CIPHERMATCH

Accelerating Homomorphic Encryption-Based String Matching via Memory-Efficient Data Packing and In-Flash Processing

Mayank Kabra
Rakesh Nadig, Harshita Gupta, Rahul Bera, Manos Frouzakis, Vamanan Arulchelvan, Yu Liang, Haiyu Mao, Mohammad Sadrosadati, and Onur Mutlu

ETH zürich SAFARI KING'S College LONDON

Executive Summary

Problem: Secure exact string matching using homomorphic encryption (HE) lacks scalability due to performance bottlenecks in **two key areas**:

a) **Use of complex homomorphic multiplication** resulting in high computation cost
b) **Data movement bottleneck** from large encrypted database stored in solid-state drive (SSD)

Goal: Develop an **algorithm-hardware co-design** to provide **scalable, parallelizable and efficient** HE-based secure ***exact*** string-matching

Key Idea: Use (a) ***only*** **homomorphic addition** and (b) **perform in-flash processing** by exploiting the **operational principles of NAND-flash memory** to accelerate secure ***exact*** string matching

CIPHERMATCH: A new algorithm-hardware co-design
that significantly improves the performance of HE-based secure *exact* string matching by

a) **using *only* homomorphic addition** to reduce the high computation cost
b) **optimizing** the **data packing scheme** to reduce memory footprint
c) **designing** a new **in-flash-processing (IFP) architecture** to reduce data movement

Key Results:

a) CIPHERMATCH algorithm: **42.9x speedup** & **39.4x energy savings** than best software
b) CIPHERMATCH with IFP: **136.9x speedup** & **256.4x energy savings** over CM-SW

Talk Outline

Background, Problem & Goal

Key Idea

CIPHERMATCH System: Overview

CIPHERMATCH: Algorithm

CIPHERMATCH: Hardware

Evaluation Results

Talk Outline

Background, Problem & Goal

Key Idea

CIPHERMATCH System: Overview

CIPHERMATCH: Algorithm

CIPHERMATCH: Hardware

Evaluation Results

Exact String Matching

Exact string matching is used in many security critical applications, such as

Databases

e.g., searching a query in sensitive databases

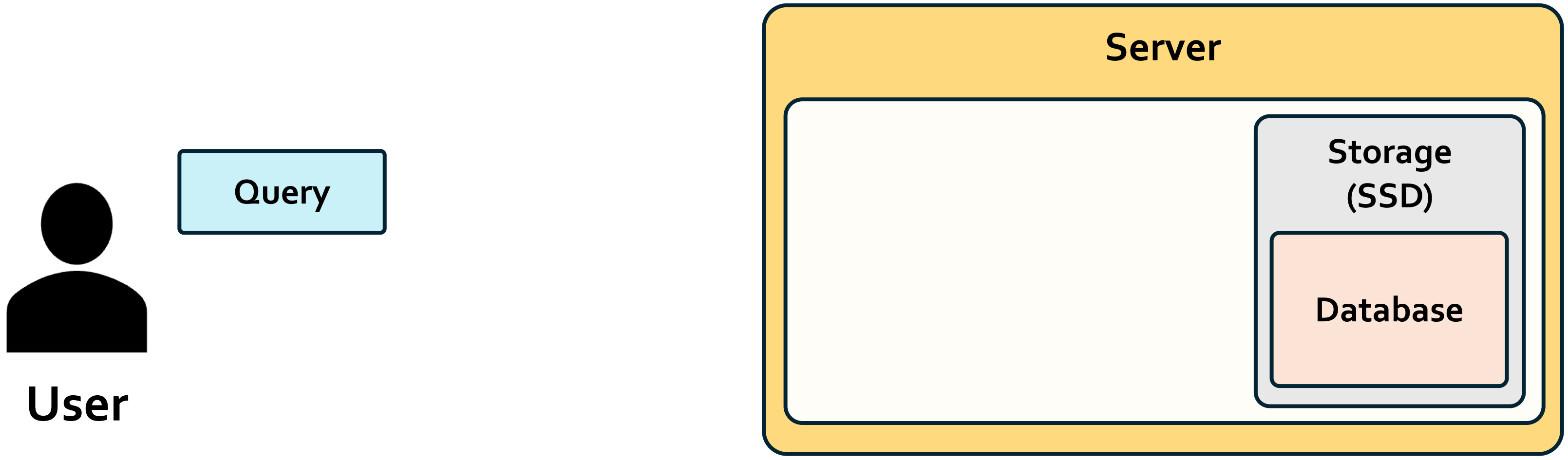
[Koudas+, VLDB 2003]
[Chen+, TIP 2013]

Bioinformatics

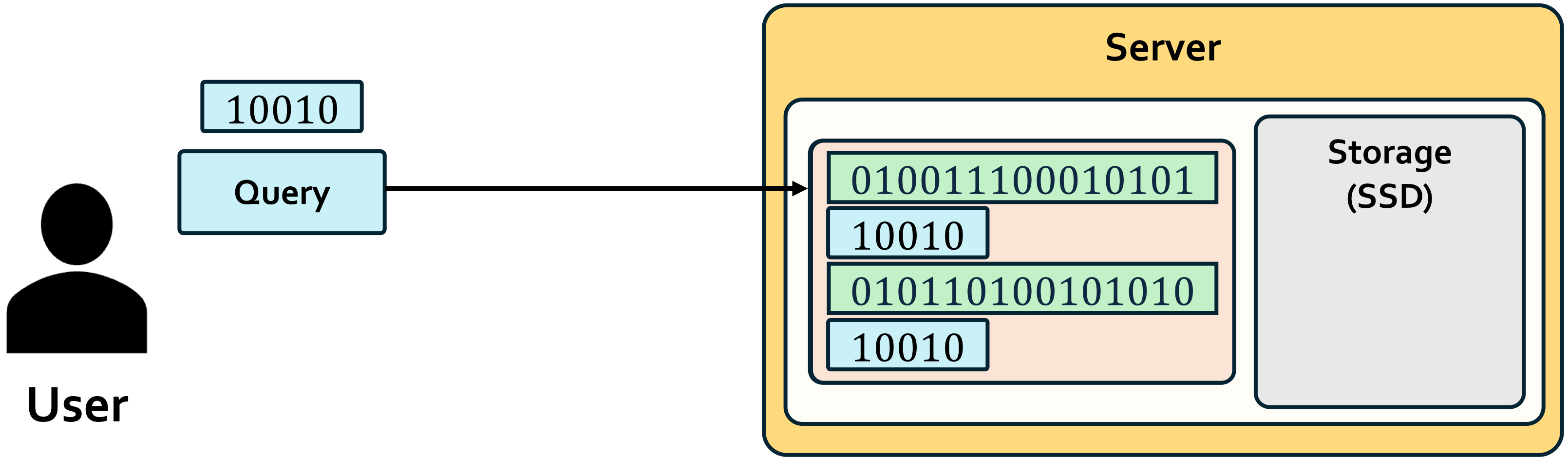
e.g., identifying similarities in DNA sequences

[Bhukya+, IJCA 2011]
[Cali+, MICRO 2020]

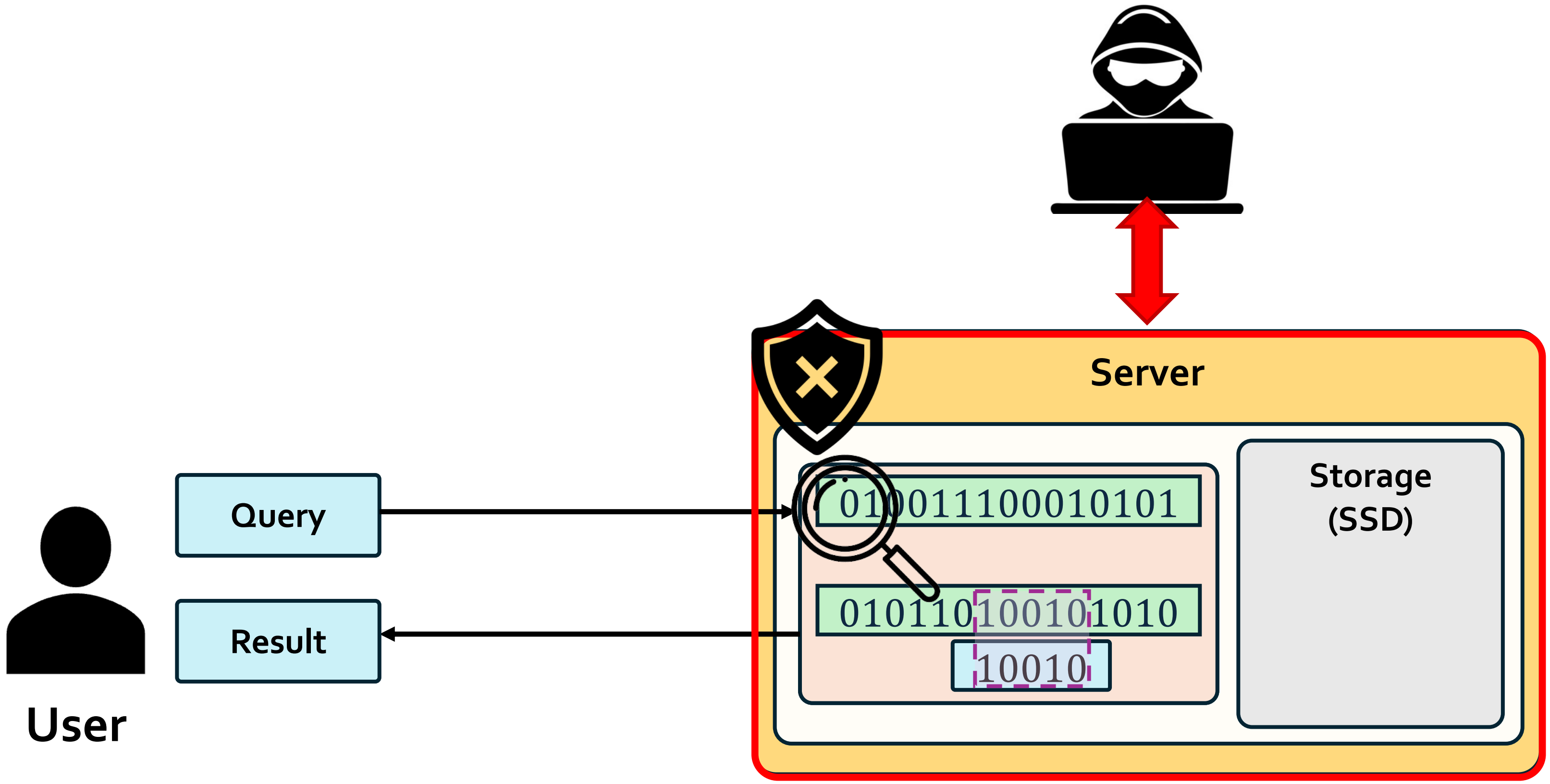
Exact String Matching



Exact String Matching

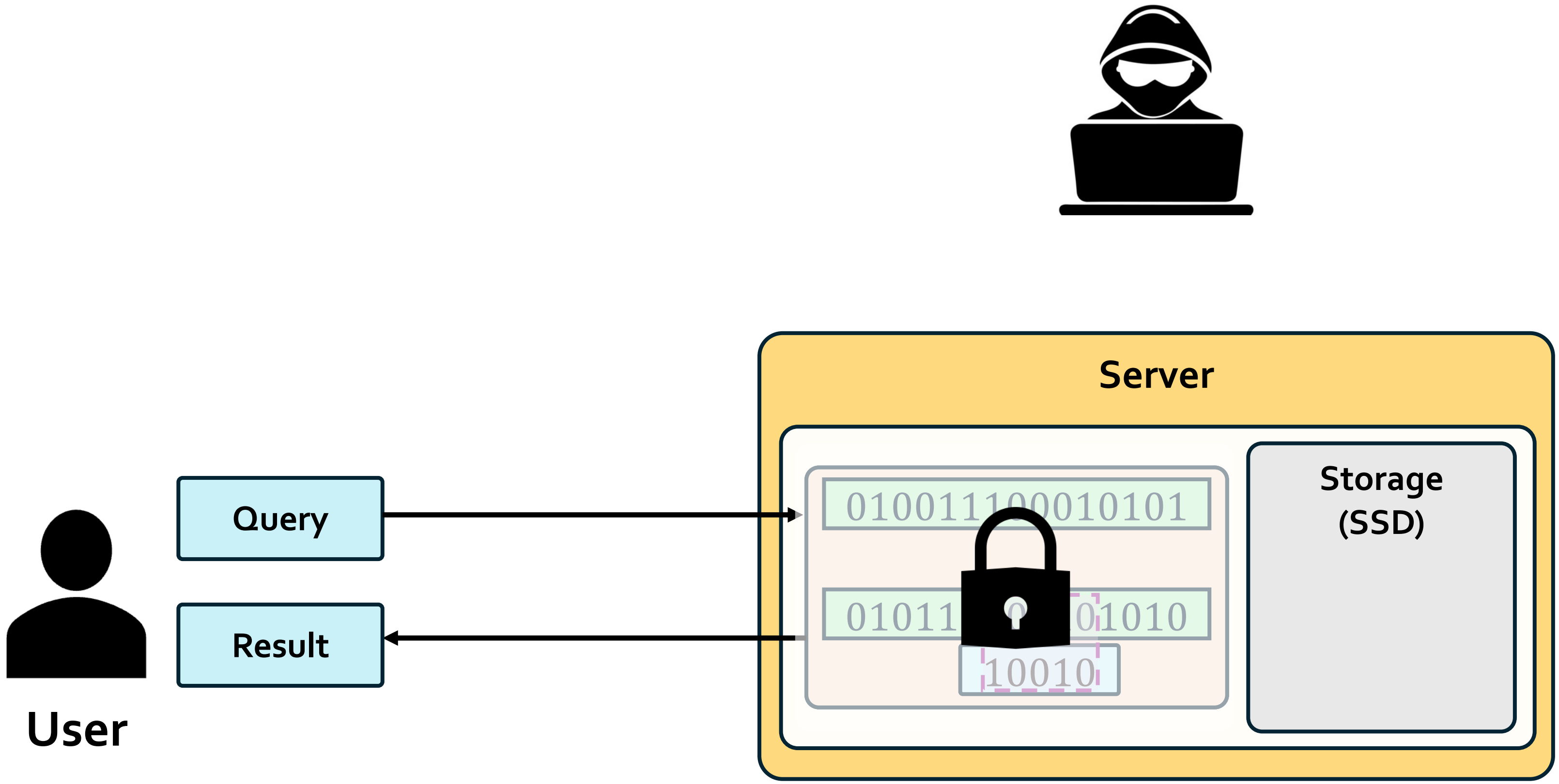


Exact String Matching



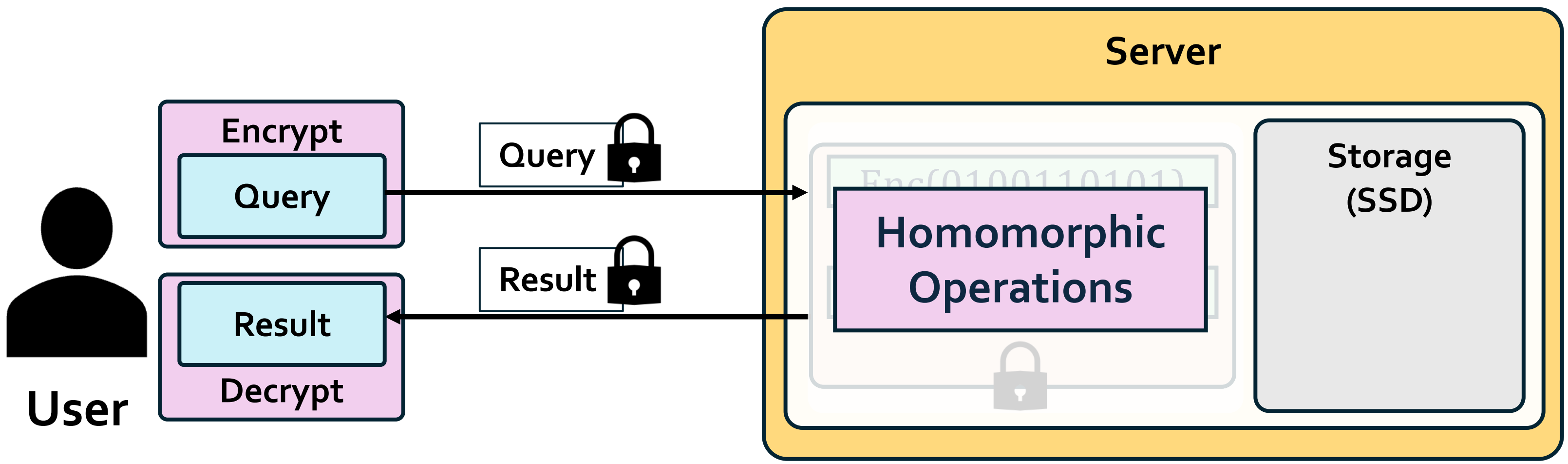
Performing computation on plaintext can lead to data leakage

Exact String Matching



Homomorphic encryption (HE) can be leveraged to perform secure *exact* string matching

Secure Exact String Matching



Allows users to compute on
encrypted data without decrypting it

Approaches to HE-based String Matching

Secure string matching using HE can be performed using **two** key approaches

Boolean Approach

Encrypt **individual bits** and use **homomorphic XOR and AND operations**

1) High memory footprint

3) Supports flexible query size

Arithmetic Approach

Encrypt **multiple packed bits** and use **homomorphic MUL and ADD operations**

1) Low memory footprint

3) Supports limited query size

More detailed analysis in the paper

Prior Works on HE-based String Matching

Arithmetic Approach [Yasuda+, CCSW 2013 ; Kim+, TDSC 2017 ; Bonte+, CCS 2020]

Boolean Approach [Pradel+, TrustCom 2021 ; Aziz+, Information 2024]

Approaches to HE-based String Matching

Secure string matching using HE can be performed using **two** key approaches

Boolean Approach

Encrypt **individual bits** and use **homomorphic XOR and AND operations**

1) High memory footprint
2) High computation cost
3) Supports flexible query size

Arithmetic Approach

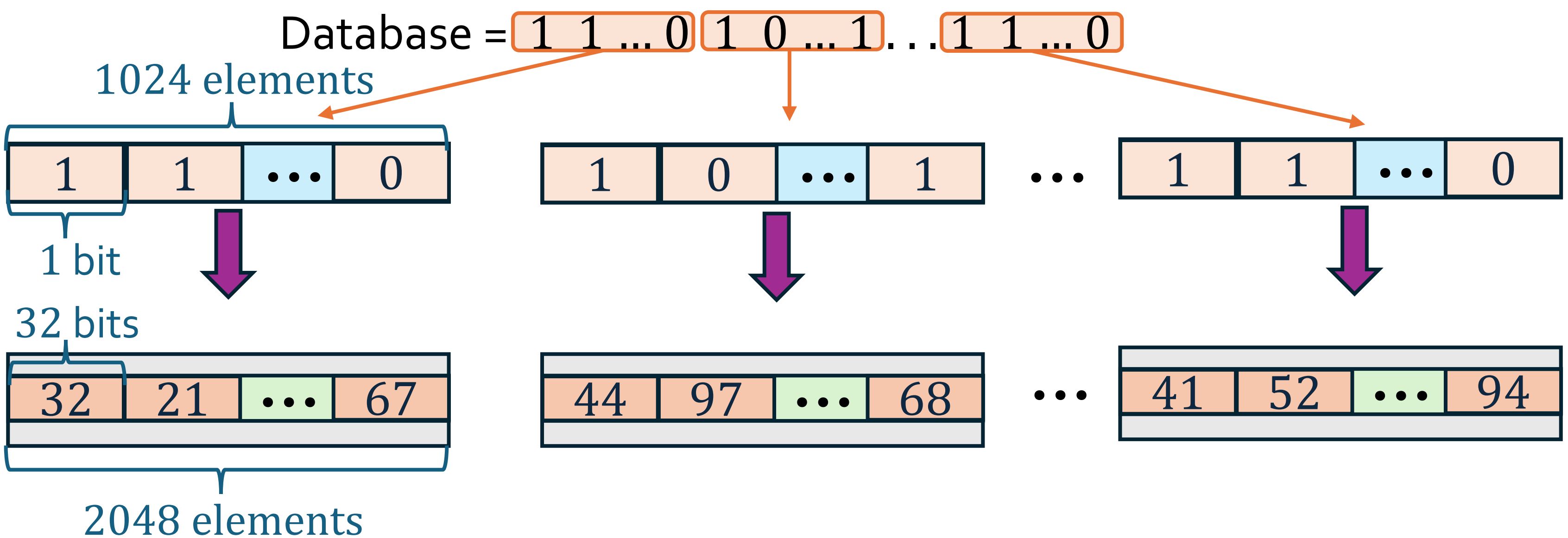
Encrypt **multiple packed bits** and use **homomorphic MUL and ADD operations**

1) Low memory footprint
2) Low computation cost
3) Supports limited query size

Arithmetic Approach

Arithmetic Approach

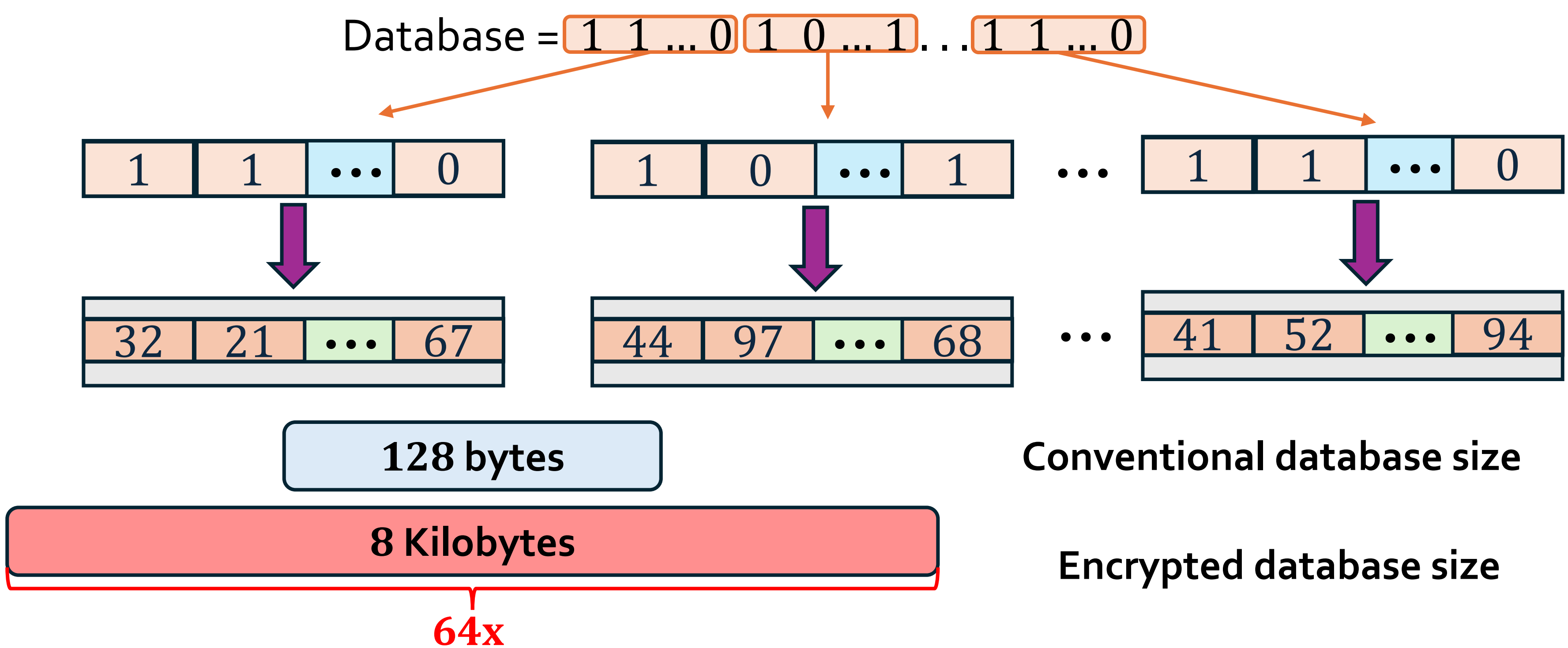
1. Encrypt **multiple packed bits**



Arithmetic Approach

Arithmetic Approach

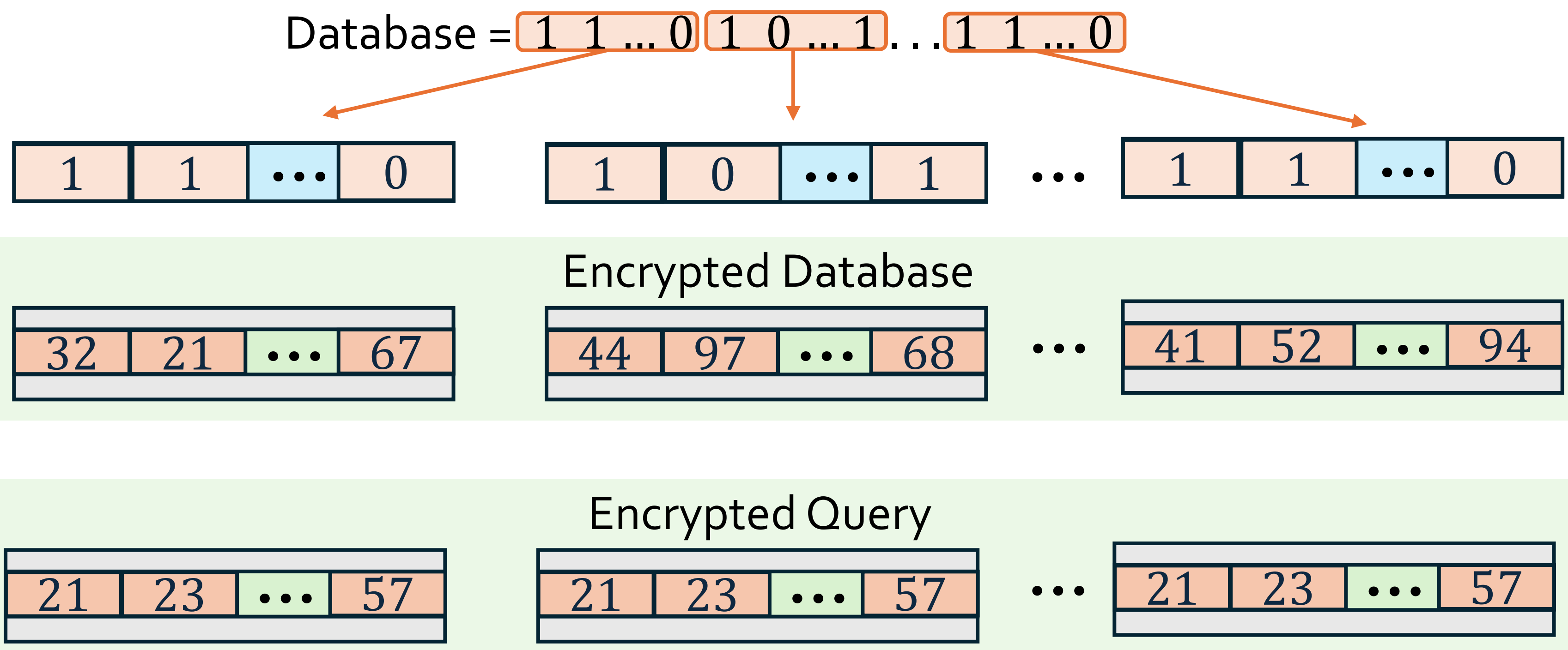
1. Encrypt **multiple packed bits**



Arithmetic Approach

Arithmetic Approach

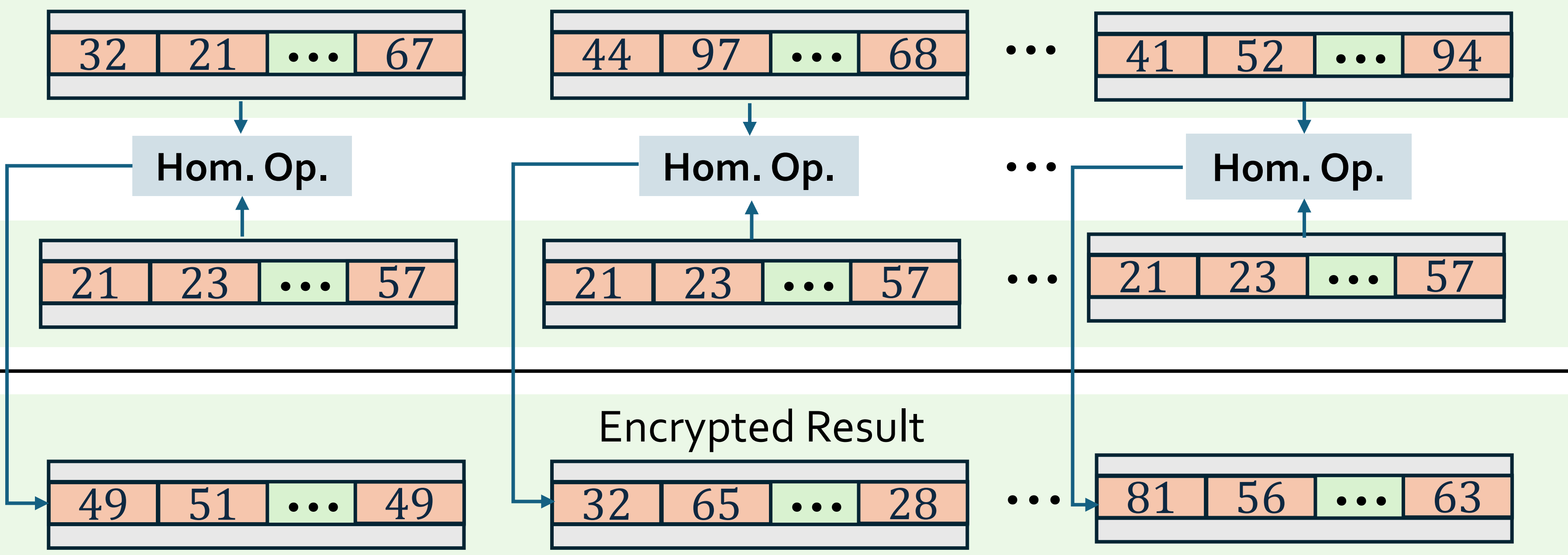
1. Encrypt **multiple packed bits**
2. Perform **homomorphic MUL and ADD** operations



Arithmetic Approach

Arithmetic Approach

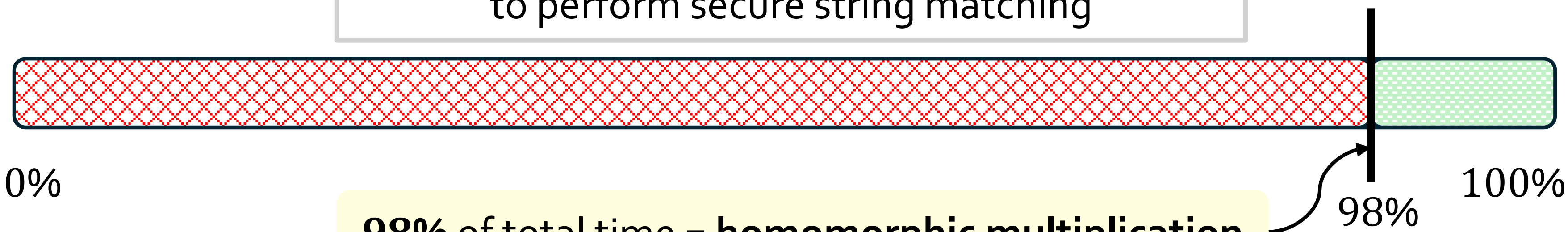
1. Encrypt **multiple packed bits**
2. Perform **homomorphic MUL and ADD** operations



Execution Time of Arithmetic Approach

Arithmetic Approach [Yasuda+, CCSW 2013 ; Kim+, TDSC 2017 ; Bonte+, CCS 2020]

Total **execution time of [Yasuda+, CCSW 2013]** to perform secure string matching



98% of total time = **homomorphic multiplication**

Homomorphic multiplication is **100x slower** than homomorphic addition on a CPU-system

Key Problem (I): Homomorphic multiplication

Arithmetic Approach [Yasuda+, CCSW 2013 ; Kim+, TDSC 2017 ; Bonte+, CCS 2020]

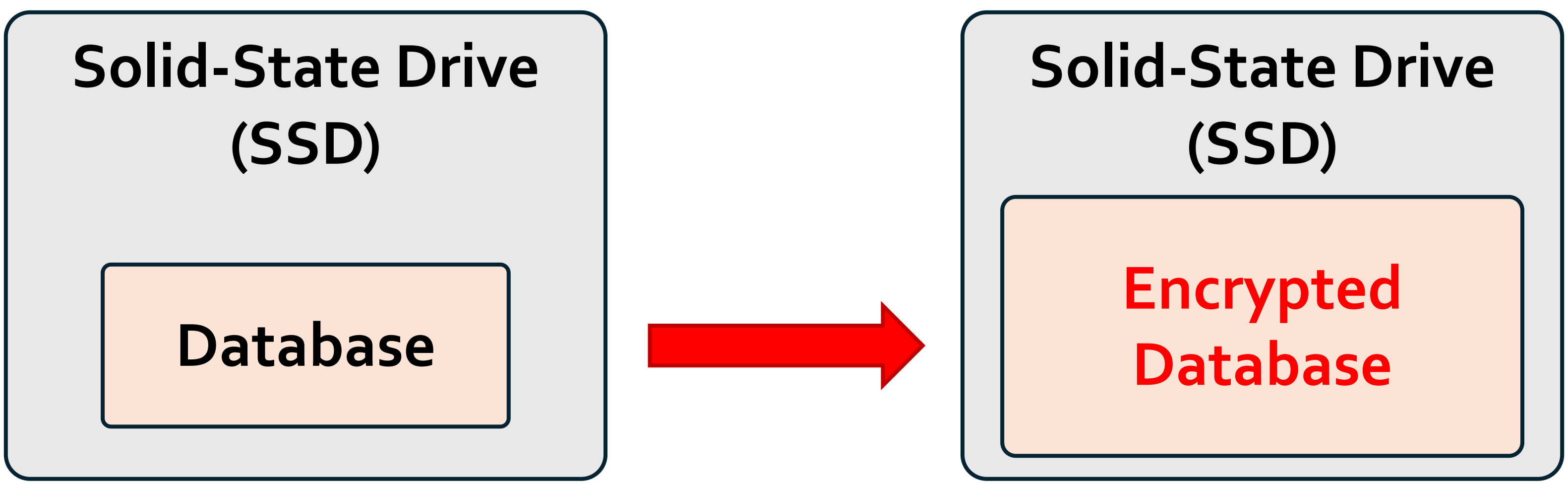
Homomorphic multiplication limits scalability of HE-based string matching algorithm

Homomorphic multiplication is 100x slower than homomorphic addition

Databases are Stored in Storage (SSD)

Databases are **large** and **stored in SSDs**

Homomorphic encryption
further increases the database size

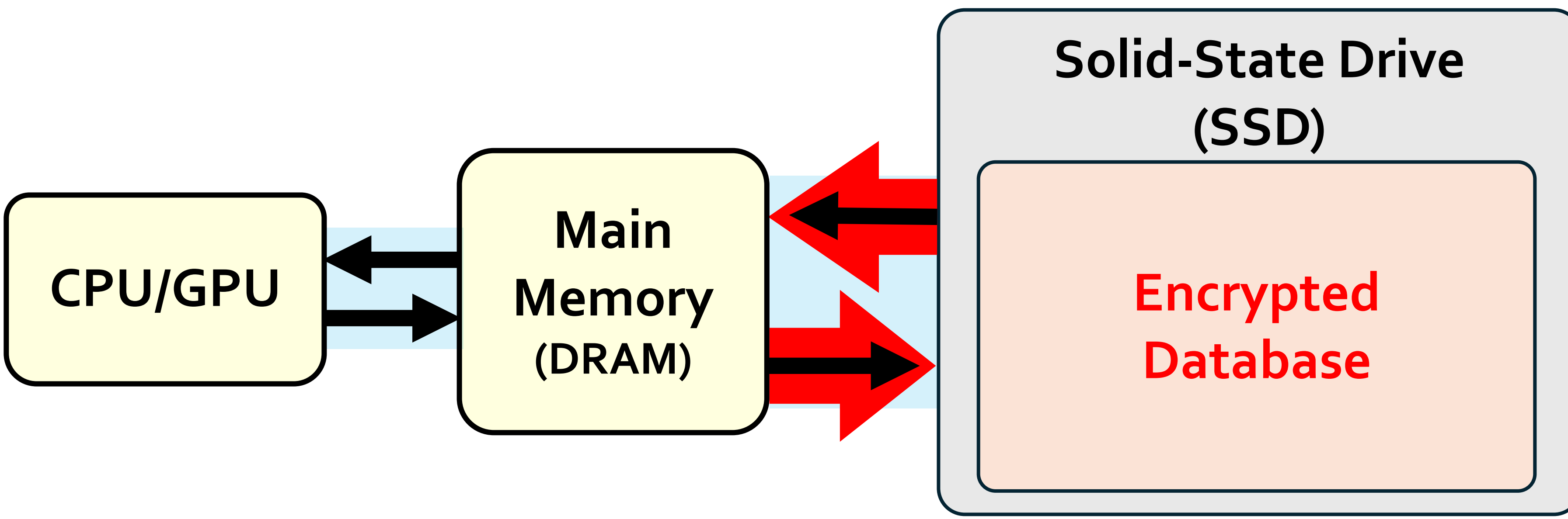


Key Problem (II): Data Movement Bottleneck

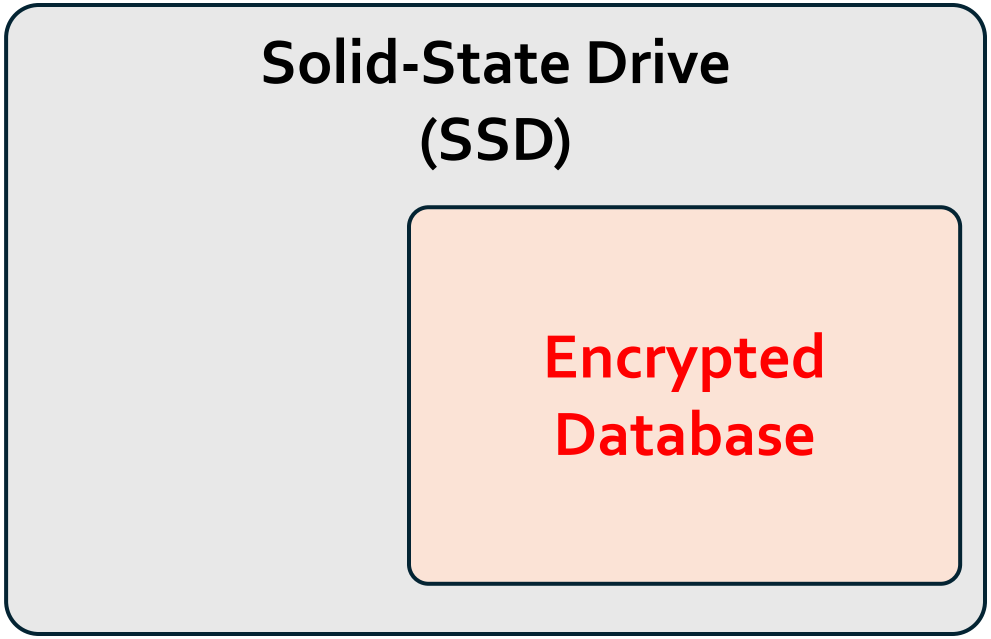
Databases are large and stored in SSDs

External I/O bandwidth of SSD
is the ***main bottleneck*** for reading large encrypted database

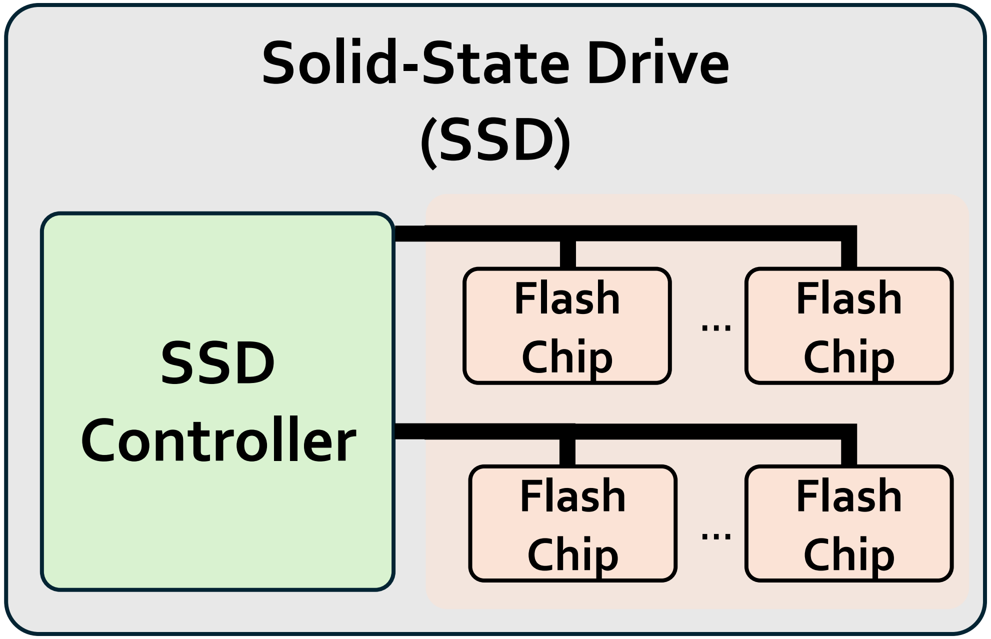
further increases the database size



Prior Works on Reducing Data Movement

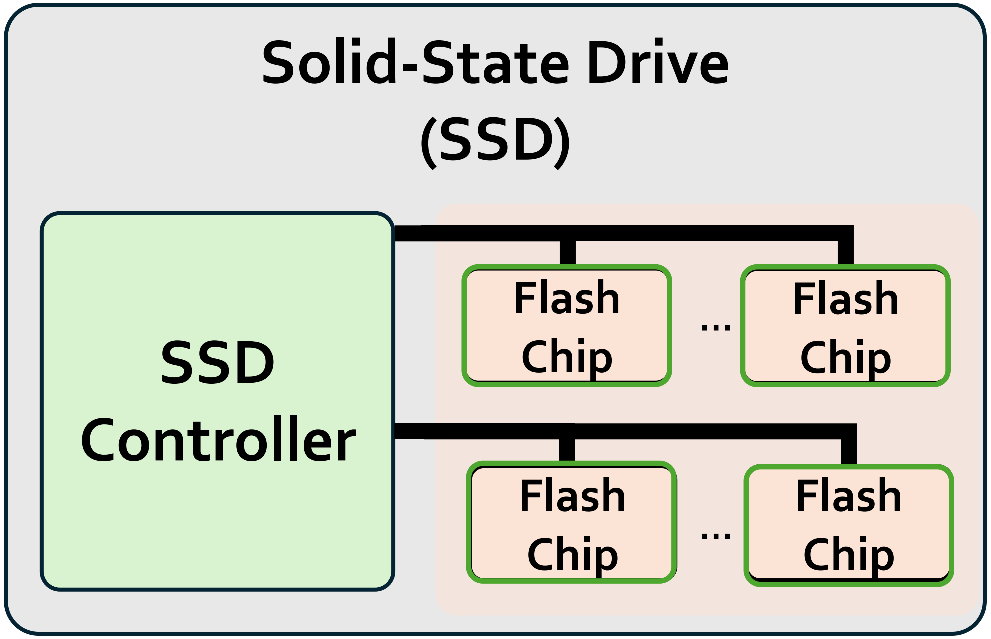


Prior Works on Reducing Data Movement



Prior Works on Reducing Data Movement

In-Flash Processing (IFP) [Park+, MICRO 2022 ; Gao+, MICRO 2021]
enables computation inside SSD by exploiting
the operational principles of NAND-flash memory



Our Goal

Develop an
IFP-based algorithm-hardware co-designed system
that can perform scalable, parallelizable and efficient
secure *exact* string matching

Talk Outline

Background, Problem & Goal

Key Idea

CIPHERMATCH: System Overview

CIPHERMATCH: Algorithm

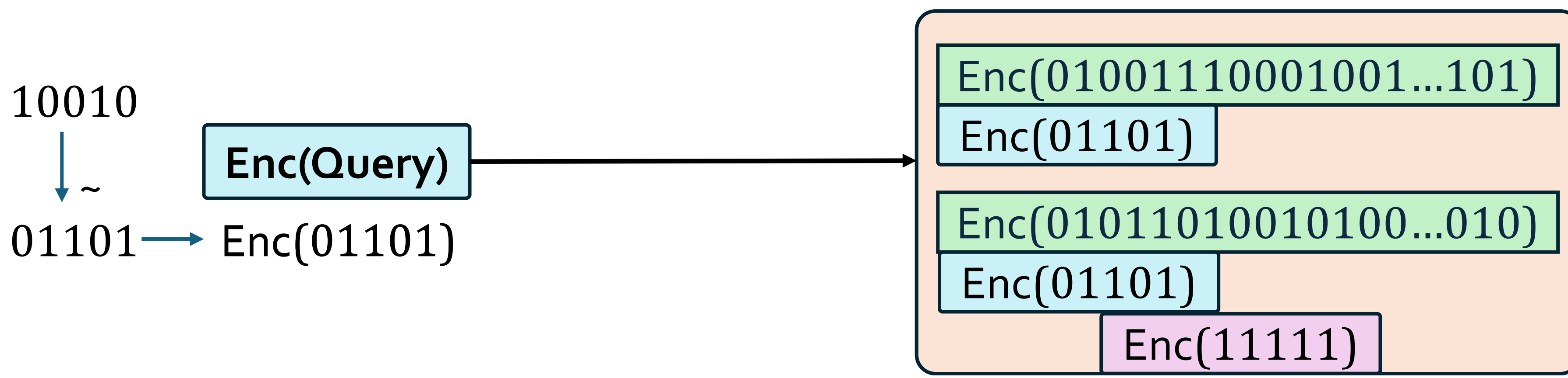
CIPHERMATCH: Hardware

Evaluation Results

Key Observation

In a conventional database,
we perform *only* addition to get a string match

This observation can be extended to perform
secure *exact* string matching
using *only* homomorphic addition



Perform **homomorphic addition**
Observe **Enc(1111's),** if Enc(query) matches

Key Idea (1/2)

Use *only* homomorphic addition to perform secure *exact* string matching

Key Observation

Homomorphic addition is highly parallelizable

Exploit inherent parallelism of NAND-flash memory

- Improves the **performance** of secure string matching
- Reduces **data movement**

Key Idea (2/2)

Use *only* homomorphic addition to perform secure *exact* string matching

Use in-flash processing (IFP) to reduce data movement and accelerate secure *exact* string matching

CIPHERMATCH

An algorithm-hardware co-design

Improves the performance of
HE-based **secure *exact* string matching**

Reduces memory footprint
- by **optimizing** the **data packing scheme** used before encryption

Eliminates costly homomorphic multiplication
- by **designing** secure string-matching algorithm using ***only* homomorphic addition**

Reduces data movement and leverages massive bit and array-level parallelism
- by **designing** an **in-flash processing architecture**

Talk Outline

Background, Problem & Goal

Key Idea

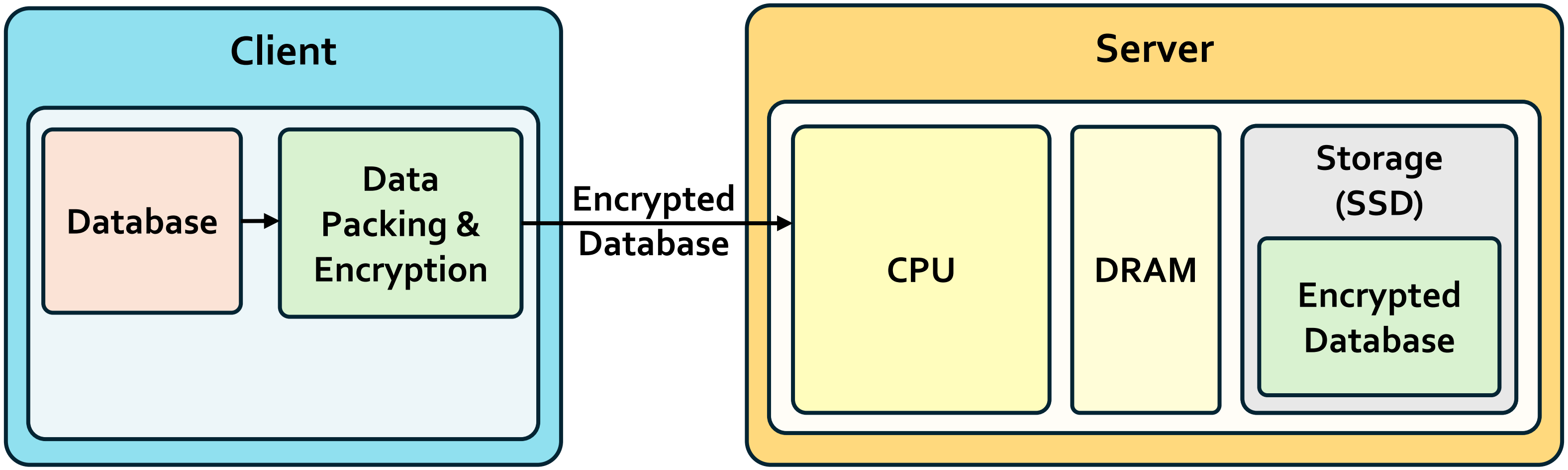
CIPHERMATCH: System Overview

CIPHERMATCH: Algorithm

CIPHERMATCH: Hardware

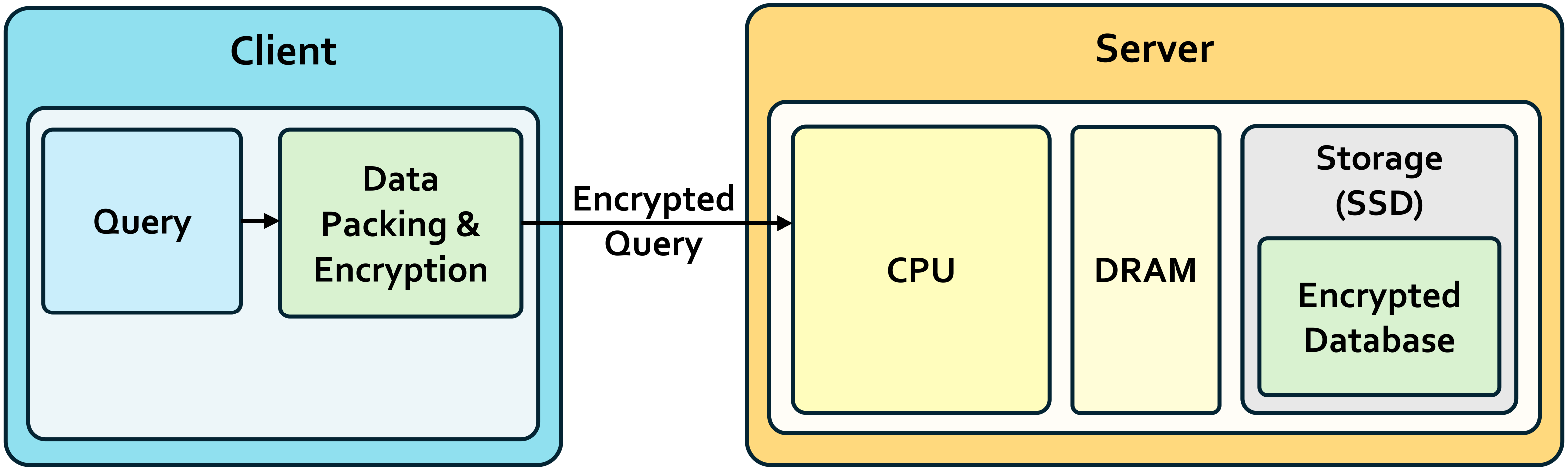
Evaluation Results

CIPHERMATCH: System Overview



Efficiently pack the database
to reduce the memory footprint after encryption

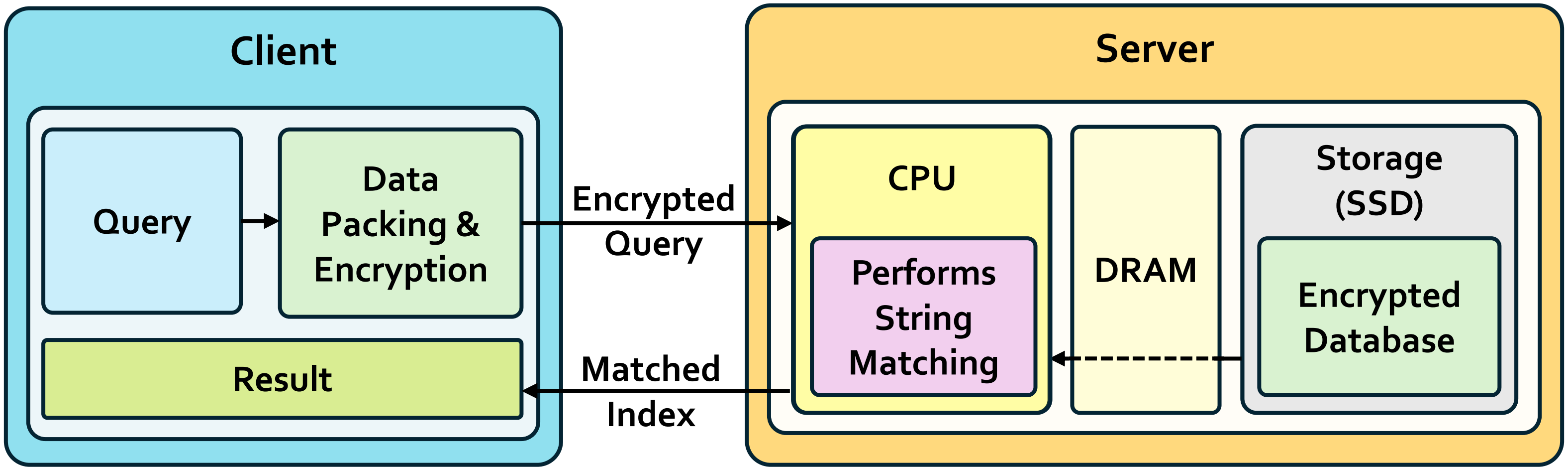
CIPHERMATCH: System Overview



Efficiently pack the database
to reduce the memory footprint after encryption

Efficiently pack the query
to perform parallel secure string matching on encrypted database

CIPHERMATCH: System Overview

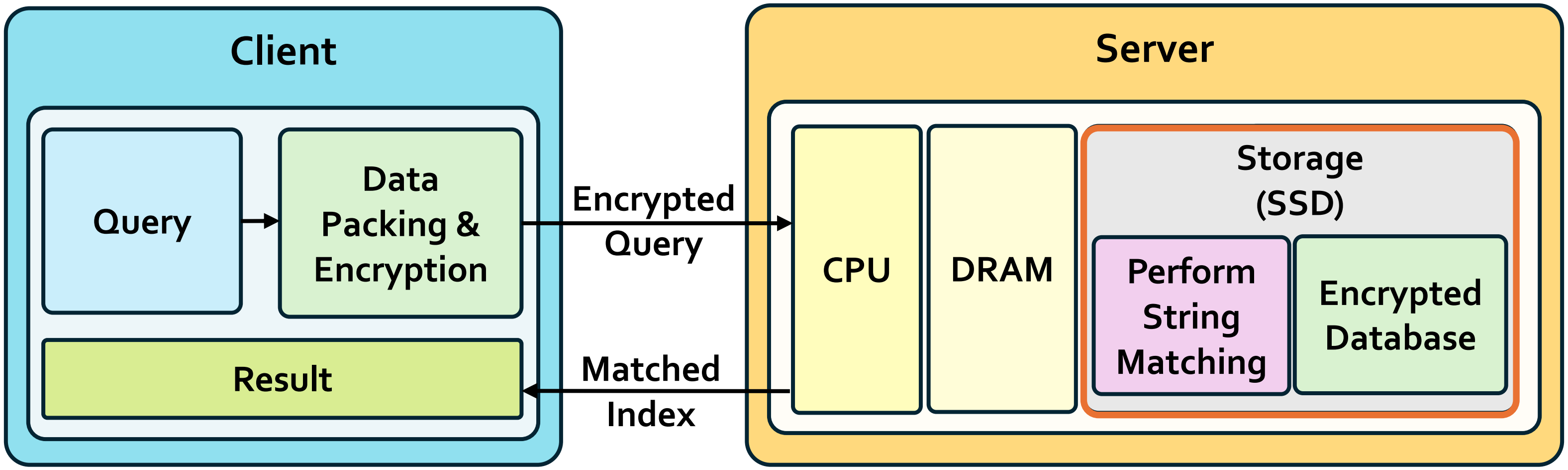


Efficiently pack the database
to reduce the memory footprint after encryption

Efficiently pack the query
to perform parallel secure string matching on encrypted database

Perform secure *exact* string matching
using *only* homomorphic addition

CIPHERMATCH: System Overview



Efficiently pack the database
to reduce the memory footprint after encryption

Efficiently pack the query
to perform parallel secure string matching on encrypted database

Perform secure *exact* string matching
using *only* homomorphic addition

Accelerate secure *exact* string matching by performing computations inside SSD
by exploiting operational principles of NAND-flash memory

CIPHERMATCH: Key Steps

1 Memory-Efficient Data Packing Scheme

2 Secure *Exact* String-Matching Algorithm

3 In-Flash Processing

Talk Outline

Background, Problem & Goal

Key Idea

CIPHERMATCH: System Overview

CIPHERMATCH: Algorithm

CIPHERMATCH: Hardware

Evaluation Results

CIPHERMATCH: Key Steps

1

Memory-Efficient Data Packing Scheme

Memory-Efficient Data Packing Scheme

- **Efficiently pack the query and database** to reduce the memory footprint and enable parallel string matching

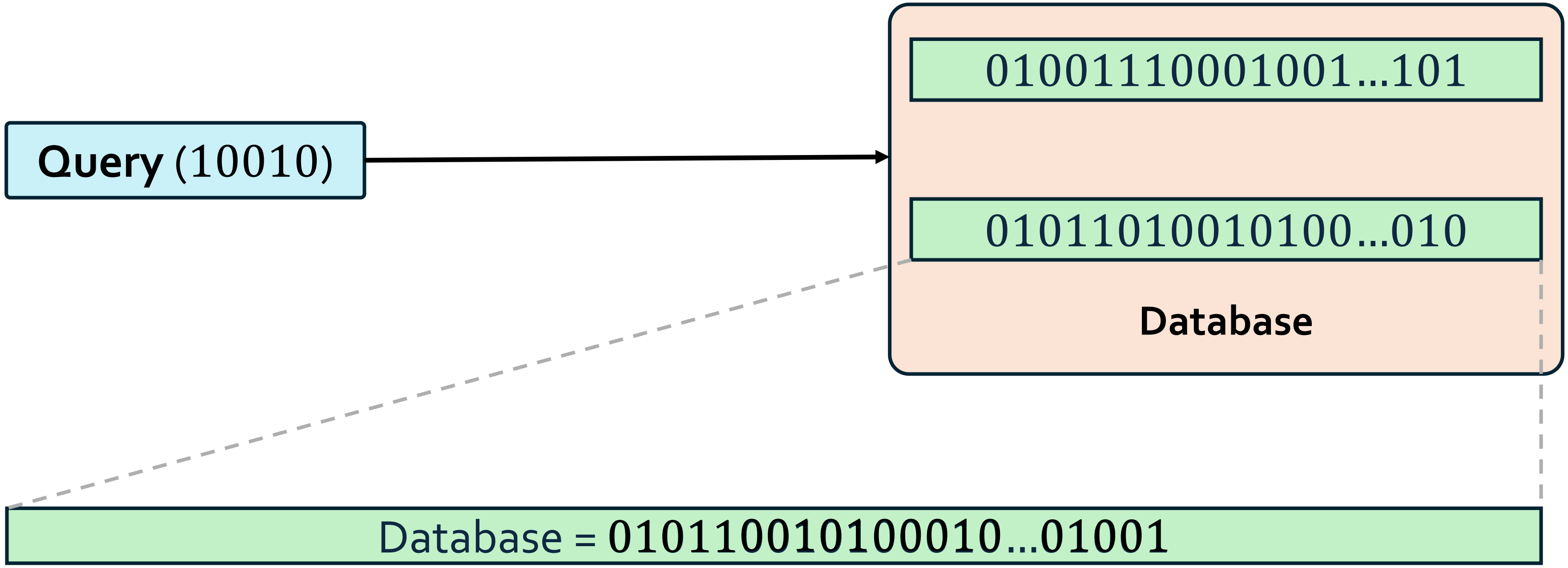
2

Secure *Exact* String-Matching Algorithm

3

In-Flash Processing

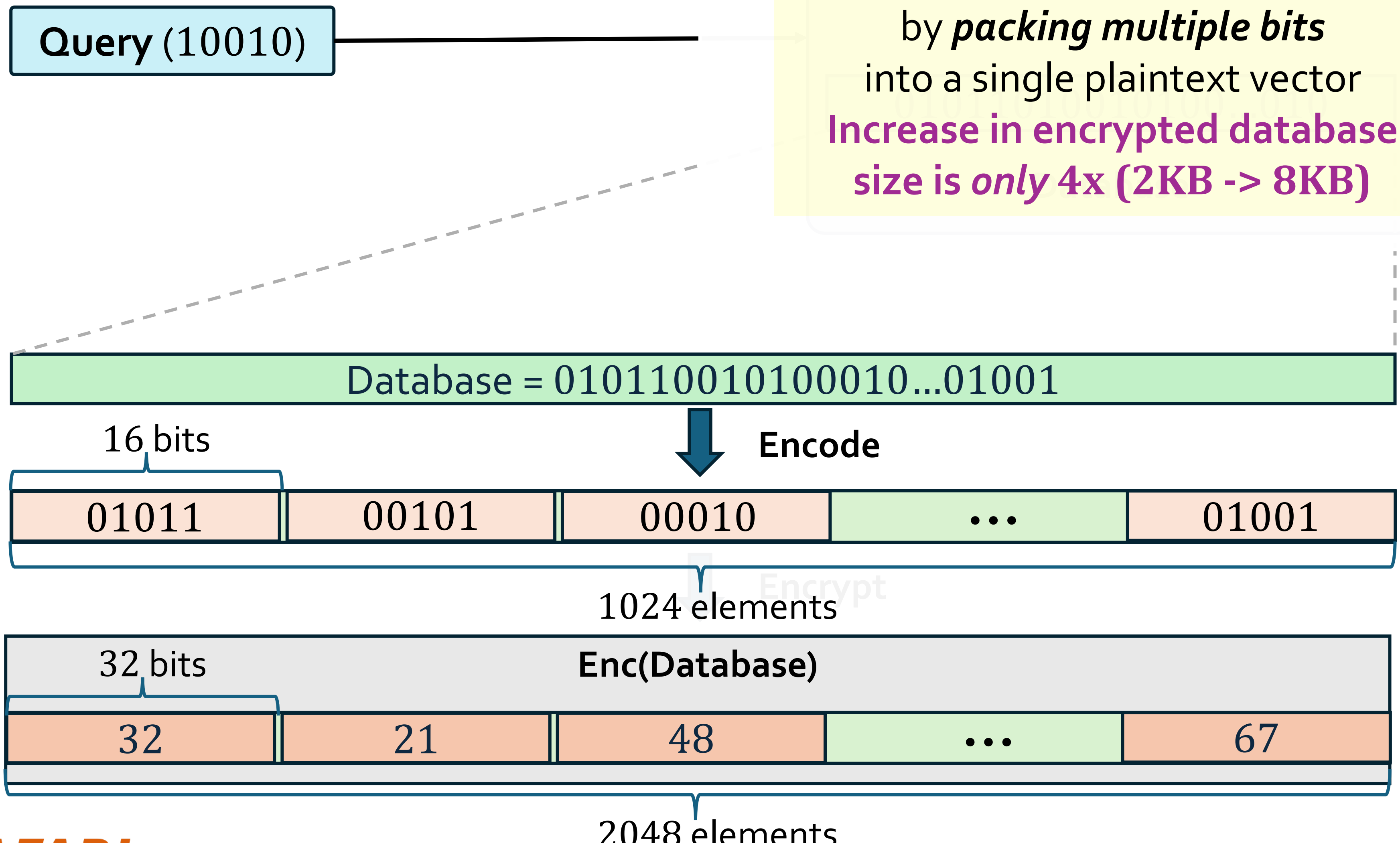
Memory-Efficient Data Packing Scheme ①



Memory-Efficient Data Packing Scheme

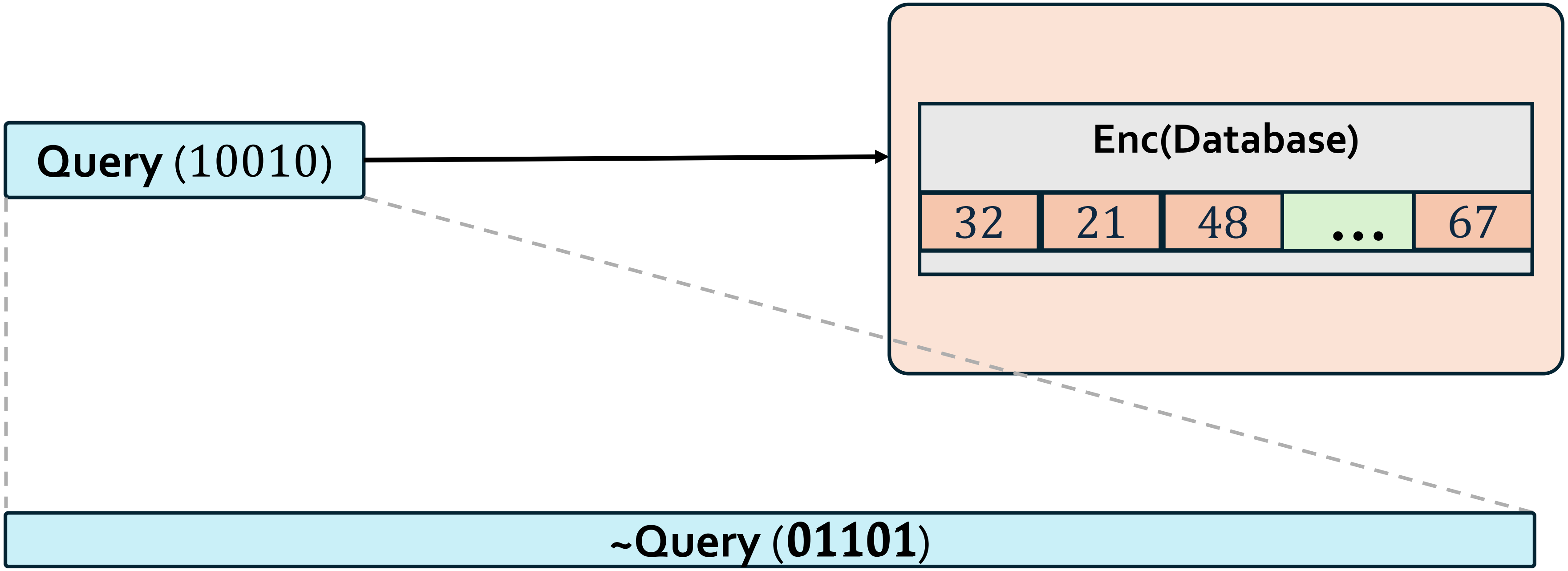
1

Encode database
by ***packing multiple bits***
into a single plaintext vector
Increase in encrypted database size is *only* 4x (2KB -> 8KB)



Memory-Efficient Data Packing Scheme

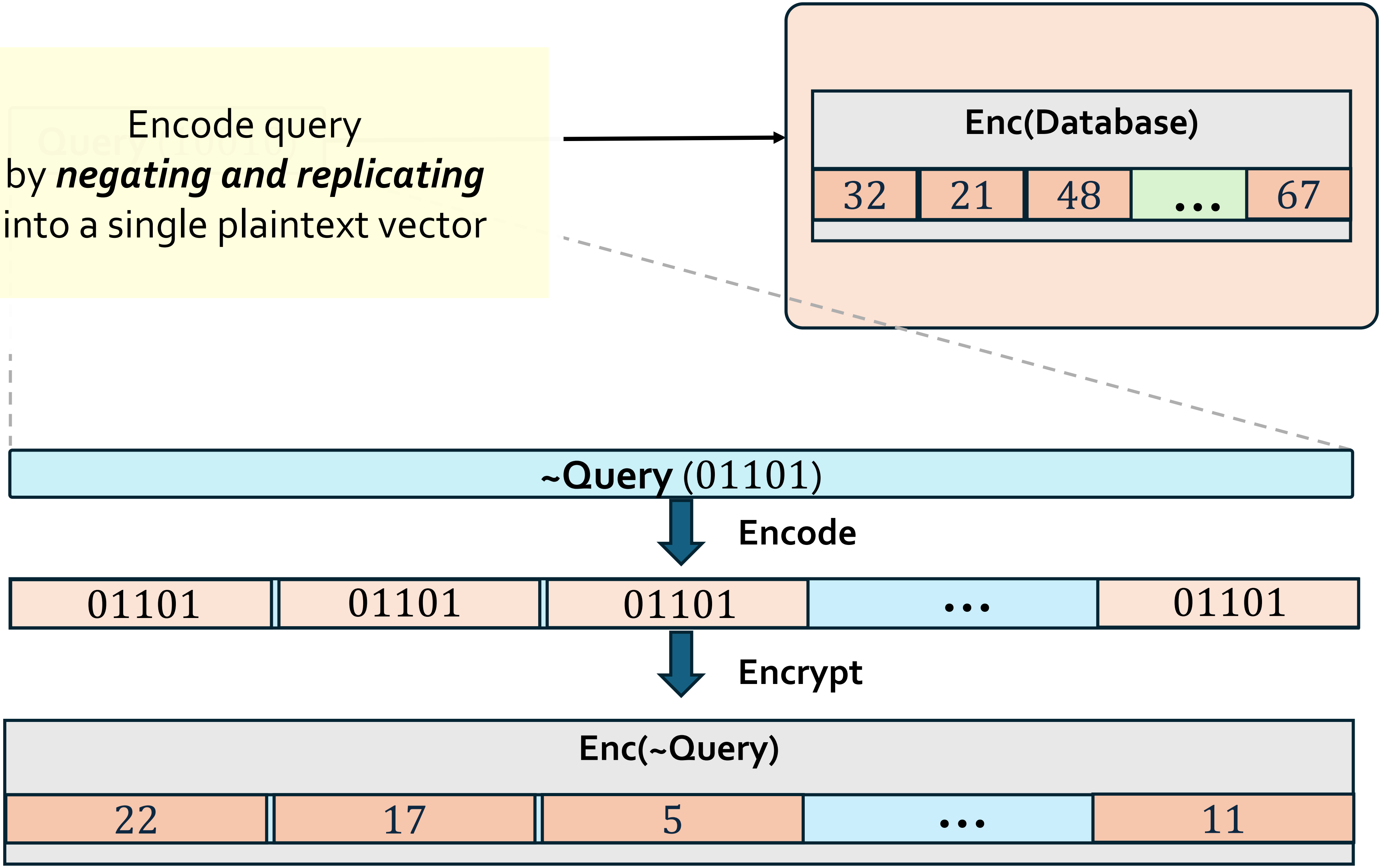
1



Memory-Efficient Data Packing Scheme

1

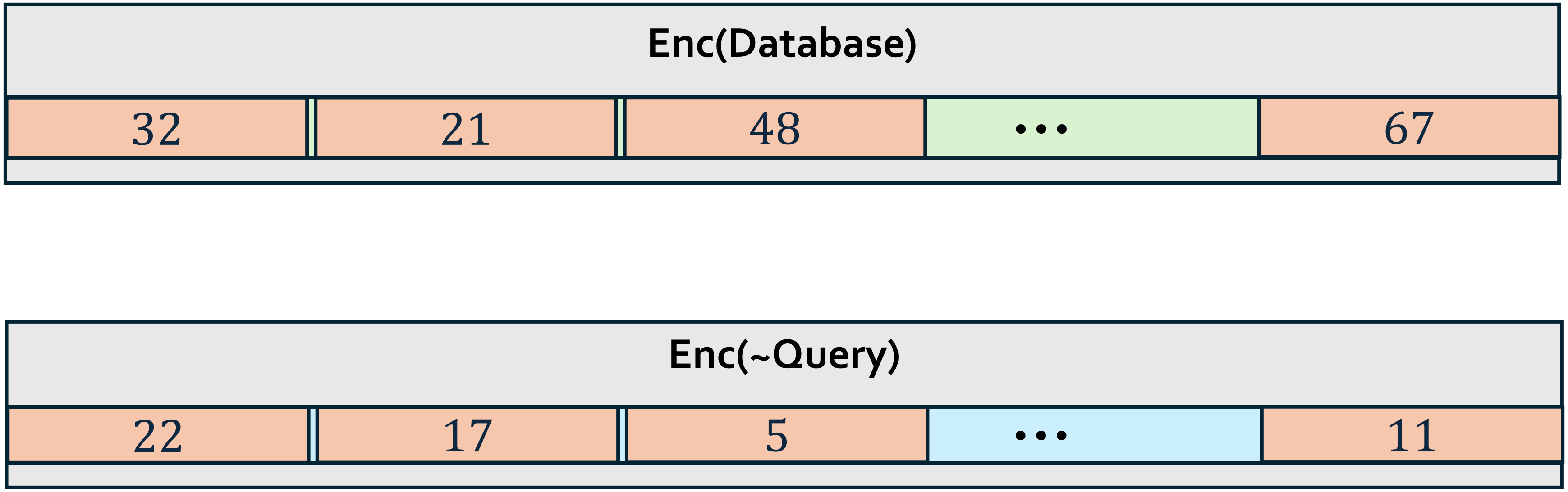
Encode query
by ***negating and replicating***
into a single plaintext vector



SAFARI

Memory-Efficient Data Packing Scheme

1



CIPHERMATCH: Key Steps

1

Memory-Efficient Data Packing Scheme

- Efficiently pack the query and database
to reduce the memory footprint and enable parallel string matching

2

Secure *Exact* String-Matching Algorithm

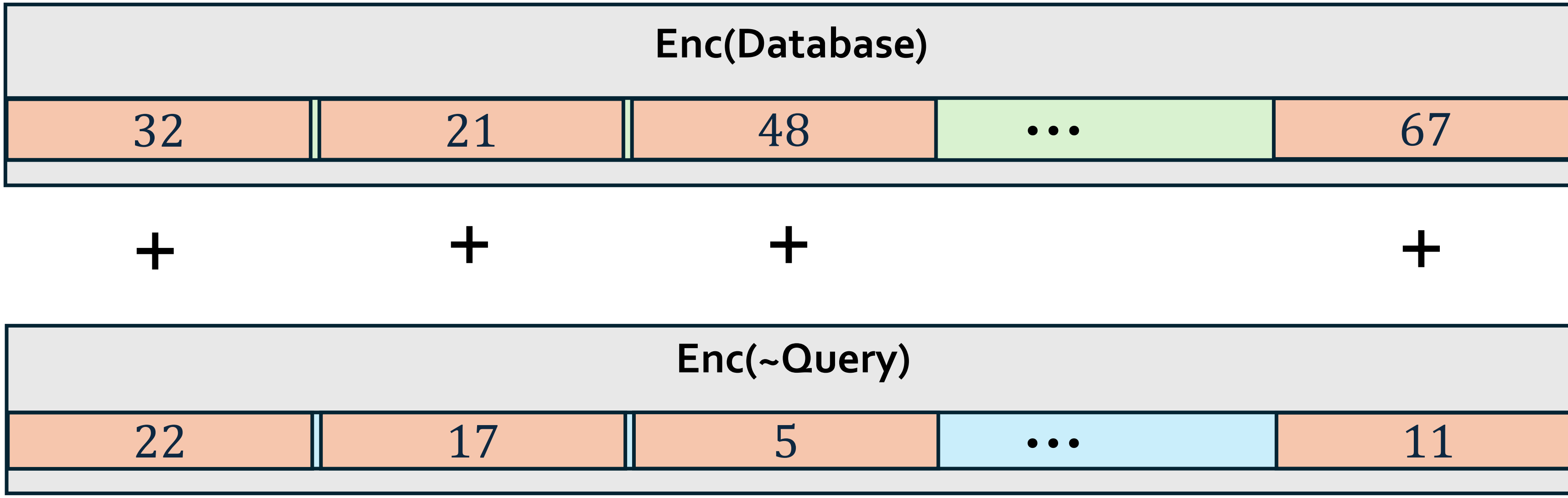
- **Uses *only* homomorphic addition and identifies the match**
to eliminate costly homomomorphic multiplication

3

In-Flash Processing

Secure Exact String-Matching Algorithm

2



Homomorphic addition

is inherently **element-wise addition**

Secure Exact String-Matching Algorithm

2

Enc(Database)

32	21	48	…	67

\+ \+ \+ \+

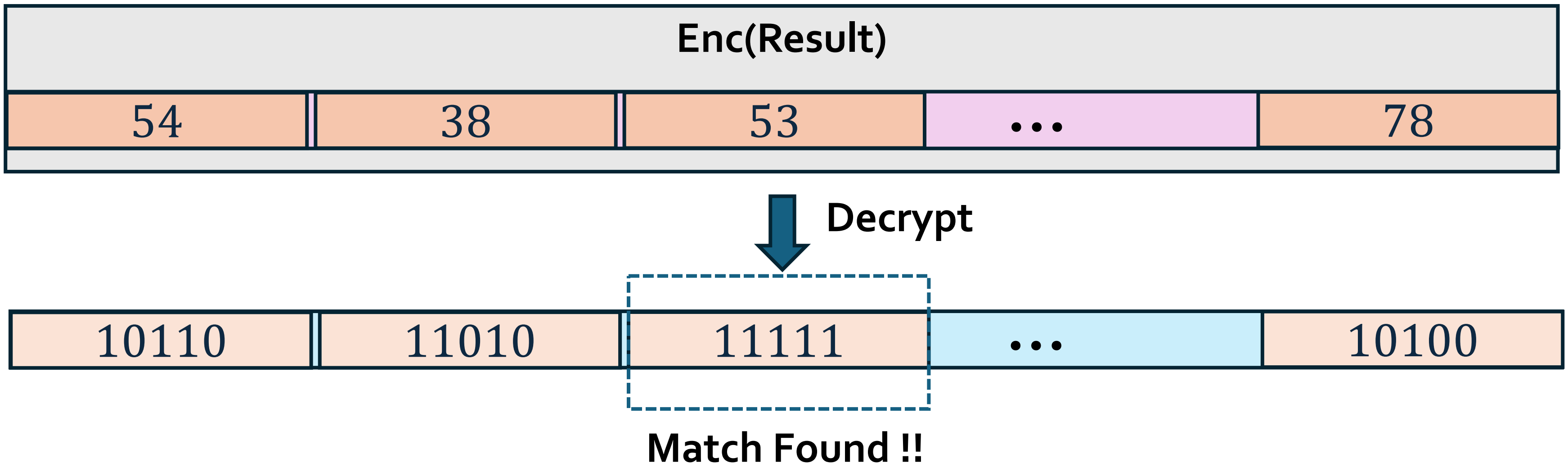
Enc(~Query)

22	17	5	…	11

Enc(Result)

54	38	53	…	78

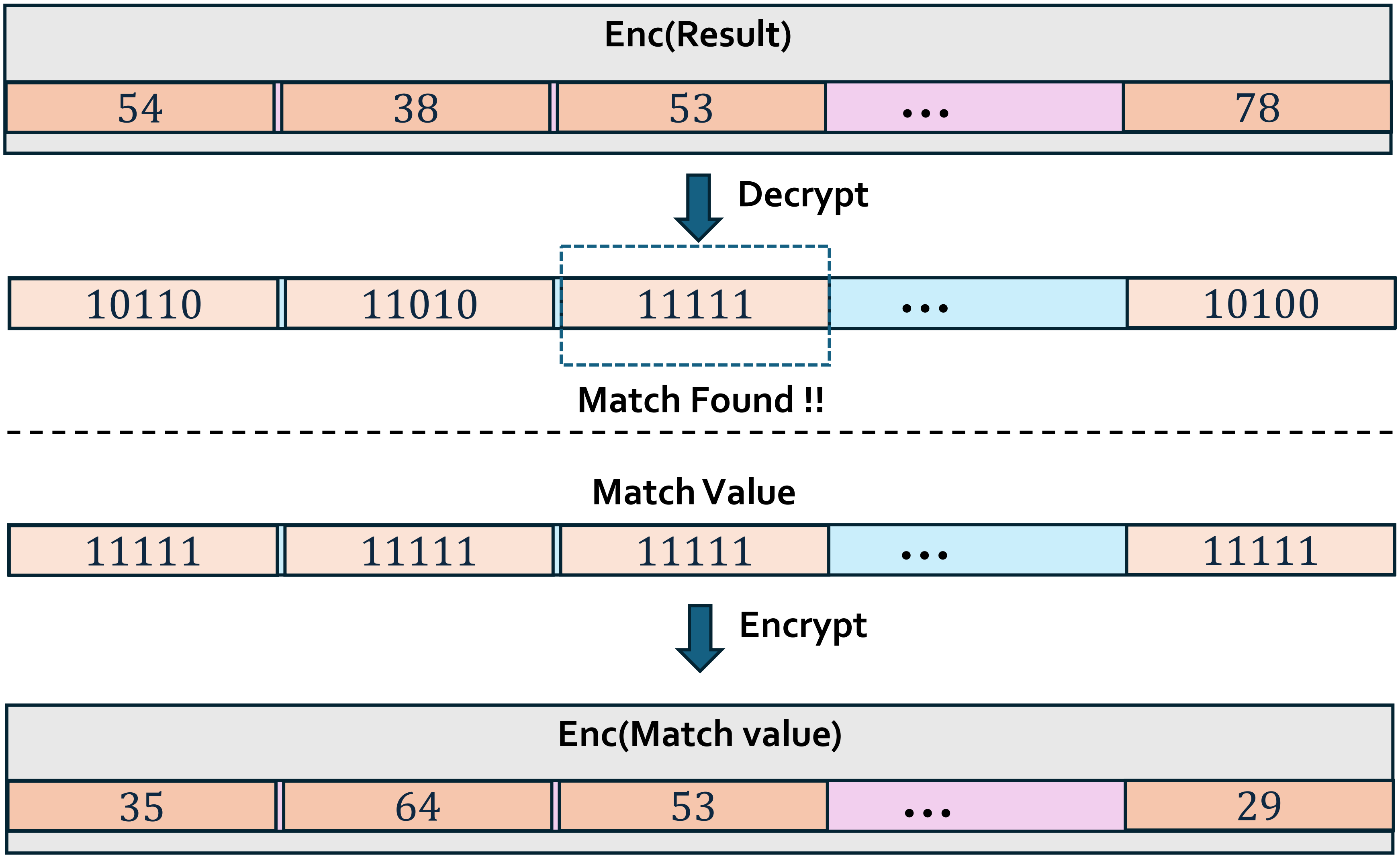
Secure Exact String-Matching Algorithm ②



However, we want to find the **match using Enc(Result)** on the server

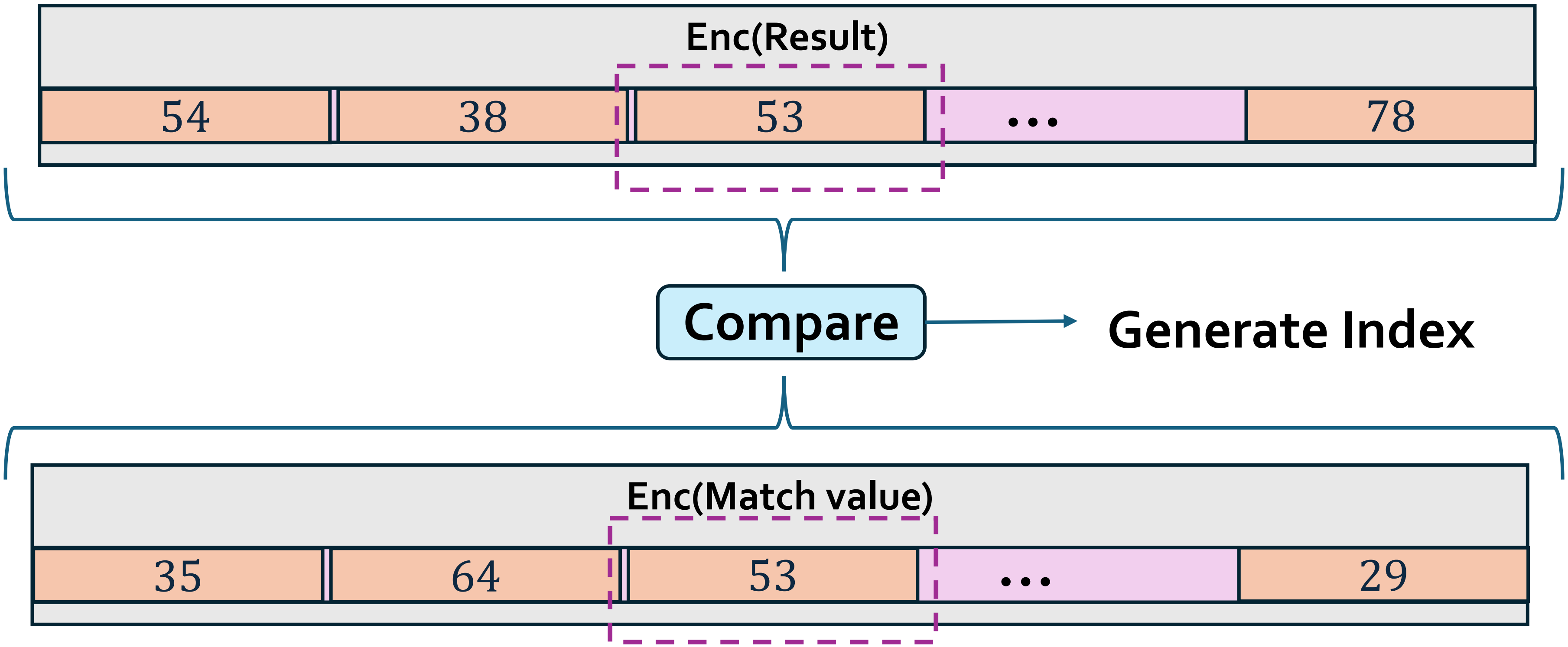
Identify the Match

2



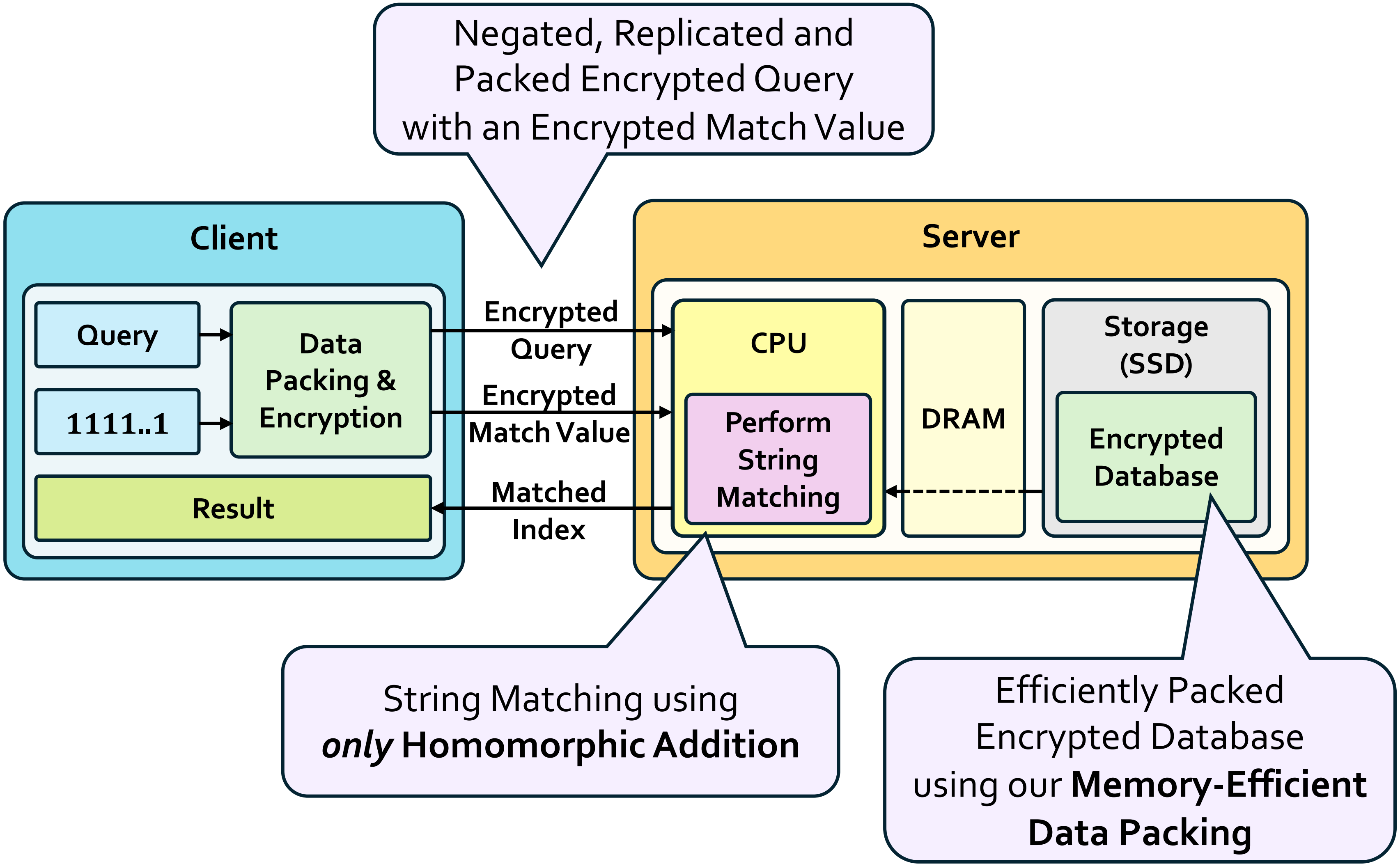
Identify the Match

2



Compare and **send the final index** back to client

CIPHERMATCH: Algorithm (Summary)



Talk Outline

Background, Problem & Goal

Key Idea

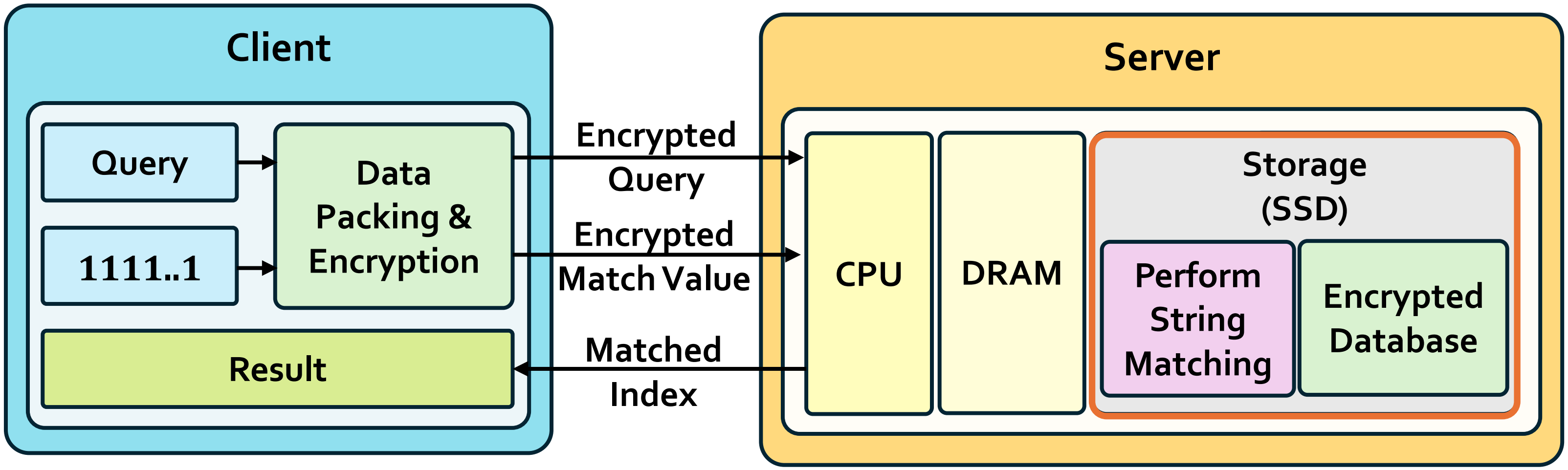
CIPHERMATCH: System Overview

CIPHERMATCH: Algorithm

CIPHERMATCH: Hardware

Evaluation Results

CIPHERMATCH: Hardware Overview



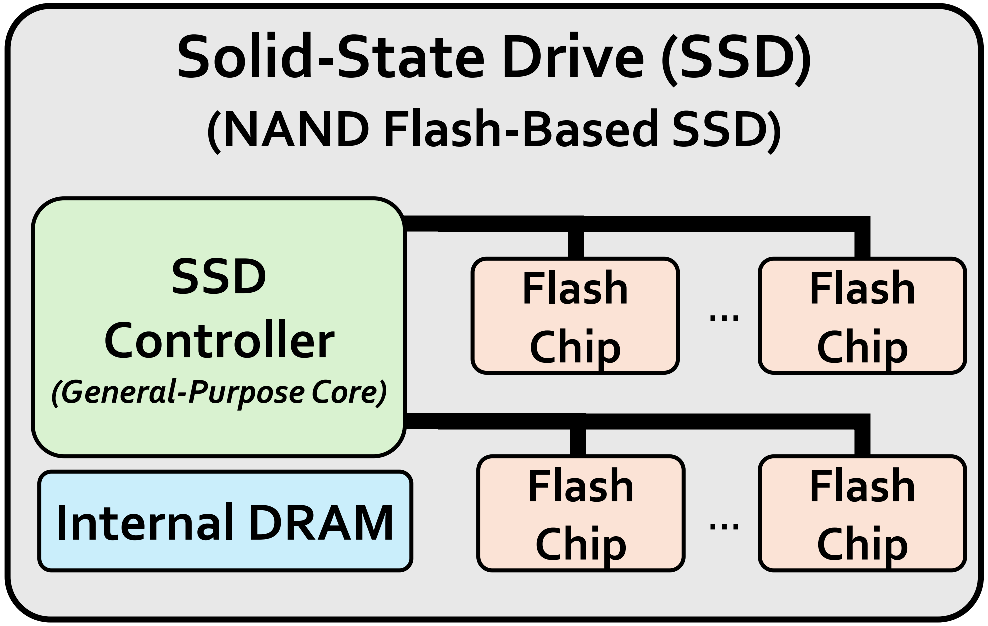
Perform secure *exact* string matching inside SSD using in-flash processing (IFP)

Overview of a Modern Solid State Drive (SSD)

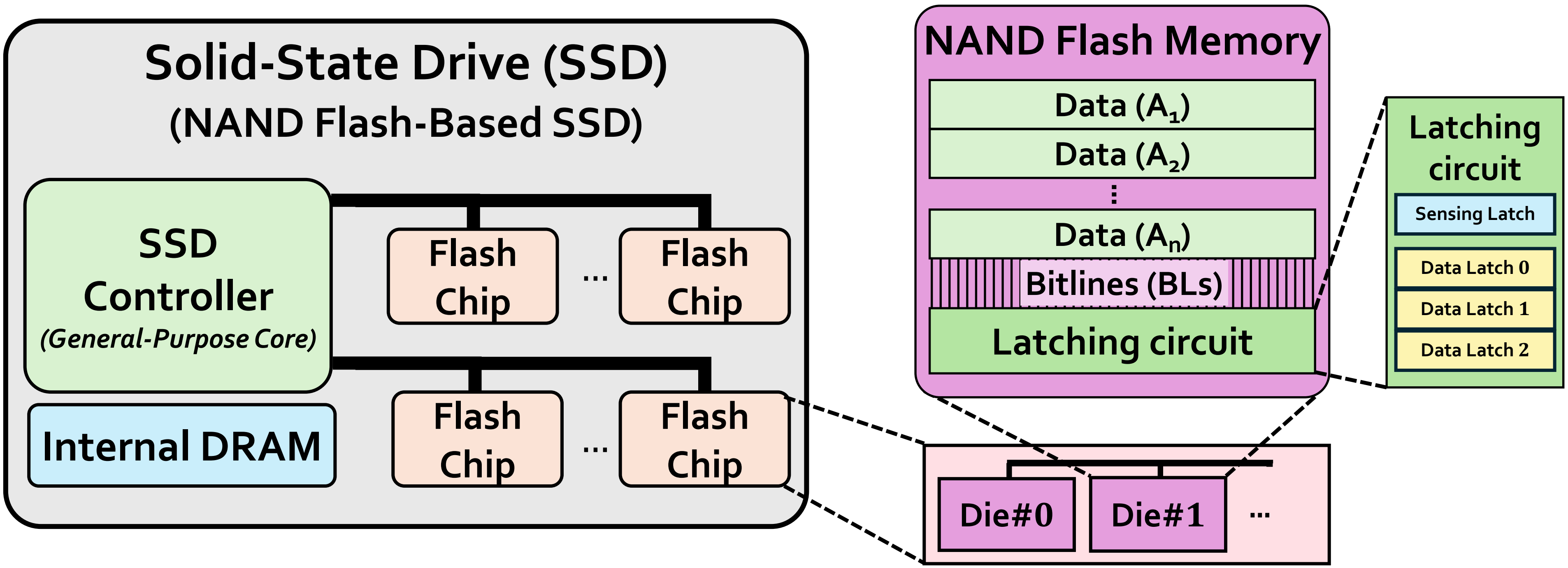
Solid-State Drive (SSD)

(NAND Flash-Based SSD)

Overview of a Modern Solid State Drive (SSD)

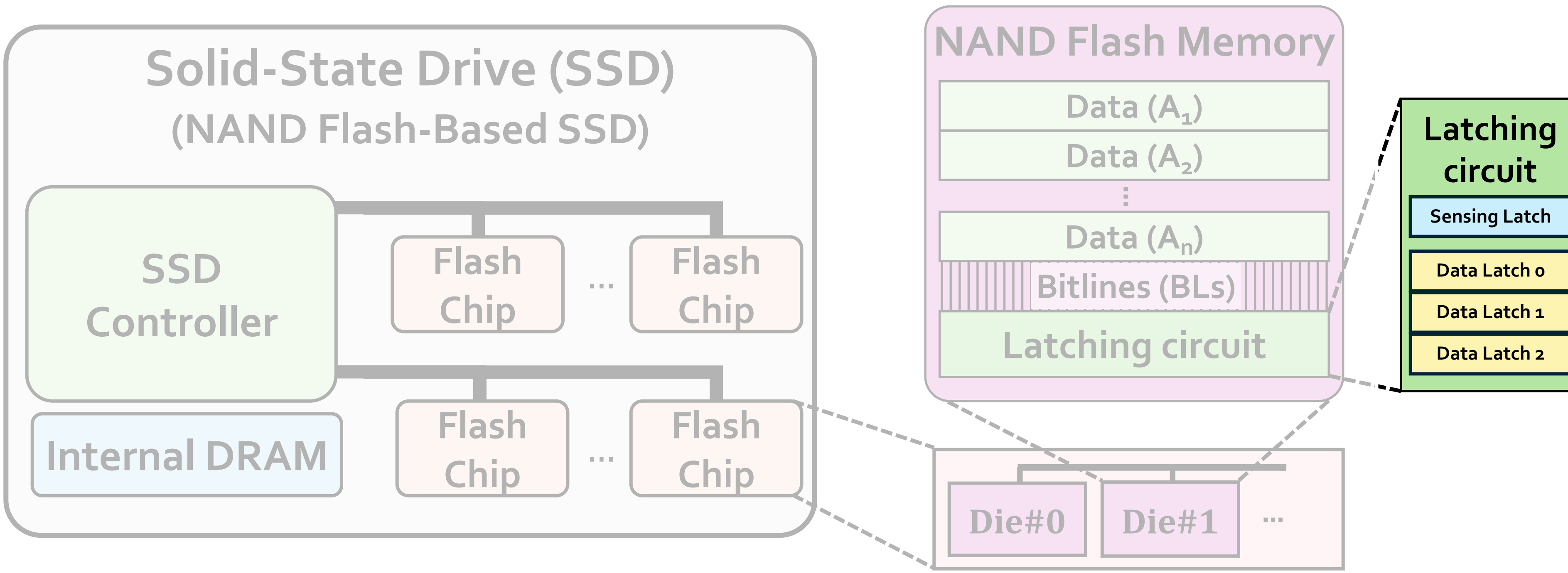


Overview of a Modern Solid State Drive (SSD)

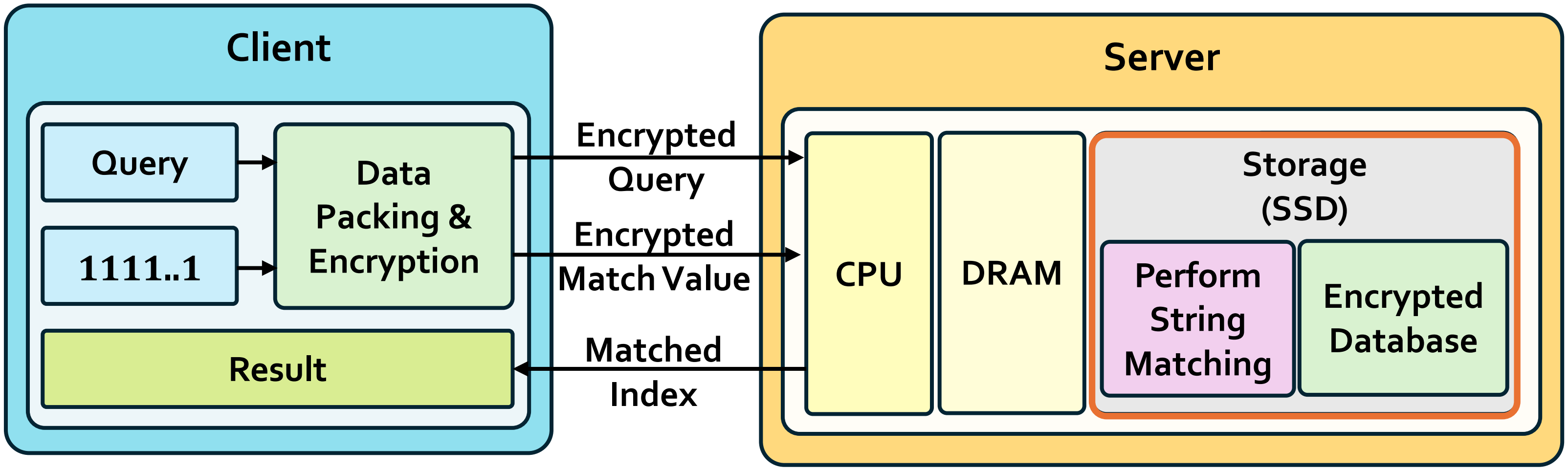


Limitations of Prior Work

Prior work [Gao+, MICRO 2021] **uses latching circuit to perform *only* bitwise operations**



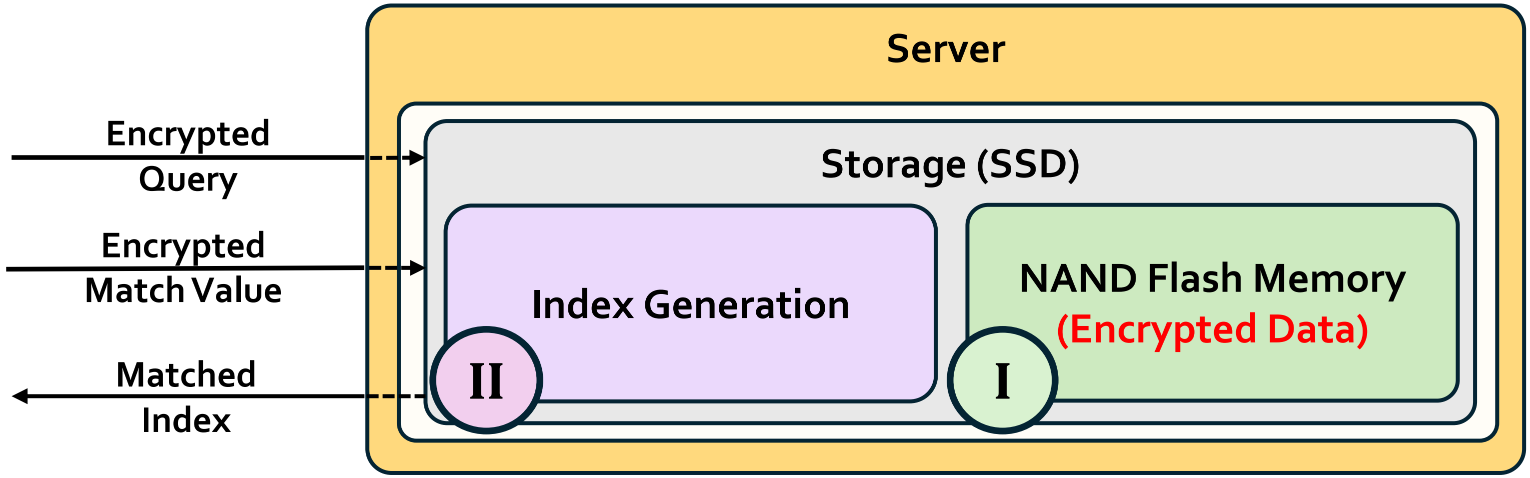
Advantages of Secure String Matching in SSD



Homomorphic addition can be **parallelized**

Exploit **bit-level and array-level parallelism** of NAND-flash memory

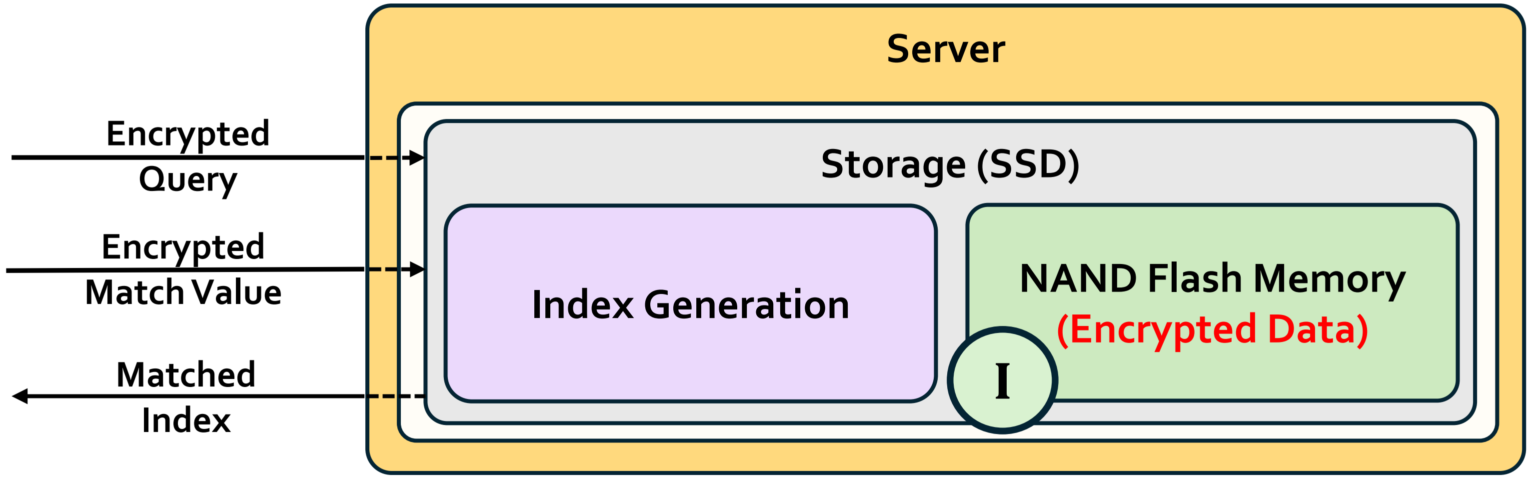
CIPHERMATCH: Hardware Overview



I Perform **homomorphic additions** inside **NAND-flash memory**

II Generate the **final index** by **comparing it with match value**

CIPHERMATCH: Hardware Overview



I Perform **homomorphic additions** inside **NAND-flash memory**

Perform **element-wise addition** inside NAND-flash memory

CIPHERMATCH: Key Steps

1

Memory-Efficient Data Packing Scheme

- Efficiently pack the query and database to reduce the memory footprint and enable parallel string matching

2

Secure *Exact* String-Matching Algorithm

- Uses *only* homomorphic addition and identifies the match to eliminate costly homomomorphic multiplication

3

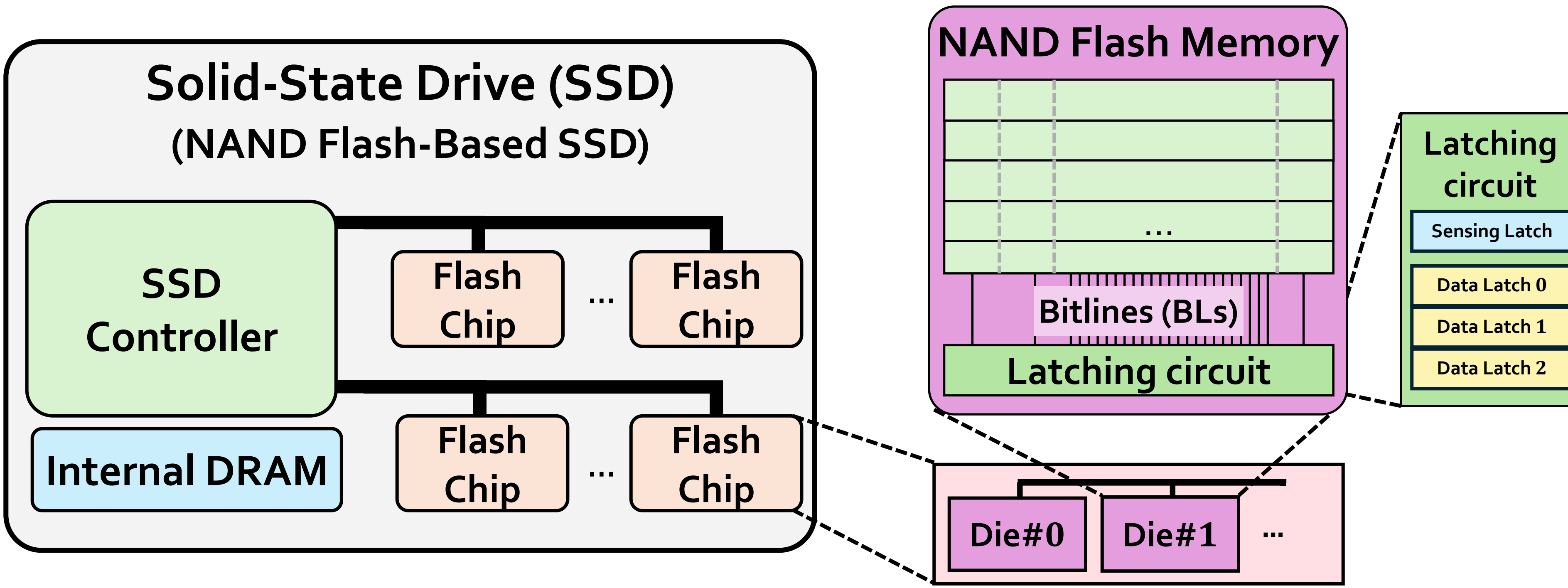
In-Flash Processing

-Exploit the operational principles of NAND-flash memory **to perform homomorphic addition**

CIPHERMATCH: Element-Wise Addition

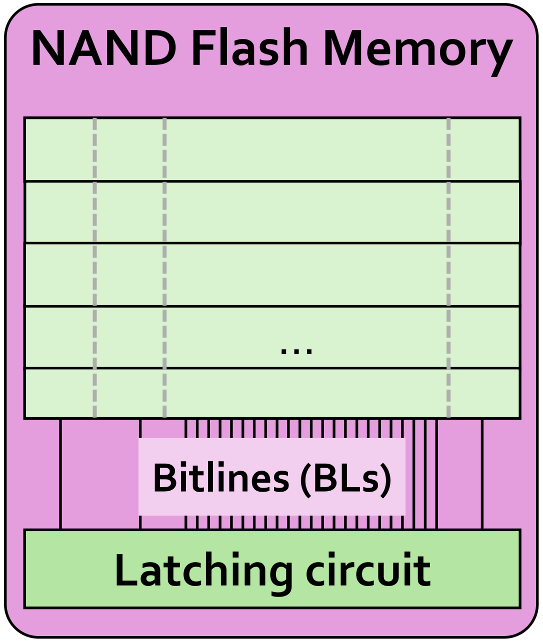
3

We use **bit-serial addition**
to **avoid carry propagation** across different bitlines

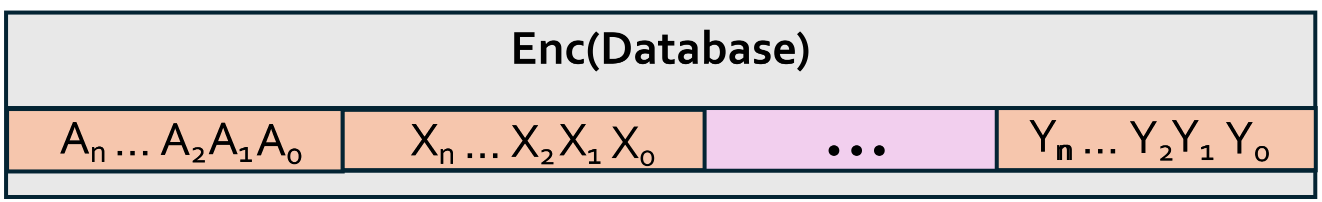


CIPHERMATCH: Bit-Serial Addition

3

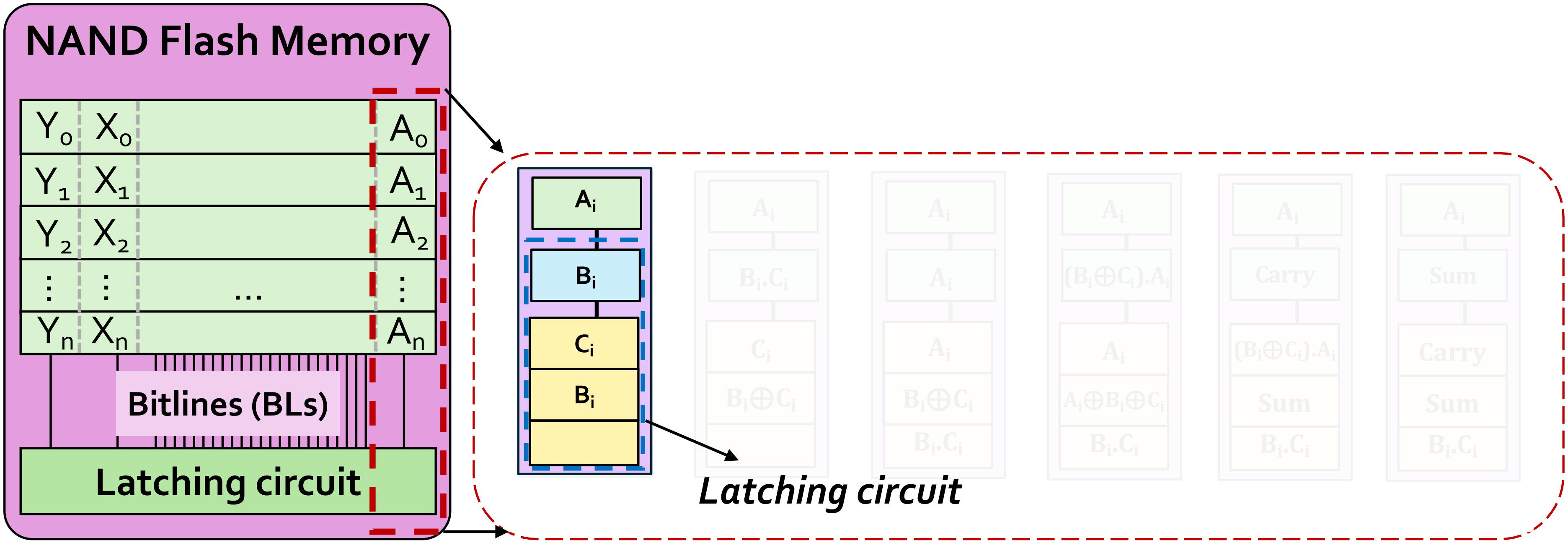


Lay out the data vertically in NAND-flash memory



CIPHERMATCH: Bit-Serial Addition

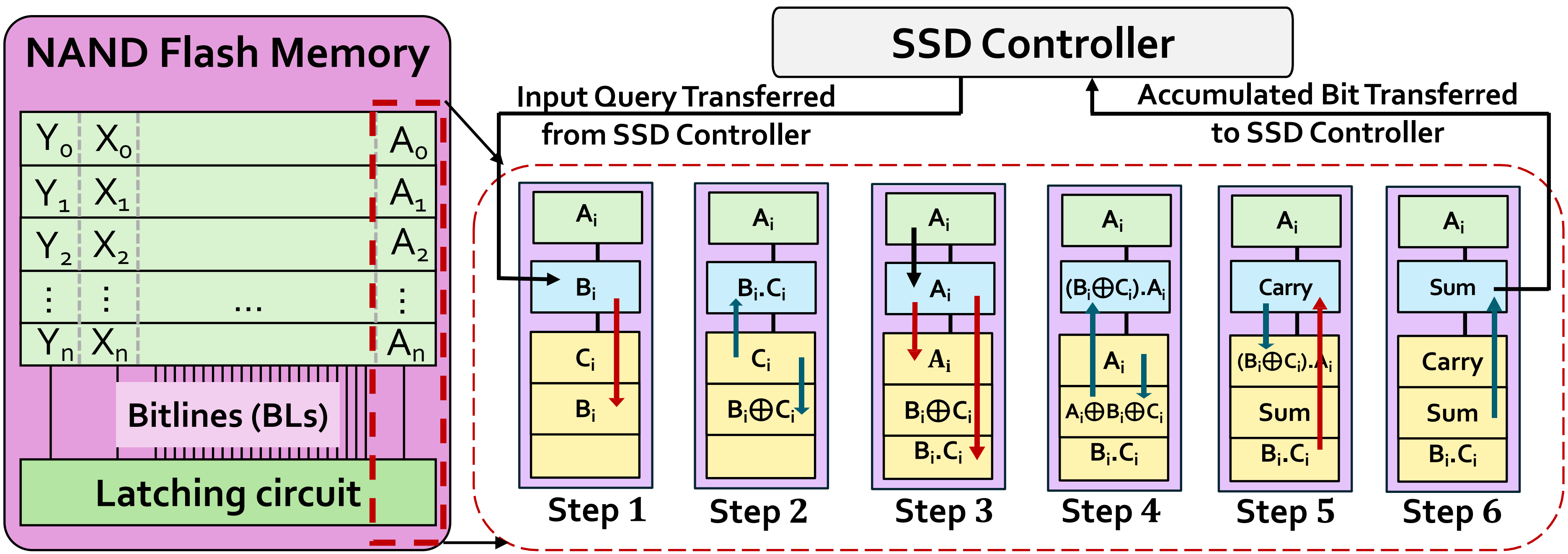
3



Lay out the data vertically in NAND-flash memory

CIPHERMATCH: Bit-Serial Addition

3



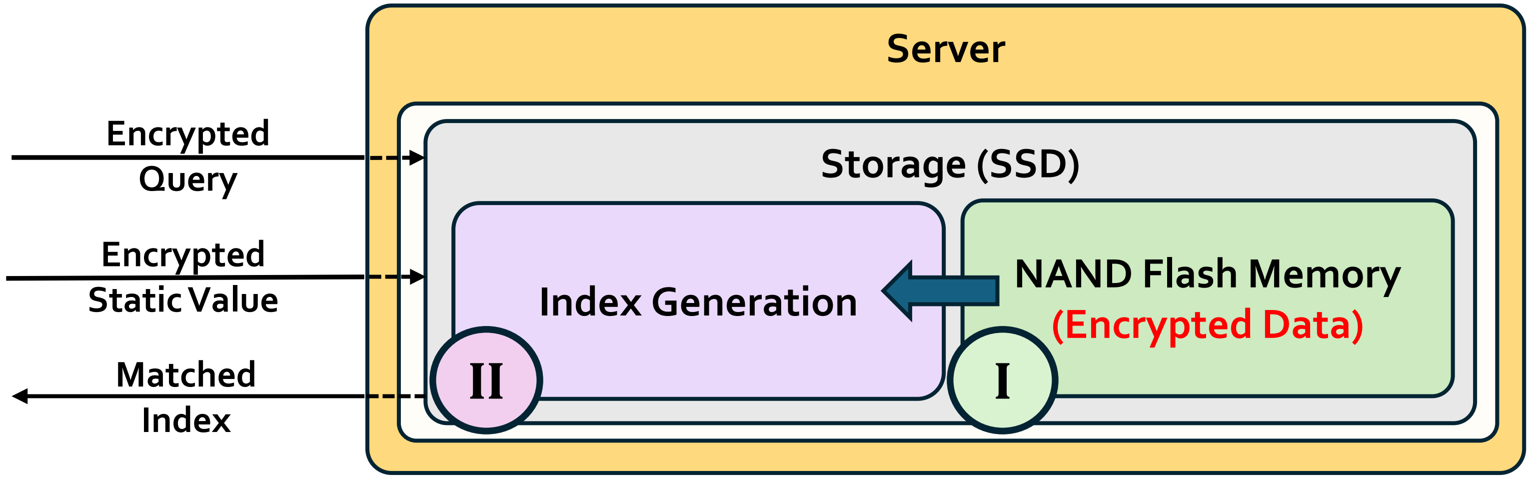
Lay out the data vertically in NAND-flash memory

Send the query from SSD controller to the latches

Perform Steps 1-6 to perform bit-serial addition

Accumulate the sum bit and send the accumulated bit to the SSD controller

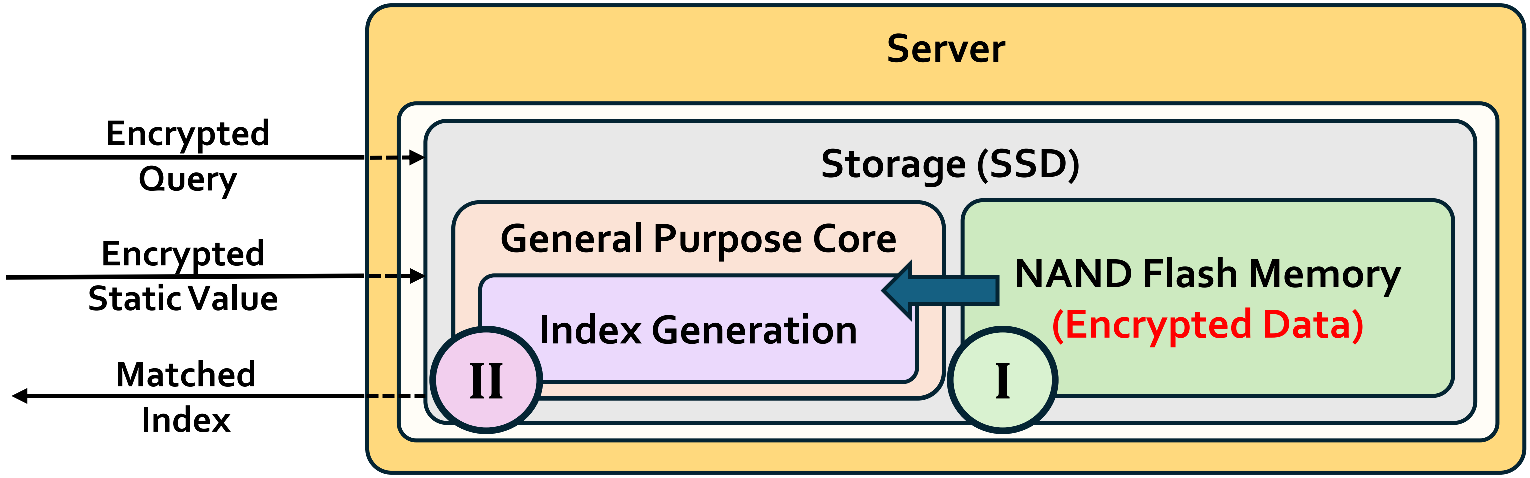
CIPHERMATCH: Hardware (Summary)



I — Perform **homomorphic additions** inside **NAND-flash memory** ✓

II — Generate the **final index** by **comparing it with match value**

CIPHERMATCH: Hardware (Summary)



II

Generate the **final index**
by **comparing it with match value**

Use **general purpose cores** to **identify the final match**

Talk Outline

Background, Problem & Goal

Key Idea

CIPHERMATCH: System Overview

CIPHERMATCH: Algorithm

CIPHERMATCH: Hardware

Evaluation Results

Evaluation Methodology (1/2): Real System

Our Implementation

Intel Xeon, 6 cores, 3.2 GHz **32GB DDR4 DRAM** **2TB PCIe 4.0 SSD**

We evaluate software-based CIPHERMATCH implementation (CM-SW) by modifying the Microsoft SEAL library

Baselines

- **Arithmetic (using SEAL):** State-of-the-art **arithmetic approach** [Yasuda+, CCSW 2013]
- **Boolean (using TFHE-rs):** State-of-the-art **Boolean approach** [Aziz+, Information 2024]

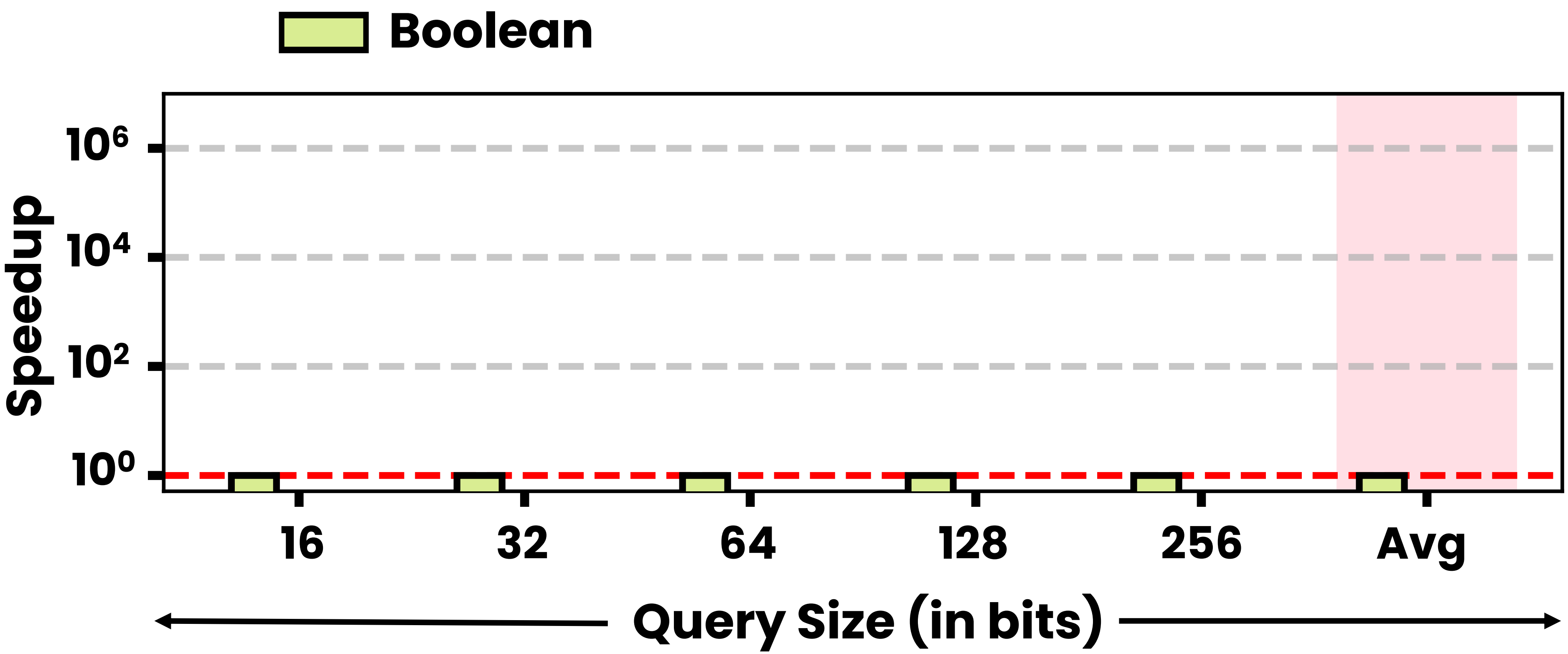
Workloads

- **Varying query size (16-256 bits)*** for encrypted database size of **128 GB**
- **Varying encrypted database size (8-128 GB)*** for **16-bit query** and 1000 queries

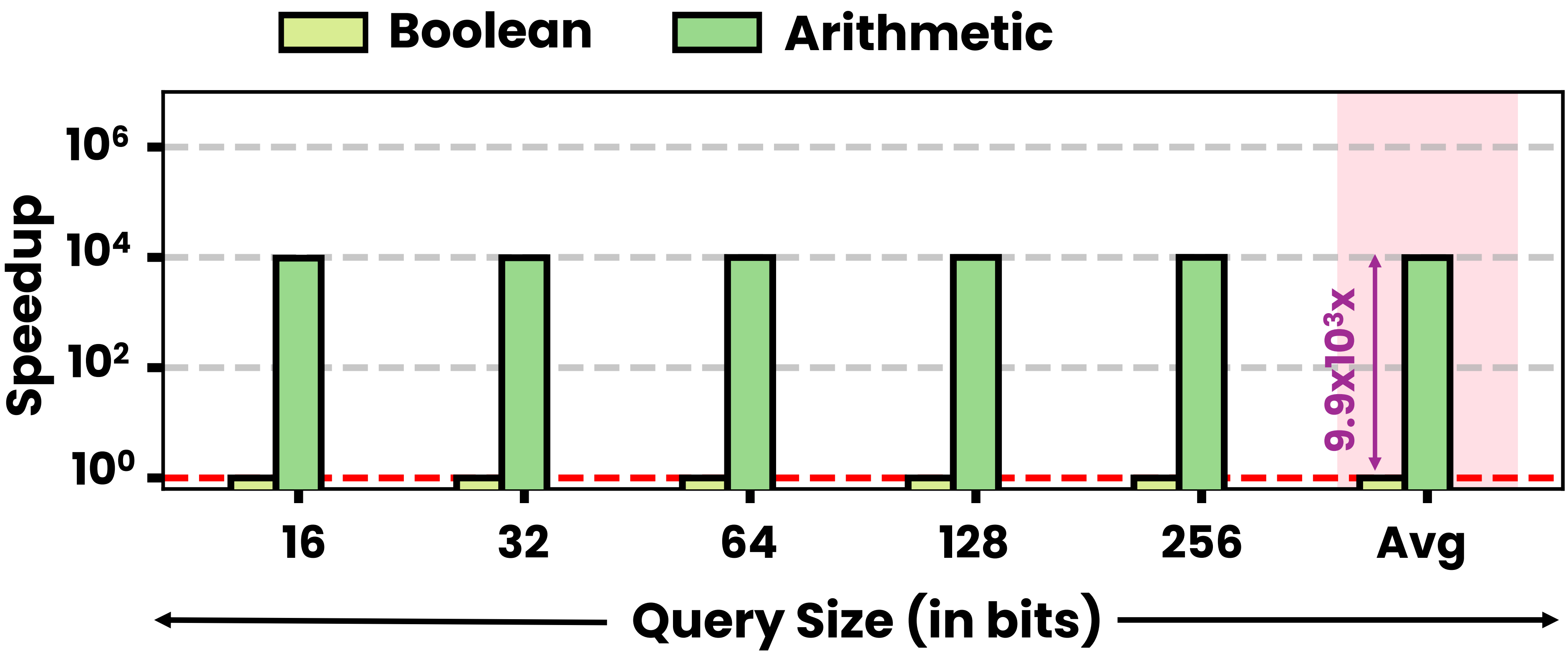
*** including all circular shifted queries**

SAFARI

Speedup for Different Query Sizes

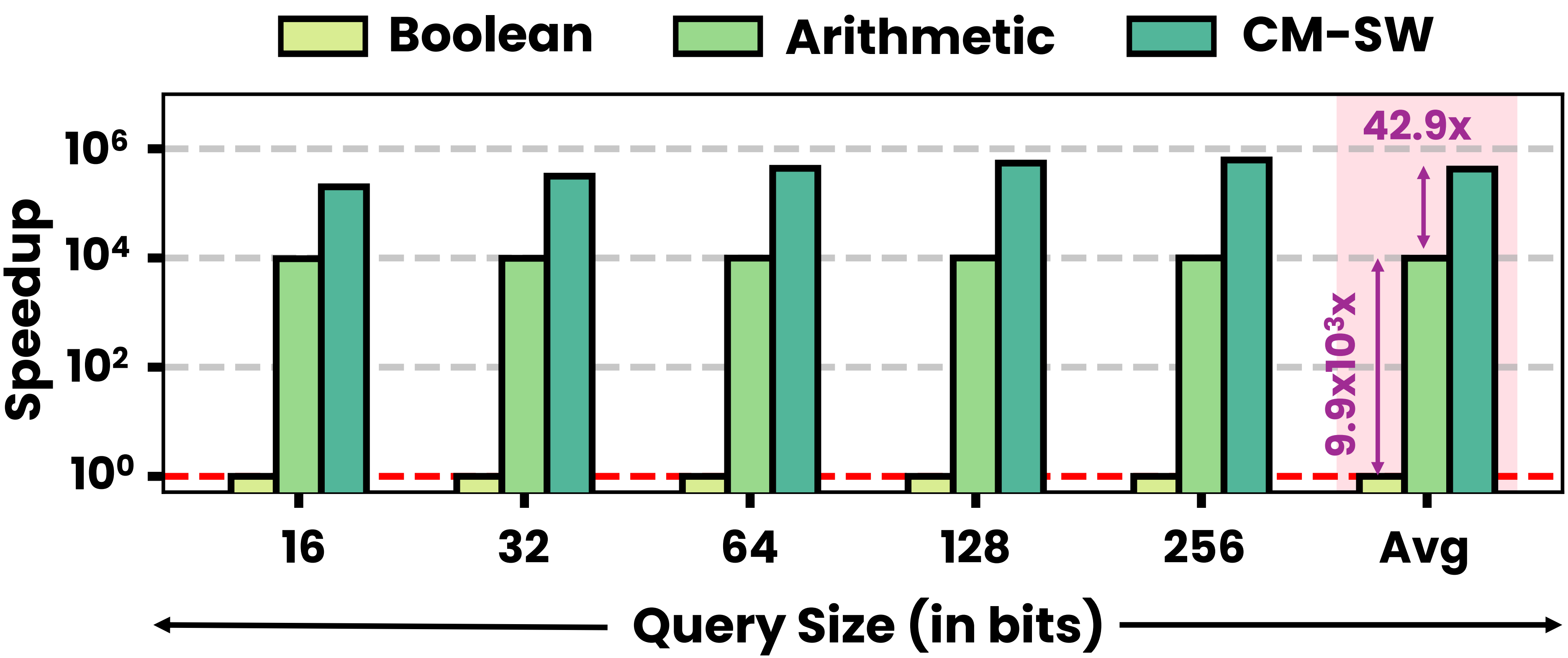


Speedup for Different Query Sizes (1/3)



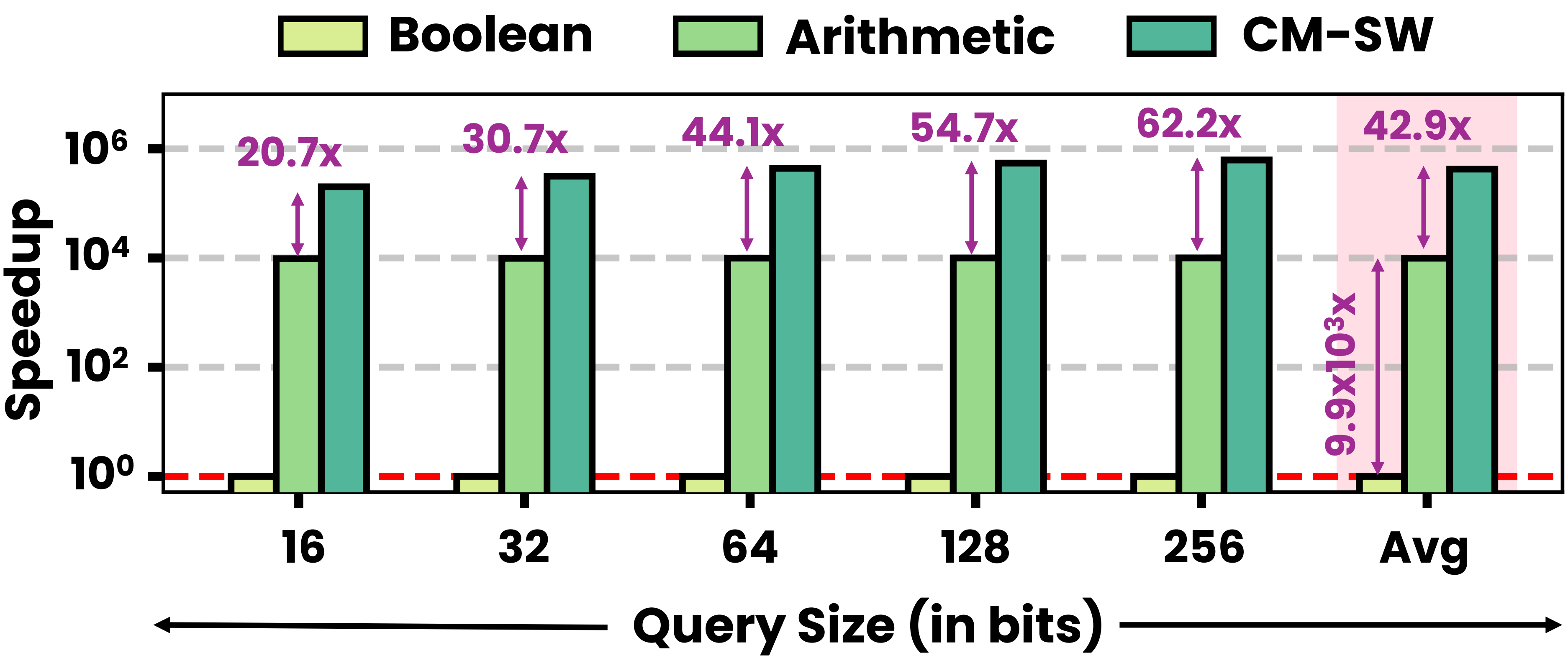
Arithmetic technique outperforms Boolean technique by orders of magnitude

Speedup for Different Query Sizes (2/3)



CM-SW outperforms the best prior arithmetic technique by 42.9x

Speedup for Different Query Sizes (3/3)



CM-SW speedup increases with query size
(due to the elimination of homomorphic multiplication)

Evaluation Methodology (2/2): Simulation

Our Implementation

We evaluate IFP-based CIPHERMATCH implementation (CM-IFP) by modeling the characteristics of the NAND-flash memory

Baselines

- **CM-SW:** CIPHERMATCH on **compute-centric system** [same as real system]
- **CM-PuM:** CIPHERMATCH on **memory-centric system** [*, 32GB DDR4-2400]
- **CM-PuM-SSD:** CIPHERMATCH on **storage-centric system** [*, SSD DRAM - 2GB LPDDR4-1866]

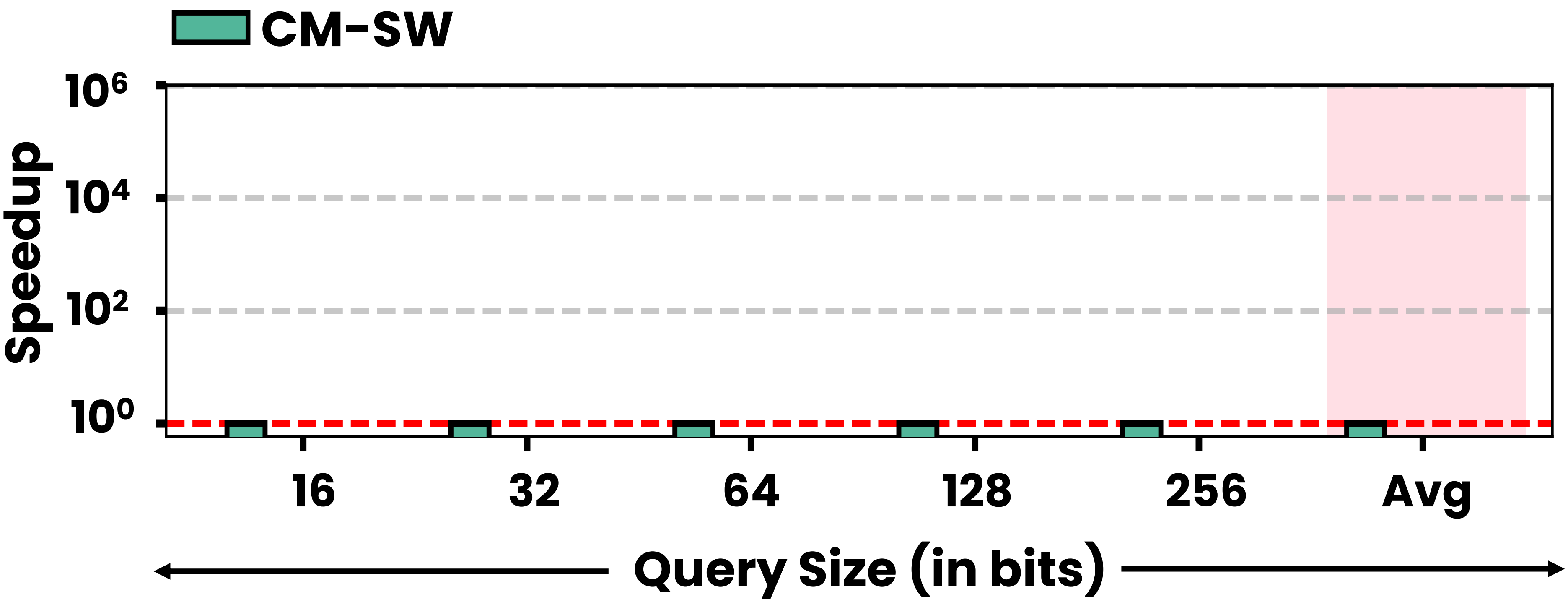
[*] - SIMDRAM framework [Hajinazar+ , ASPLOS 2021]

Workloads

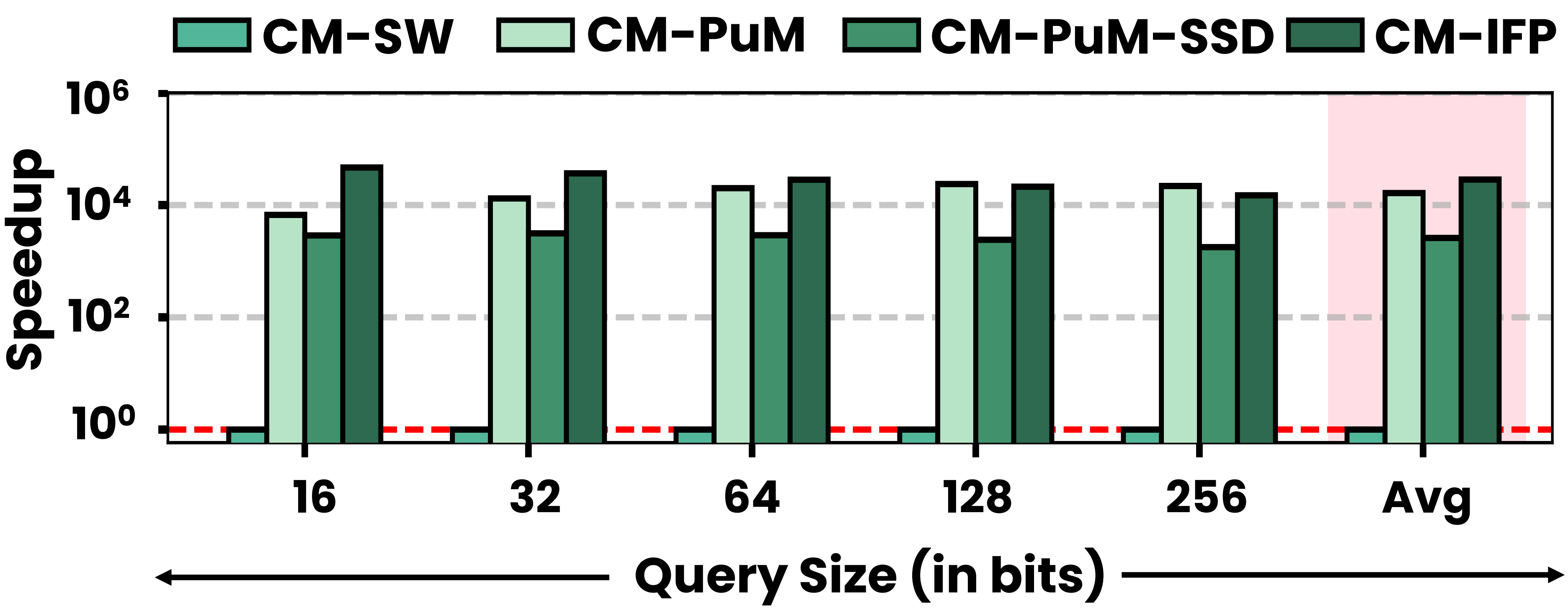
- **Varying query size (16-256 bits)*** for encrypted database size of 128 GB
- **Varying encrypted database size (8-128 GB)*** for 16-bit query and 1000 queries

*** including all circular shifted queries**

Speedup for Different Query Sizes

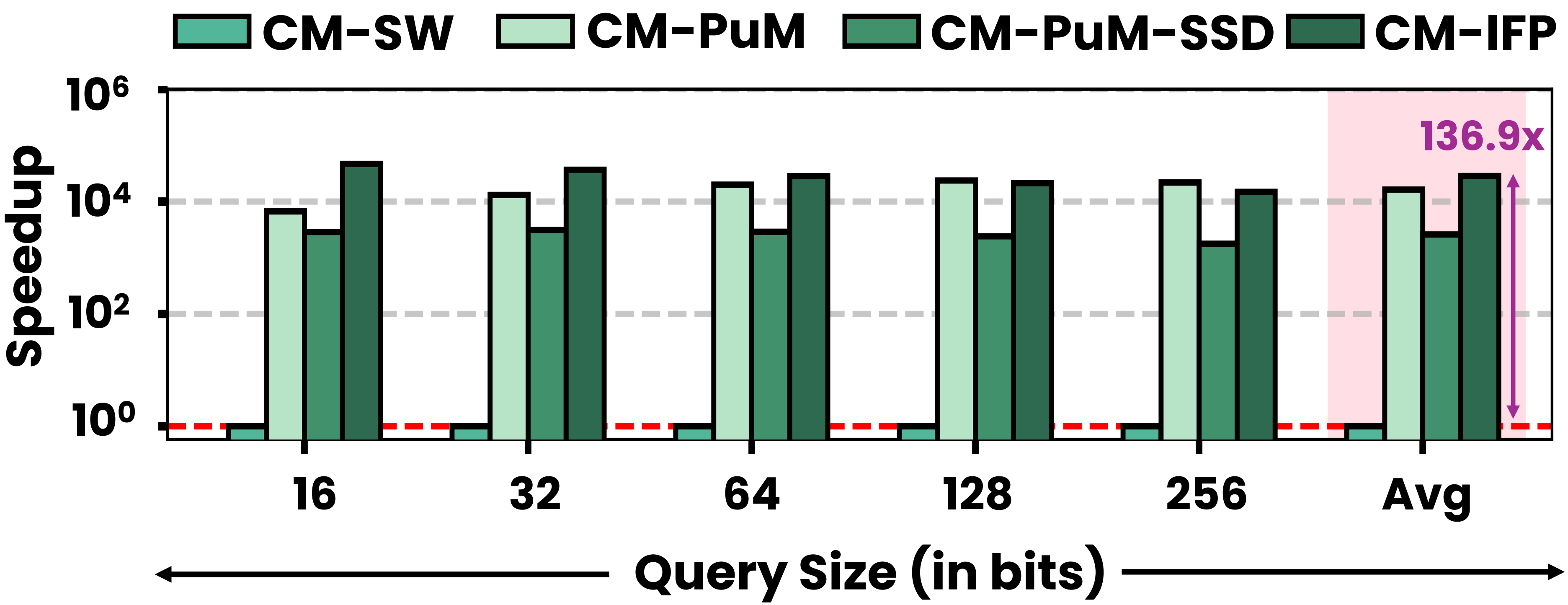


Speedup for Different Query Sizes (1/3)



All three near-data processing systems improve performance by reducing data movement

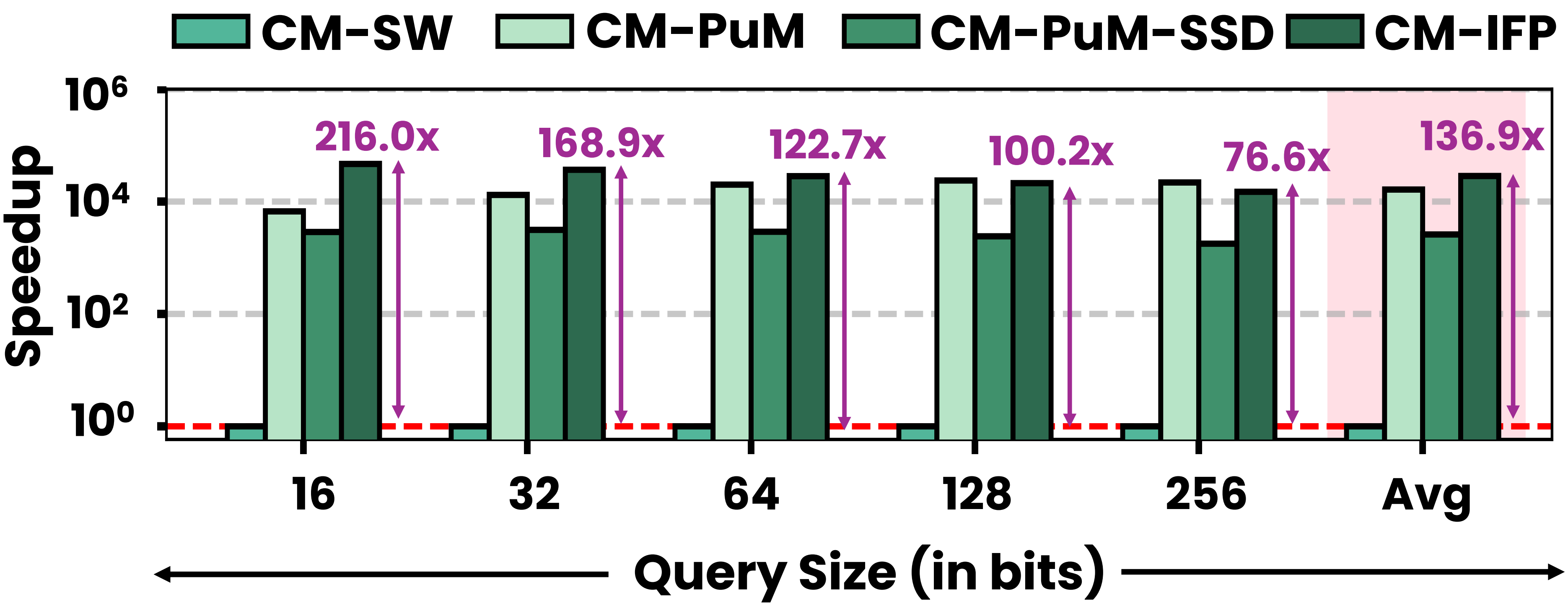
Speedup for Different Query Sizes (2/3)



CM-IFP outperforms CM-SW by 136.9x

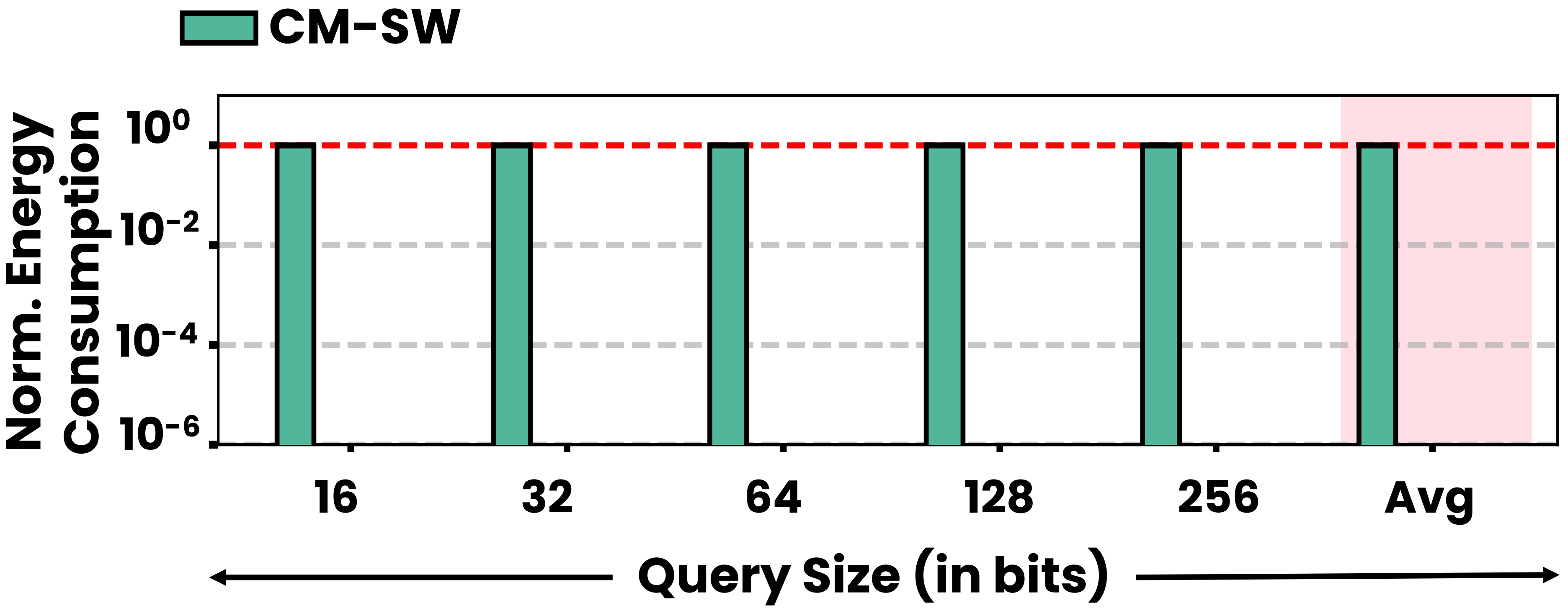
CM-IFP outperforms other near-data processing systems

Speedup for Different Query Sizes (3/3)

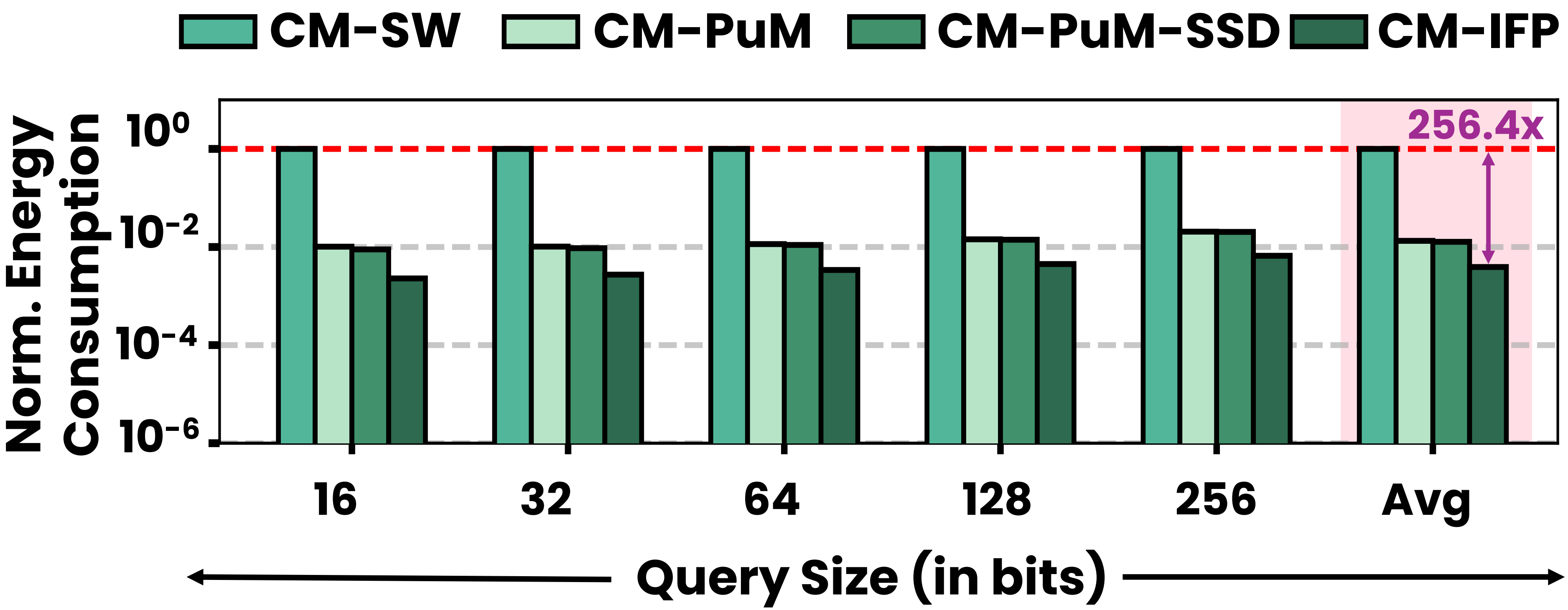


CM-IFP speedup decreases with query sizes
due to repeated flash read operations on same data
for circularly shifted queries

Energy Consumption for Different Query Sizes



Energy Consumption for Different Query Sizes



All three near-data processing systems provide large energy savings over CM-SW

More Details in Our Paper

CIPHERMATCH: Accelerating Homomorphic Encryption-Based String Matching via Memory-Efficient Data Packing and In-Flash Processing

Mayank Kabra† Rakesh Nadig† Harshita Gupta† Rahul Bera† Manos Frouzakis†
Vamanan Arulchelvan† Yu Liang† Haiyu Mao‡ Mohammad Sadrosadati† Onur Mutlu†

ETH Zurich† *King's College London‡*

Homomorphic encryption (HE) allows secure computation on encrypted data without revealing the original data, providing significant benefits for privacy-sensitive applications. Many cloud computing applications (e.g., DNA read mapping, biometric matching, web search) use exact string matching as a key operation. However, prior string matching algorithms that use homomorphic encryption are limited by high computational latency caused by the use of complex operations and data movement bottlenecks due to the large encrypted data size. In this work, we provide an efficient algorithm-hardware codesign to accelerate HE-based secure exact string matching. We propose CIPHERMATCH, which (i) reduces the increase in memory footprint after encryption using an optimized software-based data packing scheme, (ii) eliminates the use of costly homomorphic operations (e.g., multiplication and rotation), and (iii) reduces data movement by designing a new in-flash processing (IFP) architecture.

format). Since cloud servers can be shared among multiple users, sensitive user data can become vulnerable to security threats and leaks [24–26]. HE can significantly benefit privacy-sensitive applications [27–31] that require exact string matching [13–17, 19, 21–23] as the fundamental operation by directly operating on encrypted data without requiring decryption.

Unfortunately, homomorphic operations are typically $10^4\times$ to $10^5\times$ slower than their traditional unencrypted counterparts in existing systems [32]. Prior works propose two main approaches to perform secure string matching: (1) the Boolean approach (e.g., [17, 33]), and (2) the arithmetic approach (e.g., [27, 29, 34]). The Boolean approach [17, 33] packs individual bits into a polynomial, encrypts it, and uses homomorphic XNOR and AND operations to perform secure string matching on a search pattern of any size. In contrast, the arithmetic approach [27, 29, 34] packs multiple bits into a polynomial, encrypts it, and employs homomorphic multiplication and addition

https://arxiv.org/pdf/2503.08968

To Summarize …

Conclusion

CIPHERMATCH

A new algorithm-hardware codesign that significantly improves the performance of secure ***exact*** string matching algorithm

1 Pack multiple bits of database and thus eliminate the use of homomorphic multiplication

+ Reduces memory footprint
+ Provides scalable secure *exact* string-matching

2 Use in-flash processing (IFP) to accelerate secure *exact* string-matching

+ Reduces data movement
+ Leverages bit-level and array-level parallelism

Key Results

- **CIPHERMATCH-SW: 42.9x speedup** & **39.4x lower energy** than best software
- **CIPHERMATCH-IFP: 136.9x speedup** & **256.4x lower energy** than CM-SW

CIPHERMATCH

Accelerating Homomorphic Encryption-Based String Matching via Memory-Efficient Data Packing and In-Flash Processing

arXiv

Mayank Kabra

Rakesh Nadig, Harshita Gupta, Rahul Bera, Manos Frouzakis, Vamanan Arulchelvan, Yu Liang, Haiyu Mao, Mohammad Sadrosadati, and Onur Mutlu

ETH zürich

SAFARI

KING'S College LONDON

Backup slides

Summary

CM-SW provides 42.9x speedup
over the state-of-the-art approach in real systems

Due to our new memory-efficient data packing scheme
and use of *only* homomorphic additions

CM-IFP provides 136.9x speedup over CM-SW
and outperforms three near-data processing systems

Due to our new IFP design to perform in-flash operations
and exploiting large-scale bit-level parallelism

Executive Summary

Problem: Secure exact string matching using homomorphic encryption (HE) operations face performance bottlenecks in two key areas:
- **(a) High computation cost** due to use of complex homomorphic operations (e.g., multiplication)
- **(b) data movement bottleneck** due to large homomorphically encrypted data

Motivation: Reducing memory expansion from HE and performing computation where the database resides can improve the performance of secure exact string matching algorithm

Opportunity: (a) Perform memory-efficient packing of the database to reduce the increase in memory footprint after encryption and (b) perform simple computations (e.g., HE addition) inside solid state drives (SSDs – i.e., where the database is stored) to reduce data movement

CIPHERMATCH: A novel algorithm-hardware co-design that significantly improves the performance of HE-based secure string matching by using *only homomorphic addition* and leveraging the operational principles of NAND-flash memory.

Key Idea:
1) **To pack multiple bits of data in the each coefficient of ciphertext**
2) **Use in-flash processing (IFP) to perform string matching inside NAND-flash memory**

Key Benefits:
- + **Reduce memory expansion after encryption**
- + **Eliminates the use of complex HE operations**
- + **Reduces data movement bottleneck**

Key Results: **(i)** Software-based CIPHERMATCH implementation (CM-SW) achieves 42.9x speedup over state-of-the-art software approaches
(ii) CIPHERMATCH IFP implementation futher improves upon CM-SW, achieving 136.9x better performance and 256.4x lower energy consumption

Executive Summary

Problem: Secure exact string matching using homomorphic encryption (HE) operations face performance bottlenecks in two key areas:

(a) **High computation cost** due to use of complex homomorphic operations (e.g., multiplication)

(b) **Data movement bottleneck** due to large homomorphically encrypted data size

Motivation: Reducing memory expansion from HE and performing computation where the database resides can improve the performance of secure exact string matching algorithm

Opportunity: (a) Optimize memory usage by **packing encrypted data efficiently** and (b) perform secure string matching using **simple HE operations (addition) inside SSDs**, reducing data movement.

CIPHERMATCH: A novel algorithm-hardware co-design that significantly improves the performance of HE-based secure exact string matching (a) by using *only homomorphic addition* and (b) leveraging the *operational principles of NAND-flash memory* to perform secure exact string matching

Key Idea:

1) **Pack multiple bits of data in the each coefficient of ciphertext**
2) **Use in-flash processing (IFP) to perform string matching inside NAND-flash memory**

Key Benefits:

+ **Reduce memory expansion after encryption**
+ **Eliminates the use of complex HE operations**
+ **Reduces data movement bottleneck**

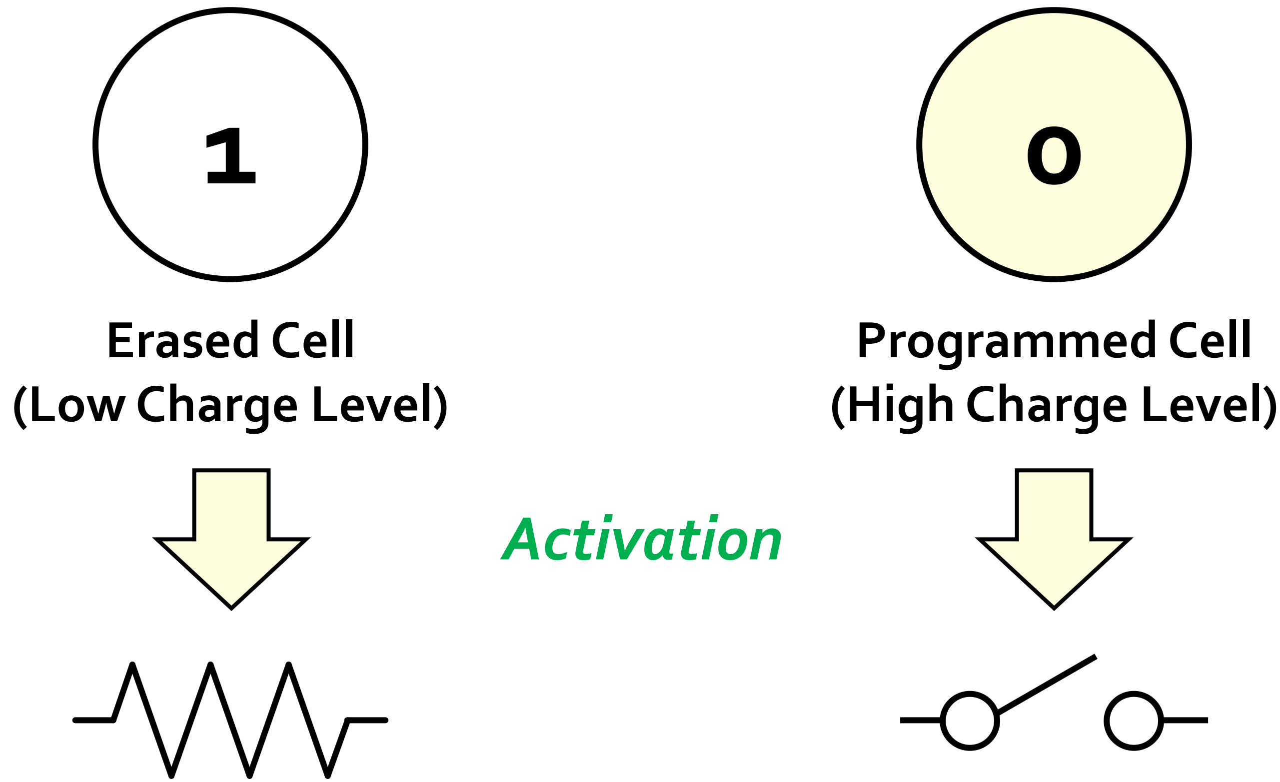
Key Results:

(i) Software-based CIPHERMATCH (CM-SW): *42.9× speedup* over existing state-of-the-art approaches

(ii) CIPHERMATCH with IFP: *136.9× faster and 256.4× lower energy consumption* than CM-SW

NAND Flash Basics: A Flash Cell

- **A flash cell stores data by adjusting the amount of charge in the cell**

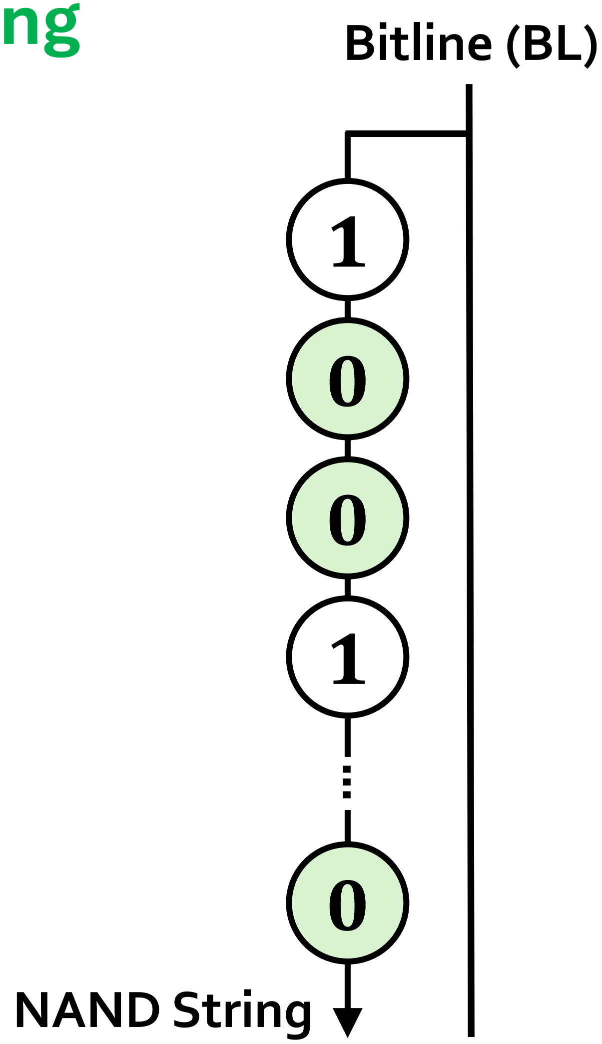


Operates as a resistor

Operates as an open switch

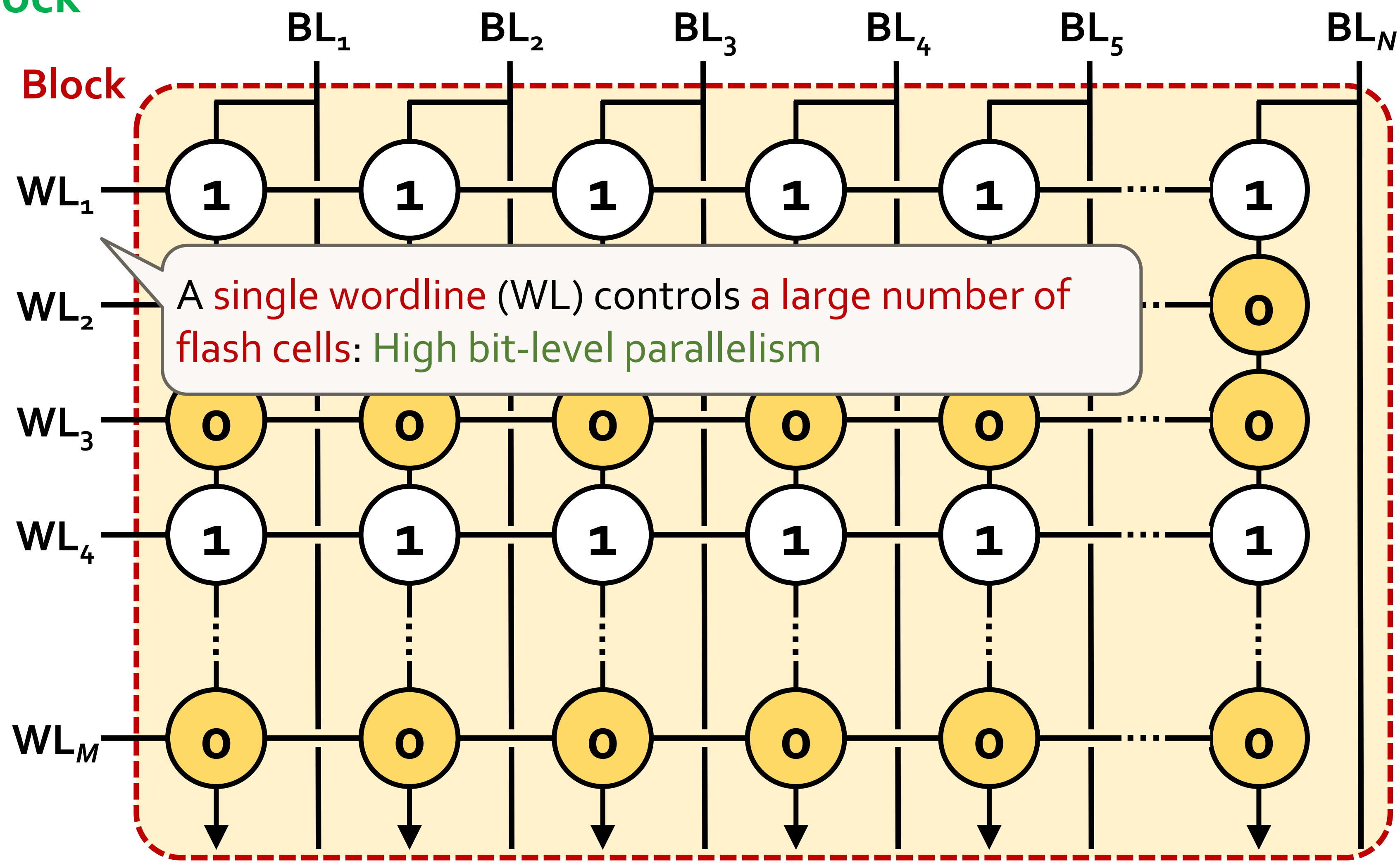
NAND Flash Basics: A NAND String

- A set of flash cells are serially connected to form a NAND String



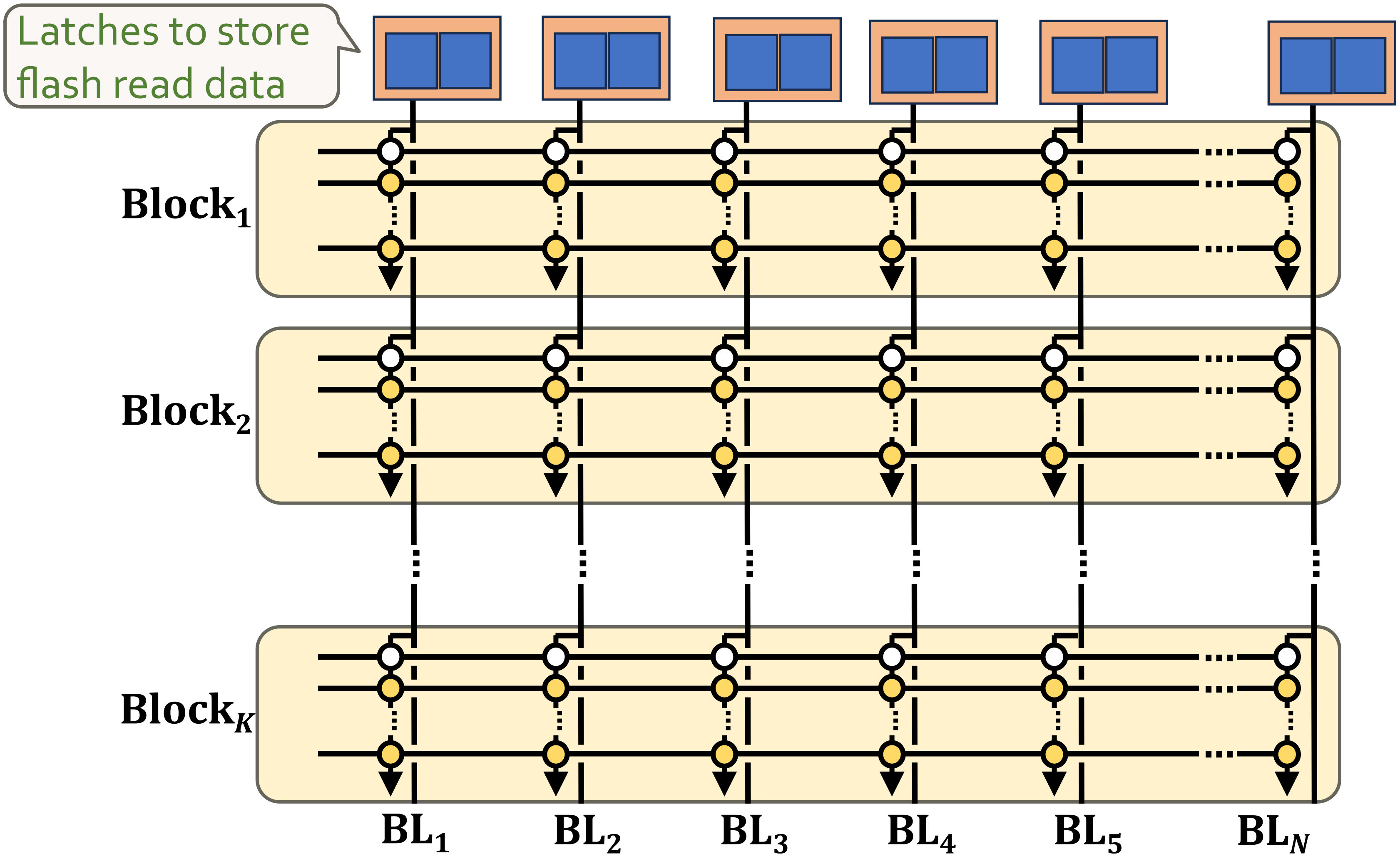
NAND Flash Basics: A NAND Block

- NAND strings connected to different bitlines comprise a NAND block

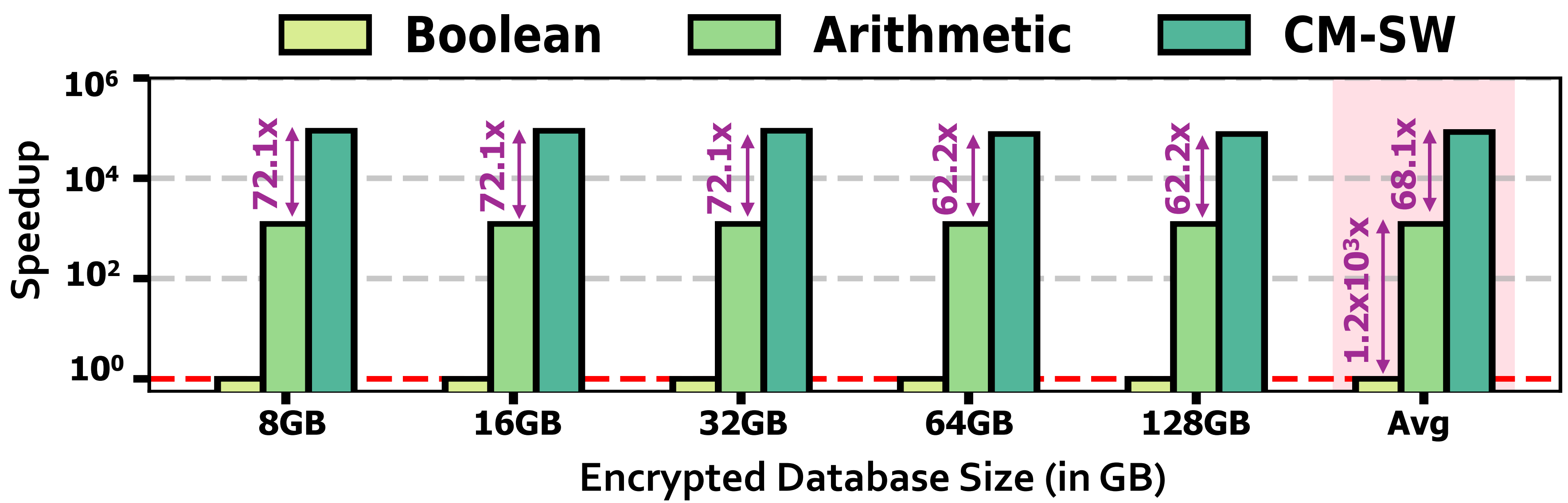


NAND Flash Basics: A NAND Plane

- A large number of blocks share the same bitlines



Speedup for Different Database Size

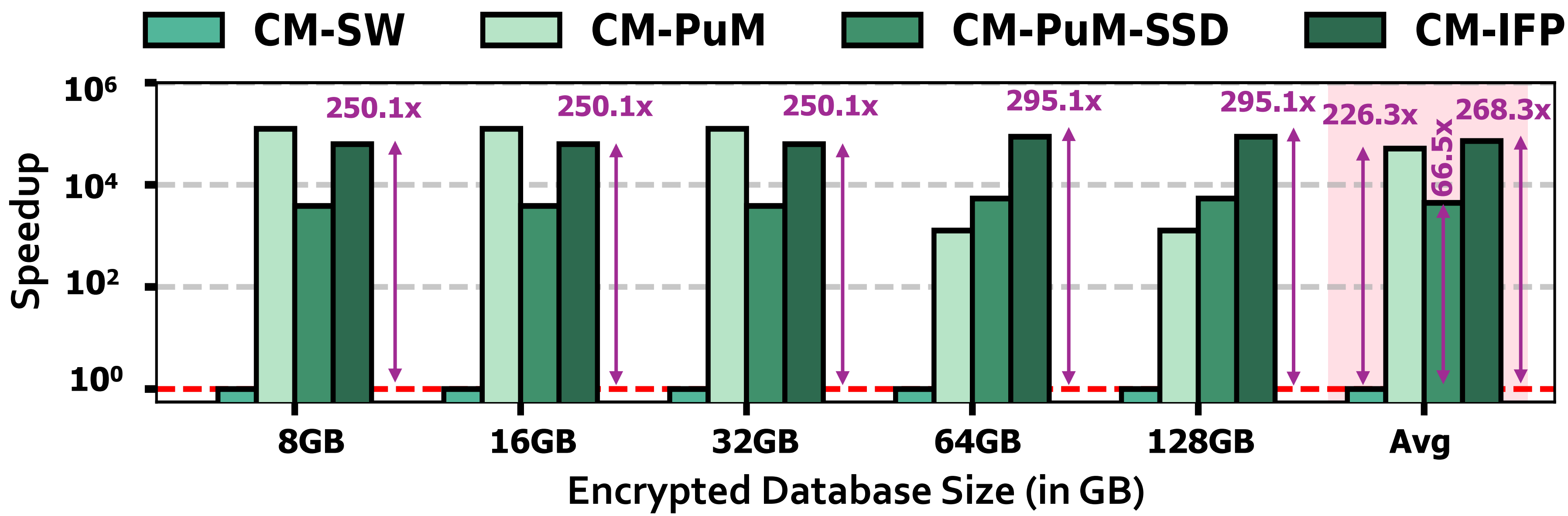


CM-SW shows average speedup of 68.1x over prior arithmetic approach

CM-SW speedup decreases as data size exceeds DRAM capacity, primarily due to increased data movement between storage and DRAM.

Query Size of 16 bits

Speedup for Different Database Size

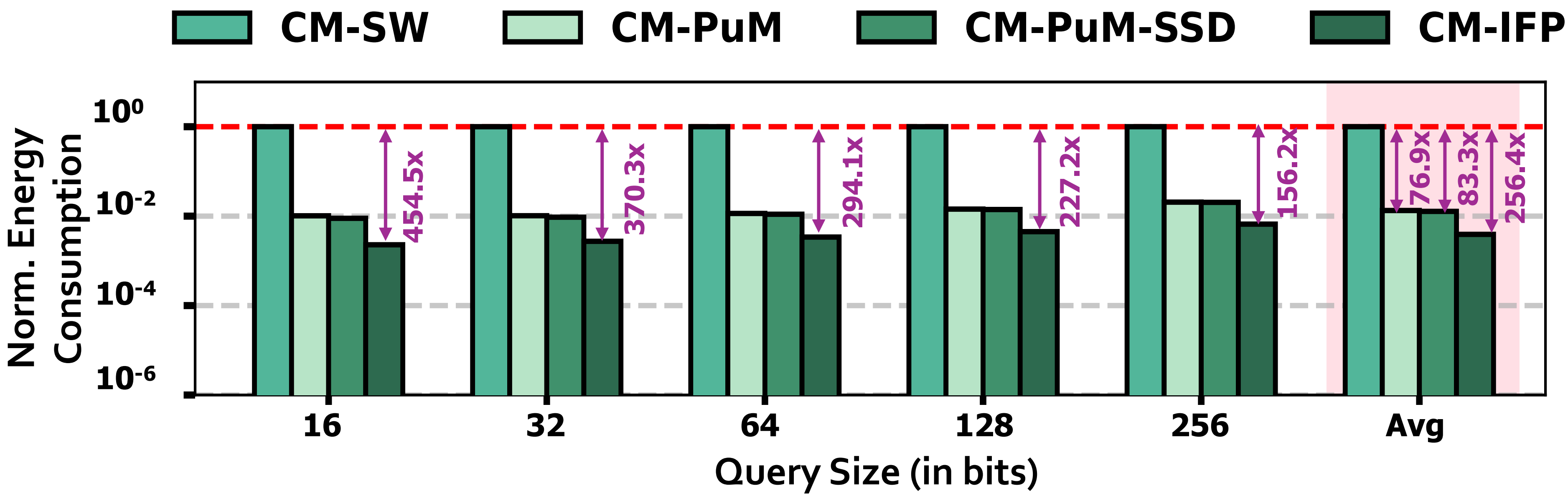


CM-IFP shows highest average speedup of 268.3x over CM-SW

CM-SW speedup decreases when data size goes beyond DRAM size due to frequent data movement between storage and DRAM

Query Size of 16 bits

Energy Consumption for Different Query Size

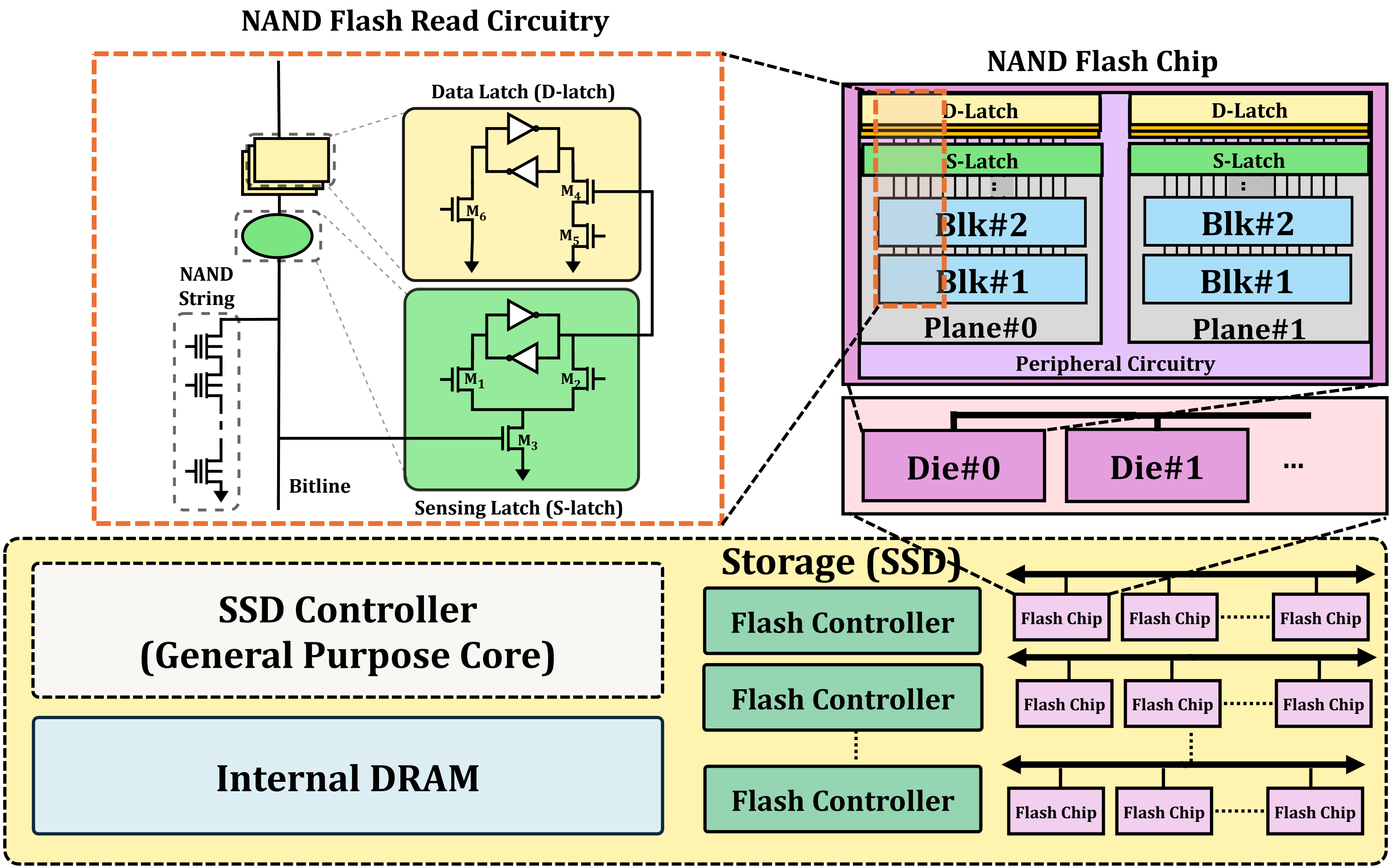


CM-IFP shows highest average energy savings of 256.4x over CM-SW

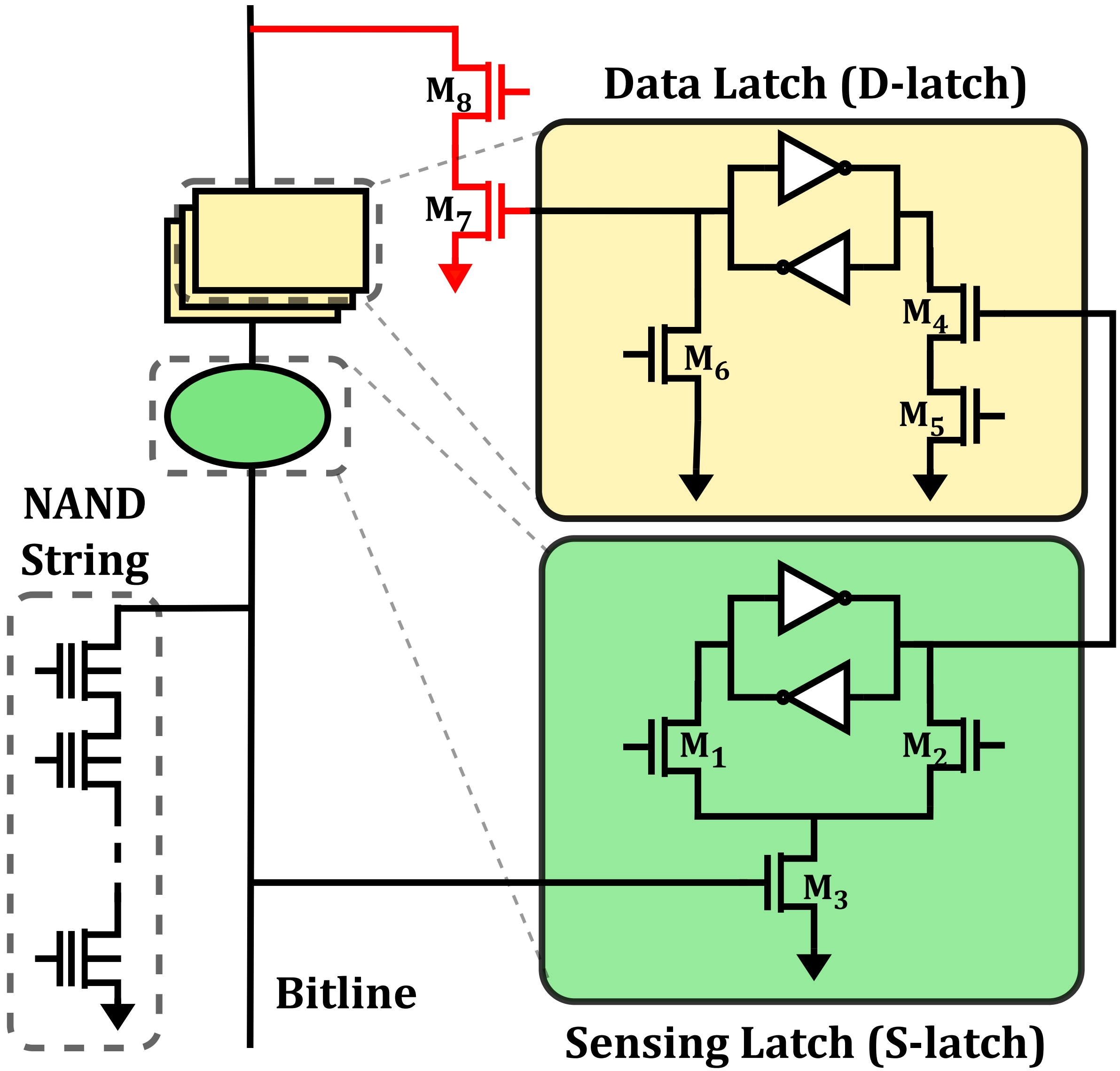
CM-IFP energy efficiency decreases with increasing query sizes d*ue to expensive flash reads*

Database Size of 128 GB

CIPHERMATCH: NAND-Flash Bitwise Operations

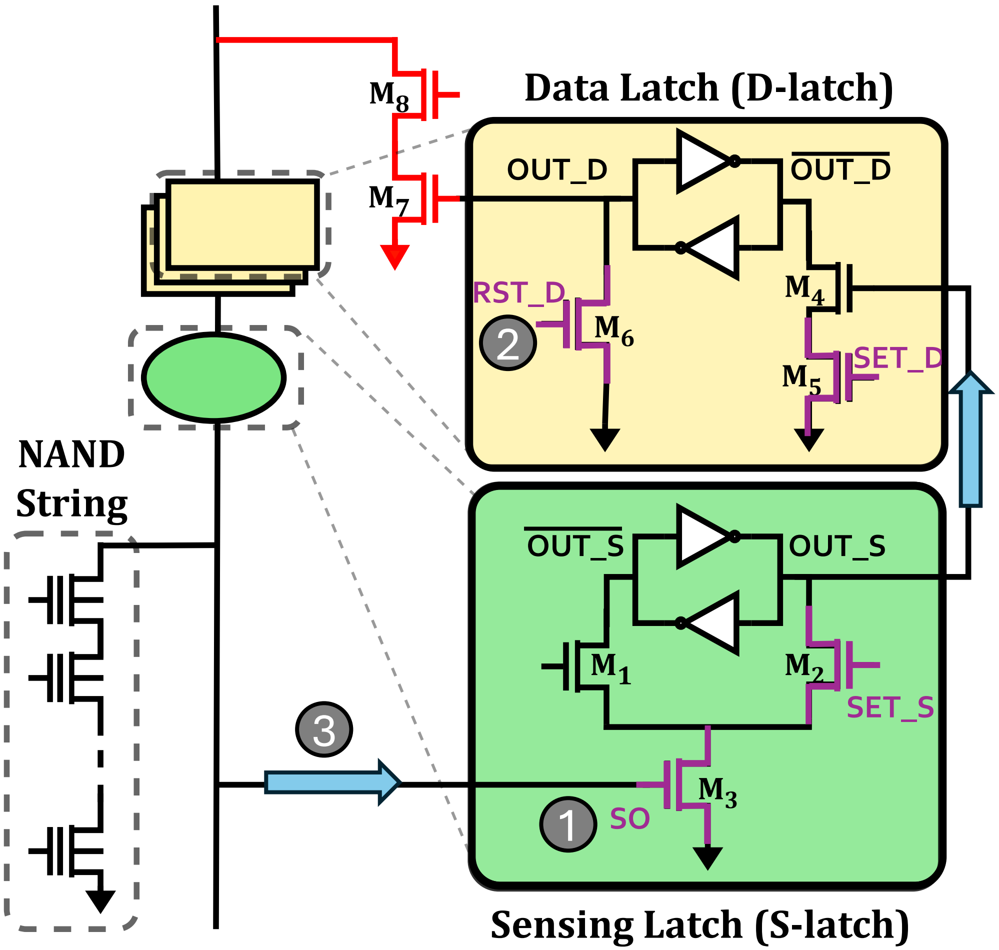


CIPHERMATCH: NAND-Flash Bitwise Operations



Design proposed by prior work [Cho+, Patent 2022]

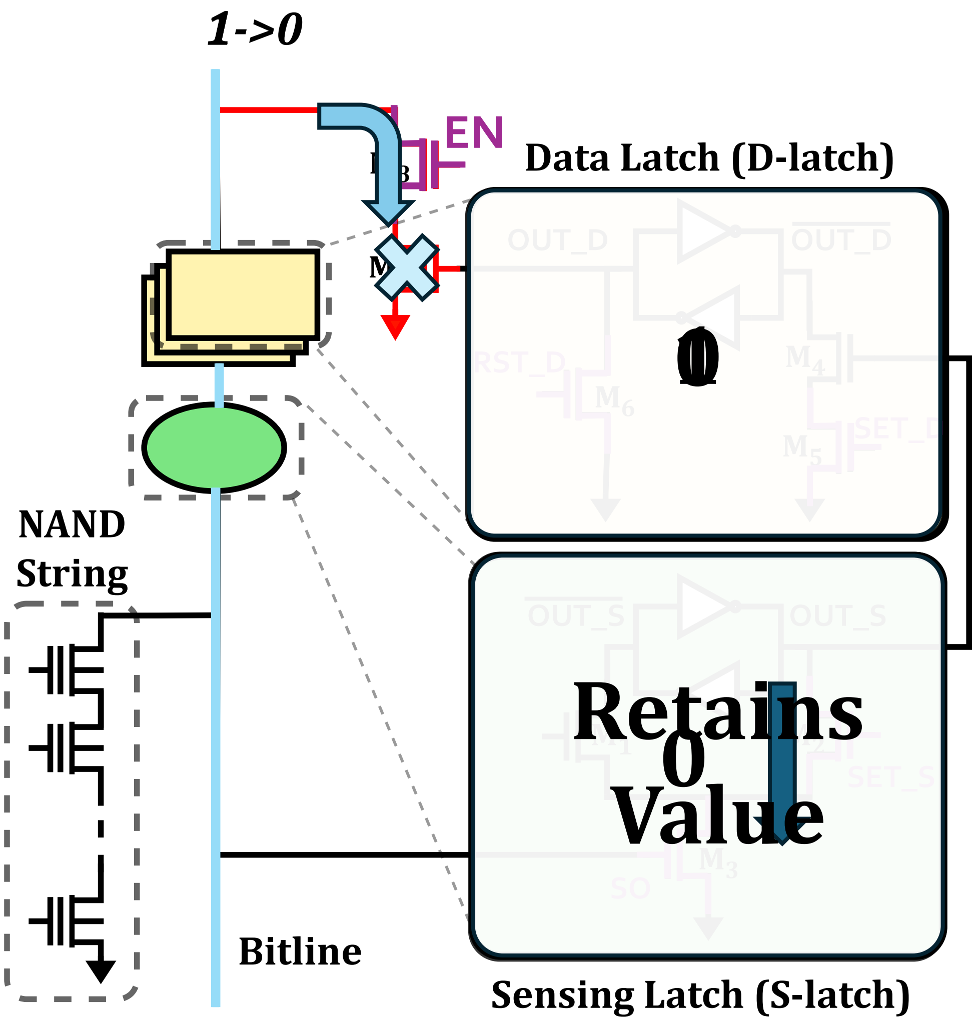
CIPHERMATCH: NAND-Flash Bitwise Operations



Bitwise AND of A and B

1. Data (A) is read and stored in S-latch
2. Data (A) is transferred from S-latch to D-latch
3. Similarly data (B) is read and stored in S-latch

CIPHERMATCH: NAND-Flash Bitwise Operations



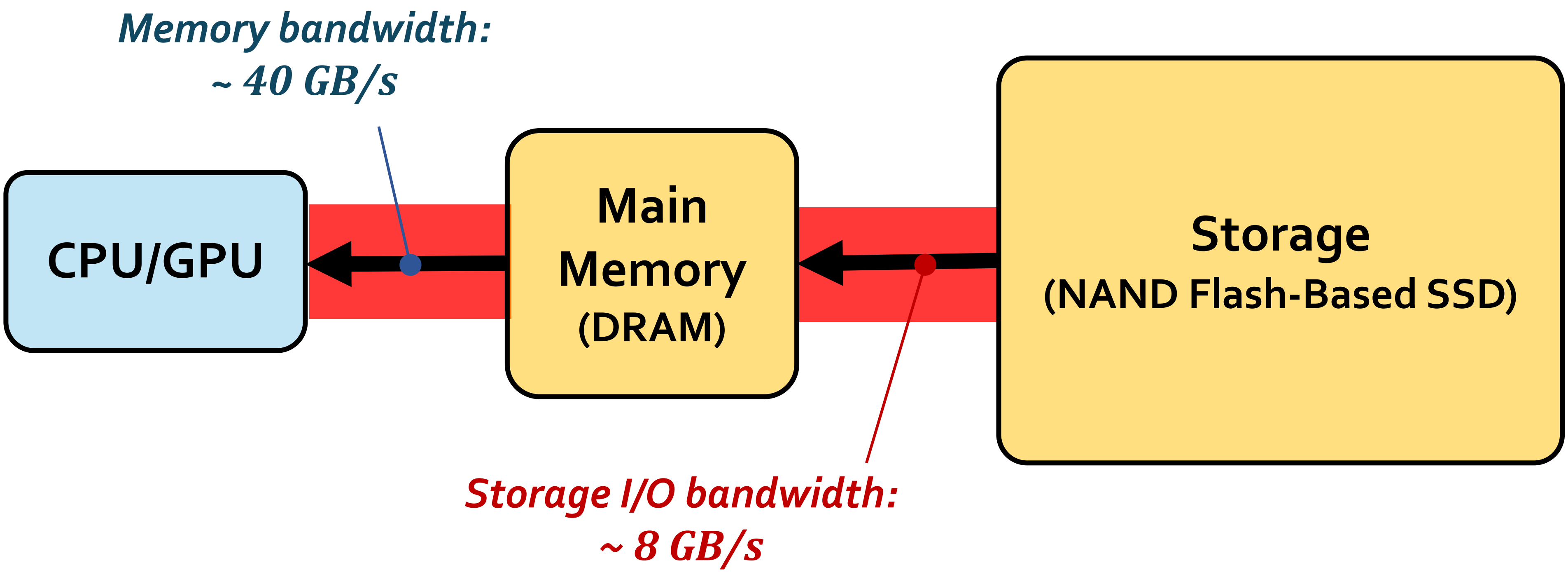
Bitwise AND of A and B

D-latch (A)	S-latch (B)	Z
0	0	0
0	1	0
1	0	0
1	1	1

Precharge the bitline to *logical 1*

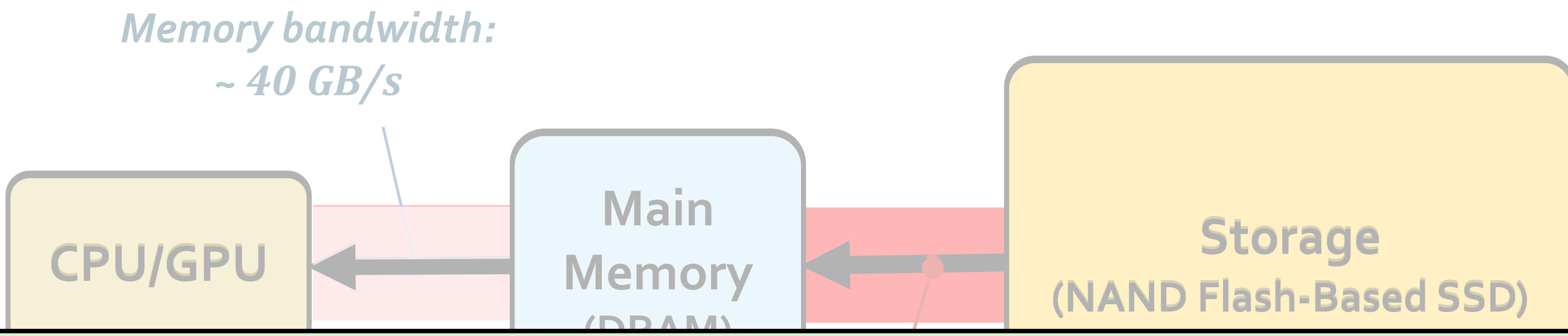
Data Movement Bottleneck

- Compute-centric systems: Move entire data from storage to CPU/GPU



Motivation (II) – Data Movement Bottleneck ②

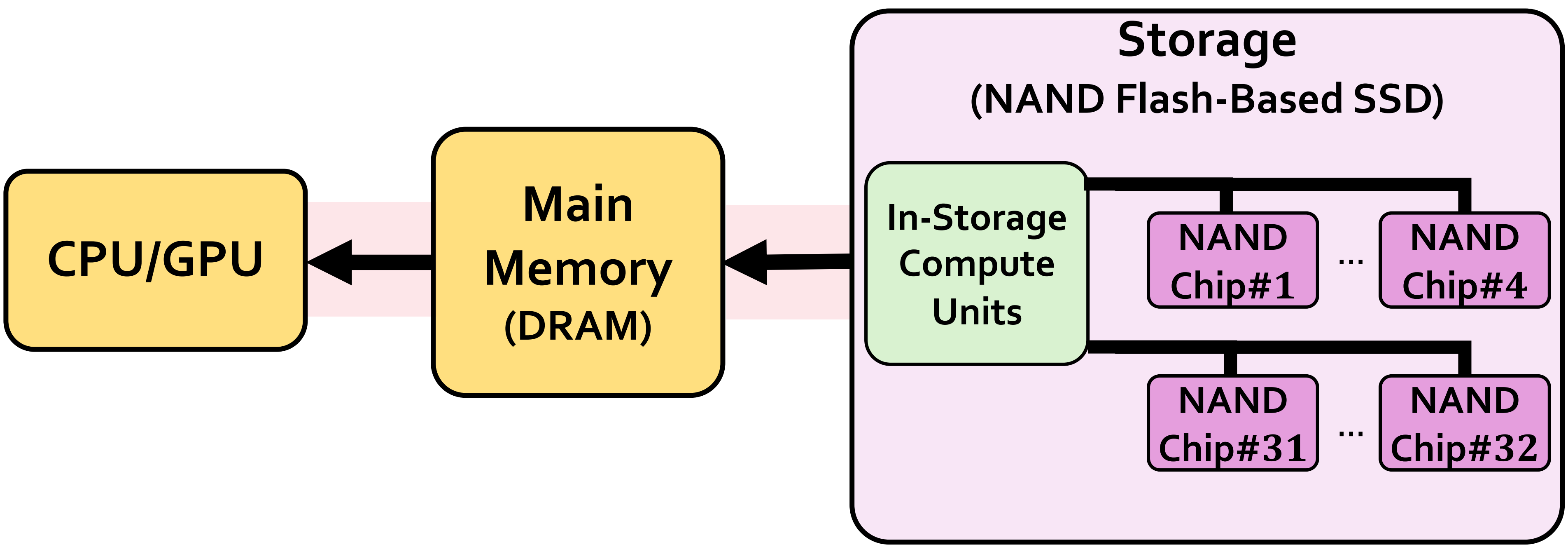
- Compute-centric systems: Move entire data from storage to CPU/GPU
- Memory-centric systems: Perform computations in main memory



External I/O bandwidth of storage systems
is the *main bottleneck* for memory intensive application

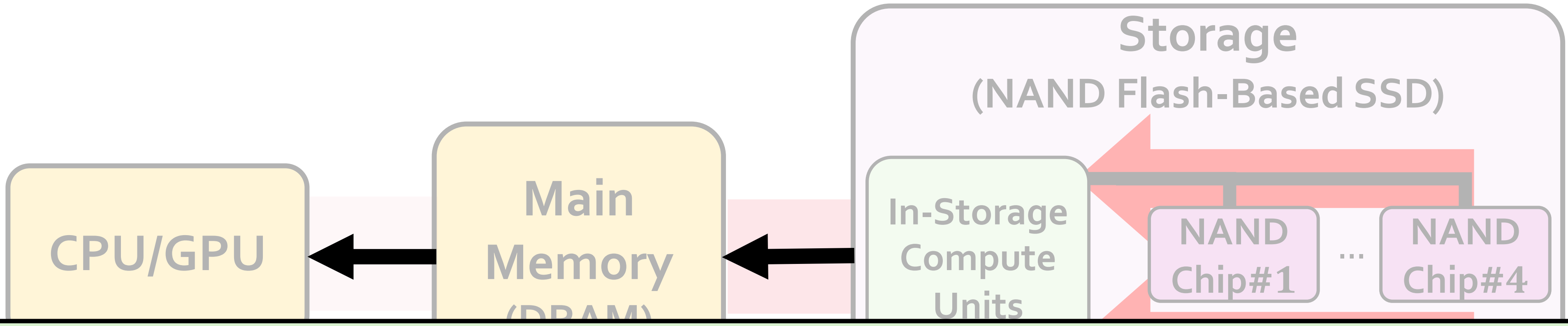
Motivation (II) – Data Movement Bottleneck ②

- Compute-centric systems: Perform computations in CPU/GPU
- Memory-centric systems: Perform computations in main memory
- Storage-centric systems: Perform computations inside storage system



Motivation (II) – Data Movement Bottleneck ②

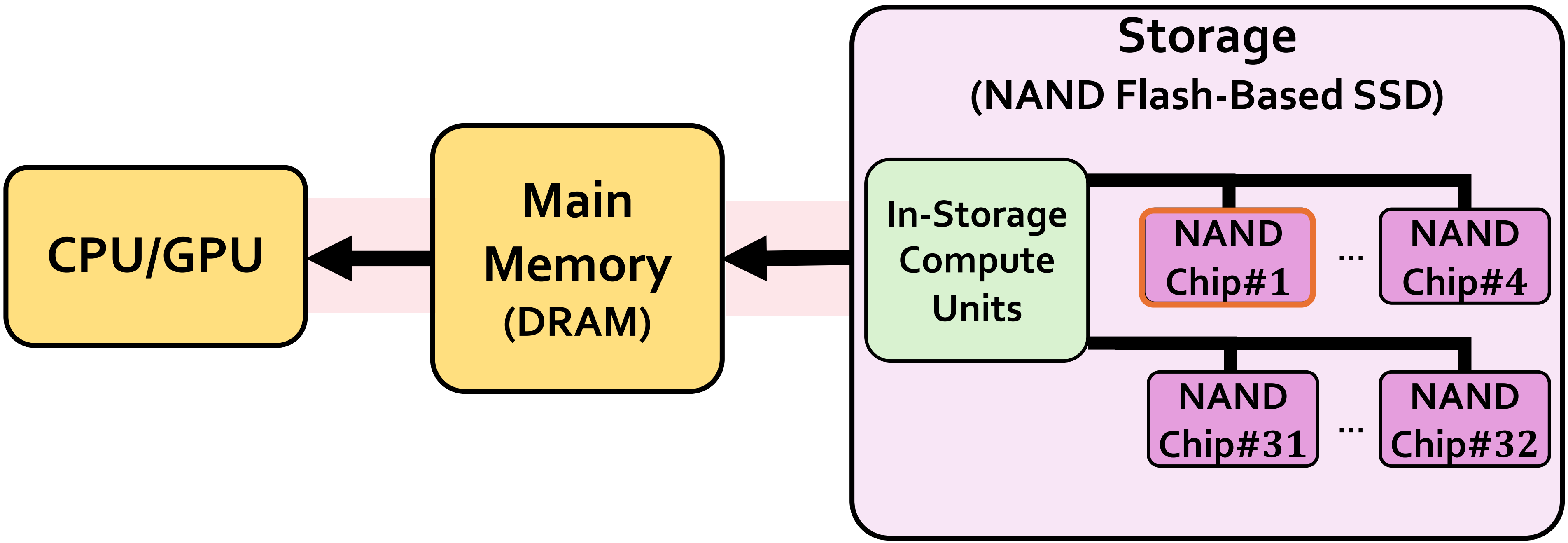
- **Compute-centric systems:** Perform computations in CPU/GPU
- **Memory-centric systems:** Perform computations in main memory
- **Storage-centric systems:** Perform computations inside storage system



SSD-internal bandwidth
becomes the *new bottleneck for computations*

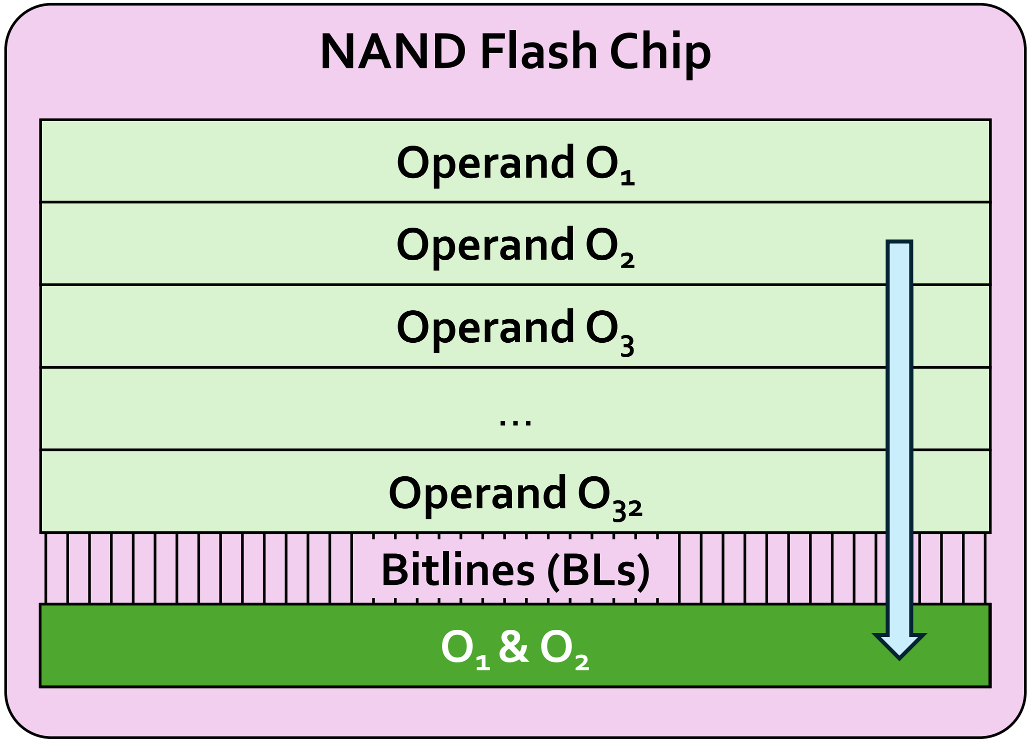
In-Flash Processing (IFP)

Perform computations inside NAND-flash chips
by using *operational principles of NAND-flash memory*



In-Flash Processing (IFP)

Prior Works ([Gao+, MICRO 2021] , [Park+, MICRO 2022]) perform ***bitwise operations using the latching circuit***



Limitations of Prior Works

- Boolean Approach [Pradel+, TrustCom 2021 ; Aziz+, Information 2024]
- Arithmetic Approach [Yasuda+, CCSW 2013 ; Kim+, TDSC 2017 ; Bonte+, CCS 2020]

Boolean Approach

1. ***High memory footprint*** due to *encryption of individual bits*

$m = 1\ 1\ 0\ 1\ 0\ 1\ 1 \dots 1\ 1\ 0$

$Enc(b_0)$ $Enc(b_1)$ ... $Enc(b_n)$

Arithmetic Approach

1. ***Low memory footprint*** due to *data packing mechanism*

$m = \boxed{1\ 1\ 0}\ \boxed{1\ 0\ 1}\ 1 \dots \boxed{1\ 1\ 0}$

$Enc(P_1(x))$ $Enc(P_2(x))$ ... $Enc(P_n(x))$

Limitations of Prior Works

- Boolean Approach [Pradel+, TrustCom 2021 ; Aziz+, Information 2024]
- Arithmetic Approach [Yasuda+, CCSW 2013 ; Kim+, TDSC 2017 ; Bonte+, CCS 2020]

Boolean Approach	Arithmetic Approach
1. ***High memory footprint*** due to *encryption of individual bits*	1. ***Low memory footprint*** due to *data packing mechanism*
2. ***High computation cost*** due to *large number of HE operations*	2. ***Low computation cost*** due to *small number of HE operations*
$Enc(b_0) \oplus Enc(b_1) \oplus Enc(b_2)$ 1000x	$Enc(P_1(x)) \times Enc(P_2(x)) \times$ 10x

Limitations of Prior Works

- Boolean Approach [Pradel+, TrustCom 2021 ; Aziz+, Information 2024]
- Arithmetic Approach [Yasuda+, CCSW 2013 ; Kim+, TDSC 2017 ; Bonte+, CCS 2020]

Boolean Approach	Arithmetic Approach
1. ***High memory footprint*** due to *encryption of individual bits*	1. ***Low memory footprint*** due to *data packing mechanism*
2. ***High computation cost*** due to *large number of HE operations*	2. ***Low computation cost*** due to *small number of HE operations*
3. ***Support flexible query sizes*** due to *unlimited computations*	3. ***Support limited query sizes*** due to *limited computations*

Approaches for HE-based String Matching

Secure string matching using HE can be performed using two key approaches

Boolean Approach

① Encrypt *individual* bits

e.g., m = 1 1 0 1 0 1 1 . . . 1 1 0

Enc(b_0) Enc(b_1) ... Enc(b_n)

Arithmetic Approach

① Encrypt *multiple packed* bits

e.g., m = [ ] [ ] 1 . . . [ ]

$P_1 = b_2x^2 + b_1x^1 + b_0$ $P_2(x) = b_5x^2 + b_4x^1 + b_3$ ... $P_n(x)$

Enc($P_1(x)$) Enc($P_2(x)$) ... Enc($P_n(x)$)

Approaches for HE-based String Matching

Secure string matching using HE can be performed using two key approaches

Boolean Approach

1. Encrypt individual bits
2. Perform ***homomorphic XOR and AND*** operation

$$
\begin{array}{cccc}
Enc(b_0) & Enc(b_1) & \dots & Enc(b_n) \\
\oplus & \oplus & \dots & \oplus \\
Enc(q_0) & Enc(q_1) & \dots & Enc(q_n) \\
\hline
Enc(r_0) \;\&\; & Enc(r_1) \;\&\; & \dots \;\&\; & Enc(r_n)
\end{array}
$$

Result

Arithmetic Approach

1. Encrypt multiple packed bits
2. Perform ***homomorphic multiplication and addition*** operation

$$
\begin{array}{cccc}
Enc(P_1(x)) & Enc(P_2(x)) & \dots & Enc(P_n(x)) \\
\times & \times & \dots & \times \\
Enc(Q_1(x)) & Enc(Q_2(x)) & \dots & Enc(Q_n(x)) \\
\hline
Enc(R_1(x)) \;+\; & Enc(R_2(x)) \;+\; & \dots \;+\; & Enc(R_n(x))
\end{array}
$$

Result

Prior Works on HE-based String Matching

- Boolean Approach [Pradel+, TrustCom 2021 ; Aziz+, Information 2024]
- Arithmetic Approach [Yasuda+, CCSW 2013 ; Kim+, TDSC 2017 ; Bonte+, CCS 2020]

Boolean Approach

1. ***High memory footprint*** due to *encryption of individual bits*

$m = 1\ 1\ 0\ 1\ 0\ 1\ 1 \ldots 1\ 1\ 0$

$Enc(b_0)$ $Enc(b_1)$ ... $Enc(b_n)$

Arithmetic Approach

1. ***Low memory footprint*** due to *data packing mechanism*

$m = \boxed{1\ 1\ 0}\ \boxed{1\ 0\ 1}\ 1 \ldots \boxed{1\ 1\ 0}$

$Enc(P_1(x))$ $Enc(P_2(x))$ ... $Enc(P_n(x))$

Prior Works on HE-based String Matching

- Boolean Approach [Pradel+, TrustCom 2021 ; Aziz+, Information 2024]
- Arithmetic Approach [Yasuda+, CCSW 2013 ; Kim+, TDSC 2017 ; Bonte+, CCS 2020]

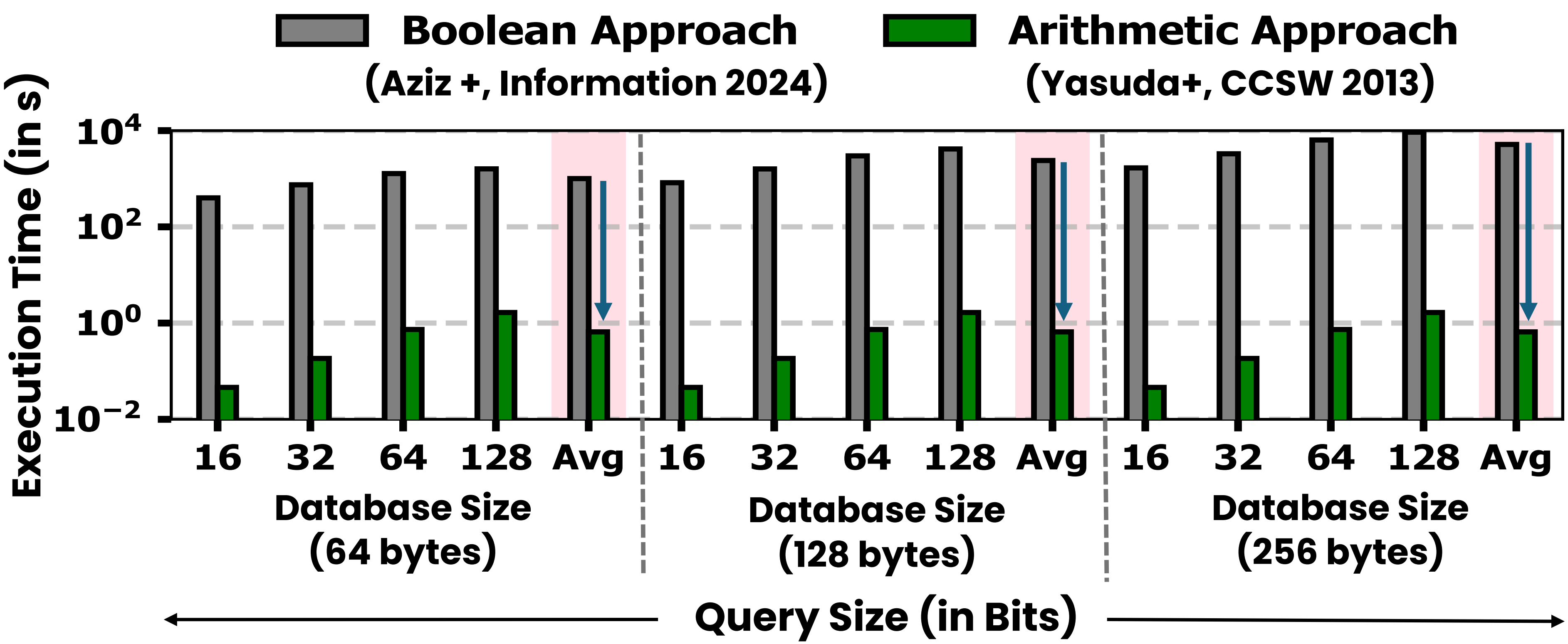
Boolean Approach

1. ***High memory footprint*** due to *encryption of individual bits*
2. ***Support flexible query sizes*** due to *unlimited computations*

Arithmetic Approach

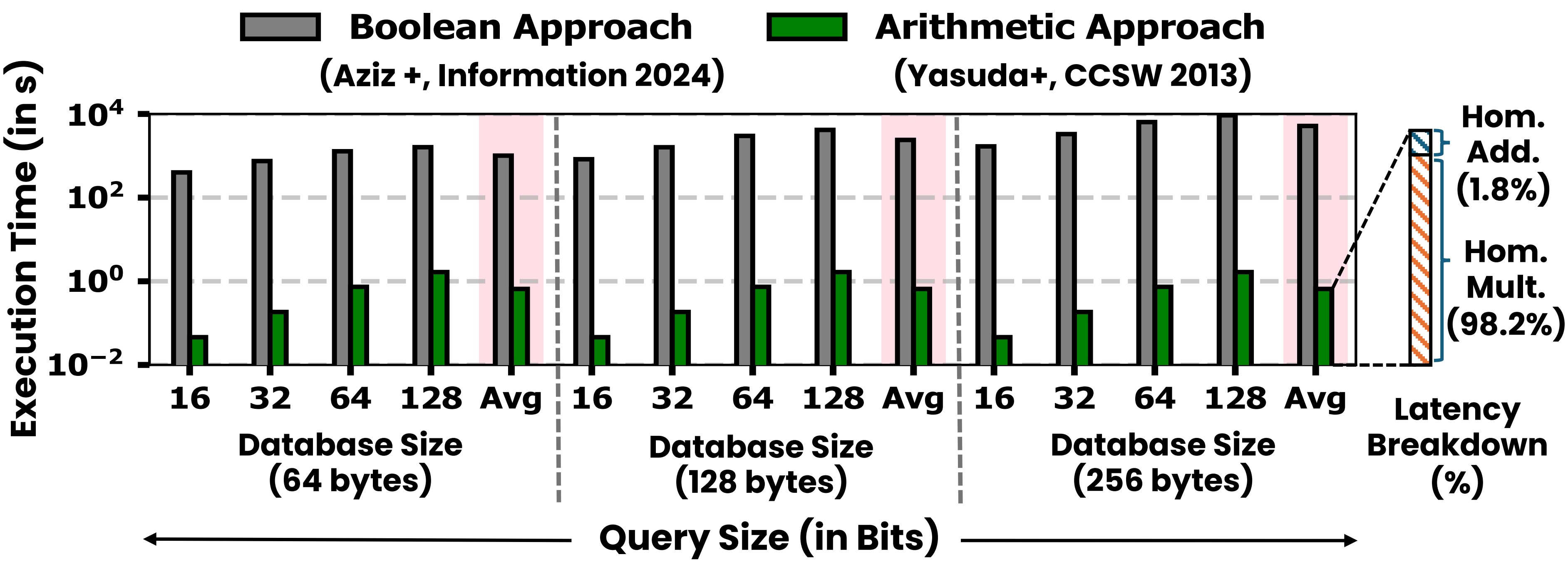
1. ***Low memory footprint*** due to *data packing mechanism*
2. ***Support limited query sizes*** due to *limited computations*

Arithmetic Approach Performs Better

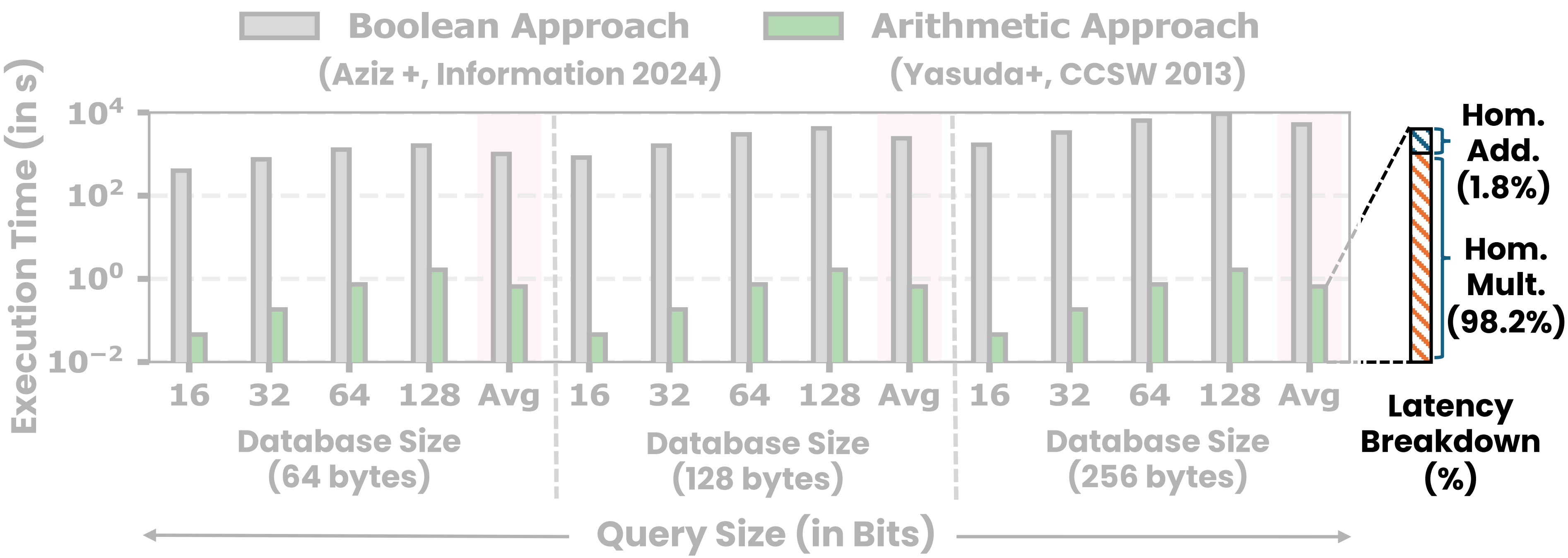


Arithmetic approach *performs better*
with larger database sizes due to fewer HE operations

Latency Breakdown of Arithmetic Approach



Key Problem (I): Complex HE Operations



Prior arithmetic approaches
use *costly homomorphic multiplication operations*
which limits *the scalabilty of HE-based string matching*

Key Observation

String matching can be performed using addition operation

If we negate the data, add it to the original data,
we get a string of 1 1 1 1's

$$
\begin{array}{rl}
m = & 01\ 01\ 00\ 10\ 10\ 01\ 01 \\
+ \sim m = & \underline{10\ 10\ 11\ 01\ 01\ 10\ 10} \\
 & \boxed{11\ 11\ 11\ 11\ 11\ 11\ 11}
\end{array}
$$

Value which can be checked

Key Observation

String matching can be performed using addition operation

$$
\begin{array}{rl}
m = & 01\ 01\ 00\ 10\ 10\ 01\ 01 \\
+\ {\sim}m = & \underline{10\ 10\ 11\ 01\ 01\ 10\ 10} \\
 & \boxed{11\ 11\ 11\ 11\ 11\ 11\ 11}
\end{array}
$$

Value which can be checked

Secure string matching can be performed using **HE addition** operation

$$
\begin{array}{rrl}
 & \mathrm{Enc}(m) = & (5\,x^{1024} + 10\,x^{1023} + \ \dots + 19\,,\ \dots) \\
+ & \mathrm{Enc}({\sim}m) = & \underline{(6\,x^{1024} + 11\,x^{1023} + \ \dots + 3\,,\ \dots)} \\
 & & \boxed{(11\,x^{1024} + 21\,x^{1023} + \ \dots + 22\,,\ \dots)}
\end{array}
$$

↓

Decrypt

↓

$$(1111\dots1x^{1024} + 1111\dots1\,x^{1023} + \ \dots + 1111\dots1)$$

Key Observation

String matching can be performed using addition operation

$$
\begin{array}{rl}
m = & 01\ 01\ 00\ 10\ 10\ 01\ 01 \\
+ \sim m = & 10\ 10\ 11\ 01\ 01\ 10\ 10 \\
\hline
 & \boxed{11\ 11\ 11\ 11\ 11\ 11\ 11}
\end{array}
$$

Value which can be checked

This output after homomorphic addition is an encrypted value of 1111's ***that can be used for matching***

$+ \quad Enc(\sim m) = (6\ x^{1024} + 11\ x^{1023} + \ldots + 3\ , \ldots)$

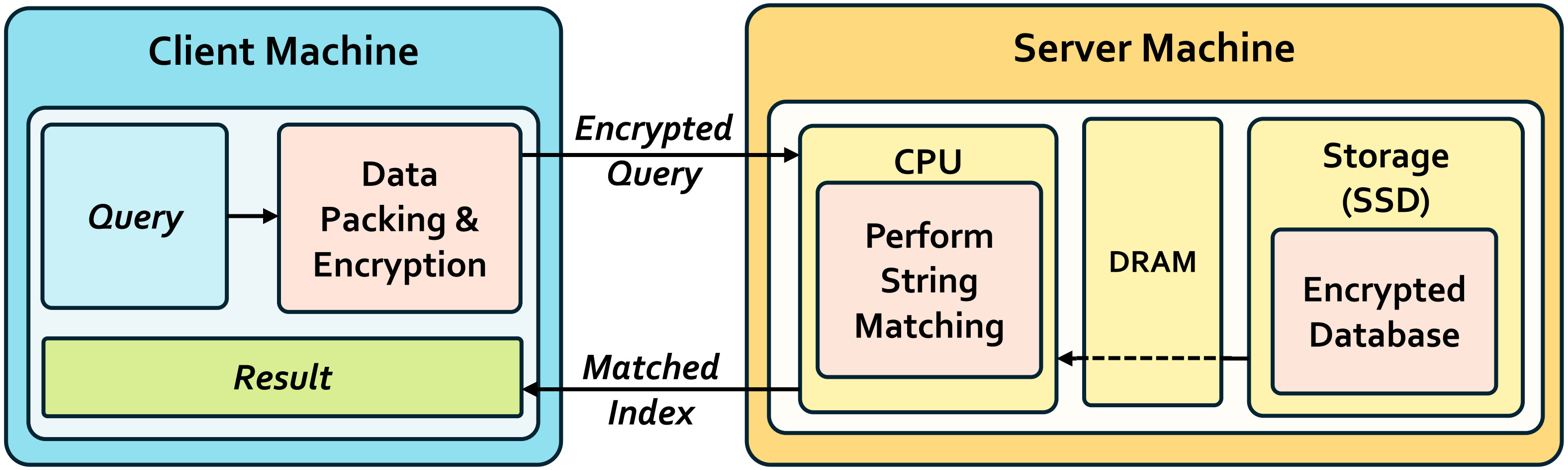
$\boxed{(11\ x^{1024} + 21\ x^{1023} + \ldots + 22\ , \ldots)}$

Decrypt

$(1111\ldots1x^{1024} + 1111\ldots1\ x^{1023} + \ldots + 1111\ldots1)$

Memory-Efficient Data Packing Scheme

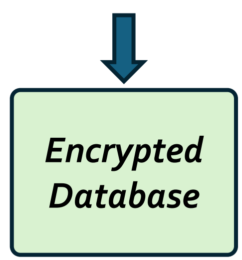
1



Assume, **Database (d)** = 1 0 1 0 1 1 1 0 0 1 0 1 1 1 1 … 1 0 1 0 1

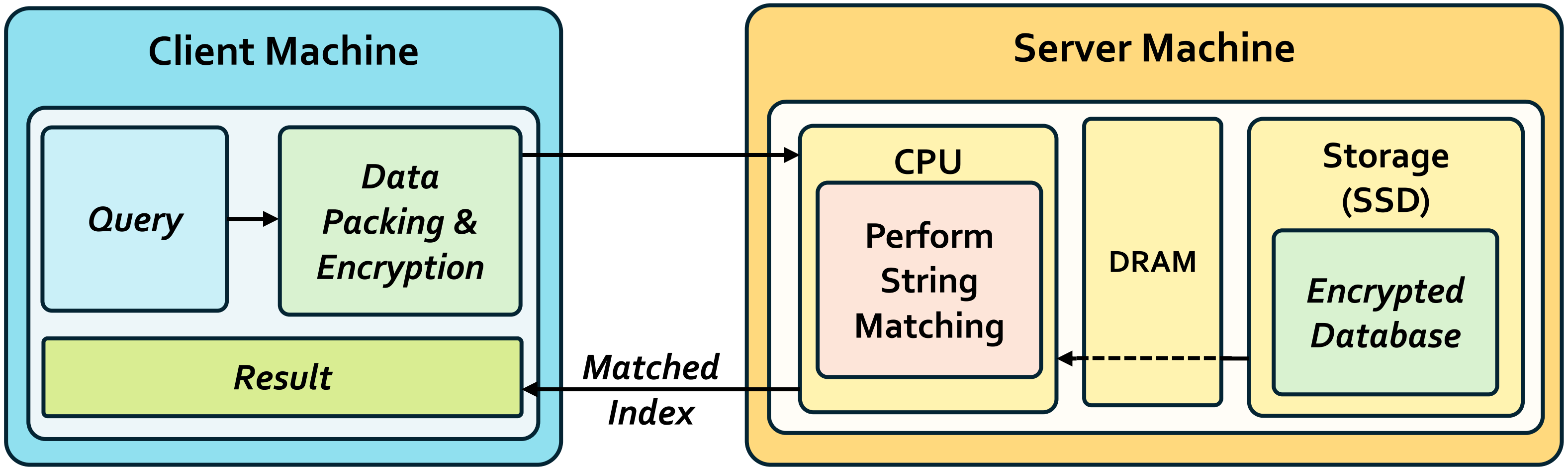
Encode database into multiple **plaintext polynomials (P(x))** by ***packing multiple bits*** into a single polynomial coefficient

e.g., $P(x) = 1\,0\,1\,0\,1 \ldots 1\, x^{1024} + 1\,0\,0\,1\,0 \ldots 1\, x^{1023} + \ldots. + 1\,1 \ldots 1\,0\,1\,0\,1$



CIPHERMATCH: Data Packing Scheme

1



Assume, **Query (q)** = 1 1 1 0 0 1 1 ⟶ 0 0 0 1 1 0 0

The query (q) is ***negated, replicated***
and encoded into the **plaintext polynomials (Q(x))**

e.g., Q(x) = $0\ 0\ 0\ 1\ 1 \ldots 0\ x^{1024} + 0\ 0\ 0\ 1\ 1 \ldots 0\ x^{1023} + \ \ldots. + \ 0\ 0\ 0\ 1\ 1 \ldots 0$

⬇

Encrypted Query

CIPHERMATCH: Identify the Match

2

String matching can be performed using addition operation

$$
\begin{array}{rl}
m = & 01\ 01\ 00\ 10\ 10\ 01\ 01 \\
+ \sim m = & 10\ 10\ 11\ 01\ 01\ 10\ 10 \\
 & 11\ 11\ 11\ 11\ 11\ 11\ 11
\end{array}
$$

Value which can be checked

This output after homomorphic addition is an encrypted value of 1111's
that can be used for matching

$+ \quad Enc(\sim m) = (6\ x^{1024} + 11\ x^{1023} + \ldots + 3\ , \ldots)$

$\boxed{(11\ x^{1024} + 21\ x^{1023} + \ldots + 22\ , \ldots)}$

Decrypt

$(1111\ldots1x^{1024} + 1111\ldots1\ x^{1023} + \ldots + 1111\ldots1)$

CIPHERMATCH: Identify the Match

2

String matching can be performed using addition operation

$$
\begin{array}{rl}
m = & 0\,1\;\;0\,1\;\;0\,0\;\;1\,0\;\;1\,0\;\;0\,1\;\;0\,1 \\
+\;\; \sim m = & 1\,0\;\;1\,0\;\;1\,1\;\;0\,1\;\;0\,1\;\;1\,0\;\;1\,0 \\
\hline
& \boxed{1\,1\;\;1\,1\;\;1\,1\;\;1\,1\;\;1\,1\;\;1\,1\;\;1\,1}
\end{array}
$$

Value which can be checked

Secure String matching can be performed using **HE addition** operation

$$
\begin{array}{rcl}
\text{Enc}(m) & = & (5\,x^{1024} + 10\,x^{1023} + \;....\; + 19\,,\,...) \\
+\;\text{Enc}(\sim m) & = & (6\,x^{1024} + 11\,x^{1023} + \;....\; + 3\,,\,...) \\
\hline
& & \boxed{(11\,x^{1024} + 21\,x^{1023} + \;....\; + 22\,,\,...)}
\end{array}
$$

Decrypt

$$(1010...1x^{1024} + \boxed{1111...1}\,x^{1023} + \;....\; + 1001...0)$$

CIPHERMATCH: Identify the Match

2

Encrypted Database $\mathrm{Enc}(m) = (5x^{1024} + 10x^{1023} + \ldots + 19, \ldots)$

Encrypted Query $+\mathrm{Enc}(\sim q) = (2x^{1024} + 14x^{1023} + \ldots + 3, \ldots)$

Result $= (7x^{1024} + 24x^{1023} + \ldots + 22, \ldots)$

Decrypt

$(1010\ldots1x^{1024} + \boxed{1111\ldots1}\,x^{1023} + \ldots + 1001\ldots0)$

Match polynomial $= 111\ldots11\,x^{1024} + 111\ldots11\,x^{1023} + \ldots + 111\ldots11$

Encrypt

Match value $= (91x^{1024} + 24x^{1023} + \ldots + 32, \ldots)$

CIPHERMATCH: Identify the Match

2

Encrypted Database $\mathrm{Enc}(m) = (5x^{1024} + 10x^{1023} + \ldots + 19, \ldots)$

Encrypted Query $+\mathrm{Enc}(\sim q) = (2x^{1024} + 14x^{1023} + \ldots + 3, \ldots)$

Result $= (7x^{1024} + 24x^{1023} + \ldots + 22, \ldots)$

Decrypt

$(1010\ldots1x^{1024} + \boxed{1111\ldots1}\,x^{1023} + \ldots + 1001\ldots0)$

Match polynomial $= 111\ldots11\,x^{1024} + 111\ldots11\,x^{1023} + \ldots + 111\ldots11$

Encrypt

Match value $= (91x^{1024} + 24x^{1023} + \ldots + 32, \ldots)$

CIPHERMATCH: Identify the Match

2

Result $= (7\,x^{1024} + \boxed{24}\,x^{1023} + \ldots + 22\,, \ldots)$

Compare → **Generate Index**

Match value $= (91\,x^{1024} + \boxed{24}\,x^{1023} + \ldots + 32\,, \ldots)$

Compare the ***match value***

Qualitative Analysis of Prior Work

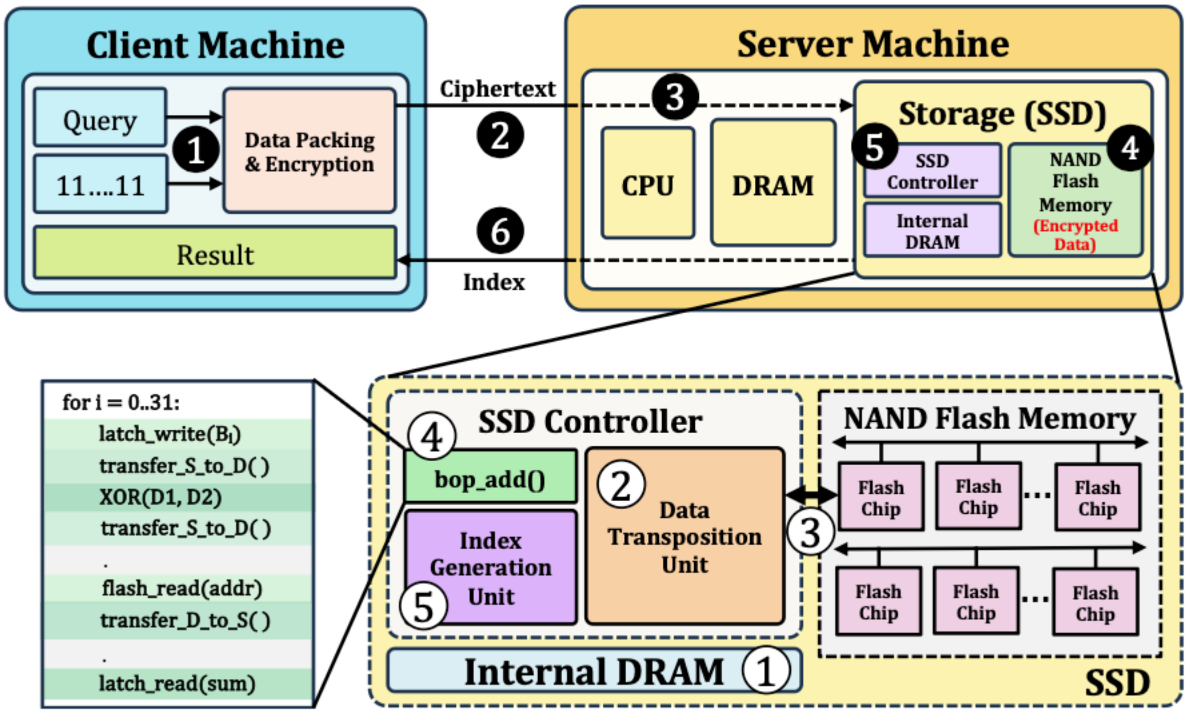


Figure 6: System-level overview of CIPHERMATCH.

System-Level Overview of CIPHERMATCH

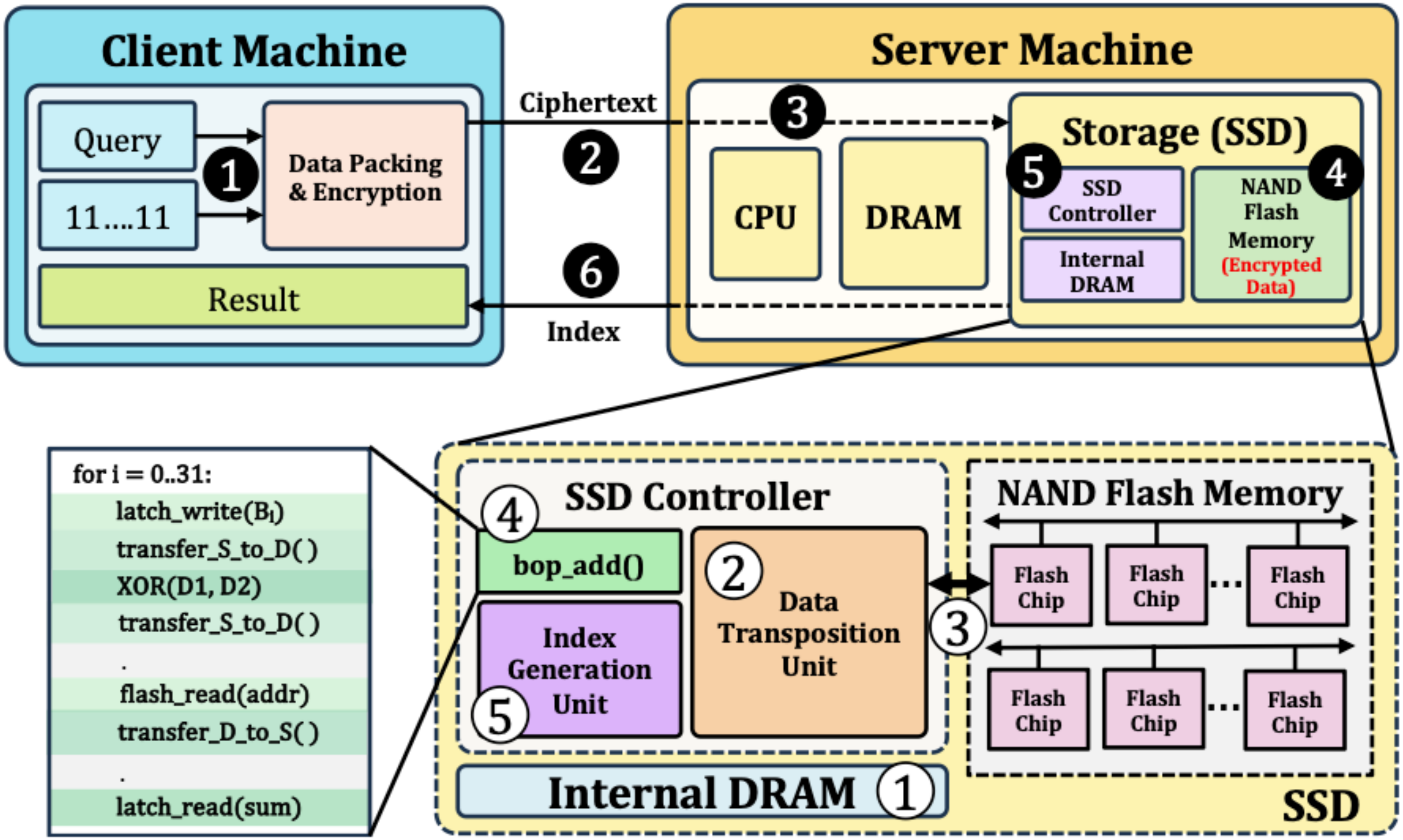


Figure 6: System-level overview of CIPHERMATCH.

Evaluation Configuration

CPU: Intel(R) Xeon(R) Gold 5118	*Microarchitecture:* Intel Skylake [149]
	x86-64 [150], 6 cores, out-of-order, 3.2 GHz
	L1 Data + Inst. Private Cache: 32kB, 8-way, 64B line
	L2 Private Cache: 256kB, 4-way, 64B line
	L3 Shared Cache: 8MB, 16-way, 64B line
Main Memory	32GB DDR4-2400, 4 channels
Storage (SSD)	Samsung 980 Pro PCIe 4.0 NVMe SSD 2 TB [102]
Operating System (OS)	Ubuntu 22.04.1 LTS

Table 2: Real CPU system configuration.

Evaluation Configuration

CM-PuM	32 GB DDR4-2400, 4 channel, 1 rank, 16 banks; Peak throughput: 19.2 GB/s
	Latency: T_{bbop}: 49 ns; **Energy:** E_{bbop}: 0.864 nJ; where *bbop* is bulk bitwise operation
	SSD External-Bandwidth: 7-GB/s external I/O bandwidth; (4-lane PCIe Gen4)
CM-IFP and CM-PuM-SSD	48-WL-layer 3D TLC NAND flash-based SSD; 2 TB
	SSD Internal DRAM: 2GB LPDDR4-1866 DRAM cache; 1 channel, 1 rank, 8 banks
	NAND-Flash Channel Bandwidth: 1.2-GB/s Channel IO rate
	Controller Cores: ARM Cortex-R5 series @1.5GHz; 5 Cores [153]
	NAND Config: 8 channels; 8 dies/channel; 2 planes/die; 2,048 blocks/plane; 196 (4×48) WLs/block; 4 KiB/page
	Latency: T_{read} (SLC mode): 22.5 μs [60]; $T_{AND/OR}$: 20 ns [62]; $T_{latchtransfer}$: 20 ns [62]; T_{XOR}: 30 ns [60]; T_{DMA}: 3.3 μs; T_{bit_add} (CM_IFP): 29.38μs
	Energy: E_{read} (SLC mode): 20.5μJ/channel [60]; $E_{AND/OR}$: 10nJ/KB [62]; $E_{latchtransfer}$: 10nJ/KB [62]; E_{XOR}: 20nJ/KB [60]; E_{DMA}: 7.656μJ/channel; E_{index_gen} (SSD controller): 0.18μJ/page size; E_{bit_add} (CM_IFP): 32.22μJ/channel

Table 3: Simulated system configurations.