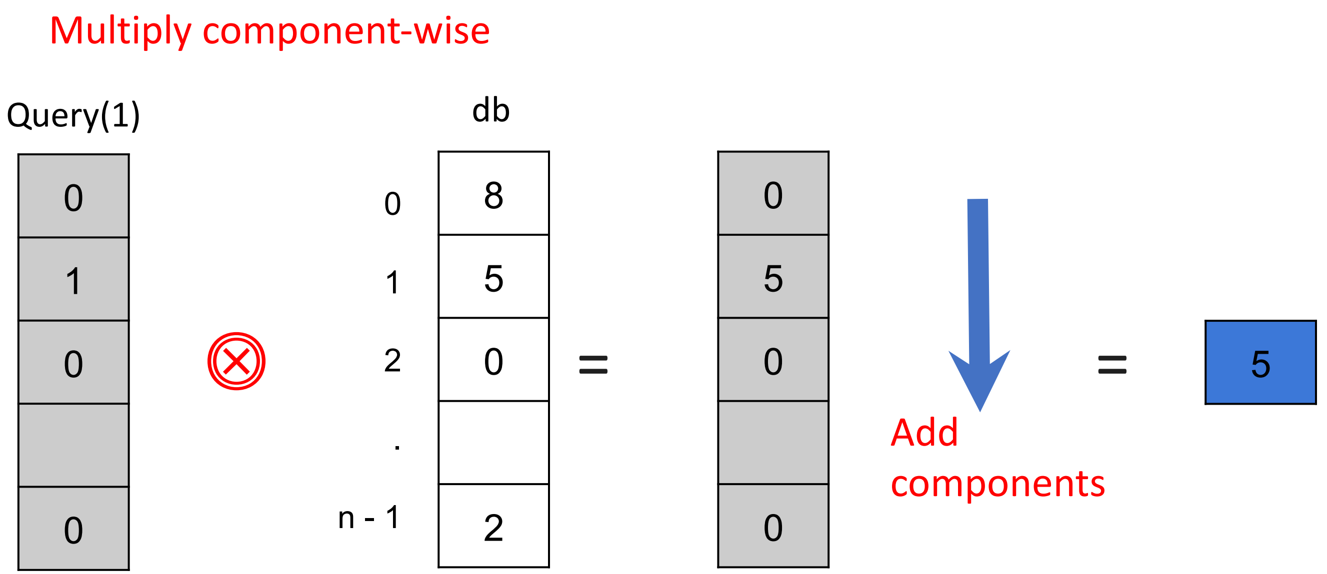
Warmup for (non-trivial) PIR

Assume that we do not care about privacy yet; only correctness



Retrieval is equivalent to computing a dot product

Warmup for (non-trivial) PIR in more detail



Dot product requires two types of operations:

- ➔ *Multiplications (8 x 0, 5 x 1, etc.)*
- ➔ *Additions (e.g., 0 + 5 + …)*

Recall: Homomorphic Encryption

A form of encryption which allows computations over encrypted data

Two classes of homomorphic encryption

Fully Homomorphic Encryption [Gentry'09]

- Supports computations for any arbitrary function
- Challenge: Can be quite inefficient

Partially Homomorphic Encryption

- Supports a particular type of operation

Additive Homomorphic encryption

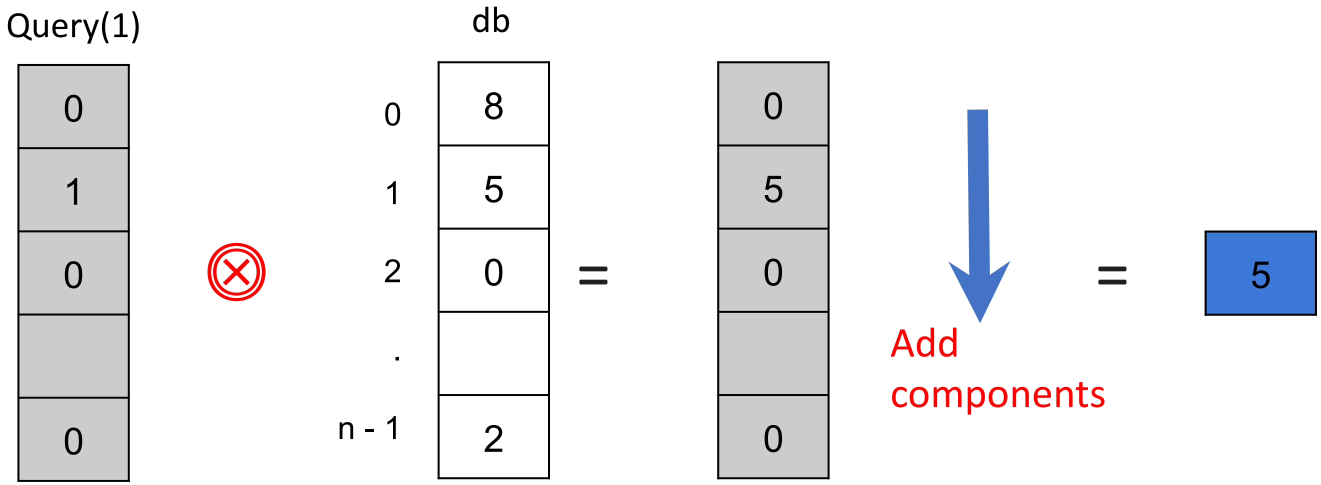
Enc(4) ⊕ Enc(8) = Enc(4 + 8) = Enc(12)

Multiplicative Homomorphic encryption

Enc(4) ⊗ Enc(8) = Enc(4 x 8) = Enc(32)

The warmup for (non-trivial) PIR

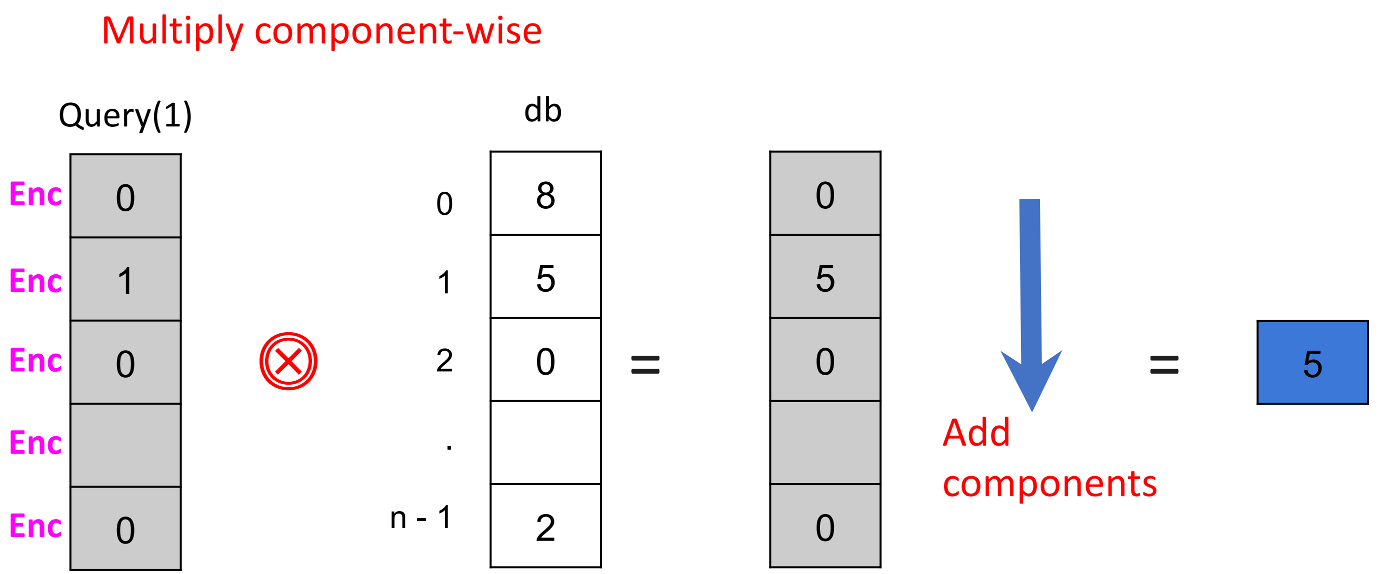
Multiply component-wise



Dot product requires two types of operations:

- ➔ *Multiplications (8 x 0, 5 x 1, etc.)*
- ➔ *Additions (e.g., 0 + 5 + …)*

Recall the warmup for (non-trivial) PIR



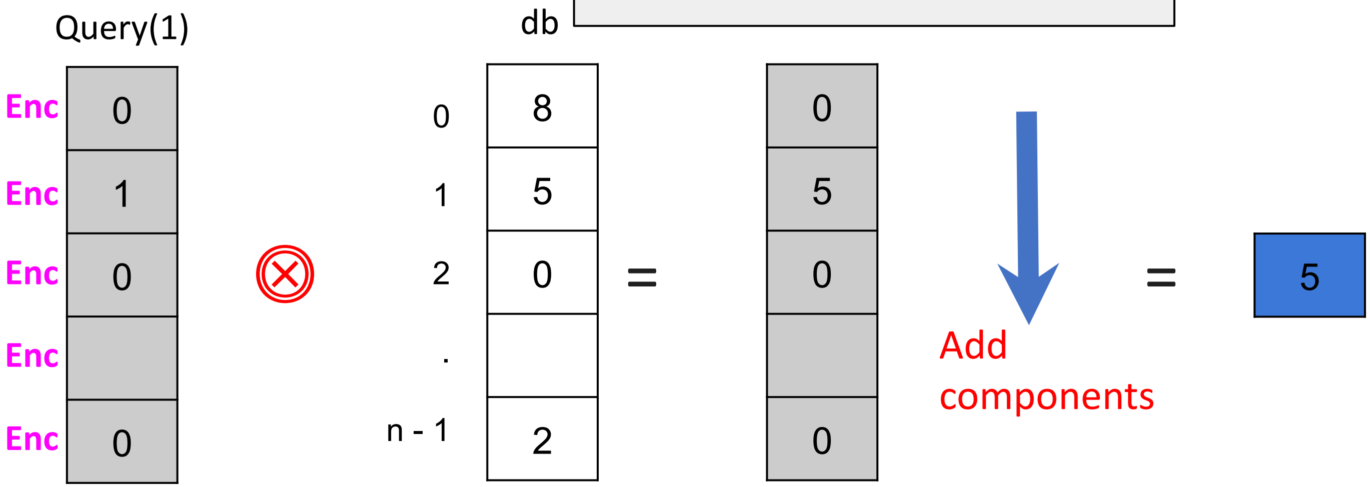
Dot product requires two types of operations:

- ➔ *Multiplications (8 x 0, 5 x 1, etc.)*
- ➔ *Additions (e.g., 0 + 5 + …)*

Recall the warmup for (non-trivial) PIR

Homomorphically multiply component-wise

$\text{Enc}(m)^k = \text{Enc}(m * k)$



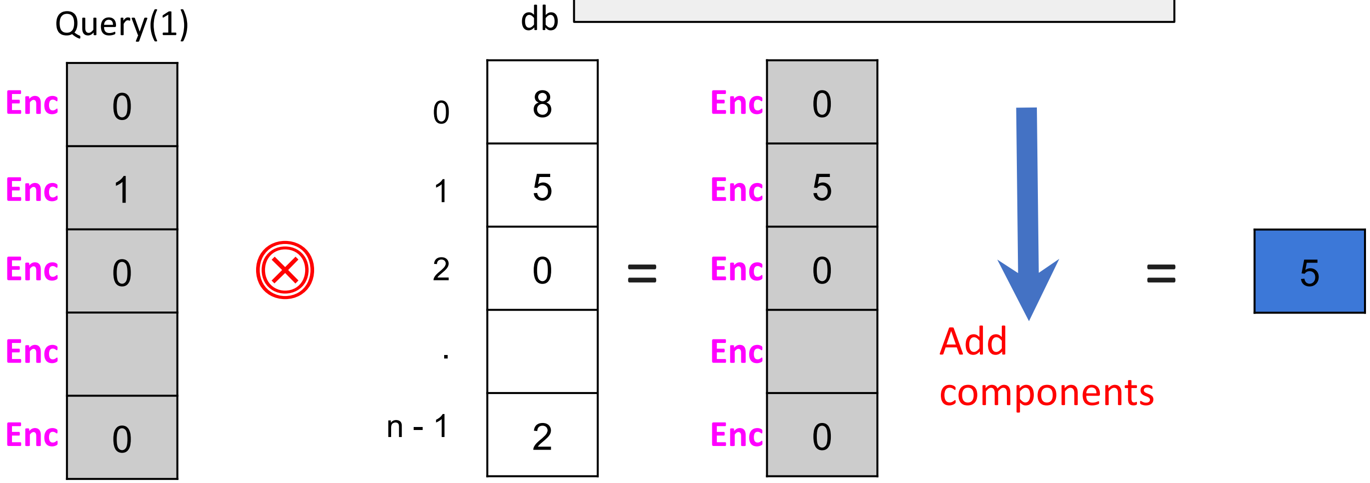
Dot product requires two types of operations:

- ➔ *Multiplications (8 x 0, 5 x 1, etc.)*
- ➔ *Additions (e.g., 0 + 5 + …)*

Recall the warmup for (non-trivial) PIR

Homomorphically multiply component-wise

$$Enc(m)^k = Enc(m * k)$$



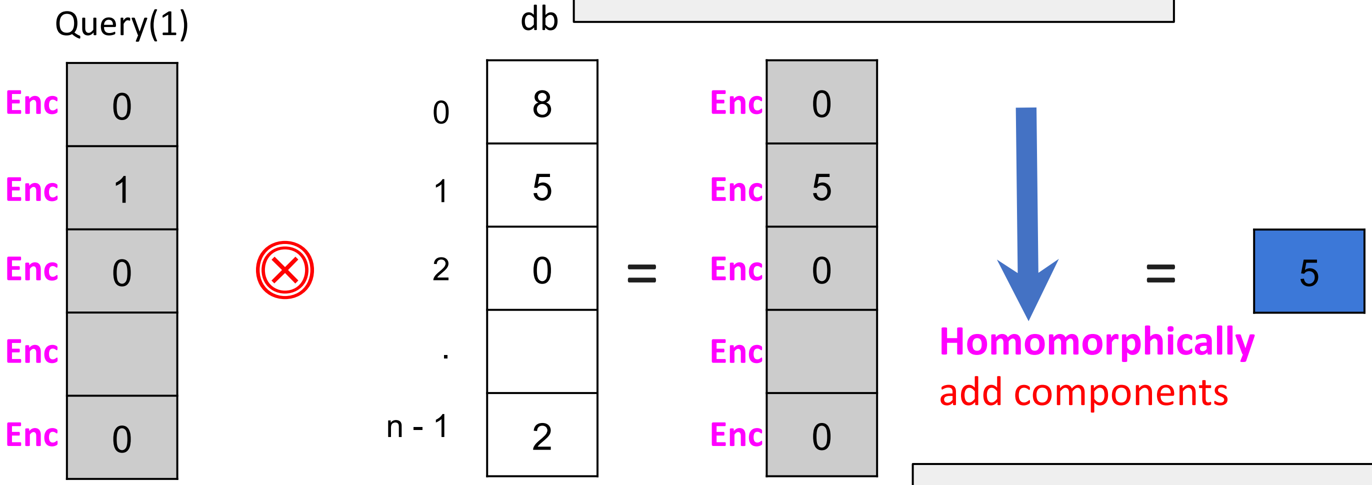
Dot product requires two types of operations:

- ➔ *Multiplications (8 x 0, 5 x 1, etc.)*
- ➔ *Additions (e.g., 0 + 5 + …)*

Recall the warmup for (non-trivial) PIR

Homomorphically multiply component-wise

$\text{Enc}(m)^k = \text{Enc}(m * k)$



$\text{Enc}(m_1) \times \text{Enc}(m_2) = \text{Enc}(m_1 + m_2)$

Dot product requires two types of operations:

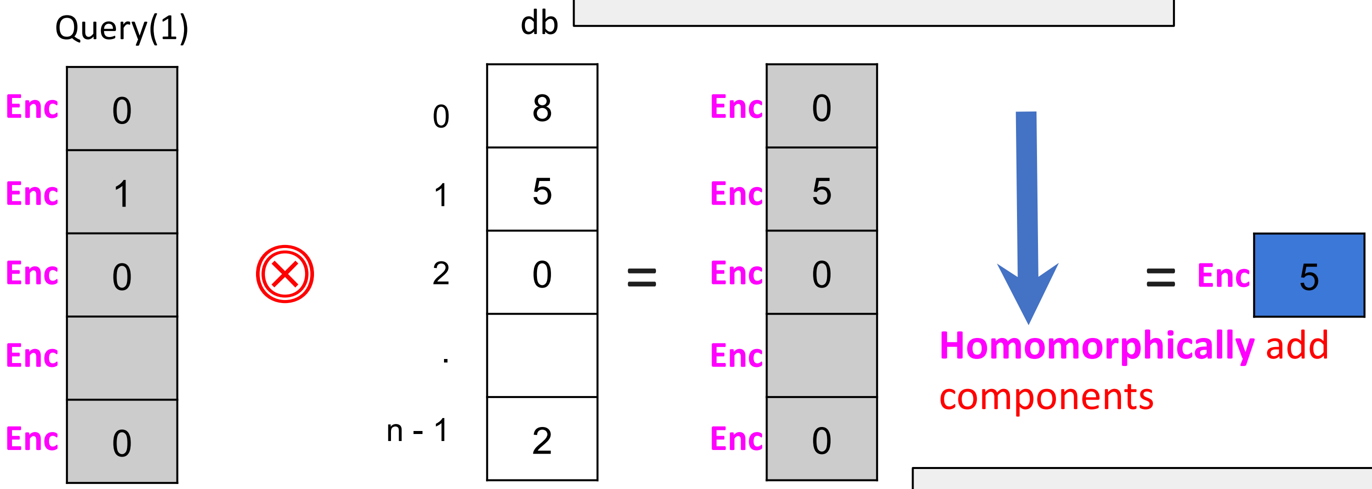
➔ *Multiplications (8 x 0, 5 x 1, etc.)*

➔ *Additions (e.g., 0 + 5 + …)*

Recall the warmup for (non-trivial) PIR

Homomorphically multiply component-wise

$\text{Enc}(m)^k = \text{Enc}(m * k)$



Homomorphically add components

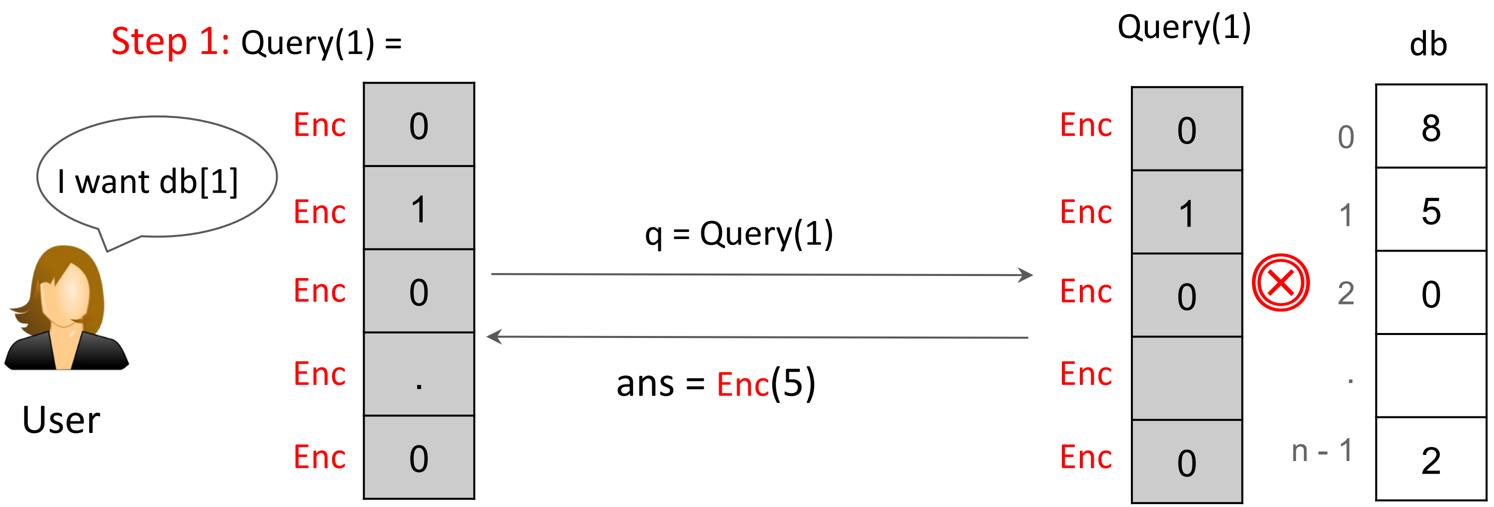
$\text{Enc}(m_1) \times \text{Enc}(m_2) = \text{Enc}(m_1 + m_2)$

Dot product requires two types of operations:

- ➔ *Multiplications (8 x 0, 5 x 1, etc.)*
- ➔ *Additions (e.g., 0 + 5 + …)*

Putting it all together: A PIR protocol

Step 1: Query(1) =



Step 2:
Answer(db, q) is a
secure dot product

Untrusted Server

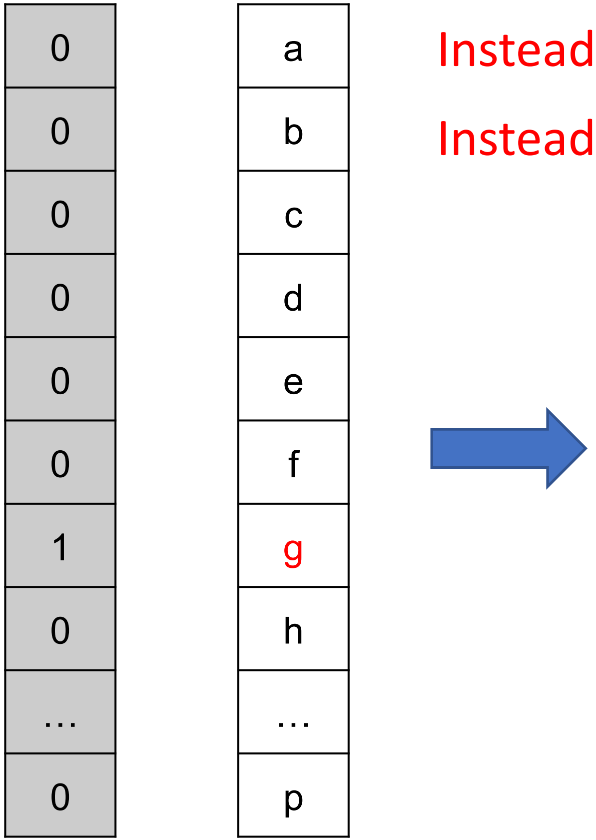
Step 3:
db[1] = Decode(ans) = Decrypt(ans)

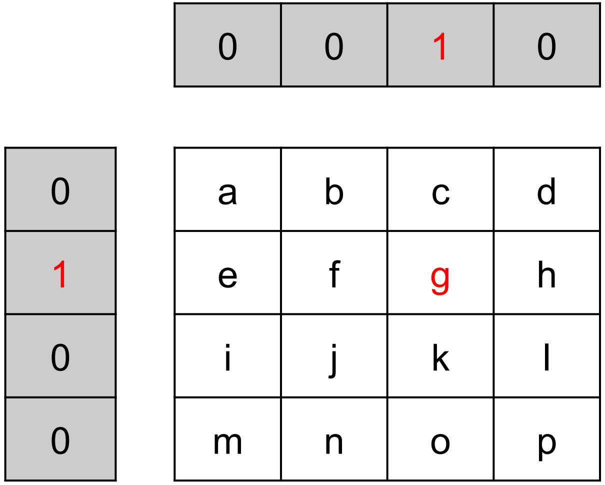
Retrieval is equivalent to computing a secure dot product

Can we reduce query size? How?

Instead of 1 dim database, view it in 2 dims.

Instead of 1 query, use 2 queries.





Two-stage query execution

0
1
0
0

*

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

=

0	0	0	0
e	f	g	h
0	0	0	0
0	0	0	0

Add rows

e	f	g	h

In first pass, extract the row of interest

Two-stage query execution

e	f	g	h

*

0	0	1	0

=

0	0	g	0

Add columns

g

So, query size is down from n to $2\sqrt{n}$.

Part 2: Retrieval by key

k_1	1
k_2	2
k_3	3
…	…
k_n	n

k

Client retrieves:

- v, if (k,v) at Server
- ∅, otherwise

Give me value for key k →

k_0	v_0
k_1	v_1
k_2	v_2
…	…
k_{n-1}	v_{n-1}

Untrusted Server

Part 2: How can the client privately retrieve the value corresponding to a given key?

This area originated as Private retrieval by keywords in 1998 (Chor et al. TOC '98)

Private Keyword retrieval can be performed by two stages:

Stage 1: Retrieve the key location

k

Give me the location for key k →

← *i*

k_0	v_0
k_1	v_1
k_2	v_2
…	…
k_{n-1}	v_{n-1}

Stage 2: Perform PIR with location

Has (key, location) mapping

Give me the *i*-th value →

0	v_0
1	v_1
2	v_2
…	…
n-1	v_{n-1}

This area originated as Private retrieval by keywords in 1998 (Chor et al. TOC '98)

Private Keyword retrieval can be performed by two stages:

Stage 1: Retrieve the key location

k

Give me the location for key k →

← *i*

k_0	v_0
k_1	v_1
k_2	v_2
…	…
k_{n-1}	v_{n-1}

Stage 2: Perform PIR by index

Has (key, location) mapping

Give me the *i*-th value →

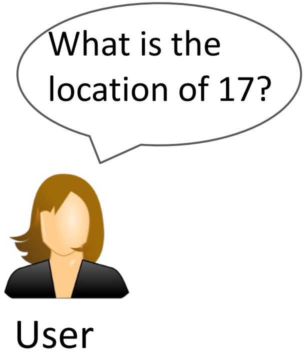
0	v_0
1	v_1
2	v_2
…	…
n-1	v_{n-1}

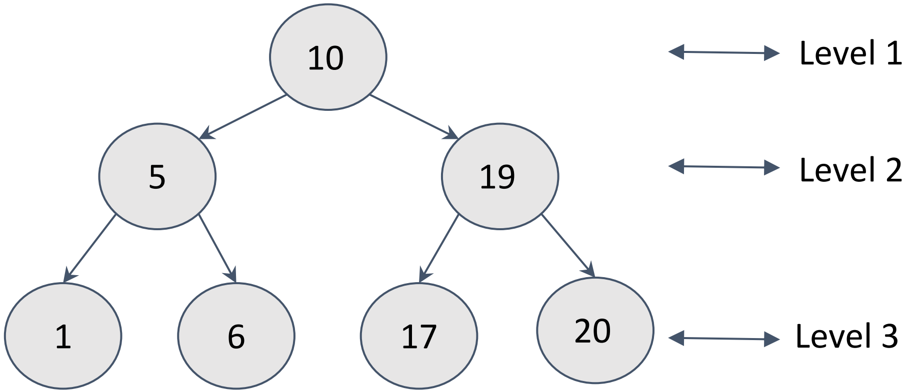
Key location can be retrieved using PIR-by-index

(Chor et al. TOC ‘98)

Assume keys are integers and arranged in a BST

K = {1, 5, 6, 10, 17, 19, 20}





Untrusted Server

Key location can be retrieved using PIR-by-index

(Chor et al. TOC ‘98)

User

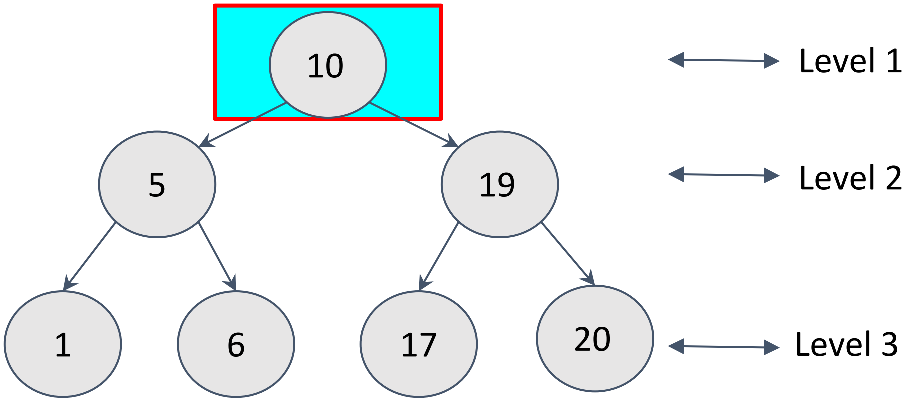
Level 1: Retrieve element at index 0 (trivial)

10 < 17

Go right

Assume keys are integers and arranged in a BST

K = {1, 5, 6, 10, 17, 19, 20}



Untrusted Server

Key location can be retrieved using PIR-by-index

(Chor et al. TOC ‘98)

Assume keys are integers and arranged in a BST

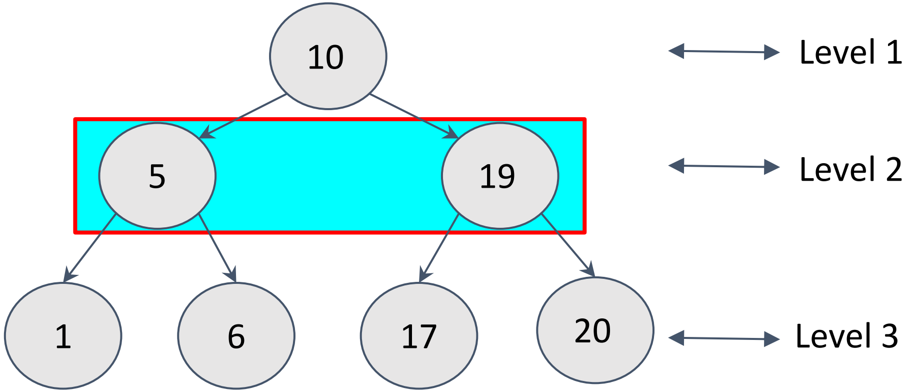
K = {1, 5, 6, 10, 17, 19, 20}

User

Level 2: Retrieve element at index 1 using PIR-by-index

17 < 19

Go left



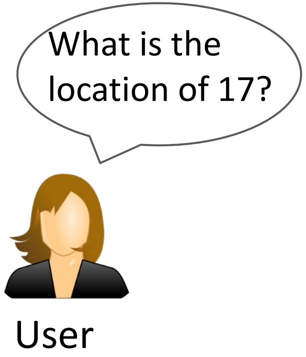
Untrusted Server

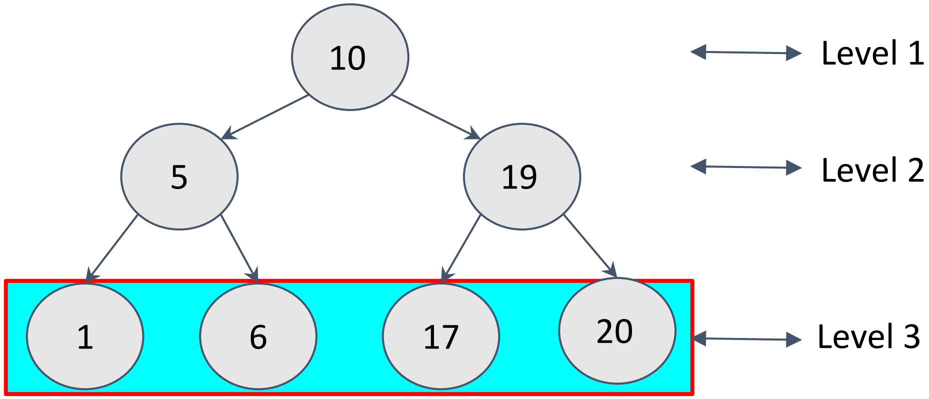
Key location can be retrieved using PIR-by-index

(Chor et al. TOC ‘98)

Assume keys are integers and arranged in a BST

K = {1, 5, 6, 10, 17, 19, 20}





Level 3: Retrieve element at index 2 using PIR-by-index

17 = 17 (found it!)

Path from root to leaf is index of k in keyset K

Untrusted Server

This area originated as Private retrieval by keywords in 1998 (Chor et al. TOC '98)

Private Keyword retrieval can be performed by two stages:

Stage 1: Retrieve the key location

k

Give me the location for key k

i

k_0	v_0
k_1	v_1
k_2	v_2
…	…
k_{n-1}	v_{n-1}

Stage 2: Perform PIR by index

Has (key, location) mapping

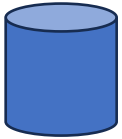
Give me the *i*-th value

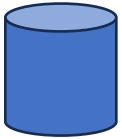
0	v_0
1	v_1
2	v_2
…	…
n-1	v_{n-1}

Information Theoretic-PIR (IT-PPIR)

- Need *k >= 2* servers with at most *t* colluding servers
- Ex: *k = 2* and *t = 1*

1	00
2	10
3	01
4	10





1	00
2	10
3	01
4	10

Information Theoretic-PIR (IT-PPIR)

- Generate an n-bit array, S, with randomly picked 0's and 1's
- Create S' → Same as S except at index i → S'[i] = S[i] complement such that S xor S' has 1 only index i
- Send S to server 1 and S' to server 2

1	00
2	10
3	01
4	10

1	00
2	10
3	01
4	10

S

1
0
1
0

S'

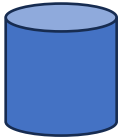
1
1
1
0

Information Theoretic-PIR (IT-PPIR)

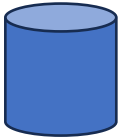
- Each server xors all values with index value 1 and sends to the client
- Client xors the two values to find the value at index i

1
0
1
0

1	00
2	10
3	01
4	10



00 xor 01 = 01



1	00
2	10
3	01
4	10

1
1
1
0

00 xor 10 xor 01 = 11

01 xor 11 = 10

Distributed point functions

Given 2 values *a* and *b*, a point function $P_{a,b}(x)$ is given by:

$$P_{a,b}(x) = \begin{cases} b & \text{for } x = a \\ 0 & \text{for } x \neq a \end{cases}$$

It's 0 everywhere except at *a*, where the value is *b*

A **distributed point function** distributes the function into *function shares*, and allows different parties to compute functions of their shared information, without revealing the information itself to either process

A DPF consists of a family of functions f_k, parameterized by key *k*, and a way to derive two keys k_0 and k_1 such that

$$P_{a,b}(x) = f_{k0}(x) + f_{k1}(x)$$

Function Secret Sharing

- A generalization of DPF such that a function *f* is split into *p* functions (split between *p* parties) s.t.

  $$f(x) = \sum f_i(x) \text{ where } i \text{ goes from 1 to } p$$

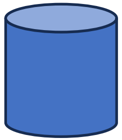
- Any strict subset of f_i *s* do not reveal anything about *f*

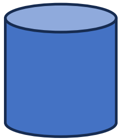
- Main difference b/w DPF and FSS is that in DPF *f(x) = 1*, whereas in FSS *f(x)* can be any value

DPF/FSS for PIR

- A DPF: $f_{a,1}(x) = 1$ when $x=a$ and 0 otherwise. *a* is our db key to find
- Let the domain of *x* be 5 (i.e., 1,2,3,4,5). These are keys of a kv-store
- Client wants to retrieve key 2 from the server without revealing 2

1	10
2	20
3	15
5	10



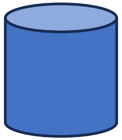


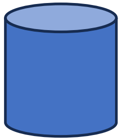
1	10
2	20
3	15
5	10

DPF/FSS for PIR

- Generate two keys k_0 and k_1 over the *entire* domain of x such that at input=2, the $k_0[2]+ k_1[2] = 1$ and $k_0[i]+ k_1[i] = 0$ everywhere else
- Send k_0 to server 1 and k_1 to server 2

1	10
2	20
3	15
5	10





1	10
2	20
3	15
5	10

k_0

4
-2
-6
9
-3

k_1

-4
3
6
-9
3

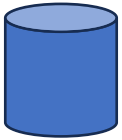
DPF/FSS for PIR

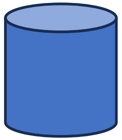
- Derive two functions $f_{k0}(x)$ *and* $f_{k1}(x)$
- Each server evaluates its own function, $f_{kb}(x)$ where $b=\{0,1\}$ for each stored db key and sends summed result
- Client computes $f(x) = f_{k0}(x) + f_{k1}(x)$

k_0

4
-2
-6
9
-3

key	val
1	10
2	20
3	15
5	10





key	val
1	10
2	20
3	15
5	10

-4
3
6
-9
3

k_1

$f_{k0}(x)$:
return $k_0[x]*val[x]$

-120+140=20!

$f_{k1}(x)$:
return $k_1[x]*val[x]$

4*10+(-2)*20+ (-6)*15+(-3)*10
= -120

-4*10+3*20+ 6*15+3*10
= 140

- Above slides only gives you an intuition
- Main benefit of DPF/FSS is that key size is ***not*** the entire domain (i.e., $2^{|x|}$)
- They are compressed to be of polynomial length

- Seminal papers:
  - DPF: https://www.iacr.org/archive/eurocrypt2014/84410245/84410245.pdf
  - FSS: https://www.iacr.org/archive/eurocrypt2015/90560300/90560300.pdf

Summary

- PIR: Retrieve a value from an external database without revealing to the db owner the object retrieved

- Computation and information theoretic PIR

- DPF/FSS can be used to generate PIR schemes