

PIRacle: A Fast and Scalable Private Information Retrieval System for Key-Value Stores

Zehao Chen[✉], Zhaoyan Shen[✉], Yi Wang[✉], *Member, IEEE*, Hang Lu[✉], and Lei Ju[✉]

Abstract—Keyword Private Information Retrieval (PIR) systems that leverage Homomorphic Encryption (HE) allow for the execution of private queries on public databases hosted by untrusted servers. However, existing solutions are significantly hindered by high computational overhead and inefficiencies in handling variable-length data, which impose limitations on their practical deployment. To address these challenges, we propose PIRacle, a high-performance protocol and deployment-optimized private retrieval system for key-value stores. First, PIRacle introduces a lightweight computational protocol that substantially reduces computational overhead by utilizing constant-weight codes and Fermat's Little Theorem. Additionally, to address the inefficiencies associated with variable-length data, PIRacle applies a two-pass sorting strategy combined with computation pruning, effectively mitigating the computational expansion typically induced by padding overhead in conventional designs. Moreover, PIRacle incorporates a workload-aware scheduling framework that dynamically distributes computation across parallel execution units, enabling efficient scaling with available hardware resources. Experimental results demonstrate that PIRacle reduces server-side latency by up to $88.5\times$ compared to the state-of-the-art solution.

Index Terms—Homomorphic encryption, private information retrieval, homomorphic equality operator.

Received 26 May 2025; revised 9 November 2025; accepted 11 January 2026. Date of publication 21 January 2026; date of current version 13 March 2026. This work was supported in part by the National Natural Science Foundation of China under Grant 62432005, Grant 62372272, and Grant 62272271; in part by Shandong Provincial Natural Science Foundation under Grant ZR2024MF099 and Grant ZR2023LZH014; in part by the Department of Science and Technology of Shandong Province under Grant SYS202201; in part by the Research Project of Quan Cheng Laboratory under Grant QCL20250101 and Grant QCL20250301; and in part by the Taishan Scholars Program under Grant TSQN202211281. Recommended for acceptance by M. Ka n che. (Corresponding author: Lei Ju.)

Zehao Chen is with the School of Cyber Science and Technology and the State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao 266237, China, and also with Quan Cheng Laboratory, Jinan 250101, China (e-mail: czh-rot@mail.sdu.edu.cn).

Zhaoyan Shen is with the School of Computer Science and Technology, Shandong University, Qingdao 266237, China (e-mail: shen-zhaoyan@sdu.edu.cn).

Yi Wang is with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China (e-mail: yi-wang@szu.edu.cn).

Hang Lu is with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with Zhongguancun Laboratory, Beijing 100094, China (e-mail: luhang@ict.ac.cn).

Lei Ju is with Quan Cheng Laboratory, Jinan 250101, China (e-mail: sr-julei@qcl.edu.cn).

Digital Object Identifier 10.1109/TC.2026.3656406

I. INTRODUCTION

THE widespread adoption of cloud computing has led to a significant transformation in data query practices, with remote servers increasingly serving as central platforms for large-scale data storage and access [1], [2]. However, this shift has also introduced substantial privacy risks, especially in the context of public data querying scenarios [3], [4], [5], [6]. In such scenarios, clients submit plaintext queries to the cloud, potentially exposing their query interests to the server, which may compromise privacy. To address this issue, Private Information Retrieval (PIR) [7], [8] has emerged as a promising solution. It allows clients to submit encrypted queries and retrieve results from a public plaintext database, ensuring that the server learns nothing about the client's query intent.

Currently, PIR systems are primarily categorized into two main types: Index PIR [9], [10], [11], [12], [13], [14], [15] and Keyword PIR [7], [16], [17], [18]. Index PIR designs assume that the server database consists of n -entry arrays, allowing clients to retrieve the i -th entry from the untrusted server without revealing their points of interest. This approach relies on Homomorphic Encryption (HE) schemes [19], [20], which enable the extraction of the target entry by performing computations over all entries in the database. However, in practical applications such as contact lists, video libraries, and websites, datasets are typically organized as key-value pairs. In these cases, clients seek to retrieve the value associated with a specific key rather than an entry by index, rendering Index PIR impractical. To address this limitation, Keyword PIR extends Index PIR to enable keyword-based retrieval. It introduces a homomorphic equality operator (EQ) [21], which allows the server to perform homomorphic equality comparisons between the client's query and each key in the database, generating an encrypted index. The server can then use this encrypted index, along with Index PIR, to retrieve the corresponding value privately.

Despite the advancements in Keyword PIR that make it suitable for real-world scenarios, scaling these solutions to large-scale commercial applications continues to present a significant challenge. The most prominent issue remains the inefficiency in computational performance. For instance, the state-of-the-art Pantheon system [17], deployed on an AWS instance with 48 vCPUs, requires over an hour to process a single private query on a key-value store containing 1 million entries. In contrast, traditional non-encrypted systems can typically process such a query within seconds. The root cause of this inefficiency lies in the substantial computational overhead introduced by

the *EQ*. We observe that, due to the high computational cost of HE operations, such as *Mult* and *Bootstrapping*, the *EQ* operator accounts for over 93% of the total execution time (details in § II-D1), making it the primary bottleneck in query processing and significantly limiting system performance.

Another key challenge lies in the inefficiency of retrieval for variable-length datasets. To exploit the SIMD capabilities [19], [20] of HE, existing PIR systems typically model the database as a two-dimensional matrix of “uniform” records. However, this assumption rarely holds in practice—real-world data from platforms like Amazon [22], Google [23], and Facebook [24], [25] exhibit significant variability, with standard deviations often exceeding 30% of the mean. To meet the requirement of HE for structural regularity, servers must pad all data items to match the longest record during processing. While this formally restores the matrix consistency required by HE, it substantially increases data volume and storage overhead. More importantly, since PIR inherently performs a linear scan over the entire dataset, the data inflation introduced by padding further enlarges the computational burden, degrading overall system efficiency (see § II-D2 for details).

To address these challenges, we propose PIRacle, a fast and scalable keyword PIR system designed to support variable-length key-value entries. PIRacle achieves its efficiency and scalability through the integration of the following key techniques. First, building upon prior constant-weight code (CWC) [26] and Fermat’s little theorem (FLT) [27]-based mechanisms, PIRacle refines the *EQ* to achieve lightweight computation—requiring only a few *Mult* operations—and a fixed, shallow multiplicative depth that eliminates bootstrapping. This refinement preserves the theoretical basis of existing CWC- and FLT-based approaches (e.g., [16], [17]) while substantially improving their practical efficiency and scalability. Secondly, for variable-length data, PIRacle adopts a two-pass stable sorting to decouple the padding characters from the payload. This design enables PIRacle to skip computations on padding characters through computation pruning, thus avoiding meaningless computational overhead. Finally, PIRacle integrates recent advances in HE acceleration to achieve substantial computational gains. It introduces a workload-aware task scheduling framework that efficiently distributes workloads across parallel compute units, enabling scalable performance as additional resources become available. This design significantly improves throughput and makes PIRacle well-suited for large-scale, deployment-oriented applications.

In summary, the contributions of this work are as follows:

- We optimize the homomorphic equality operator. Compared to prior approaches, our design achieves fixed multiplicative depth and reduces equality check overhead by 1~2 orders of magnitude.
- We introduce a two-pass stable sorting mechanism coupled with computation pruning, enabling the system to skip redundant operations on padding and significantly improve performance on variable-length datasets.
- We develop a dynamic scheduling mechanism that ensures balanced workload distribution across heterogeneous compute environments.

- Extensive evaluations on five real-world datasets show that PIRacle achieves up to $88.5\times$ speedup over state-of-the-art keyword PIR baselines.

II. BACKGROUND Motivation

A. Fully Homomorphic Encryption

Homomorphic encryption (HE) [19], [20], a cryptographic method that enables the computation of encrypted data, is often employed as a fundamental component in constructing PIR solutions. In this paper, we focus on the BFV [19] scheme among various HE schemes, due to its widespread adoption in existing PIR solutions.

In the BFV scheme, a plaintext x is represented as a vector of length N , where N belong to $2^{10}, 2^{11}, \dots, 2^{15}$. Each element within this vector is an integer modulo a prime T , i.e., each element $< T$. This vector format constitutes the fundamental data unit processed by the BFV scheme. By employing a public key pk , the `BFV.encrypt` function can encrypt the plaintext vector x into the corresponding ciphertext $\llbracket x \rrbracket$. This ciphertext consists of two vectors, with each coefficient being an integer modulo Q , where Q significantly exceeds T to ensure the robustness and security of the encryption. The ciphertext $\llbracket x \rrbracket$ can be decrypted back into the original plaintext x using the private key sk and `BFV.decrypt` function. Assume $N = 4$, x is $[1, 2, 3, 4]$, and the BFV scheme supports the following arithmetic operations:

- **Add (+)/Sub (-)**: receives ciphertexts or plaintexts and outputs their sum/difference. e.g., $\llbracket x \rrbracket + \llbracket x \rrbracket = \llbracket x + x \rrbracket$, encrypting $[2, 4, 6, 8]$, respectively.
- **PMult (·)**: receives a ciphertext and a plaintext and outputs their product. e.g., $\llbracket x \rrbracket \cdot x = \llbracket x \cdot x \rrbracket$, encrypting $[1, 4, 9, 16]$.
- **Mult (·)**: receives two ciphertexts and outputs their product. e.g., $\llbracket x \rrbracket \cdot \llbracket x \rrbracket = \llbracket x \cdot x \rrbracket$, encrypting $[1, 4, 9, 16]$.
- **Rotate (ϕ)**: receives one ciphertext $\llbracket x \rrbracket$ and an integer k and outputs $\phi_k(\llbracket x \rrbracket)$. e.g., $\phi_k(\llbracket x \rrbracket)$ encrypts $[4, 1, 2, 3]$.
- **Bootstrapping (BS)**: In the BFV scheme, each ciphertext contains a noise component and a predefined noise threshold. This noise component grows asymmetrically during arithmetic operations, particularly with `PMult` and `Mult`. Once the noise component exceeds the predefined threshold, the ciphertext becomes undecryptable. Therefore, the BFV scheme only supports a limited number of serial multiplication operations on ciphertexts. To address the issue of noise accumulation, *Bootstrapping* [28], serving as the most expensive homomorphic operations, can reset the noise to a low level to further support subsequent calculations. Table I presents the computational cost with different BFV operations.

B. Homomorphic Equality Operators

Equality checking over encrypted values is a fundamental operation in a wide range of encrypted data processing tasks,

TABLE I
COST OF OPERATIONS IN SEAL. $N = 2^{15}$, $\log_2 Q = 881$

Operation	Output	CPU cost (ms)	Noise
Sub($\llbracket x \rrbracket$, $\llbracket y \rrbracket$)	$\llbracket x - y \rrbracket$	1.2	-
Add($\llbracket x \rrbracket$, $\llbracket y \rrbracket$)	$\llbracket x + y \rrbracket$	1.2	-
PMult($\llbracket x \rrbracket$, y)	$\llbracket x \cdot y \rrbracket$	39.3	$> \log_2 T$
Mult($\llbracket x \rrbracket$, $\llbracket y \rrbracket$)	$\llbracket x \cdot y \rrbracket$	272.1	$> \log_2 T$
Rotate($\llbracket x \rrbracket$, k)	$\phi_k(\llbracket x \rrbracket)$	168.3	-
Bootstrapping($\llbracket x \rrbracket$)	$\llbracket x \rrbracket_{BS}$	> 1 minute	-

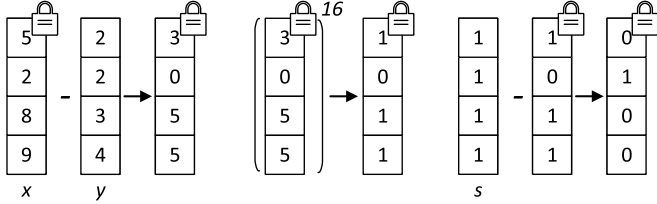


Fig. 1. The equality check procedure of $\llbracket x \rrbracket$ and y .

including secure search [29], [30] and Keyword Private Information Retrieval (PIR) [16], [17]. In this context, the homomorphic equality operator, denoted as EQ , is defined:

$$EQ(\llbracket a \rrbracket, b) = \begin{cases} \llbracket 1 \rrbracket, & \text{if } a = b, \\ \llbracket 0 \rrbracket, & \text{if } a \neq b. \end{cases}$$

Currently, the most advanced approach [17], [21] leverages Fermat's Little Theorem [31], which states that for any prime modulus T and integer $a \not\equiv 0 \pmod T$, it holds that $a^{T-1} \equiv 1 \pmod T$. This property integrates naturally with the BFV scheme, where plaintext arithmetic is performed modulo T . Therefore, given two plaintext vectors x and y of length N , with entries in $\mathbb{Z}_T$, element-wise equality in the encrypted domain can be computed via:

$$EQ(\llbracket x \rrbracket, y) = s - \llbracket x - y \rrbracket^{T-1} \quad (1)$$

where s is $[1, 1, \dots, 1]$. For illustration, let $N = 4$, $T = 17$, $x = [5, 2, 8, 9]$, and $y = [2, 2, 3, 4]$. As in Fig. 1, $EQ(\llbracket x \rrbracket, y)$ returns a ciphertext encrypting the vector $[0, 1, 0, 0]$.

The equality operator described above can be extended to support encrypted comparison of multi-segment vectors. Consider two vectors m and n of length N , where each element is a large integer represented using $l = t \times \log_2(T-1)$ bits. Since the BFV scheme operates over plaintext slots of integers less than T , each element must first be decomposed into t smaller segments, each fitting within $\log_2(T-1)$ bits. Let the decomposed forms be $m = \{m_1, m_2, \dots, m_t\}$ and $n = \{n_1, n_2, \dots, n_t\}$, where each $m_i, n_i \in \mathbb{Z}_T$. The extended equality operator is then defined as:

$$EQ(\llbracket m \rrbracket, n) = \prod_{i=1}^t (s - \llbracket m_i - n_i \rrbracket^{T-1})_{BS} \quad (2)$$

Each pair of segments $(\llbracket m_i \rrbracket, n_i)$ is compared independently using the single-segment equality logic, and the final result is obtained by multiplying the outputs of all sub-comparisons.

• **Observation #1:** The equality operator is computationally expensive. First, it incurs a large number of homomorphic multiplications (Mult), with a total of $\log_2(T-1) \cdot t + t - 1$ operations required. Second, it exhibits a substantial multiplicative depth of $\log_2(T-1) + t - 1$, which often necessitates repeated invocations of expensive Bootstrapping to manage ciphertext noise for each sub-comparison. Notably, both the multiplicative cost and circuit depth grow linearly with the bitwidth l of the input integers.

C. Private Information Retrieval

1) *PIR Definition:* Private Information Retrieval (PIR) is designed to address the problem of privacy-preserving data retrieval over public databases [7]. A PIR system typically involves three entities: the client, the server, and the database hosted on the server. The system operates under the **semi-honest threat model** [8], [9], [10], [11], [12], [13], [16], [17], in which the server correctly follows the protocol but may attempt to infer sensitive information about the client's query by observing the protocol's execution behavior. Under this assumption, PIR guarantees that clients can retrieve specific entries from the server-hosted database DB without revealing any information about their query intent.

2) *PIR Categories:* While all PIR systems share the goal of ensuring query privacy, their specific implementations vary significantly depending on how databases are organized and accessed. Consequently, PIR schemes are commonly classified into two major categories based on database structure and query approach: Index PIR and Keyword PIR.

Index PIR. Index PIR [9], [10], [11], [12], [32], [33], [34] enables clients to privately retrieve the i -th entry from a public database structured as an n -entry array $DB = \{d_0, d_1, \dots, d_{n-1}\}$, without revealing the index i to the server. While efficient in terms of computation and communication, Index PIR assumes that the client knows the exact index of the target entry. In many real-world scenarios, however, clients only possess high-level identifiers (e.g., keys) and must first download the complete key set to establish a local key-to-index mapping [7], [35]. This additional step significantly increases communication overhead and limits the scheme's scalability in large or frequently updated databases.

Keyword PIR. To overcome the limitations of Index PIR, Keyword PIR schemes [16], [17] have been proposed, which assume the database is structured as a set of key-value pairs, i.e., $DB = \{(k_1, v_1), (k_2, v_2), \dots, (k_{n-1}, v_{n-1})\}$, where each key k_i serves as a unique identifier. Keyword PIR enables clients to retrieve values using encrypted queries based solely on the target key, without prior knowledge of index positions. This keyword-based retrieval is made possible by equality operators (EQ) in § II-B, which allows the server to compare the encrypted query key against all plaintext keys in the database, thereby computing the encrypted indices. These results are then used to retrieve the corresponding value via the Index PIR. However, as demonstrated in **Observation #1**, Keyword PIR schemes typically exhibit lower retrieval efficiency than Index

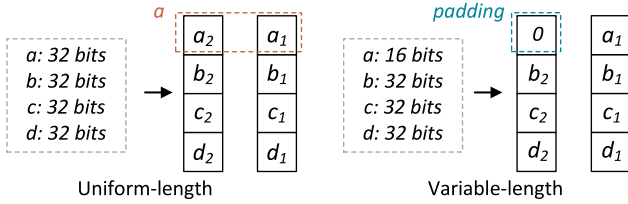


Fig. 2. Encoding process. Assume $N = 4, T = 2^{16} + 1$. Here, a_1, a_2 denote the two 16-bit segments of a .

PIR due to the significant computational overhead introduced by the EQ .

• **Observation #2:** Existing PIR schemes operate under the implicit constraint of uniform data length, primarily due to the batching semantics of homomorphic encryption and the structural regularity required for vectorized computation [9], [10], [12], [16], [17]. In these schemes, the server encodes the database as a structured matrix, enabling SIMD-style operations across ciphertext slots. While this representation poses no issue for uniform-length datasets, it becomes problematic when applied to variable-length entries—which are prevalent in real-world applications. As illustrated in Fig. 2, to achieve alignment across slots, the server must pad each entry with leading zeros to match the longest record, thereby inflating the size of the encoded dataset. Since PIR protocols typically perform linear scans over the entire database, this inflation proportionally increases the computational cost of each query, degrading overall efficiency.

D. Empirical Analysis of Keyword PIR Bottlenecks

To validate the performance challenges raised in Observation #1 and #2, we conduct experiments based on Pantheon [17], one of the most advanced keyword PIR systems to date. We use two variable-length datasets (e.g., UDB [24], ZiDB [24], UP2X [25], USR [25]) from Facebook and a uniform-length synthesized dataset (i.e., Uni) to evaluate the system, each containing 1 million key-value pairs. Full experimental settings and dataset details are described in § V.

1) *Inefficiency in Equality Operator:* Fig. 3(a) reports the average query latency for executing a private query using Pantheon across different datasets. Notably, the homomorphic equality operator (EQ) accounts for over 93% of the total server-side execution time in all cases. This dominant overhead arises from its reliance on costly BFV operations, particularly Mult and Bootstrapping, whose computational expense is detailed in Table I. By contrast, the subsequent Index PIR phase (denoted as “Rest Cost”) is significantly less demanding, as it primarily involves lightweight operations such as PMult and Add. These results clearly indicate that EQ constitutes the primary computational bottleneck in current systems.

This leads to our first design goal: to develop a new equality operator that minimizes the use of costly primitives such as Mult and Bootstrapping, and instead leverages more lightweight homomorphic operations wherever possible (§ III).

2) *Inefficiency in Variable-Length Handling:* As discussed in Observation #2, current HE-based PIR schemes handle

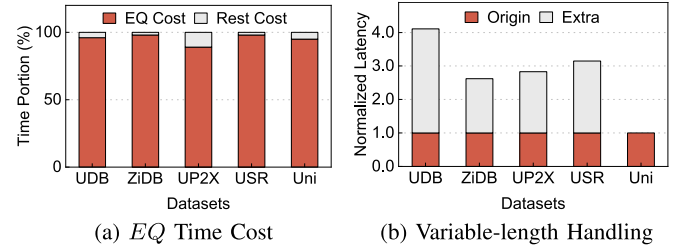


Fig. 3. (a) The proportion of EQ in the total time cost. (b) Retrieval time increases when handling variable-length datasets. The experimental details refer to § V.

variable-length records by padding each key and value to match the longest entry in the dataset, ensuring alignment for vectorized homomorphic operations. While this approach ensures correctness, it also introduces significant computational waste by performing redundant operations on padded characters. Fig. 3(b) illustrates this effect: *Origin* denotes computation on actual payload data, while *Extra* corresponds to computations over padding. Across the four variable-length datasets, padding increases the total homomorphic workload by factors of $4.11\times$, $2.62\times$, $2.83\times$, and $3.15\times$, respectively. In contrast, the fixed-length baseline incurs no such overhead. These results confirm that padding-based alignment is highly inefficient for variable-length data, and the inefficiency grows more severe as the variance in entry lengths increases.

Accordingly, our second design goal is to enable PIR systems to avoid redundant computations on padding and reduce overall system latency (§ IV).

III. PIRACLE CONSTRUCTIONS

A. Equality Operator Design

1) *Constant-Weight Code:* Constant-weight Code [26] plays a fundamental role in our lightweight homomorphic equality operator (EQ). Constant-weight code is a specialized encoding technique that transforms each integer to a m -bit code with a constant Hamming weight w . This can be viewed as a combinatorial problem, where w positions out of m are set to “1”, while the remaining positions are set to “0”. Therefore, for a fixed Hamming weight w , to have a binary constant-weight code with a size of at least n , we must choose the length m , such that $\binom{m}{w} \geq n$. For example, when $m = 4$ and $w = 2$, the total number of unique constant-weight code is determined by the binomial coefficient $\binom{m}{w} = 6$, allowing six distinct integers to be represented. In this case, the integer 1 is encoded as 0011, while 2 is encoded as 0101. In the following paper, we denote the binary Constant-weight code with length m and Hamming weight w by $CW(m, w)$.

2) *Arithmetic Constant-Weight Equality Operator:* For two Constant-weight codes, $x, y \in CW(m, w)$. Algorithm 1 describes the arithmetic equality operator applied in plaintext over constant-weight codes.

Proof: In Algorithm 1, if x and y are equal, the positions of the bits set to “1” in their codes are identical. Consequently, the inner product, e , will equal w . Conversely, if x and y are not

Algorithm 1: Equality Operator in plaintext

Input: $x, y \in CW(m, w)$
 $e \leftarrow \sum_{i=0}^{m-1} x[i] \cdot y[i]$
 $r \leftarrow 1 - (e - w)^{T-1} \bmod T$
Output: $I \in \{0, 1\}$

TABLE II
COMPUTATIONAL COMPLEXITY OF DIFFERENT EQ

	PMult	Mult	Bootstrapping	Mul Depth
Pantheon	0	$t \cdot p + t - 1$	$\approx t$	$p + t - 1$
CwPIR	$m \approx l$	$w \approx l/2$	$< \log w$	$w + 1$
PIRacle	m	p	0	$p + 1$

equal, the sum of the bits will be in the range $[0, w - 1]$. Therefore, when using $(e - w)$ as input to Fermat's Little Theorem, as defined in § II-B, it holds that

$$r = \begin{cases} 1 & e = w \\ 0 & e \in \{0, 1, \dots, w - 1\} \end{cases}$$

Homomorphic Equality Operator. The above arithmetic equality operator can be easily extended to the encrypted domain. Given two constant-weight codes, $x, y \in CW(m, w)$, we wish to check the equality between $\llbracket x \rrbracket$ and y , where $\llbracket x \rrbracket$ is in form of m ciphertexts encrypting $x[i]$, $i \in [0, m)$, respectively. The $EQ(\llbracket x \rrbracket, y)$ can be divided into two steps:

$$\begin{cases} (1) & \llbracket e \rrbracket = \sum_{i=0}^{m-1} \llbracket x[i] \rrbracket \cdot y[i] \\ (2) & \llbracket r \rrbracket = 1 - \llbracket e - w \rrbracket^{T-1} \end{cases} \quad (3)$$

Notably, as described in § II-A, BFV plaintexts and ciphertexts are batch-packed with N elements in $\mathbb{Z}_T$ to enable batch processing using SIMD parallel processing capabilities. This enables our homomorphic EQ based on constant-weight codes to compare N elements simultaneously.

3) *Complexity Analysis:* We compare the computational complexity of our proposed EQ operator with that of the conventional EQ used in Pantheon [17] and CwPIR [16]. The comparison focuses on the number of Mult, PMult, and Bootstrapping operations required to compare an encrypted integer against N candidates, each represented by l bits. The overall results are summarized in Table II.

In Pantheon, each l -bit integer is divided into $t = \lceil l / \log_2(T - 1) \rceil$ chunks according to the plaintext modulus T , where $p = \log_2(T - 1)$. Each segment-wise equality comparison requires p Mult operations, leading to a total of $t \cdot p + t - 1$ Mult. In contrast, CwPIR first encodes all integers into binary codewords $\in CW(m, w)$ using constant-weight codes. Each codeword is divided into m segments for sub-comparisons, and the partial results are aggregated through cumulative multiplications. This process requires approximately m PMult and w Mult operations, where $m \approx l$ and $w \approx l/2$. It is worth noting that in both Pantheon and CwPIR, the multiplicative depth grows linearly with the input bit width l . As a consequence, to mitigate decryption

failures caused by noise growth, frequent bootstrapping is required—up to $\mathcal{O}(l)$ times in practical deployments.

Our proposed EQ operator fundamentally differs in design. It combines the representational efficiency of constant-weight codes with the algebraic properties of Fermat's little theorem (FLT). The overall computation proceeds in two stages: first, lightweight subset comparisons are performed via constant-weight encoding; second, FLT is employed to efficiently aggregate the sub-results. This two-stage design offers two key advantages: (1) Compared with Pantheon, which solely relies on FLT, our method significantly reduces the cost of subset comparisons. Specifically, the original $t \cdot p + t - 1$ Mult operations are replaced by approximately m PMult (with $m \approx l$). (2) Compared with CwPIR, our design greatly optimizes the result aggregation. Instead of performing about w serial Mult operations, only p Mult operations are required. Most importantly, the multiplicative depth is reduced to $p + 1$, a constant independent of the bit width l . With appropriate homomorphic encryption parameter selection, such a depth can be supported without invoking any Bootstrapping operations.

B. Keyword PIR Constructions

In this section, we describe our Keyword PIR protocol using the EQ based on constant-weight codes, i.e., PIRacle. It builds upon existing Keyword PIR frameworks but introduces several optimizations to improve performance, consisting of six stages as follows. For ease of description, we assume that the database DB on the server side holds n key-value items with uniform length, where each value is l_v bits.

1) *Setup in Server Side:* In this stage, the server converts all keys in its database into constant-weight codes $\in CW(m, w)$. To ensure the mapping is injective, $\binom{m}{w}$ must be greater than $\max(key)$, where $\max(key)$ represents the largest key in the dataset. This process is described in Algorithm 2. For each $k_i \in$ the database DB , it assigns the k_i -th valid constant-weight code from a stored list of codes to k_i .

2) *Query Generation in Client Side:* In this stage, the client constructs its query and sends it to the server. First, it applies the Algorithm 2 to transform its target key to constant-weight code. Next, the client encrypts each bit of the constant-weight code into a separate ciphertext. Assume the target key of client is $x \in CW(m, w)$, the corresponding m ciphertexts encrypting $[x[0], x[0], \dots, x[0], \dots, [x[m - 1], x[m - 1], \dots, x[m - 1]]$, respectively.

Query Compress. In practical implementation, sending m ciphertexts directly to the server imposes a significant network bandwidth burden, especially when m is large. To reduce this overhead, PIRacle leverages the batching feature of the BFV encryption scheme, which allows packing N integers into a single ciphertext. This design significantly reduces the communication overhead for query transmission. Note that, on the server side, this ciphertext q will be expanded back into its original form of m ciphertexts to facilitate equality checks against all keys. The above process can be generalized for any m and N . As a result of communication compression, the PIRacle client only needs

Algorithm 2: Setup in PIRacle server

Input: $k_i \in DB$, $m, w \in \mathbb{N}$

- 1 $j \leftarrow k_i$; $l \leftarrow w$; $CW_i \leftarrow 0^m$
- 2 **for** $m' \leftarrow m - 1$ **to** 0 **do**
- 3 **if** $j \geq \binom{m'}{l}$ **then**
- 4 $CW_i[m'] \leftarrow 1$
- 5 $j \leftarrow j - \binom{m'}{l}$
- 6 $l \leftarrow l - 1$
- 7 **if** $l = 0$ **then**
- 8 **Break**

Output: $CW_i(m, w)$

Algorithm 3: Query Expansion in PIRacle server

Input: q , $mask[i] = x^i, i \in [0, m)$

- 1 **for** $i \leftarrow 0$ **to** m **do**
- 2 $q[i] \leftarrow q \cdot mask[i]$
- 3 **for** $j \leftarrow 0$ **to** $\log_2 \frac{N}{2}$ **do**
- 4 $q[i] \leftarrow \phi_{2^j}(q[i]) + q[i]$

Output: $q[m]$

Algorithm 4: Equality Check in PIRacle server

Input: $q[m]$, $K[\beta][m]$

- 1 **for** $i \leftarrow 0$ **to** β **do**
- 2 $sum[i] \leftarrow \sum_{j=0}^{m-1} q[j] \cdot K[i][j]$
- 3 $I[i] \leftarrow pt_{one} - (sum[i] - pt_w)^T$

Output: $I[\beta]$

to send $\lceil m/N \rceil$ ciphertexts to the server, instead of the full set of m ciphertexts.

3) *Query Expansion in Server Side:* In this stage, the server expands the ciphertext received q such that each bit of the related constant-weight code is in a separate ciphertext. Algorithm 3 describes the query expansion procedure. Specifically, it iteratively extracts an element in q via m specific $mask$ vector (line 2) and fills the ciphertext with this element by *Rotate* and *Add* (line 4), eventually generating m ciphertexts encrypting $[x[0], x[0], \dots, x[0]], \dots, [x[m-1], x[m-1], \dots, x[m-1]]$, respectively, denoted by $q[m]$. Note that the query expansion is lightweight and is independent of the scale of the dataset since the expansion procedure depends solely on m (i.e., the bitwidth of the constant-weight code) and not on the total number of keys stored on the server.

4) *Equality Check in Server Side:* In this stage, the server performs the homomorphic equality operator (*EQ*) to check the equality between the $q[m]$ and all keys in the database. First, to facilitate subsequent computation, the server encodes all keys $\in CW(m, w)$ into plaintext vectors. Based on the length N of the BFV plaintext vector, all keys are encoded into $\lceil n/N \rceil = \beta$ vector sets, each containing m vectors, denoted as $K[\beta][m]$.

Algorithm 5: Index PIR in PIRacle server

Input: $I[\beta]$, $V[\beta][t]$

- 1 **for** $i \leftarrow 0$ **to** t **do**
- 2 $res[i] = \sum_{j=0}^{\beta-1} I[j] \cdot V[j][i]$
- 3 **for** $i \leftarrow 0$ **to** t **do**
- 4 $Ans = \phi_i(res[i]) + Ans$

Output: Ans

Then, the server performs *EQ* based on constant-weight code proposed in § III-A between $q[m]$ and $K[\beta][m]$ and outputs $I[i]$, $i \in [0, \beta)$ for each sets. Algorithm 4 illustrates the equality check procedure, where pt_{one}, pt_w are plaintext vectors $[1, \dots, 1]$ and $[w, \dots, w]$, respectively.

5) *Index PIR in Server Side:* In this stage, the server uses the $I[\beta]$ to extract the target value. First, similar to $K[\beta][m]$, the server first encodes all values into β vector sets, each containing $\lceil l_v / \log_2(T-1) \rceil = t$ vectors, denoted as $V[\beta][t]$. Next, the server employs FastPIR [12], which is preferred due to its lower processing time, to generate the result Ans . The Index PIR procedure is outlined in Algorithm 5. Specifically, the server iteratively computes the product between $I[\beta]$ and $V[\beta][t]$, summing the intermediate results into t ciphertexts $res[t]$. Throughout this process, the $\beta - 1$ non-target sets are masked, ensuring that the sum $res[t]$ encapsulates only the target value, with each ciphertext containing partial bits of that value. Subsequently, the server consolidates these results into a single ciphertext Ans by performing successive *Rotate* and *Add* operations. Finally, the server sends Ans and I back to the client. Here, $I = \sum_{i=0}^{\beta-1} I[i]$ serves as an indicator for the starting position of Ans .

6) *Decrypt Ans in Client Side:* After receiving the Ans and I from the PIRacle server, the client first decrypts Ans to obtain the plaintext vector. Then, the client decrypts the I to get the index information and shifts the vector based on the index to get the desired value finally. If the value is nil, i.e., the query key is not present in the key-value store.

IV. VARIABLE-LENGTH SUPPORT IN PIRACLE

A. Two-Pass Stable Sorting

As highlighted in § II-D2, due to the batching nature of homomorphic encryption (HE) schemes, most existing PIR protocols tend to assume uniform record lengths, often achieved by padding entries to the maximum bitwidth. [9], [10], [12], [16], [17]. While straightforward, this approach comes at a cost: padding bits—despite carrying no meaningful data—are interleaved with payload bits within the plaintext vector and are treated as a whole during the computation with the encrypted query from clients. This entanglement leads to inflated computations. PIRacle addresses this inefficiency by structurally decoupling padding bits from actual payloads. The central design insight is to group entries with similar bitwidths, so that the high-order padding can be isolated into dedicated zero plaintext vectors. These zero plaintext vectors can then be recognized and eliminated from computation at runtime.

Algorithm 6: Two-Pass Stable Sort

Input: Key-value dataset DB , mode $keyFirst$ (boolean)

```

1 if  $keyFirst$  then
2   stable sort  $DB$  by  $len(key)$ ;
3   stable sort  $DB$  by  $len(value)$ ;
4 else
5   stable sort  $DB$  by  $len(value)$ ;
6   stable sort  $DB$  by  $len(key)$ ;

```

Output: Sorted dataset DB

To realize this, PIRacle adopts a two-pass stable sorting mechanism, inspired by radix sort, outlined in Algorithm 6. Unlike traditional radix sorting, which operates on fixed digit priorities, PIRacle's sort order is determined by a dataset-specific boolean flag, $keyFirst$, which controls whether key or value bitwidths are prioritized. If $keyFirst$ is true, the system first performs a stable sort on the key bitwidths, followed by a second stable sort on the value bitwidths. Conversely, if $keyFirst$ is false, the order is reversed: values are sorted first, then keys. Notably, the value of $keyFirst$ is determined by the distribution characteristics of the dataset, we analyze it in § IV-B.

Once sorted, the dataset exhibits a structured ordering in which key or value entries increase monotonically in bitwidth. This organization causes the high-order padding of shorter entries to be aligned toward the beginning. As a result, when a batch of N entries is encoded into plaintext vectors, the high bits of shorter entries remain zero, and entire zero-padded regions are concentrated into dedicated zero vectors. These zero vectors are easy to detect and can be skipped during homomorphic computation, enabling effective pruning and improving overall system performance. The downstream effects of this alignment are discussed in the context of the equality check and Index PIR phases in the following sections.

B. Computation Pruning

1) *Variable-Length Key Handling:* In the equality check stage, the server checks the equality between the encrypted queries $q[m]$ and plaintext matrix $K[\beta][m]$. Although all keys are represented by constant-weight codes $CW(m, w)$, the bitwidth of the original payload significantly affects the distribution of zeros in $CW(m, w)$. Notably, shorter keys result in high-order zero padding in their $CW(m, w)$ form. Therefore, after the two-pass stable sorting, the high-order padding of these shorter keys will be clustered and encoded into zero plaintext vectors. As illustrated in the right half of Fig. 4, the plaintext vector $K[0][0]$ in the matrix $K[0][m]$ —corresponding to the high-order padding—is encoded as a zero vector. Consequently, when computing: $sum[i] \leftarrow \sum_{j=0}^{m-1} q[j] \cdot K[i][j]$, any multiplication involving such a zero vector (e.g., $K[0][0]$) has no impact on the result and can therefore be safely skipped. This optimization saves one PMult and one Add, without introducing any semantic deviation or correctness risk.

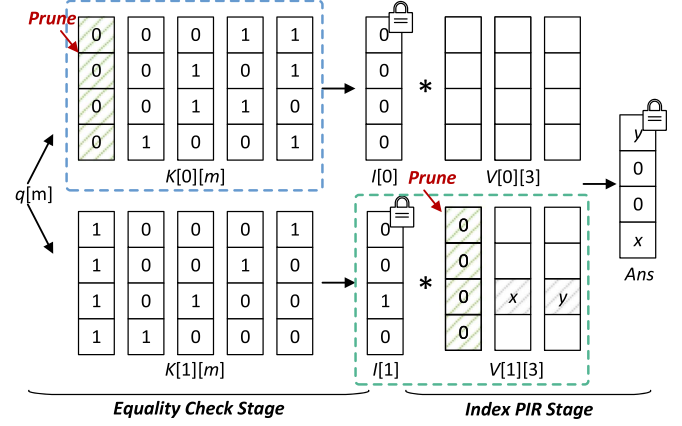


Fig. 4. The overall procedure when the PIRacle server handles 8 variable-length entries. In this example, we assume the encrypted query corresponds to $\{10100\}$, and the parameters (N, m, w) are set to $(4, 5, 2)$.

2) *Variable-Length Value Handling:* A similar optimization applies in the Index PIR stage. On the left side of Fig. 4, the high-order padding of several short values appears in the first column of $V[1][3]$, i.e., $V[1][0]$, and is encoded as a zero vector. Consequently, during the computation: $res[i] = \sum_{j=0}^{\beta-1} I[j] \cdot V[j][i]$, the computation involving $V[0][0]$ can be safely omitted, yielding the same computational saving of one PMult and one Add. Notably, as discussed above, this padding-oriented pruning mechanism is entirely determined by the length distribution of the public dataset DB handled by the server and is completely independent of the client's encrypted query $q[m]$. As a result, it does not reveal any additional information about the client's query. For a detailed security analysis, please refer to § IV-D.

3) *Trade-Off:* In practice, optimizing computation for variable-length keys and values involves an inherent trade-off: Since the key and value of each entry must be aligned in $K[\beta][m]$ and $V[\beta][t]$, computation pruning in the equality check and Index PIR stages often cannot be maximized simultaneously. When sorting is performed in a key-first manner, the grouped values may exhibit substantial bitwidth variance, which limits pruning efficiency in the Index PIR stage. The reverse also holds true. Fundamentally, the key factor in prioritization is which component—keys or values—results in more zero vectors after sorting. If the key bitwidths follow a long-tail distribution or have a high concentration of short entries, sorting by key bitwidth tends to yield tighter clustering and more pruning opportunities. Conversely, if the value lengths are more heterogeneous, value-first sorting is likely to provide greater savings during Index PIR. In practical deployment, the PIRacle server dynamically evaluates the bitwidth variance of keys and values to determine the $KeyFirst$.

C. Workload-Aware Scheduling

Despite benefiting from various algorithmic optimizations and its efficient handling of variable-length data, the overall performance of PIRacle remains fundamentally constrained by the intensive cost of homomorphic encryption (HE) operations, which are several orders of magnitude slower than equivalent

plaintext computations [36]. To overcome this performance bottleneck in large-scale deployments, we introduce a load-aware scheduling framework that enables efficient parallel execution, significantly accelerating PIRacle's workflow. This framework is agnostic to underlying hardware architectures and can be seamlessly extended to multi-core processors or multi-device cloud environments.

Given that PIRacle must scan the entire dataset to perform private information retrieval, a natural parallelization strategy involves partitioning the data into fragments and distributing computational tasks accordingly. However, the inherent characteristics of variable-length data in PIRacle introduce a non-trivial challenge. Specifically, with computation pruning enabled, the workload of each fragment becomes uneven, depending on the actual lengths and sparsity patterns of the keys and values it contains. This imbalance significantly degrades parallel execution performance and leads to inefficient resource utilization.

To address this challenge, we propose a scheduling strategy based on structure-aware load estimation and dynamic task distribution. The key insight is that, although different $K[i]/V[i]$ sets exhibit inherent variability in computational cost, this variation can be systematically estimated. After the initial setup phase, the server profiles each data fragment by counting the number of non-zero vectors it contains. Since zero vectors are explicitly pruned from computation, this count serves as a reliable proxy for the task's computational complexity, which can be approximately modeled as a linear function of the non-zero elements. Based on this profiling, we adopt the idea of iterative load-balancing that dynamically assigns the $K[i]$ and $V[i]$ computation tasks to independent execution units (e.g., threads, processes, or compute nodes). At each iteration, the task with the largest estimated cost is assigned to the execution unit with the current minimum cumulative load. This least-loaded-first strategy ensures effective load balancing, mitigates execution skew, and avoids both underutilization and overload of compute resources. It is inherently adaptive and can operate efficiently over a wide range of parallel execution environments, regardless of the specific hardware substrate.

We also optimize the query expansion phase, which—though lightweight in absolute terms—can become a performance bottleneck when the number of key-value items is small. In this phase, PIRacle transforms a compressed query ciphertext into m expanded ciphertexts using m distinct masks (see § III-B). Since these expansion operations are mutually independent, they can be readily parallelized. We incorporate them into our unified scheduling framework, distributing them across available execution units to reduce latency and maintain throughput even in small-scale workloads.

D. Security Analysis

Following the threat model described in § II-B, PIRacle protects the query privacy of clients by ensuring two forms of computational indistinguishability. (1) **Query/Response Indistinguishability**: PIRacle guarantees that the ciphertext sizes of both the query and its corresponding response remain constant

across all queries. These sizes are determined solely by public parameters m and the longest value. (2) **Computation Indistinguishability**: PIRacle ensures that the server's computation is independent of the client's input. For any encrypted queries, the server applies a fixed sequence of operations to each vector set during query processing. This uniform computation flow depends only on the internal structure of the database (i.e., key-value distribution), not on the specific client query.

Security Proof. We now define three security games (denoted as $\mathcal{G}_0$, $\mathcal{G}_1$, and $\mathcal{G}_2$) between a challenger and an adversary to sketch a formal proof of the PIRacle's privacy guarantee. $\mathcal{G}_0$: the adversary provides two keys k_1 and k_2 of equal bitwidth. The challenger randomly selects one of them and then performs a query using the chosen key with the PIRacle scheme. $\mathcal{G}_1$: the adversary provides two keys of equal bitwidth, k_1 and k_2 . The challenger randomly selects a key from the key space to perform a query with the PIRacle scheme. $\mathcal{G}_2$: the adversary provides two keys of unequal bitwidth, k_3 and k_4 . The challenger randomly selects one of them and then performs a query using the chosen key with the PIRacle scheme. In these three games, the adversary attempts to guess which key the challenger used and the correct guess events are denoted as S_0 , S_1 , and S_2 , respectively. Clearly, $\Pr[S_1] = 1/2$, since the challenger's selected key in $\mathcal{G}_1$ is independent of the adversary's provided keys, reducing the adversary to mere random guessing. In $\mathcal{G}_0$, since the challenger encrypts the selected one using BFV, the advantage of the adversary to distinguish it from the random key in $\mathcal{G}_1$ can be quantified by the $|\Pr[S_0] - \Pr[S_1]| = |\Pr[S_0] - 1/2| \leq \epsilon_{HE}^1$, so the adversary cannot win the $\mathcal{G}_0$ with non-negligible advantage. In $\mathcal{G}_2$, since the processing of any query by the PIRacle scheme is indistinguishable, i.e., the operations performed by the server on the plaintext polynomial of its database are identical for different queries, $|\Pr[S_2] - \Pr[S_0]| \leq \epsilon_{HE}$. Therefore, the adversary cannot also win the $\mathcal{G}_2$ with a non-negligible advantage. In summary, relying on a robust BFV scheme, PIRacle ensures that client query privacy is not leaked.

V. EVALUATION

A. Evaluation Methodology

Implementation. To evaluate the effectiveness of our proposed techniques across different computational settings, we implemented two versions of the PIRacle system. The CPU-based implementation leverages the Microsoft SEAL library [37], a widely adopted homomorphic encryption framework. The GPU-based version integrates recent advances in homomorphic encryption acceleration through the HEonGPU library [38], enabling efficient parallel processing of BFV operations on modern GPUs.

Parameters. The performance and efficiency of PIRacle depend on two key parameter sets: m and w of $CW(m, w)$, and the BFV parameters (N, T, Q) . For $CW(m, w)$, the parameter m must be sufficiently large to ensure that the encoding of all keys is injective. At the same time, a smaller

¹ ϵ_{HE} denotes the maximum advantage an adversary can gain in distinguishing between ciphertexts generated by the HE scheme, which is negligible in the context of cryptographic security.

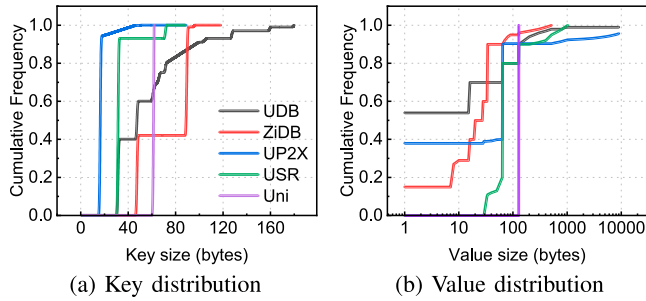


Fig. 5. The key and value size distribution.

m value reduces both computational complexity and communication overhead during the equality-checking stage. Therefore, given a dataset with maximum key $\max(\text{key})$, we set $w = m/2$ and choose the smallest m satisfying the inequality $\binom{m}{w} \geq \max(\text{key})$, thereby ensuring injectivity while minimizing resource costs. For the BFV parameters, we set the polynomial modulus degree to $N = 2^{15}$, the plaintext modulus to $T = 2^{16} + 1$, and use a 881-bit ciphertext modulus Q . This setup follows SEAL [10], achieving a 128-bit security level and enabling support for circuits with more than 20 multiplicative depth.

PIRacle and Baselines. To enable fair and consistent performance evaluation, we compare both the CPU and GPU versions of PIRacle against two state-of-the-art keyword-PIR schemes: Pantheon [17] and CwPIR [16]. Note that the equality operator proposed in CwPIR is not directly amenable to efficient Index-PIR pipelines, so we use an optimized variant of CwPIR in our implementation. [39]. For CPU evaluations, we use the multi-threaded implementations of Pantheon and CwPIR. For GPU evaluations, we extend both schemes with support for HEonGPU [38]. All experiments are conducted on Amazon EC2 instances. The CPU-based evaluations run on `c5.12xlarge` instances (48 vCPUs, 96 GiB RAM), while the GPU-based evaluations are deployed on `p4d.24xlarge` instances equipped with 8 NVIDIA A100 GPUs and 1.1 TiB of system memory. This setup allows us to measure system throughput, latency, and scalability under realistic deployment conditions across both compute environments.

Dataset. To evaluate PIRacle under realistic conditions, we use five datasets: four derived from large-scale Facebook workloads (UDB, ZiDB, UP2X, and USR), and one synthetic dataset, Uni, with uniform-length key-value entries. The real-world datasets reflect diverse production scenarios and are modeled using the trace-driven methodology from prior work [24], [25]. Each dataset contains 1 million key-value pairs, which are loaded into a Redis-backed server database for query processing. Fig. 5 plots the cumulative distribution functions (CDFs) of key and value sizes.

Metric. We focus on two key performance metrics. (1) Server-side latency: refers to the time required for the server to process a query and generate the corresponding response. (2) Communication overhead: measures the amount of data

exchanged between the client and server during the query-response process, reflecting the system’s scalability and the potential bottleneck caused by data transfer.

B. Overall Performance

We begin by evaluating the server-side latency and communication overhead per private query among the Pantheon, CwPIR, and PIRacle systems across various datasets. Note that in PIRacle, the server executes a fixed sequence of operations regardless of the client’s query content. Each experiment involves 100 private queries, and the average results are reported.

Server-side Latency. As shown in Fig. 6(a), under the CPU setting, PIRacle achieves substantial improvements in server-side efficiency over Pantheon and Conflux, realizing $28.5\times \sim 58.8\times$ and $22.7\times \sim 33.5\times$ speedups across dataset, respectively. These gains are driven by two design choices: (1) **A more efficient EQ construction.** Each EQ in PIRacle requires only 16 Mult operations and has a fixed multiplicative depth of 17, thereby completely avoiding Bootstrapping and substantially reducing computational complexity. By contrast, each EQ in Pantheon invokes $\approx 17b$ Mult operations where $b = \lceil l/16 \rceil$ and l denotes the bitwidth of the key being compared; each EQ in CwPIR requires at least w ($\approx l/2$) Mult operations with $w + 1$ multiplicative depth. Because the multiplicative depth in Pantheon and CwPIR grows with l , the Bootstrapping is necessary, incurring nontrivial overheads. (2) **Computation pruning for variable-length data.** PIRacle introduces computation pruning for variable-length data. Specifically, it sorts all entries based on their bitwidths, enabling the system to identify and skip zero plaintext vectors during processing. This technique significantly reduces redundant computation introduced by data padding or structural inflation. Under the GPU setting in Fig. 6(b), the same trend holds. Since PIRacle eliminates bootstrapping—which scales poorly on GPUs due to its compute intensity and heavy evaluation-key memory footprint—while baselines still invoke it in the equality-check phase, PIRacle’s advantage is amplified: it attains $43.5\times$ – $88.5\times$ and $26.1\times$ – $53.6\times$ speedups over Pantheon and Conflux, respectively.

Communication Overhead. In terms of communication, Client-server communication is platform-agnostic and therefore identical on CPU and GPU. As shown in Fig. 6(c) (CPU results), Pantheon, CwPIR, and PIRacle incur the same communication cost. This follows from their shared interaction pattern: the client sends one encrypted query (i.e., encrypted k_i), and the server returns an encrypted index and an encrypted answer—three ciphertexts in total. Given identical BFV parameters (same polynomial degree and modulus chain), each ciphertext is 6 MB, yielding $3 \times 6 \text{ MB} = 18 \text{ MB}$ overall.

C. Effectiveness of Individual Technologies

In this section, we quantify the performance contributions of PIRacle’s two core optimizations: (i) a lightweight homomorphic equality operator based on constant-weight codes, and (ii) a computation pruning mechanism tailored for variable-length

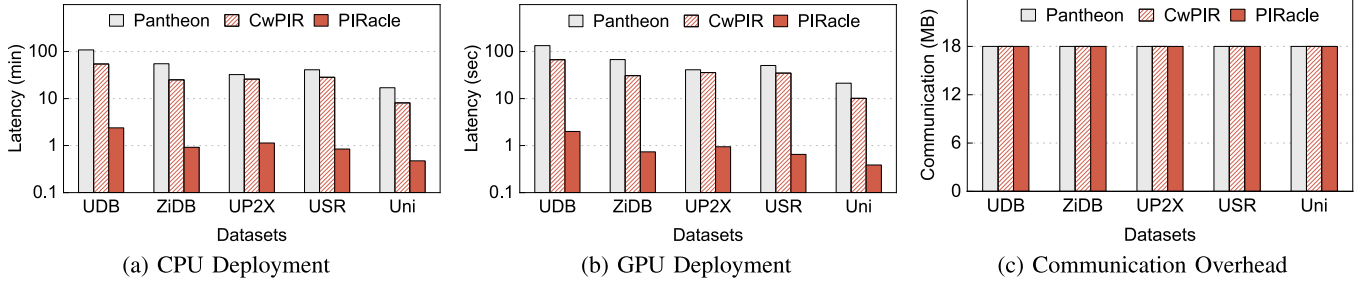


Fig. 6. The performance comparison among the Pantheon, CwPIR, and PIRacle on CPU and GPU platforms.

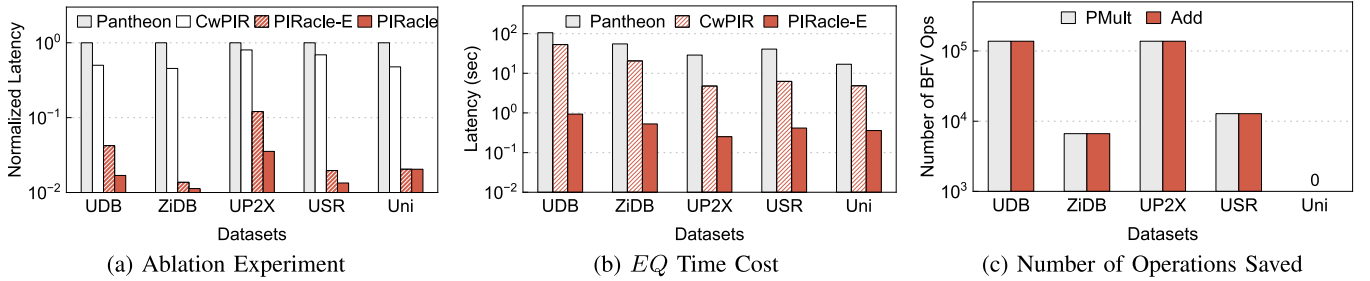


Fig. 7. The ablation analysis among Pantheon, CwPIR, and PIRacle on GPU platforms.

data. All experiments are conducted under a GPU deployment setting using the following system configurations:

- **Pantheon** and **CwPIR**: Original protocol implementations, both deployed on AWS EC2 p4d.24xlarge.
- **PIRacle-E**: A variant of PIRacle that incorporates only the optimized equality operator.
- **PIRacle**: The full implementation of PIRacle, with both optimizations enabled.

We focus exclusively on server-side latency, as communication overhead remains identical across all configurations. Experimental results are summarized in Fig. 7.

1) *Impact of the Constant-Weight Equality Operator*: Fig. 7(a) reports the normalized server-side latency per private query across five datasets. With only our optimized equality operator enabled (i.e., PIRacle-E), we observe substantial speedups over Pantheon and CwPIR: on UDB, ZippyDB, UP2X, USR, and Uni, the improvements reach $8.3\times \sim 73.1\times$ and $6.6\times \sim 35.1\times$, respectively. As highlighted earlier, these gains stem from a lighter-weight construction of *EQ*: unlike Pantheon and CwPIR, which require many Mult operations and inevitably resort to Bootstrapping as multiplicative depth grows, our *EQ* in PIRacle-E uses only a small, fixed number of Mult operations and completely eliminates Bootstrapping. To isolate the cost of equality operator, Fig. 7(b) plots the absolute runtime spent on equality check for Pantheon, CwPIR, and PIRacle-E. Across all datasets, PIRacle-E reduces this cost by $47.1\times \sim 114\times$ relative to Pantheon and by $13.46\times \sim 56.65\times$ relative to CwPIR, corroborating the above analysis.

2) *Impact of Computation Pruning for Variable-Length Data*: To assess the effectiveness of computation pruning, we compare the full PIRacle against PIRacle-E (which excludes pruning and retains only the optimized *EQ*). As shown in Fig. 7(a), PIRacle achieves additional speedups of $2.51\times$,

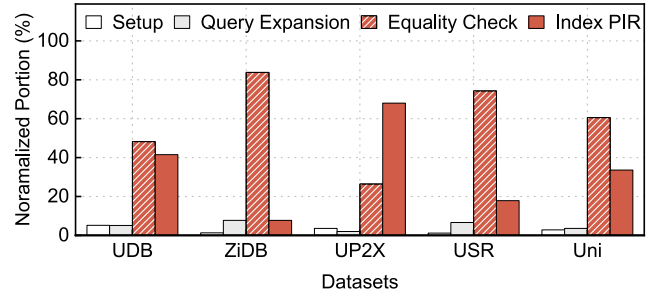


Fig. 8. Breakdown analysis of PIRacle system.

$1.21\times$, $3.49\times$, and $1.46\times$ on UDB, ZippyDB, UP2X, and USR, respectively. These gains arise from PIRacle's ability to detect and skip zero plaintext slices during query processing, thereby shrinking the effective workload and reducing the number of operator invocations on the server. To more precisely quantify the benefit of pruning, we collect the total number of skipped operations across all five datasets. Fig. 7(c) summarizes these statistics (note: this panel isolates the pruning effect; it compares PIRacle to PIRacle-E and is not about the *EQ* optimization). Depending on dataset structure, pruning eliminates up to 10^5 PMult and Add operations, underscoring its tangible efficiency benefits. Unless otherwise specified, KeyFirst is set to false in these experiments.

D. Breakdown Analysis in PIRacle

To better understand PIRacle's computational behavior and identify stage-specific bottlenecks, we analyze server-side latency across four stages: Setup, Query Expansion, Equality Check, and Index PIR. The results are presented in Fig. 8.

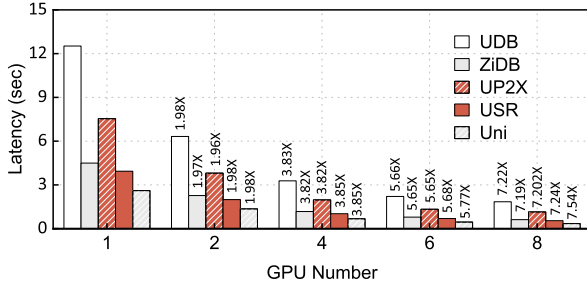


Fig. 9. Hardware scalability of the PIRacle system.

We begin by observing that the Setup and Query Expansion phases incur the lowest computational costs across all datasets, accounting for only 2.8% and 5.4% of the total server-side latency, respectively. This is expected: the Setup phase involves only plaintext preprocessing and parameter initialization, while Query Expansion consists of lightweight ciphertext operations that are independent of dataset size or bitwidth distribution.

More notably, the Equality Check stage—previously a dominant bottleneck in prior systems—now contributes only a moderate portion of the total latency in PIRacle. This reduction is primarily due to the introduction of a constant-weight-code-based *EQ*, which significantly lowers the associated computational overhead. Notably, the relative contributions of Equality Check and Index PIR vary significantly across datasets, depending on the structural characteristics of the key-value entries. For example, in the UP2X dataset, where the average value bitwidth significantly exceeds that of the keys, the Index PIR phase becomes the dominant cost contributor, accounting for 68% of total latency. In contrast, datasets such as UDB, ZippyDB, and USR exhibit more balanced key-value bitwidth distributions. In these cases, PIRacle’s value-first pruning strategy effectively reduces the Index PIR cost. Consequently, although the *EQ* in PIRacle is lightweight, its share of the overall latency increases, making it the most prominent stage in relative terms.

E. Scalability Analysis

In this section, we evaluate the scalability of the PIRacle system from both hardware and software perspectives, analyzing its performance variations as the number of integrated GPUs and the dataset size increase.

Hardware Scalability. To evaluate the scalability of PIRacle in parallel computing environments, we vary the number of active GPUs in the system (1, 2, 4, 6, and 8) and measure the corresponding server-side latency. Fig. 9 presents the results across all five datasets. As the number of GPUs increases, query latency decreases nearly linearly, demonstrating PIRacle’s ability to scale effectively with hardware resources. Notably, despite the inherent load imbalance introduced by variable-length data, our workload-aware scheduling framework ensures balanced task distribution across GPUs. This allows PIRacle to fully exploit parallelism, achieving significant performance gains through horizontal scaling.

Software Scalability. To assess PIRacle’s software scalability, we gradually increase the number of key-value entries in

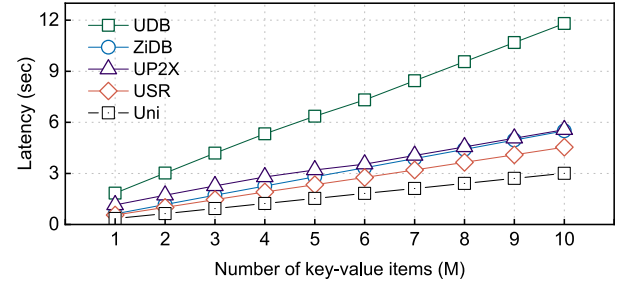


Fig. 10. Software scalability of the PIR system.

the database while preserving the original distribution. Specifically, we scale the dataset size from 1 million to 10 million entries and measure the server-side latency on an AWS EC2 p4d.24xlarge instance. The results are shown in Fig. 10. As expected, server-side latency increases roughly linearly with dataset size, consistent with the “all-for-one” nature of PIR protocols, where the server processes the entire database for each query. Nevertheless, PIRacle remains well-suited for large-scale deployment. One contributing factor is that PIRacle’s variable-length data optimizations become increasingly effective as the dataset grows. Larger datasets exhibit more concentrated bitwidth distributions within partitions, making value-based sorting more effective at exposing zero plaintext vectors. This structural effect enables PIRacle to mitigate a portion of the computational cost even as the overall data volume scales up.

F. Resource Overheads

In this section, we compare the monetary cost of executing private queries under Pantheon and PIRacle, considering both compute and network resource usage. Network costs are estimated using Amazon’s standard pricing model, which charges \$0.09 per GiB for data egress (downloads), with uploads being free. For compute costs, we consider two EC2 instance types used in our evaluation: the CPU-based c5.12xlarge instance at \$2.04/h and the GPU-based p4d.24xlarge instance at \$32.77/h.

Table III presents the per-query dollar cost of processing private queries across five datasets—UDB, ZiDB, UP2X, USR, and Uni—each containing 1 million key-value pairs. The results show a stark cost disparity between Pantheon and PIRacle on both CPU and GPU platforms. On the CPU platform, Pantheon incurs high per-query costs ranging from \$0.76 (Uni) to \$4.9 (UDB). In contrast, PIRacle reduces the cost dramatically on the same hardware, with per-query costs between \$0.021 and \$0.11. The cost advantage of PIRacle is even more evident on the GPU platform. It achieves per-query costs as low as \$0.003 (Uni) and remains below \$0.06 across all datasets—up to 60× cheaper than Pantheon, whose GPU costs range from \$0.16 to \$0.95.

G. Dynamic Database Support

We evaluate PIRacle’s performance under dynamic dataset updates using multi-version concurrency control (MVCC) to

TABLE III
DOLLAR COST PER PRIVATE QUERY FOR PANTHEON AND PIRACLE
UNDER CPU AND GPU PLATFORMS

System	UDB	ZiDB	UP2X	USR	Uni
<i>CPU Platform (EC2 c5.12×large)</i>					
Pantheon (\$)	4.9	2.4	1.4	1.8	0.76
PIRacle (\$)	0.11	0.041	0.057	0.035	0.021
<i>GPU Platform (EC2 p4d.24×large)</i>					
Pantheon (\$)	0.95	0.54	0.26	0.37	0.16
PIRacle (\$)	0.016	0.06	0.009	0.005	0.003

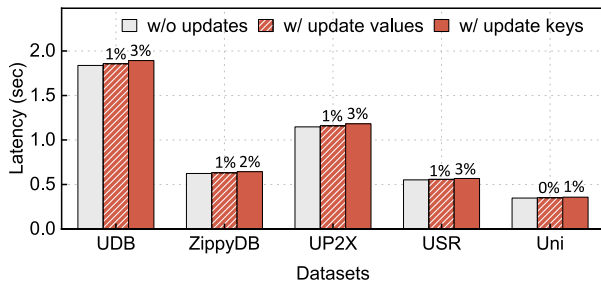


Fig. 11. Server-side latency with updates in PIRacle.

ensure uninterrupted query processing. MVCC is implemented by managing multiple Redis instances during update operations. We assess query latency across three scenarios: (1) no updates, (2) real-time value updates, and (3) real-time key updates. Each experiment is conducted over 100 queries using the UDB, ZiDB, UP2X, USR, and Uni datasets, each containing 1 million key-value pairs.

Fig. 11 reports the average query latency under different update scenarios. Across all datasets, introducing updates incurs a small performance overhead. The largest increase—up to 3%—occurs during key updates, which trigger re-execution of the Setup stage. This includes re-encoding updated keys into constant-weight representations and potentially reapplying the two-pass stable sorting procedure to preserve the structural alignment necessary for efficient computation pruning. In contrast, value updates introduce less overhead, as they do not require changes to the constant-weight encoding. Overall, the results demonstrate that PIRacle effectively supports dynamic updates with negligible impact on query performance, maintaining low latency and system responsiveness even under real-time data modifications.

H. Comparison With Non-EQ-Based Scheme

To demonstrate the generality of PIRacle, we further compare it with a class of Keyword PIR schemes that do not rely on explicit equality checking. Specifically, we select two of the most advanced and representative systems to date: SparsePIR [18] and KPIR^{Index} [40]. Both approaches pre-construct a keyword-to-index mapping to avoid explicit equality comparisons. All systems were implemented and evaluated on the same hardware platform (AWS EC2 p4d.24×large).

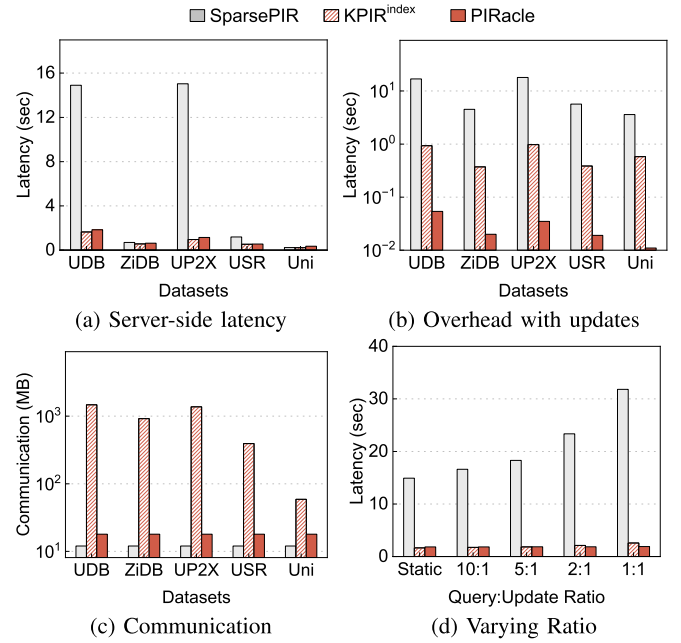


Fig. 12. Performance comparison between PIRacle and Non-EQ-based Schemes.

We first evaluate the three schemes along three dimensions and the results are presented in Fig. 12(a)(c). (1) **Server-side Latency.** Across the four variable-length datasets, PIRacle achieves server-side latency comparable to KPIR^{Index}, while consistently outperforming SparsePIR by a factor of $2.15 \times \sim 13.1 \times$. These improvements primarily stem from PIRacle's ability to efficiently handle variable-length records and to eliminate redundant computations via computation prune. In contrast, on the uniform-length Uni dataset, SparsePIR and KPIR^{Index} slightly outperform PIRacle, which aligns with expectation: although PIRacle's EQ is substantially optimized, in uniform-length scenarios, its performance still lags behind these Index PIR-style approaches. (2) **Communication Volume.** Regarding communication cost, SparsePIR incurs the lowest overhead (≈ 12 MB) since it only requires sending one encrypted query and receiving one encrypted response. KPIR^{Index}, on the other hand, incurs substantially higher communication overhead, including GB-scale server-to-client hints, as well as encrypted queries and responses. In contrast, PIRacle maintains a communication volume comparable to SparsePIR, demonstrating similarly efficient bandwidth utilization. (3) **Dynamic Updatability.** As discussed in § V-G, PIRacle's retrieval paradigm inherently supports dynamic datasets, incurring negligible computational cost during updates. In contrast, both SparsePIR and KPIR^{Index} rely on static keyword-to-index mappings that must be fully reconstructed whenever the dataset is updated, introducing 1 ~ 2 orders of magnitude higher overhead compared with PIRacle.

To further examine the impact of different operational scenarios, we use the UDB dataset and generate workloads based on YCSB with several representative update ratios, including a fully static setting and query-to-update ratios of 10:1, 5:1,

2:1, and 1:1. We then measure the average latency of executing private queries under varying update frequencies, as shown in Fig. 12(d). The results indicate that when updates are infrequent, i.e., in quasi-static datasets, KPIR^{Index} achieves slightly lower latency than PIRacle, while SparsePIR performs the worst. However, as the update ratio increases, the frequent index reconstructions required by SparsePIR and KPIR^{Index} lead to a substantial degradation in their performance. In contrast, PIRacle maintains stable efficiency across all update ratios. The higher the update frequency, the more pronounced the performance advantage of PIRacle becomes. These results demonstrate that PIRacle consistently delivers either optimal or near-optimal performance across a wide range of scenarios, underscoring its robustness and broad applicability.

VI. RELATED WORK

Keyword PIR. Prior research has extensively studied keyword-based private queries using Index PIR. Chor et al. [8] proposed a generic transformation that integrates search data structures with Index PIR, but it requires $\lceil \log_2(n+1) \rceil$ PIR rounds, where n is the database size. To reduce this latency, later works introduced single-round protocols using homomorphic equality (EQ) operators. In these schemes, the server evaluates homomorphic equality across all keys and then invokes Index PIR [10], [12], [41] to retrieve values. Systems such as CwPIR [16] and Pantheon [17] follow this paradigm but suffer from heavy computation overhead due to expensive EQ evaluations. PIRacle adopts the same general framework but introduces a lightweight EQ operator based on constant-weight codes, substantially improving efficiency and enabling fast keyword retrieval in real-world deployments. Recent work [18], [42], [43], [44] explores precomputation-based approaches that bypass equality checks. For example, ChalametPIR [43] employs binary fusion filters for probabilistic key indexing. These designs achieve fast queries but lack support for dynamic updates, as any data modification requires complete offline reconstruction. In contrast, PIRacle attains comparable query performance while natively supporting incremental updates, making it suitable for practical, dynamic settings. Finally, recent work [45] has sought to further reduce PIR costs by relaxing security assumptions—for instance, through the adoption of weaker indistinguishability guarantees. These approaches are technically orthogonal to our design and can be integrated with PIRacle to achieve complementary performance improvements under relaxed privacy models.

PIR in Variable-Length Datasets. Variable-length retrieval has been relatively underexplored in the PIR literature. The most relevant study is by Henry et al. [46], which, in the IT-PIR (multi-server) setting, introduces multi-block query techniques to alleviate the communication overhead caused by variable-length records. In contrast, our work lies within the CPIR (single-server, FHE-based) paradigm and tackles a distinct bottleneck—computation and storage amplification due to ciphertext-level padding—using two-pass stable sorting and computation pruning. These studies address distinct challenges

under fundamentally different assumptions and can be regarded as complementary efforts.

GPU-accelerated PIR. Some studies have utilized GPU acceleration for PIR protocols that are not based on Homomorphic Encryption (HE) [47], [48], [49], [50]. For instance, Maximilian et al. [47] constructed a PIR protocol using Distributed Point Functions on a dual-server setup and achieved over $20\times$ performance speedup via GPU acceleration. Inspired by these advances, we extend PIRacle with support for GPU-accelerated homomorphic operations under the BFV scheme. By leveraging recent developments in GPU-based HE libraries [38], we deploy PIRacle on a multi-GPU platform to further improve its computational throughput and reduce per-query latency in large-scale settings.

VII. CONCLUSION

In this paper, we present PIRacle, a high-performance Keyword PIR system to support privacy-preserving scenarios in cloud environments. Built on the Keyword PIR framework, PIRacle addresses several inefficiencies, including high computational complexity and suboptimal variable-length data retrieval, through optimizations in both algorithm design and data layout. Additionally, to better align with cloud environments, PIRacle integrates a load-balancing task scheduling framework and leverages multi-core parallel processing, fully utilizing hardware acceleration to boost performance. Compared to the state-of-the-art solution, Pantheon, experimental evaluations based on real-world workloads demonstrate that PIRacle achieves up to an $88.5\times$ speedup in performance.

REFERENCES

- [1] R. Nishtala et al., “Scaling MEMCACHE at Facebook,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, 2013, pp. 385–398.
- [2] G. DeCandia et al., “Dynamo: Amazon’s highly available key-value store,” in *Proc. ACM Symp. Oper. Syst. Princ. (SOSP)*, 2007, pp. 205–220.
- [3] S. Angel and S. Setty, “Unobservable communication over fully untrusted infrastructure,” in *Proc. USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, 2016, pp. 551–569.
- [4] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, “PIR-Tor: Scalable anonymous communication using private information retrieval,” in *Proc. USENIX Secur. Symp. (Secur.)*, 2011, pp. 475–490.
- [5] A. Henzinger, E. Dauterman, H. Corrigan-Gibbs, and N. Zeldovich, “Private web search with tiptoe,” in *Proc. 29th Symp. Oper. Syst. Princ. (SOSP)*, 2023, pp. 396–416.
- [6] K. Zhong, Y. Ma, and S. Angel, “Ibex: Privacy-preserving ad conversion tracking and bidding,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2022, pp. 3223–3237.
- [7] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, “Private information retrieval,” *J. ACM (JACM)*, vol. 45, no. 6, pp. 965–981, 1998.
- [8] B. Chor, N. Gilboa, and M. Naor, “Private information retrieval by keywords,” *IACR Cryptol. ePrint Arch.*, 1997.
- [9] C. A. Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, “XPIR: Private information retrieval for everyone,” in *Proc. Privacy Enhancing Technol. (PoPETs)*, 2016, pp. 155–174.
- [10] S. Angel, H. Chen, K. Laine, and S. Setty, “PIR with compressed queries and amortized query processing,” in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2018, pp. 962–979.
- [11] M. H. Mughees, H. Chen, and L. Ren, “OnionPIR: Response efficient single-server PIR,” in *Proc. ACM SIGSAC Conf. Comput. Communications Secur. (CCS)*, 2021, pp. 2292–2306.

- [12] I. Ahmad, Y. Yang, D. Agrawal, A. E. Abbadi, and T. Gupta, "Addra: Metadata-private voice communication over fully untrusted infrastructure," in *Proc. USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, 2021, pp. 313–329.
- [13] A. Ali et al., "Communication-computation trade-offs in PIR," in *Proc. 30th USENIX Secur. Symp. (Secur.)*, 2021, pp. 1811–1828.
- [14] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, "PIR with compressed queries and amortized query processing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2018, pp. 962–979.
- [15] J. P. Stern, "A new efficient all-or-nothing disclosure of secrets protocol," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur. (ASIACRYPT)*, vol. 1514, 1998, pp. 357–371.
- [16] R. A. Mahdavi and F. Kerschbaum, "Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators," in *Proc. USENIX Secur. Symp. (Secur.)*, 2022, pp. 1723–1740.
- [17] I. Ahmad, D. Agrawal, A. E. Abbadi, and T. Gupta, "Pantheon: Private retrieval from public key-value store," *Proc. VLDB Endowment (VLDB)*, vol. 16, no. 4, pp. 643–656, 2022.
- [18] S. Patel, J. Y. Seo, and K. Yeo, "{Don't} be dense: Efficient keyword {PIR} for sparse databases," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur.)*, 2023, pp. 3853–3870.
- [19] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *Proc. Cryptogr. Track RSA Conf. (CT-RSA)*, Cham: Springer, 2019, pp. 83–105.
- [20] J. H. Cheon et al., "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur. (ASIACRYPT)*, Cham: Springer, 2017, pp. 409–437.
- [21] K. Morimura, D. Maeda, and T. Nishide, "Improved integer-wise homomorphic comparison and division based on polynomial evaluation," in *Proc. 17th Int. Conf. Availability, Rel. Secur. (ARES)*, 2022, pp. 1–10.
- [22] T. U. Haque, N. N. Saber, and F. M. Shah, "Sentiment analysis on large scale amazon product reviews," in *Proc. IEEE Int. Conf. Innov. Res. Develop. (ICIRD)*, 2018, pp. 1–6.
- [23] A. Ferrari, G. O. Spagnolo, and S. Gnesi, "Pure: A dataset of public requirements documents," in *Proc. IEEE 25th Int. Requirements Eng. Conf. (RE)*, 2017, pp. 502–505.
- [24] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rockSDB key-value workloads at Facebook," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2020, pp. 209–223.
- [25] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.
- [26] A. E. Brouwer, J. B. Shearer, N. J. Sloane, and W. D. Smith, "A new table of constant weight codes," *IEEE Trans. Inf. Theory*, vol. 36, no. 6, pp. 1334–1380, Nov. 2006.
- [27] E. W. Weisstein, "Fermat's little theorem". Accessed: Jan. 30, 2026. [Online]. Available: <https://mathworld.wolfram.com/>
- [28] J. Kim, J. Seo, and Y. Song, "Simpler and faster BFV bootstrapping for arbitrary plaintext modulus from CKKS," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2024, pp. 2535–2546.
- [29] A. Akavia, D. Feldman, and H. Shaul, "Secure data retrieval on the cloud: Homomorphic encryption meets coresets," *IACR Trans. Cryptograph. Hardware Embedded Syst. (CHES)*, 2019, pp. 80–106.
- [30] A. Akavia, C. Gentry, S. Halevi, and M. Leibovich, "Setup-free secure search on encrypted data: Faster and post-processing free," *Proc. Privacy Enhancing Technol. (PoPETs)*, vol. 2019, no. 3, pp. 87–107, 2019.
- [31] U. Daepf, P. Gorkin, U. Daepf, and P. Gorkin, "Fermat's little theorem," in *Reading, Writing, and Proving: A Closer Look at Mathematics*, New York, NY, USA: Springer, 2011, pp. 315–323.
- [32] J. Lin et al., "Inspire: In-storage private information retrieval via protocol and architecture co-design," in *Proc. 49th Annu. Int. Symp. Computer Archit. (ISCA)*, 2022, pp. 102–115.
- [33] A. Davidson, G. Pestana, and S. Celi, "FrodoPIR: Simple, scalable, single-server private information retrieval," in *Proc. Privacy Enhancing Technol. (PoPETs)*, 2023, pp. 365–383.
- [34] M. Zhou, A. Park, W. Zheng, and E. Shi, "Piano: Extremely simple, single-server PIR with sublinear server computation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2023, pp. 55–55.
- [35] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: Single database, computationally-private information retrieval," in *Proc. 38th Annu. Symp. Found. Comput. Sci. (FOCS)*, Piscataway, NJ, USA: IEEE Press, 1997, pp. 364–373.
- [36] B. Reagen et al., "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 26–39.
- [37] "Microsoft SEAL (Release 4.1)," Microsoft Research, Redmond, WA, USA, Jan. 2023. [Online]. Available: <https://github.com/Microsoft/SEAL>
- [38] A. Ş. Özcan and E. Savaş, "HeonGPU: A GPU-based fully homomorphic encryption library 1.0," *Cryptology ePrint Archive*, 2024.
- [39] J. Liu, J. Li, D. Wu, and K. Ren, "PIRANA: Faster multi-query PIR via constant-weight codes," in *Proc. IEEE Symp. Secur. Privacy (SP)*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2023, pp. 39–39.
- [40] M. Hao et al., "Practical keyword private information retrieval from key-to-index mappings," in *Proc. USENIX Secur. Symp. (Secur.)*, 2025, pp. 3397–3416.
- [41] Z. Chen, H. You, Q. Wei, H. Lu, L. Ju, and Z. Shen, "SmartPIR: A private information retrieval system using computational storage devices," in *Proc. 58th IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2025, pp. 1749–1762.
- [42] D. Kogan and H. Corrigan-Gibbs, "Private blocklist lookups with checklist," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 875–892.
- [43] S. Celi and A. Davidson, "Call me by my name: Simple, practical private information retrieval for keyword queries," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2024, pp. 4107–4121.
- [44] M. Hao et al., "Practical keyword private information retrieval from key-to-index mappings," *Cryptology ePrint Archive*, 2025.
- [45] J. Zhang et al., "Femur: A flexible framework for fast and secure querying from public key-value store," in *Proc. ACM Manage. Data (SIGMOD)*, 2025, Art. no. 162, 29 pp.
- [46] R. Henry, Y. Huang, and I. Goldberg, "One (block) size fits all: PIR and SPIR with variable-length records via multi-block queries," in *Proc. 20th Annu. Netw. Distrib. System Secur. Symp., (NDSS)*, 2013, pp. 1–18.
- [47] M. Lam et al., "GPU-based private information retrieval for on-device machine learning inference," in *Proc. 29th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2024, pp. 197–214.
- [48] D. Günther, M. Heymann, B. Pinkas, and T. Schneider, "GPU-accelerated PIR with client-independent preprocessing for large-scale applications," in *Proc. 31st USENIX Secur. Symp. (USENIX Secur.)*, 2022, pp. 1759–1776.
- [49] Z. Ji, J. Zhao, P. Gao, X. Yin, and L. Ju, "Accelerating number theoretic transform with multi-GPU systems for efficient zero knowledge proof," in *Proc. 30th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2025, pp. 1–14.
- [50] Z. Chen, Z. Cao, Z. Shen, and L. Ju, "HMC-FHE: A heterogeneous near data processing framework for homomorphic encryption," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 43, no. 11, pp. 3551–3563, Nov. 2024.



Zehao Chen received the B.S. degree from the School of Software, Shandong University, Jinan, China, in 2019, and the M.S. degree in 2022 from the School of Computer Science and Technology, Shandong University, Qingdao, China, where he is currently working toward the Ph.D. degree with the School of Cyber Science and Technology. His research interests include heterogeneous computing systems, and privacy computing.



Zhaoyan Shen received the B.S. and M.S. degrees from the Department of Computer Science and Technology, Shandong University, China, in 2012 and 2015, respectively, and the Ph.D. degree from the Department of Computing, Hong Kong Polytechnic University, in 2018. He is currently a Professor of Shandong University. His research interests include big data systems, storage systems, embedded systems, system architecture, and hardware/software codesign.



multicore systems.

Yi Wang (Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from Harbin Institute of Technology, Harbin, China, in 2005 and 2008, respectively, and the Ph.D. degree in computer science from the Department of Computing, the Hong Kong Polytechnic University, Hong Kong, in 2011. He is currently a Professor with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China. His research interests include embedded systems, nonvolatile memory, and real-time scheduling for



Lei Ju received the B.E and Ph.D. degrees from the School of Computing, National University of Singapore, in 2005 and 2010, respectively. He is currently the Director of the Research Center for Cryptography Theory and Applications, Qian Cheng Laboratory, Shandong Province. He is also a Professor with the School of Cyber Science and Technology, Shandong University, China. His research interests cover computer architecture, embedded and real-time systems, and algorithm-hardware codesign of privacy computing systems.



for Processor Technologies. His research interests include fully homomorphic encryption acceleration, FPGA accelerator design, and fully homomorphic processor design. He is a member of the Youth Innovation Promotion Association of CAS, and the New Best Star of ICT.

Hang Lu received the B.S. and M.S. degrees in electronic information engineering from Beijing University of Aeronautics and Astronautics, Beijing, China, in 2008 and 2011, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2015. He is currently an Associate Professor and a Master Tutor with the ICT, CAS, University of Chinese Academy of Sciences, Beijing, China. He is also a Research Scientist with Shanghai Innovation Center