[2025.04.30] MMLAB Main Seminar – Private Information Retrieval (PIR)

ACM CCS 2024

Call Me By My Name: Simple, Practical Private Information Retrieval for Keyword Queries

Sofia Celi

Brave Software

Lisbon, Portugal

cherenkov@riseup.net

Alex Davidson

Universidade NOVA de Lisboa & Nova LINCS

Lisbon, Portugal

a.davidson@fct.unl.pt

Honggeun Park

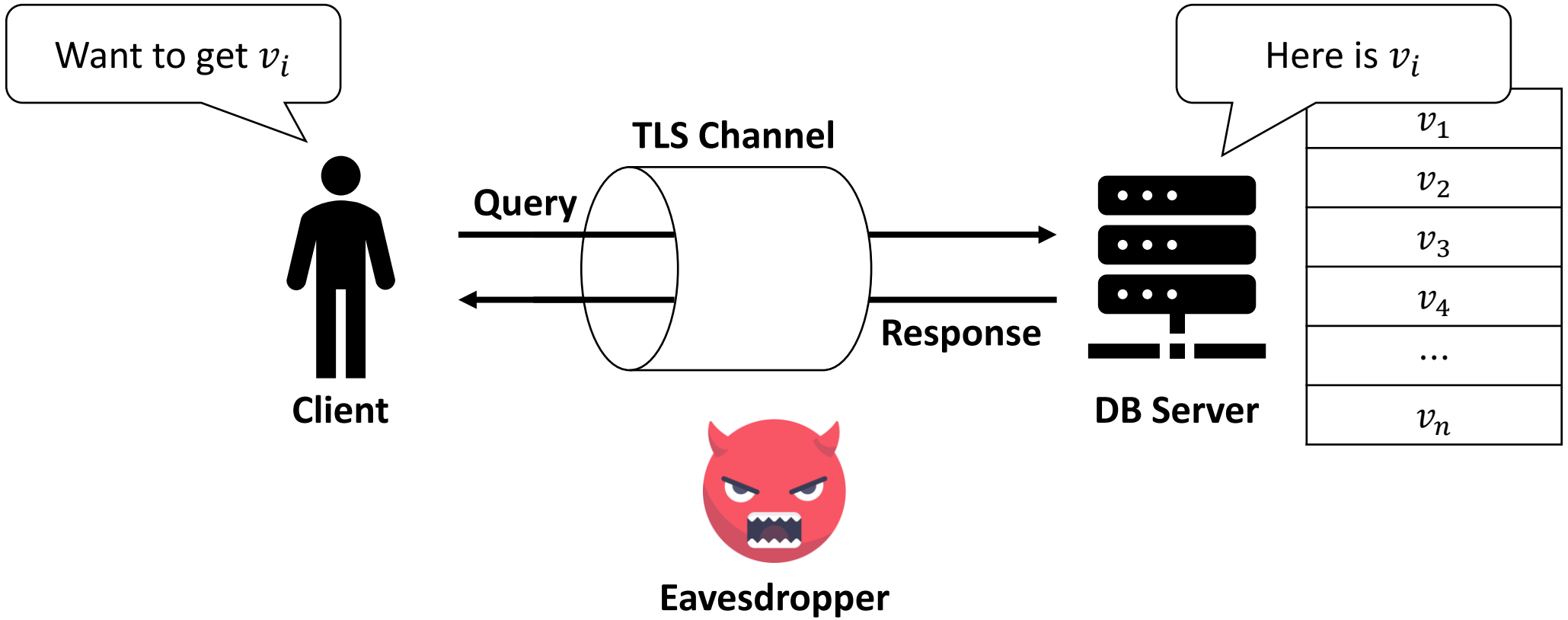
mmLab

Network Convergence & Security Lab

Table of Contents

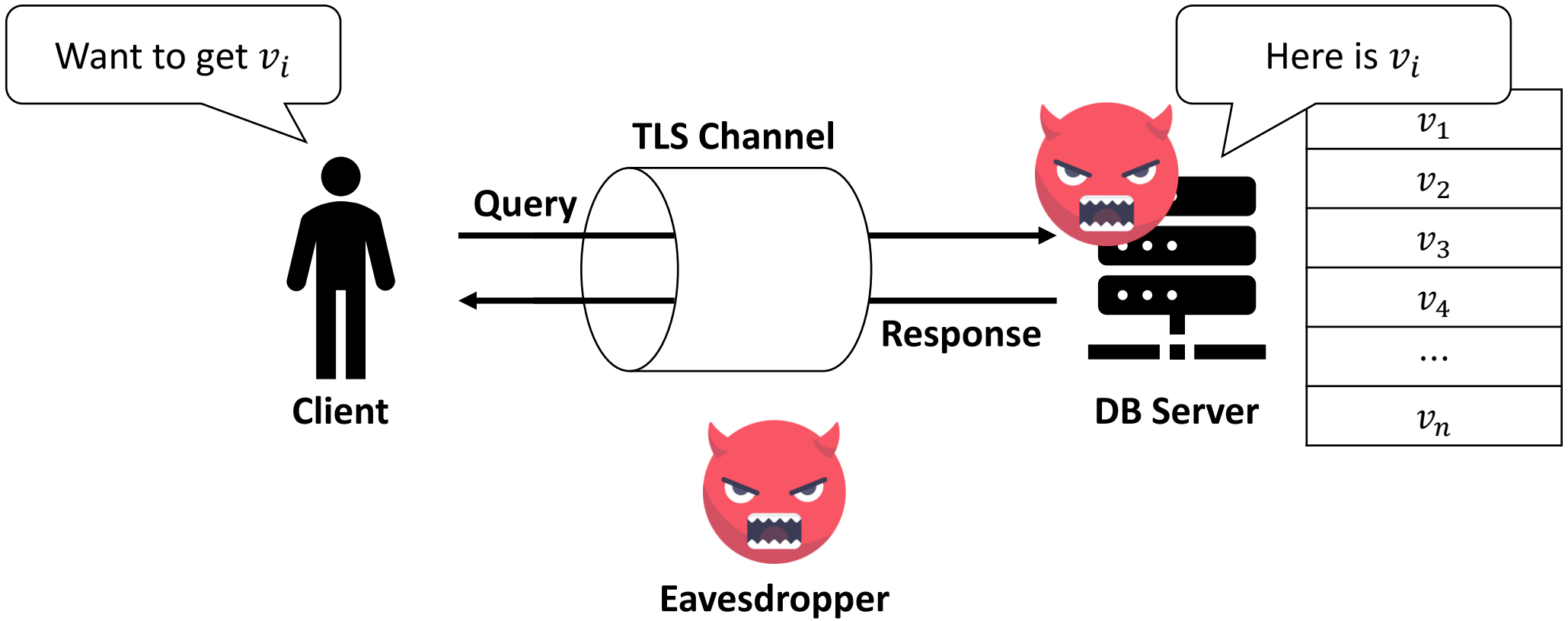
Introduction [1/2]

- Assume that client is going to query an element to database server
  - To ensure privacy, **secure communication channel** like TLS is used



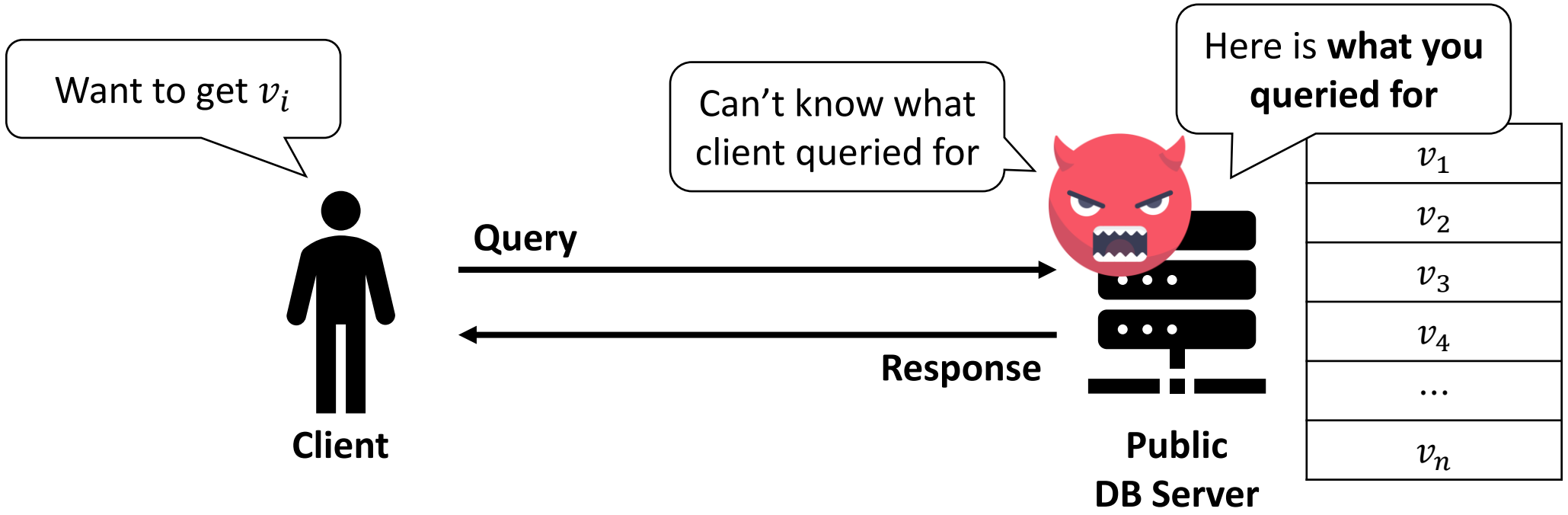
Introduction [2/2]

- In practice, DB server might not be honest
  - To ensure privacy, DB server **should not know** what client queries



Private Information Retrieval (PIR) [1/3]

- **Private Information Retrieval (PIR)**:
  - Allows private queries on **public DBs** that are hosted by an untrusted server(s)



Private Information Retrieval (PIR) [2/3]

Two Types of PIR:

1. Computationally secure PIR (CPIR)

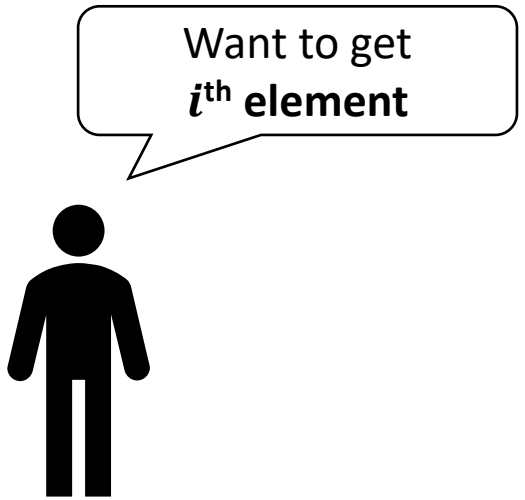
- PIR with single public DB server
- Uses homomorphic encryption
- Can be categorized into index-based CPIR and keyword-based CPIR

2. Information-Theoretically secure PIR (IT-PIR)

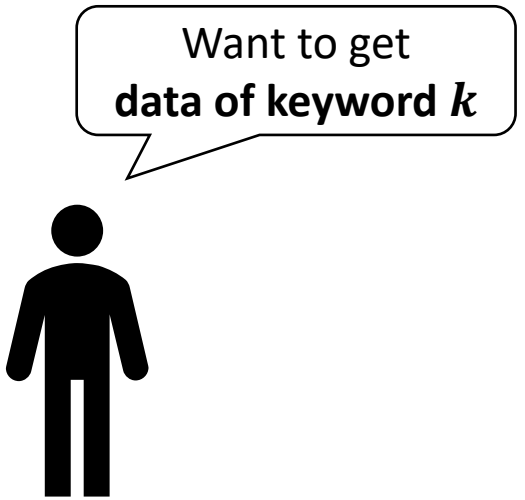
- PIR with multiple public DB servers
- Uses group of batched queries to hide what client wants
  - e.g., query (v_1, v_2) to server 1 and query (v_1, v_2, v_3) to server 2

Private Information Retrieval (PIR) [3/3]

Index-based PIR (LWEPIR)

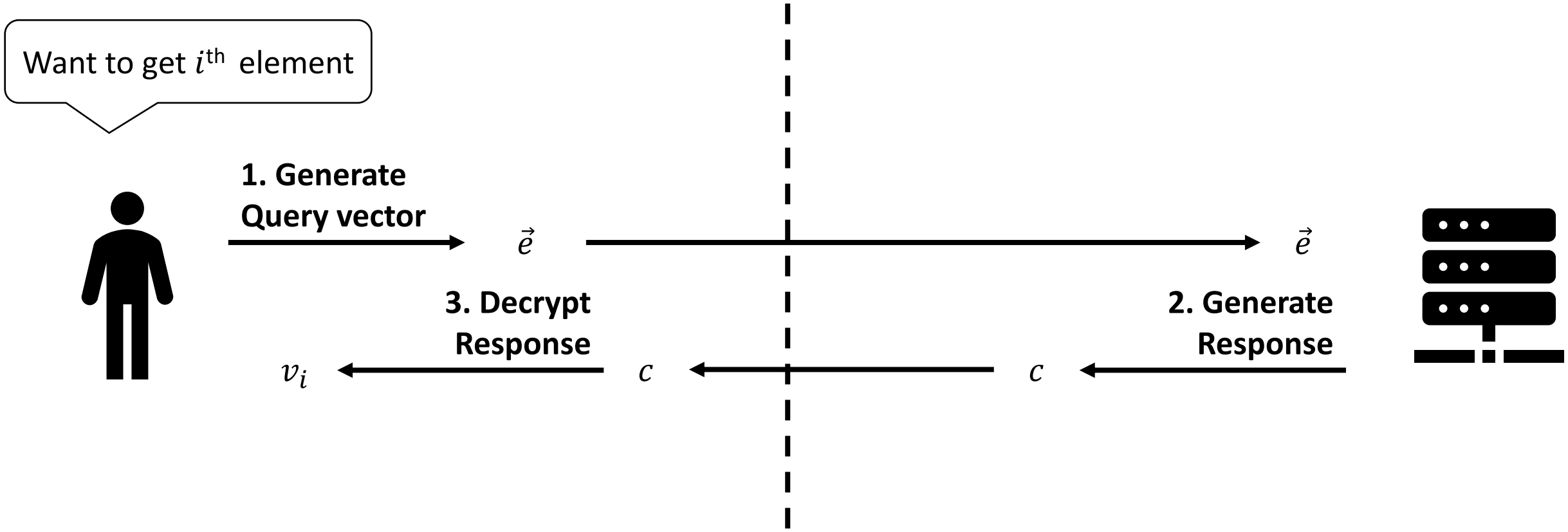


Keyword-based PIR (KWPIR)



Index-Based PIR – Overview

- Uses **LWE-based Homomorphic Encryption**
  - Uses special homomorphic encryption which supports public inner-product
  - Public inner-product: $Enc_{\mathrm{LWE}}(\vec{v}) \cdot \vec{w} = Enc_{\mathrm{LWE}}(\vec{v} \cdot \vec{w})$

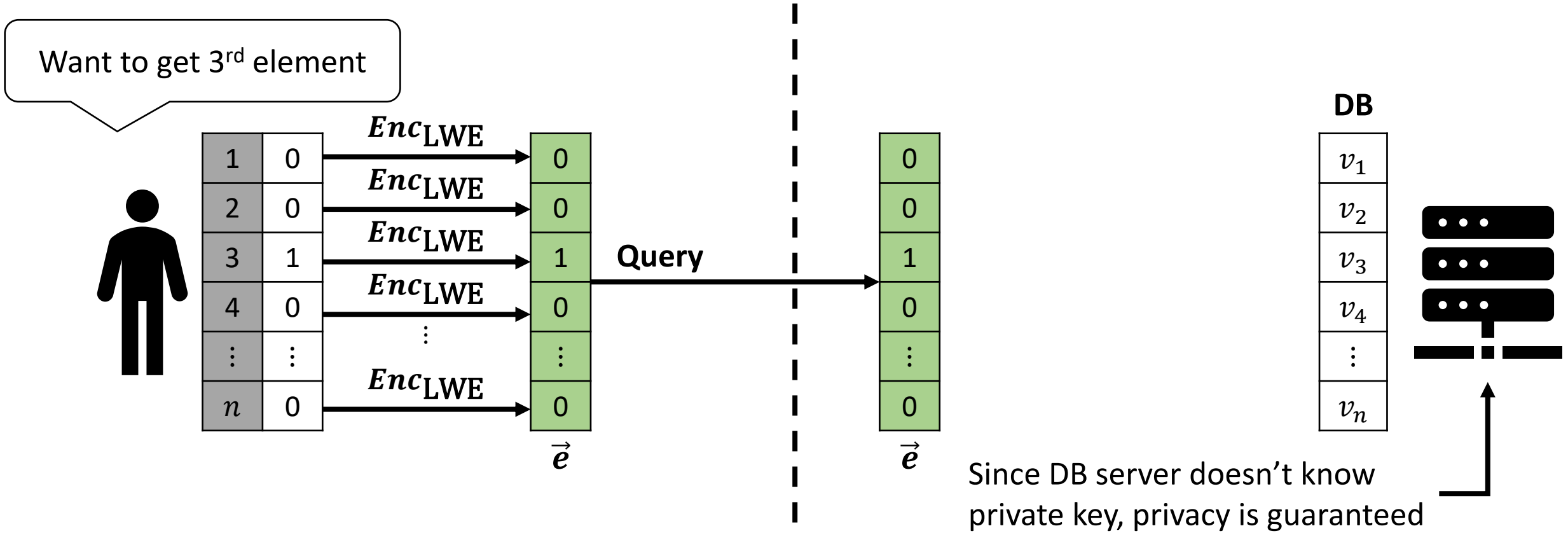


Index-Based PIR – Detail [1/3]

Encrypted Message

1. Generates Query vector $\vec{e}$

- Generates indicator vector which represent what client wants
- Encrypts each element in vector with homomorphic encryption



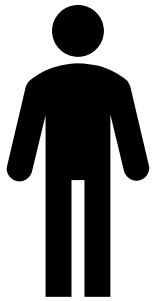
Index-Based PIR – Detail [2/3]

Encrypted Message

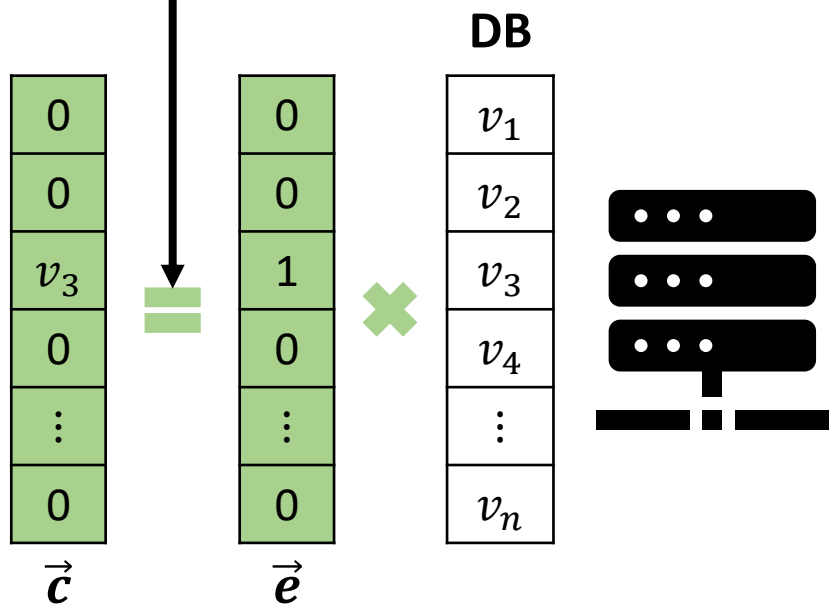
2. Generates Response c

- Multiplies each value in DB with each element in query vector $\vec{e}$ to generate $\vec{c}$

Want to get 3rd element



$$c \times Enc_{\mathrm{LWE}}(x) = Enc_{\mathrm{LWE}}(cx)$$

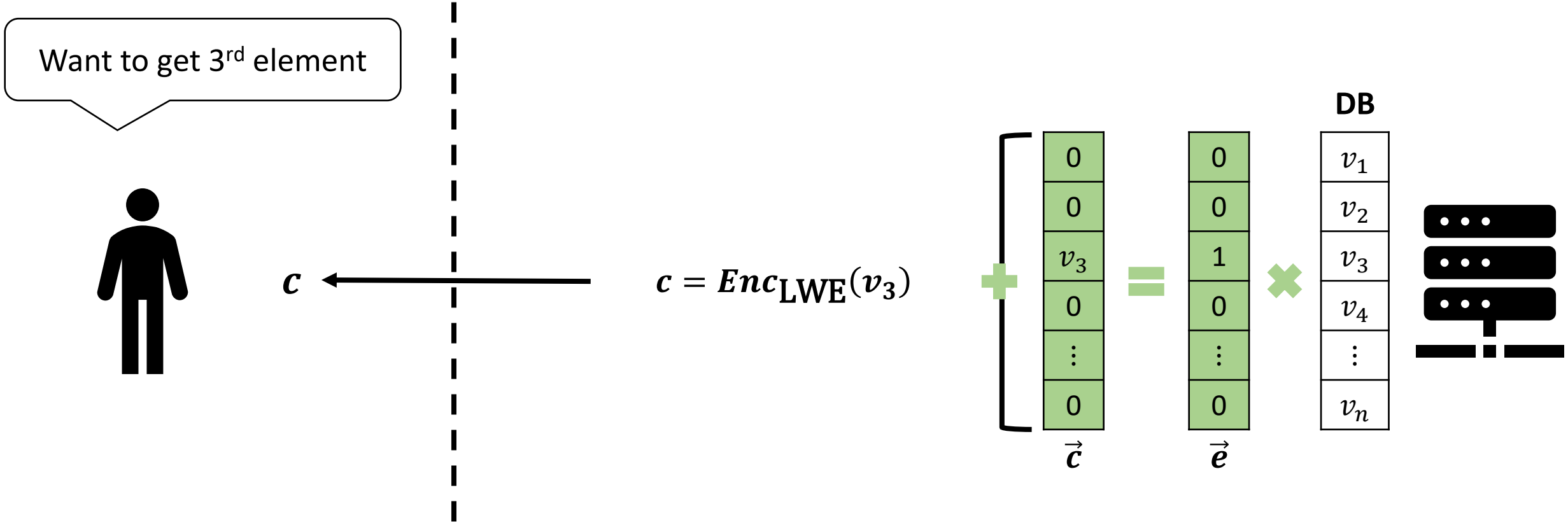


Index-Based PIR – Detail [2/3]

Encrypted Message

2. Generates Response c

- Multiplies each value in DB with each element in query vector $\vec{e}$ to generate $\vec{c}$
- Adds all element in $\vec{c}$ homomorphically to generate response c

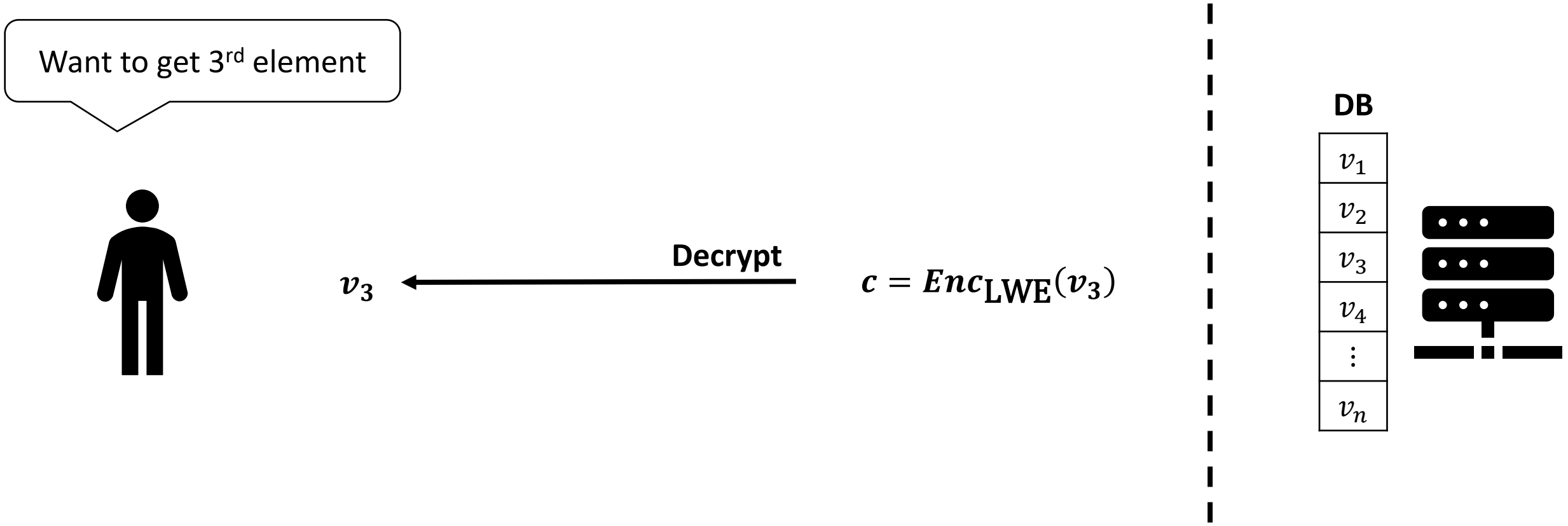


Index-Based PIR – Detail [3/3]

Encrypted Message

3. Decrypts Response c

- Decrypts response c to get queried value v_2



Keyword-Based PIR – Naïve Approach [1/3]

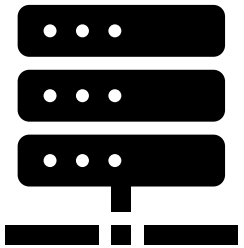
- Extends index-based PIR with 2 steps

1. **Retrieves the location of keyword in keywords**
   - Assume that keywords in DB are sorted in increasing order
   - e.g., for $k_2 = \text{banana}$, its location is 2

2. **Performs index-based PIR to that location**

Key-value map DB

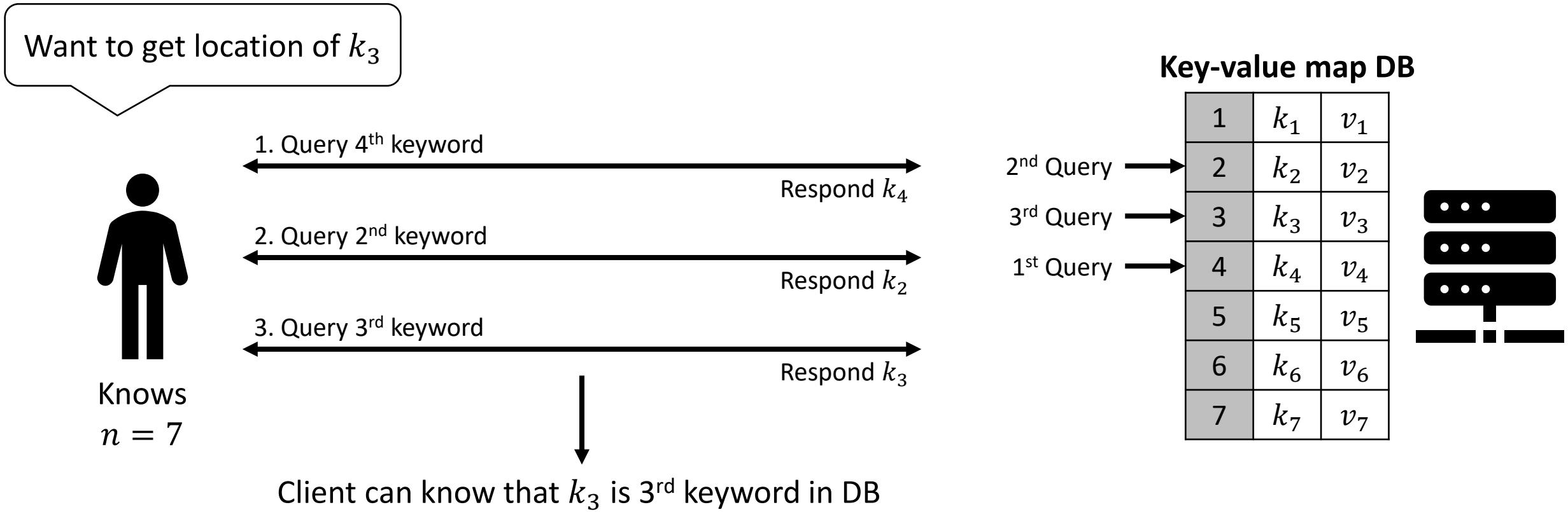
1	k_1	v_1
2	k_2	v_2
3	k_3	v_3
4	k_4	v_4
⋮	⋮	⋮
n	k_n	v_n



Keyword-Based PIR – Naïve Approach [2/3]

1. Retrieves the location of keyword in keywords

- Assume that keywords are sorted, and the number of keywords is 7
- Uses binary search to get location of keyword



Keyword-Based PIR – Naïve Approach [3/3]

1. Retrieves the location of keyword in keywords → $\boldsymbol{O(\log n)}$
   - Assume that keywords in DB are sorted in increasing order

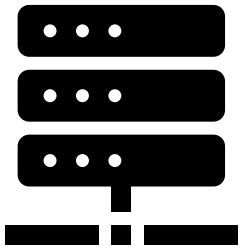
2. Performs index-based PIR to that location → **1**

- $\boldsymbol{O(\log n)}$ index-based PIRs are required!
  - **Too slow to use in practice ☹**

- **How to do better?**

Key-value map DB

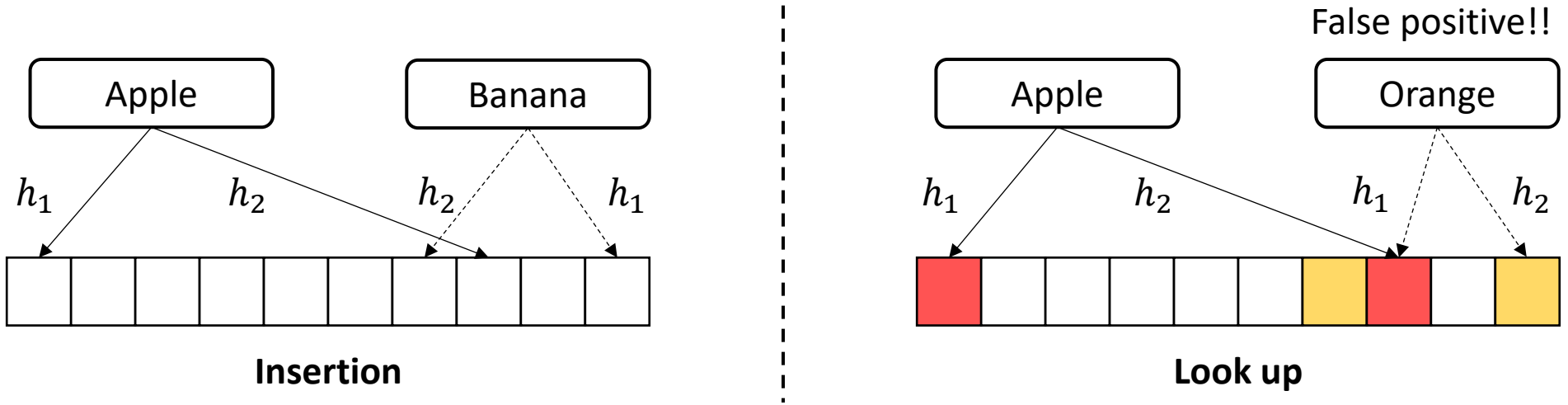
1	k_1	v_1
2	k_2	v_2
3	k_3	v_3
4	k_4	v_4
⋮	⋮	⋮
n	k_n	v_n



Keyword-Based PIR – Better Approach [1/2]

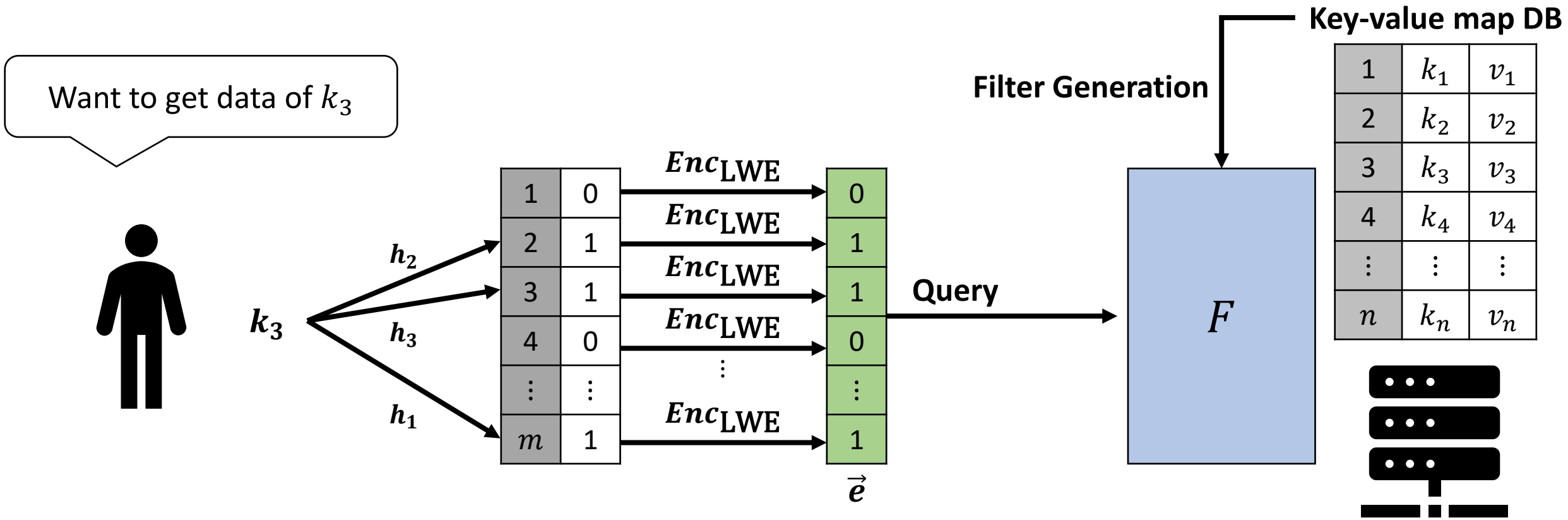
- Uses probabilistic filter (PF) like Bloom Filter
  - Probabilistic data structure for **testing membership of data**
  - Operates by maintaining k hash functions
  - Only false positive exists with false positive rate $0 < \epsilon < 1$
  - Usually used with finite set of items

Example with $k = 2$



Keyword-Based PIR – Better Approach [2/2]

- Uses probabilistic filter (PF) like Bloom Filter
  - $O(\log n)$ index-based PIRs → 1 index-based PIR with some false positive rate ϵ



1. Introduction & Background
    1) Introduction
    2) Private Information Retrieval (PIR)

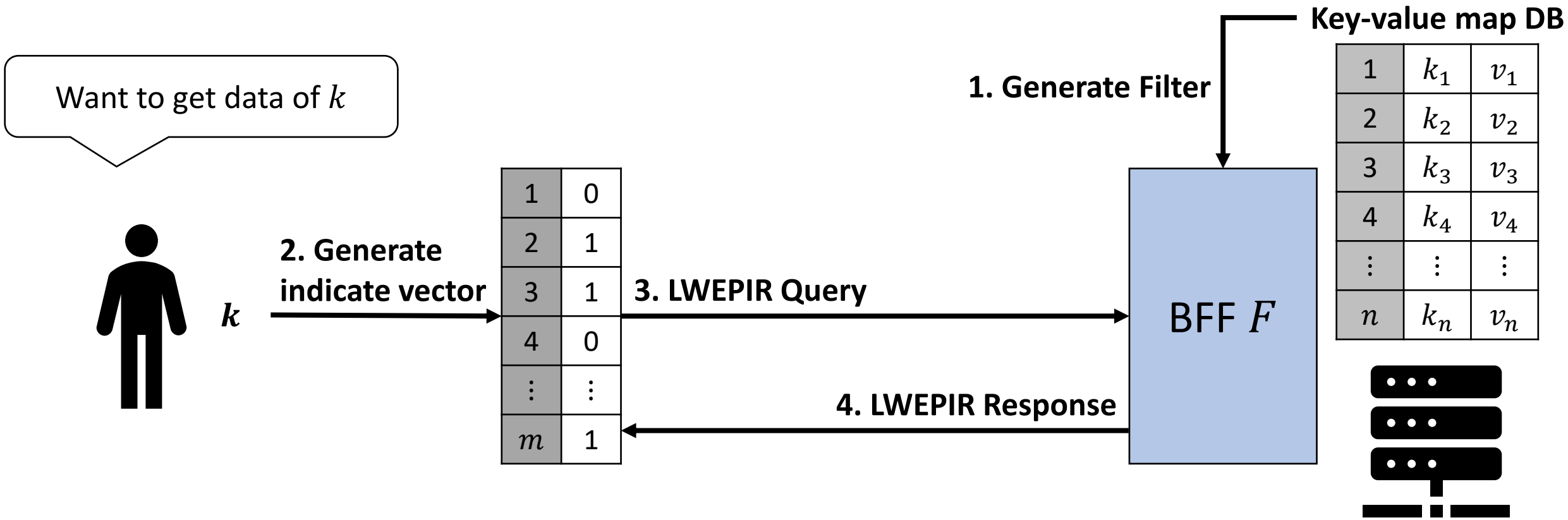
2. **Simple, Practical KWPIR: ChalametPIR**
    1) ChalametPIR
    2) Binary Fuse Filter

3. Performance Evaluation

4. Conclusion & Discussion

ChalametPIR [1/2]

- ChalemetPIR : Simple & Practical KWPIR
  - For **simplicity**, just combines Binary Fuse Filter (BFF) with LWEPIR



Binary Fuse Filter [1/3]

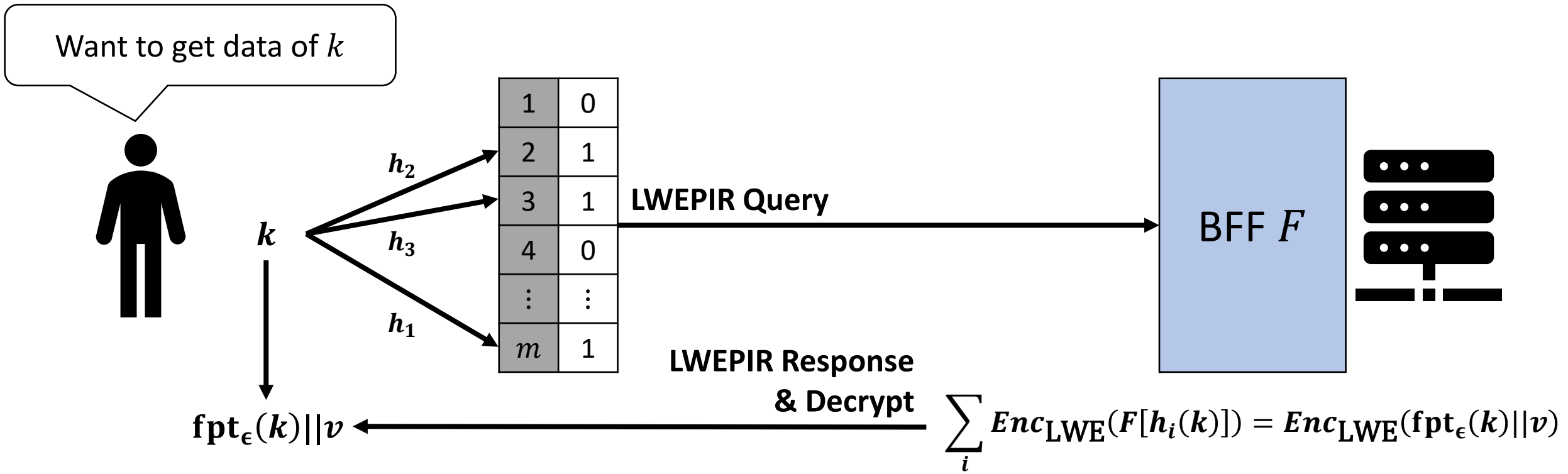
- Static probabilistic filter based on XOR filter
  - No insertion and deletion after filter construction

- More efficient than other probabilistic filters:
  1. More than twice as fast as the construction of XOR filters
  2. Query speed comparable to that of XOR filters
  3. Low memory usage per key
     - Bloom filter : 1.44α
     - XOR filter : 1.23α
     - Binary Fuse Filter : 1.075α ~ 1.125α

Binary Fuse Filter [2/3]

- Given : $k = \{3, 4\}$ hash functions & fingerprint function fpt_ϵ
- Goal : Construct filter F such that

$$F[h_1(k)] + F[h_2(k)] + F[h_3(k)] = \text{fpt}_\epsilon(k)||v \; mod \; p$$



Binary Fuse Filter [3/3]

- Main Idea:
  1. Picks a random index from $h_1(\mathrm{X}), h_2(\mathrm{X}), h_3(\mathrm{X}) \rightarrow h_3(\mathrm{X})$
  2. Encodes that block as $F[h_3(\mathrm{X})] = (\mathrm{fpt}_\epsilon(\mathrm{X})||v) - F[h_1(\mathrm{X})] - F[h_2(\mathrm{X})]$

Assumptions : $p = 16$ & $\mathrm{X} \xrightarrow{h_1,h_2,h_3} \{1, 4, 7\}$ & $\mathrm{fpt}_\epsilon(\mathrm{X})||v = 4$

6	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9

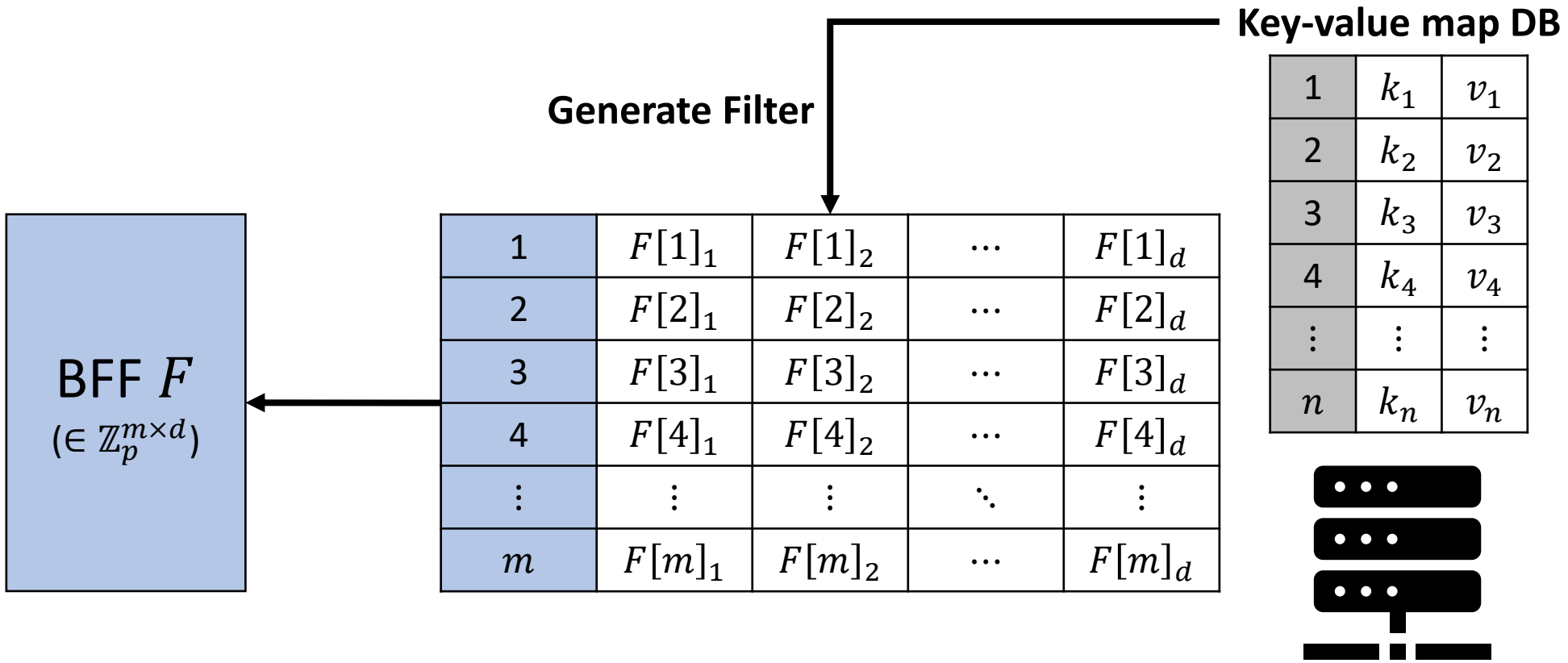
Picks 7 from $\{1, 4, 7\}$, and encodes $F[7]$ as following:

$F[7] = (\mathrm{fpt}_\epsilon(\mathrm{X})||v) - F[1] - F[4] = 4 - 6 - 0 = -2 \equiv 14\ (mod\ 16)$

6	0	0	0	0	0	**14**	0	0
1	2	3	4	5	6	7	8	9

ChalametPIR [2/2]

- In practice, $\mathrm{fpt}_\epsilon(k)||v$ is longer than $\log p$
  - Divides $\mathrm{fpt}_\epsilon(k)||v$ into $\log p$-bits blocks



1. Introduction & Background
    1) Introduction
    2) Private Information Retrieval (PIR)

2. Simple, Practical KWPIR: ChalametPIR
    1) ChalametPIR
    2) Binary Fuse Filter

3. **Performance Evaluation**

4. Conclusion & Discussion

Metrics [1/2]

- Goal of ChalametPIR : making simple and practical KWPIR
  - Simplicity: just combines BFF and LWEPIR

- How to define practicality?
  - Practicality: amount of expenses which server should cover

1. **Bandwidth Costs** (query size, response size)
2. **Online Performance** (runtime for query, response, parsing)
3. **Online Costs** (cost, rate, throughput)

Metrics [2/2]

3. Online Costs : Comparison with other KWPIR

- Performance metrics for **deploying ChalametPIR server in AWS**

1) Cost (USD) : current AWS financial cost for running a server in “c5.9xlarge”
   - CPU per-hour : $1.53/36 = $0.0425 per hour
   - Download cost : $0.09 per GB
   - Upload cost : $0 per GB (no cost)

2) Rate : ratio of retrieved record size to response size
   - How small the response $\boldsymbol{ENC}_{\mathbf{LWE}}(\mathbf{fpt}_{\epsilon}(\boldsymbol{k})||\boldsymbol{v})$ is compared to record $\boldsymbol{v}$

3) Throughput (MB/s) : ratio of database size to server’s online computation time
   - How fast server can deal with database

Experiment Setup

- **Baseline LWEPIR** : {FrodoPIR, SimplePIR}
  - FrodoPIR : Response size is much smaller than query size
  - SimplePIR : Response size and query size are similar

- **Server Specification** : {t2.2xlarge, c5.9xlarge, Macbook M1 Max}

- **Database Parameters** :
  1) Key-value map size m : $\{2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}\}$
  2) Value size w : {256B, 1kB, 30kB, 100kB}
  3) Modular value p : $\{2^9, 2^{10}\}$
  4) Number of hash functions for BFF : $\{3, 4\}$

Bandwidth Costs [1/2]

- For **FrodoPIR based ChalametPIR**,
  - Query size depends on the number of keys in DB
  - Response size depends on the size of value

$\|\mathrm{KV}\|$	# keys $\times$ |value| $(m \times w)$	**Query** (kB) $k=3$	**Query** (kB) $k=4$	**Response** (kB) $k=3$	**Response** (kB) $k=4$	# keys $\times$ |value| $(m \times w)$	**Query** (kB) $k=3$	**Query** (kB) $k=4$	**Response** (kB) $k=3$	**Response** (kB) $k=4$
	LWEPIR = **FrodoPIR**					LWEPIR = **SimplePIR**				
	$2^{16} \times 1\mathrm{kB}$	287	276	3.2	3.2	$2^{16} \times 1\mathrm{kB}$	31.89	31.17	31.89	31.17
	$2^{17} \times 1\mathrm{kB}$	579	553	3.2	3.2	$2^{17} \times 1\mathrm{kB}$	44.65	43.64	44.65	43.64
$m \uparrow$	$2^{18} \times 1\mathrm{kB}$	1157	1106	3.2	3.2	$2^{18} \times 1\mathrm{kB}$	63.78	62.34	63.78	62.34
	$2^{19} \times 1\mathrm{kB}$	2314	2212	3.56	3.56	$2^{19} \times 1\mathrm{kB}$	90.36	88.32	90.36	88.32
	$2^{20} \times 1\mathrm{kB}$	4628	4424	3.56	3.56	$2^{20} \times 1\mathrm{kB}$	127.56	124.68	127.56	124.68
	$2^{20} \times 256\mathrm{B}$	4628	4424	0.89	0.89	$2^{20} \times 256\mathrm{B}$	63.78	62.34	63.78	62.34
$w \uparrow$	$2^{17} \times 30\mathrm{kB}$	579	553	96	96	$2^{17} \times 30\mathrm{kB}$	256.18	250.4	256.18	250.4
	$2^{14} \times 100\mathrm{kB}$	72	69	291	291	$2^{14} \times 100\mathrm{kB}$	180.71	176.63	180.71	176.63

Bandwidth Costs [2/2]

- For **SimplePIR based ChalametPIR**,
  - Query size and response size depend on total size of DB $(m \times w)$
  - Query size and response size are equal for same setting

$\|\mathrm{KV}\|$	# keys $\times$ |value| $(m \times w)$	**Query** (kB) $k=3$	**Query** (kB) $k=4$	**Response** (kB) $k=3$	**Response** (kB) $k=4$	# keys $\times$ |value| $(m \times w)$	**Query** (kB) $k=3$	**Query** (kB) $k=4$	**Response** (kB) $k=3$	**Response** (kB) $k=4$
	LWEPIR = **FrodoPIR**						LWEPIR = **SimplePIR**			
	$2^{16} \times 1\mathrm{kB}$	287	276	3.2	3.2	$2^{16} \times 1\mathrm{kB}$	31.89	31.17	31.89	31.17
	$2^{17} \times 1\mathrm{kB}$	579	553	3.2	3.2	$2^{17} \times 1\mathrm{kB}$	44.65	43.64	44.65	43.64
$m \uparrow$	$2^{18} \times 1\mathrm{kB}$	1157	1106	3.2	3.2	$2^{18} \times 1\mathrm{kB}$	63.78	62.34	63.78	62.34
	$2^{19} \times 1\mathrm{kB}$	2314	2212	3.56	3.56	$2^{19} \times 1\mathrm{kB}$	90.36	88.32	90.36	88.32
	$2^{20} \times 1\mathrm{kB}$	4628	4424	3.56	3.56	$2^{20} \times 1\mathrm{kB}$	127.56	124.68	127.56	124.68
	$2^{20} \times 256\mathrm{B}$	4628	4424	0.89	0.89	$2^{20} \times 256\mathrm{B}$	63.78	62.34	63.78	62.34
$w \uparrow$	$2^{17} \times 30\mathrm{kB}$	579	553	96	96	$2^{17} \times 30\mathrm{kB}$	256.18	250.4	256.18	250.4
	$2^{14} \times 100\mathrm{kB}$	72	69	291	291	$2^{14} \times 100\mathrm{kB}$	180.71	176.63	180.71	176.63

Online Performance

- FrodoPIR based ChalametPIR with $k = 3$ hash functions
  - Client runtime (Query & Parsing)
  - Server runtime (Response)

- For server to generate response, it takes up to 1846 ms

Unit : ms

	DB ($m \times w$)	Query	Response	Parsing
Macbook M1 Max	$2^{16} \times 1024$ B	0.010597	6.5508	0.22001
	$2^{17} \times 1024$ B	0.038866	12.473	0.21894
	$2^{18} \times 1024$ B	0.051996	24.452	0.21658
	$2^{19} \times 1024$ B	0.14442	54.053	0.24204
	$2^{20} \times 1024$ B	0.24049	116.89	0.24384
EC2 “t2.t2xlarge”	$2^{16} \times 1024$ B	0.050048	37.830	0.47251
	$2^{17} \times 1024$ B	0.1787	74.733	0.47046
	$2^{18} \times 1024$ B	0.19739	143.82	0.46782
	$2^{19} \times 1024$ B	0.4219	319.82	0.50735
	$2^{20} \times 1024$ B	0.8471	634.21	0.56381
EC2 “c5.9xlarge”	$2^{20} \times 256$ B	1.3699	133.58	0.090116
	$2^{17} \times 30$ kB	0.055415	1846.6	10.663
	$2^{14} \times 100$ kB	0.0040465	760.64	35.485

Online Costs

- Comparison with other KWPIR, SparsePIR
  - Server: AWS EC2 c5.9xlarge
  - : most optimal case
  - : second-most optimal case

- In most cases, ChalametPIR is better
  1) Online Runtime : 6×-11× faster
  2) Financial Cost : 3.75×-11.4× less
     - Due to cost structure of AWS

	ChalametPIR		SparsePIR	
	FrodoPIR	SimplePIR	Onion	Spiral
Online costs: $2^{20} \times 256$ B				
Query (kB)	287	63.78	63	14
Response (kB)	0.89	63.78	127	21
Runtime (s)	0.13358	0.13358†	3.04	1.44
Rate	0.28	0.004	0.002	0.012
Throughput (MB/s)	1916	1916	84	178
Cost (USD)	1.65e−6	7.05e−6	4.68e−5	1.88e−5
Online costs: $2^{17} \times 30$ kB				
Query (kB)	579	256	63	14
Response (kB)	96	256.18	127	86
Runtime (s)	1.8466	1.8466†	41.91	11.57
Rate	0.313	0.117	0.236	0.349
Throughput (MB/s)	2218	2218	98	354
Cost (USD)	3e−5	4.37e−5	5.05e−4	1.43e−4
Online costs: $2^{14} \times 100$ kB				
Query (kB)	72	180.71	63	14
Response (kB)	291	176.63	508	242
Runtime (s)	0.76064	0.76064†	17.32	5.91
Rate	0.344	0.566	0.197	0.413
Throughput (MB/s)	2692	2692	118	347
Cost (USD)	3.4e−5	2.41e−5	0.25e−4	9.05e−5

Conclusion

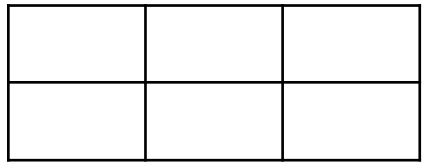
- ChalametPIR is simple KWPIR scheme
  - ChalametPIR consists of Binary Fuse Filter and LWEPIR scheme

- ChalametPIR is more efficient than state-of-the-art KWPIR scheme
  - 6×-11× faster in server's online runtime
  - Requires 3.75×-11.4× less cost when deploying

Discussion

- ChalametPIR can be more efficient
  - In this paper, authors do not use several optimizing techniques

- Experiment setup appears to be designed to highlight FrodoPIR based ChalametPIR
  - Since upload cost is 0 in AWS instance, minimizing response size is optimal
  - FrodoPIR based ChalametPIR uses more bandwidth costs in total



Thank you

ORAM vs. PIR

- For DB servers who deal with sensitive data, they store encrypted data
  - To prevent some leakages, Oblivious RAM (ORAM) is used

- For DB servers who store public data, they cannot encrypt their data
  - Encrypting each data element with client's key is inefficient
  - To ensure privacy, PIR can be used

Binary Fuse Filter – Peeling [1/3]

- Order of insertion is very important due to **cycle**

$$X \rightarrow h_1(X) = 1, h_2(X) = 4, h_3(X) = 7$$
$$Y \rightarrow h_1(Y) = 1, h_2(X) = 5, h_3(X) = 9$$

0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9

Inserting X $F[7] = \text{fpt}_\epsilon(\text{X})||v_\text{X} - F[1] - F[4] = \mathbf{4} - 0 - 0 = \mathbf{4}$

0	0	0	0	0	0	**4**	0	0
1	2	3	4	5	6	7	8	9

Inserting Y $F[1] = \text{fpt}_\epsilon(\text{Y})||v_\text{Y} - F[5] - F[9] = \mathbf{6} - 0 - 0 = \mathbf{6}$

6	0	0	0	0	0	**4**	0	0
1	2	3	4	5	6	7	8	9

$F[1] + F[4] + F[7] = \mathbf{6} + 0 + \mathbf{4} = \mathbf{10} \neq \mathbf{4}\left(= \text{fpt}_\epsilon(\text{X})||v_\text{Y}\right)$

Binary Fuse Filter – Peeling [2/3]

- Since insertion requires multiple filter blocks, order is important
  - $\mathrm{X} \xrightarrow{h_1,h_2,h_3} \{1,4,7\}, \mathrm{Y} \xrightarrow{h_1,h_2,h_3} \{1,5,9\}$

Insertion Order ↓

1	2	3	4	5	6	7	8	9
X Y			X	Y		X		Y

Since $F[1]$ is changed after inserting X,

$$F[1]' + F[4] + F[7]$$
$$= F[1]' + F[4] + (\mathrm{fpt}_\epsilon(\mathrm{X})||v - F[4] - F[7])$$
$$= F[1]' - F[1] + \mathrm{fpt}_\epsilon(\mathrm{X})||v \neq \mathrm{fpt}_\epsilon(\mathrm{X})||v$$

Binary Fuse Filter – Peeling [3/3]

- X (1, 4, 7), Y (1, 5, 9), Z (1, 4, 6)

1	2	3	4	5	6	7	8	9
X, Y, Z			X, Z	Y	Z	X		Y
Y				Y				Y
X			X			X		
Z			Z		Z			

Insertion Order ↓

↑ Decision Order

Comparison with LWEPIR

- Almost similar with its baseline LWEPIR scheme