Practical Keyword Private Information Retrieval from Key-to-Index Mappings

Meng Hao[1], Weiran Liu[2], Liqiang Peng[2], Cong Zhang[3] (✉), Pengfei Wu[1], Lei Zhang[2], Hongwei Li[4], and Robert H. Deng[1]

[1]School of Computing & Information Systems, Singapore Management University
[2]Alibaba Group
[3]Institute for Advanced Study, BNRist, Tsinghua University
[4]Peng Cheng Laboratory

Abstract

This paper introduces practical schemes for keyword Private Information Retrieval (keyword PIR), enabling private queries on public databases using keywords. Unlike standard index-based PIR, keyword PIR presents greater challenges, since the query's position within the database is unknown and the domain of keywords is vast. Our key insight is to construct an efficient and compact key-to-index mapping, thereby reducing the keyword PIR problem to standard PIR. To achieve this, we propose three constructions incorporating several new techniques. The high-level approach involves (1) encoding the server's key-value database into an indexable database with a key-to-index mapping and (2) invoking standard PIR on the encoded database to retrieve specific positions based on the mapping. We conduct comprehensive experiments, with results showing substantial improvements over the state-of-the-art keyword PIR, ChalametPIR (CCS'24), i.e., a 15 $\sim$ 178$\times$ reduction in communication and 1.1 $\sim$ 2.4$\times$ runtime improvement, depending on database size and entry length. Our constructions are practical, executing keyword PIR in just 47 ms for a database containing 1 million 32-byte entries.

1 Introduction

Private Information Retrieval (PIR) [12] enables a client to retrieve entries from a public indexed database hosted by a server, without revealing any information about which entry is being retrieved. Over the last decade, there has been a significant amount of works on PIR [5, 6, 13, 25, 27, 29–33, 45, 46]. Due to its strong privacy guarantees, PIR is an important building block in many real-world applications, including certificate transparency auditing [23], web search [22], password checkup [5], and anonymous communication [4]. Despite these security and application advantages, standard PIR unrealistically assumes that the client knows the query's index within the database. However, in most practical applications, database entries are instead indexed by keywords from a much larger domain and their indices are not immediately available [10, 11, 36].

To address this problem, Chor et al. [11] introduced the concept of keyword PIR, where each database entry is represented as a key-value pair. In keyword PIR, the client aims to privately retrieve the value v associated with a query key k from a key-value database $D := \{(k_1, v_1), \ldots, (k_n, v_n)\}$. A naive approach would involve downloading the entire mapping from keys to indices and then using a standard PIR scheme to retrieve the desired entry after identifying the index of the query key. However, this method incurs a communication cost that scales linearly with the database size. Chor et al. [11] proposed a more efficient approach based on binary tree structures, which requires $O(\log n)$ invocations of standard PIR. As shown in Section 1.3, recent advancements [5, 10, 28, 36] have further improved this by demonstrating that keyword PIR can be reduced to a constant number of standard PIR invocations, significantly enhancing efficiency.

However, current state-of-the-art keyword PIR constructions [5, 10, 28, 36] either involve complex operations using fully homomorphic encryption that result in prohibitively high computation overheads or incur substantial communication costs. For instance, the most efficient existing construction, ChalametPIR [10], still involves communication costs that scale linearly with the database size. In this paper, we systematically address the inefficiencies associated with keyword PIR. We identify that the primary challenge stems from the unavailability of the index for the query key within the database. Motivated by this, our key insight is to encode the key-value database into an indexable format and develop a compact mapping between keywords and their positions in the database. This approach allows us to construct a keyword PIR scheme with communication and computation costs nearly equivalent to those of standard PIR.

1.1 Our Contributions

This paper presents three concretely efficient keyword PIR constructions based on different key-to-index mapping strate-

✉Corresponding author

gies. Our contributions are summarized as follows.

Keyword PIR from Sparse Key-Value Store. Our first construction $\mathsf{KPIR}^{\mathsf{kvs}}$ introduces a novel key-value store (KVS) termed sparse KVS. Using this sparse KVS, we achieve keyword PIR by encoding the key-value database into a vector of comparable size to the original database and then performing a constant number of standard PIR operations on this vector.

Keyword PIR from Hashing-to-Bins. Our second construction $\mathsf{KPIR}^{\mathsf{hash}}$ employs hashing-to-bins techniques, similar to MulPIR [5], to retrieve an entire bin based on the query key. To this end, we utilize the row retrieval capability of Kushilevitz-Ostrovsky PIR (KOPIR) [24], where the database is represented as a matrix and a row of this matrix is privately retrieved. We abstract PIR with this property as Row-KOPIR and instantiate it from pre-processing-based SimplePIR [23]. $\mathsf{KPIR}^{\mathsf{hash}}$ requires a single invocation of Row-KOPIR on a slightly expanded encoded database.

Keyword PIR from Approximate Key-to-Index Mapping. Our most efficient construction $\mathsf{KPIR}^{\mathsf{index}}$ introduces a novel approximate key-to-index mapping that produces a mapping between keys and their positions with an error ε. We handle this error by utilizing a repetition-based matrix encoding combined with Row-KOPIR. $\mathsf{KPIR}^{\mathsf{index}}$ requires a single invocation of Row-KOPIR on an encoded database that is almost the same size as the original.

Extensive Evaluations. We implement our three constructions alongside the state-of-the-art scheme ChalametPIR in a unified platform. Extensive experiments demonstrate that our three constructions achieve a $15 \sim 178\times$ reduction in communication and $1.1 \sim 2.4\times$ runtime improvement, depending on database size and entry length. Figure 1 gives a quick comparison with ChalametPIR. Our constructions are practical, executing keyword PIR in just 47 ms for a database containing 1 million 32-byte entries.

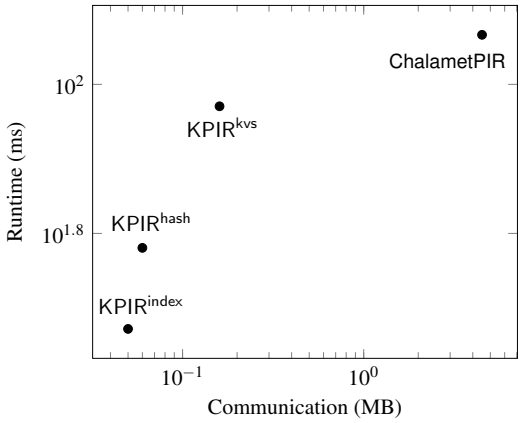


Figure 1: Communication and runtime comparisons between our constructions ($\mathsf{KPIR}^{\mathsf{kvs}}$, $\mathsf{KPIR}^{\mathsf{hash}}$, $\mathsf{KPIR}^{\mathsf{index}}$) and the state-of-the-art scheme, ChalametPIR [10], for a database size 2^{20} with 32-byte entries in the online phase. Similar to ChalametPIR, $\mathsf{KPIR}^{\mathsf{index}}$ requires the client additionally stores three hash functions. $\mathsf{KPIR}^{\mathsf{hash}}$ additionally stores one hash function in the client's memory. In $\mathsf{KPIR}^{\mathsf{index}}$, the client additionally stores a key-to-index mapping of size 324 KB. Both axes are in log-scale.

1.2 Overview of Our Techniques

We present the key techniques of our keyword PIR constructions. The high-level idea is to (1) encode the server's key-value database into an indexable database with a compact key-to-index mapping and (2) invoke standard PIR on the encoded database to retrieve specific positions according to the mapping. For clarity, we denote the original key-value database as D of size n and the new encoded database as D' of size m, where $m \geq n$. Therefore, two main efficiency metrics are the size m of the encoded matrix D' and the number α of standard PIR invocations on D'. As we will illustrate, our first two constructions achieve favorable values for m and α, respectively, while the final construction combines the best of both worlds, with $m \approx n$ and $\alpha = 1$. It is worth emphasizing that, as shown in Figure 1, all of our three constructions outperform the state-of-the-art work [10], especially for the communication cost.

Keyword PIR from Sparse Key-Value Store. Our first construction called $\mathsf{KPIR}^{\mathsf{kvs}}$ is a generic transformation from keyword PIR to any standard PIR using sparse Key-Value Stores (KVS). A KVS [18] consists of two algorithms, namely Encode and Decode. For a key-value database D of size n, the Encode algorithm takes the database D as input and produces an indexable vector D' of dimension m. The Decode algorithm takes D' and any key k as input and accesses some positions of D', providing an output that corresponds to v_i if k matches some k_i used to generate D'. With such a KVS, keyword PIR on D appears to reduce to standard PIR on D', which privately executes the Decode algorithm on any query. However, the main challenge is that the decoding of existing KVS schemes [8, 18, 41, 44] requires to access multiple positions of the encoded vector D'. This introduces a large number of standard PIR invocations, resulting in high computation and communication costs. For example, the recently proposed near-optimal KVS, random-band KVS [8], accesses about one hundred positions from D' during decoding.

To address the above challenge, we present a new KVS notion, called α-sparse KVS, particularly designed for efficient decoding. A key difference from prior KVS schemes [8, 18, 41] lies in that the Decode algorithm exclusively accesses a small constant number α of entries from the encoded vector D'. After thorough investigation, we found that Binary Fuse Filters (BFF) [20] satisfy the α-sparse KVS abstraction. Exploiting the sparsity property of α-sparse KVS, we can

achieve keyword PIR by executing α standard PIR invocations on the encoded vector D' of size m. In our instantiation, we set α to 3, and the size m of D' is less than $1.15n$. As a result, this keyword PIR strategy takes 3 invocations of any standard PIR on the encoded database with an expansion rate of less than 1.15. Note that while ChalametPIR [10] also employs BFF, it does not utilize BFF's sparsity property and rather involves an inner product between BFF output and query vector for decoding, which involves a single invocation of linear-communication FrodoPIR [14].

Keyword PIR from Hashing-to-Bins. To address the problem of multiple standard PIR invocations in the first construction, we present a keyword PIR construction called $\mathsf{KPIR}^{\mathsf{hash}}$ using hashing-to-bins techniques. Similar to MulPIR [5], we use a public hash function to assign the server's key-value pairs of the database D of size n into a hash table and then let the client privately retrieve an entire bin corresponding to the query. However, there are two critical efficiency issues: (1) If we set the number of bins as large as n, the maximum bin size of the resulting hash table is $O(\log n)$ [39], which leads to the size of the hash table (i.e., the encoded database D') becoming $O(n \cdot \log n)$. (2) If we utilize a small number of bins, as also noted in MulPIR [5], each bin thus contains a large number of entries of D, resulting in a prohibitively high standard PIR overhead.

We address this problem by utilizing the row retrieval capability of Kushilevitz-Ostrovsky PIR (KOPIR) [24], in which the database is represented as a matrix and then a row of the matrix database is privately retrieved each time. We abstract PIR with this property as Row-KOPIR and instantiate it with SimplePIR [23], the state-of-the-art pre-processing-based PIR. By utilizing this row retrieval capability, the server inserts the entries of the database D of size n into a hash table with $\sqrt{n}$ bins and combines these bins as rows into a matrix database D'. We empirically evaluate that the maximum number of entries in each bin is less than $1.87\sqrt{n}$ and the size m of D' does not exceed $1.87n$. Therefore, $\mathsf{KPIR}^{\mathsf{hash}}$ takes a single invocation of Row-KOPIR on the encoded database, with an expansion rate of less than 1.87.

Keyword PIR from Approximate Key-to-Index Mapping. We present the most efficient construction called $\mathsf{KPIR}^{\mathsf{index}}$ from a novel approximate key-to-index mapping. An ideal approach for keyword PIR is to construct a key-to-index mapping that directly associates keys with their exact positions in the key-value pairs. By sharing this mapping in advance, keyword PIR is reduced to standard PIR almost without additional costs. Essentially, this key-to-index mapping serves as a KVS, where the key-coordinate corresponds to the keys of the database D, and the value-coordinate corresponds to the positions of those keys within D. However, existing KVS schemes [8, 18, 41] require the mapping size scale linearly with the size of D. This leads to undesirable linear communication costs in keyword PIR constructions.

To address this problem, we formalize an approximate key-to-index mapping with a theoretical error bound ε while achieving sublinear communication relative to the size of the key set K. That is, for any $k \in K$, this mapping outputs an approximate position i_k such that $i_k \in [\mathsf{pos}(k)-\varepsilon, \mathsf{pos}(k)+\varepsilon]$, where $\mathsf{pos}(k)$ is the exact position of k within K. We instantiate this mapping using the Piece-wise Linear Approximation (PLA) algorithm [17], which ensures sublinear mapping size with carefully chosen parameters. Nevertheless, this approximate solution introduces a new challenge, i.e., how to privately obtain the value of the exact position only given an approximate position. We address this challenge by utilizing Row-KOPIR to retrieve all entries indexed by $i_k-\varepsilon, \ldots, i_k+\varepsilon$ in the same row. In more detail, we sort the key-value entries of the database D of size n and then transform D into a square matrix of size $\sqrt{n} \times \sqrt{n}$ in a row-wise manner. We require that for each entry D_{i_k}, all adjacent entries $D_{i_k-\varepsilon}, \ldots, D_{i_k+\varepsilon}$ should be included in the same row. This is achieved by repeatedly appending $D_{i-\varepsilon}, \ldots, D_{i-1}$ before the first entry D_i and $D_{j+1}, \ldots, D_{j+\varepsilon}$ after the last entry D_j in each row. This ensures that for each query k, the consecutive 2ε positions centered on i_k are in one row. It results in the encoded matrix D' of size $\sqrt{n} \times (\sqrt{n}+2\varepsilon)$. In our evaluation, we set ε to 4 and also achieve a very small size of the mapping. Therefore, $\mathsf{KPIR}^{\mathsf{index}}$ takes a single invocation of Row-KOPIR on the encoded database with an expansion rate between $1.03 \sim 1.36$.

1.3 Related Works

We elaborate on existing keyword PIR schemes and further discuss the essential differences between these works and our constructions.

Keyword PIR, also known as PIR on sparse databases, was first introduced in the pioneering work by Chor et al. [11]. Unlike standard PIR, which assumes that the client knows the physical index of the desired entry, keyword PIR considers a more practical scenario where databases are organized by keywords, and each entry is represented as a key-value pair. A naive solution to keyword PIR is to download all keys and invoke standard PIR after finding the corresponding index of the query key. Unfortunately, this approach results in linear communication. Chor et al. [11] proposed an alternative solution that builds a binary level-$\log n$ search tree over the entries from a database of size n, thereby reducing the computation to standard PIR. However, the main issue is that it requires $O(\log n)$ PIR queries with $O(\log n)$ communication rounds.

To address the above issues, recent keyword PIR schemes focused on achieving constant overheads. Ali et al. [5] presented two constructions based on hashing schemes. Similar to our scheme $\mathsf{KPIR}^{\mathsf{hash}}$, their first construction is based on simple hashing, where database entries are assigned to buckets using a public hash function. The client then privately retrieves the whole bucket corresponding to the query utilizing a single invocation of standard PIR. However, their

protocols result in large computation costs, because the size of the buckets is quite large and standard PIR is unfriendly to large entries. For example, for a database with one million entries, their protocol runs in 3.7 seconds, which is at least 14× slower than our $\mathsf{KPIR}^{\mathsf{hash}}$ construction. Moreover, we also analyze parameter settings in detail to optimize the overhead. Their second construction leverages Cuckoo hashing to assign database entries to buckets, ensuring that each bucket includes at most one database entry. The client then retrieves multiple positions (e.g., 3 in their evaluation) from the buckets according to the query and the used hash functions. However, this solution causes both multiple invocations of standard PIR and expanded (e.g., 1.5× larger) size of the buckets than that of the original database. Both limitations are inherent due to the usage of Cuckoo hashing.

To further improve the efficiency, several keyword PIR works [28, 36] designed novel protocols that only require a single invocation of standard PIR. Mahdavi and Kerschbaum [28] built keyword PIR using constant-weight equality operators. The core idea is that the server first executes equality testing between the encoded query and the database's keys in ciphertext and then derives the output by evaluating the inner product between the database and the encrypted equality results. Although their protocols are general for both standard and keyword PIR, the underlying complicated homomorphic operations incur significantly higher computation and communication costs than existing standard PIR [30, 32], as illustrated by recent studies [10, 36]. Subsequently, Patel et al. [36] proposed SparsePIR to optimize the communication and computational overhead of the above works. SparsePIR breaks down the database into smaller partitions and encodes each partition as linear combinations. This encoding is compatible with advanced standard PIR schemes [30, 32], which employ recursion and batching techniques. Despite the improved costs, the recent work [10] also shows that there is still an order of magnitude in the performance deprecation between SparsePIR and state-of-the-art PIR schemes [14, 26, 34].

The state-of-the-art keyword PIR is proposed by Celi and Davidson [10], called ChalametPIR. ChalametPIR first transforms a key-value database D of size n into a vector D' of size $m := O(n)$ via KVS. In their work, KVS is also instantiated with Binary Fuse Filters [20]. In contrast to our constructions, they do not leverage the sparsity property defined in our sparse KVS to improve efficiency. Then, the client invokes a variant of FrodoPIR [14], where the client sends an encrypted vector ct of size m computed according to the query key and the server evaluates the inner-product $\mathsf{ct} \cdot D'$. However, although their FrodoPIR-based keyword PIR is able to push most computation into the pre-processing phase, the online communication (from sending ct) is still linear to the database size. This becomes a key bottleneck for practical applications and significantly affects scalability. As shown in our evaluation, we achieve up to two orders of magnitude improvement in communication, while ensuring comparable and even better computation costs.

ChalametPIR also estimates the communication overhead if the underlying FrodoPIR is replaced with SimplePIR. Unfortunately, the ChalametPIR framework is incompatible with a single invocation of SimplePIR. It currently lacks a theoretically analyzed or empirically validated SimplePIR-based construction. We give a detailed explanation in Appendix A. It is worth noting that our $\mathsf{KPIR}^{\mathsf{kvs}}$ invokes three times SimplePIR to avoid this issue. Despite multiple sublinear-communication index PIR invocations, we still offer significant communication advantages over linear-communication FrodoPIR-instantiated ChalametPIR.

2 Preliminaries

2.1 Notations

We use κ and λ to denote the computational and statistical security parameters, respectively. We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. For a vector d, $d[i]$ denotes the i-th element. For a matrix D, $D[i, \cdot]$ and $D[\cdot, j]$ represent the i-th row and j-th column of D, respectively. By $a \leftarrow A$, we denote that a is sampled from the set A uniformly at random. $a \leftarrow \mathrm{A}(x)$ denotes that a is the output of the randomized algorithm A on input x, and $a := b$ denotes that a is assigned by b. we use negl to denote a negligible function. $\langle x, y \rangle$ denotes the inner product of x and y. For two distributions X and Y, we write $X \approx_c Y$ and $X \approx_s Y$ if X and Y are computationally and statistically indistinguishable, respectively.

2.2 Keyword Private Information Retrieval

Similar to recent works [14, 23, 26], we consider keyword PIR with a query-independent setup phase, in which the server can pre-process the database and send hints to the client.

Definition 1. *A keyword Private Information Retrieval (keyword PIR) scheme, over key space $\mathcal{K}$, value space $\mathcal{V}$ and database size n, consists of the following four routines, all taking the computational security parameter κ as an implicit input.*

- $\mathsf{Setup}(D) \to (\mathsf{hint}_S, \mathsf{hint}_C)$*: On input a database $D := \{(k_1, v_1), \ldots, (k_n, v_n)\} \in (\mathcal{K} \times \mathcal{V})^n$, output pre-processed hints $\mathsf{hint}_S, \mathsf{hint}_C$ for the server and the client, respectively.*
- $\mathsf{Query}(k) \to (\mathsf{st}, \mathsf{qu})$*: On input a query $k \in \mathcal{K}$, output a secret client's state* st *and a query* qu.
- $\mathsf{Answer}(\mathsf{qu}, \mathsf{hint}_S) \to \mathsf{ans}$*: On input the query* qu *and the hint* hint_S*, output an answer* ans.
- $\mathsf{Recover}(\mathsf{st}, \mathsf{hint}_C, \mathsf{ans}) \to v$*: On input the state* st*, the hint* hint_C *and the answer* ans*, output a value $v \in \mathcal{V}$.*

A keyword PIR scheme should satisfy the following correctness and security properties.

Correctness. A keyword PIR scheme is *correct* if for any database $D := \{(k_1, v_1), \ldots, (k_n, v_n)\}$ and all queries k_i for $i \in [n]$, it holds that

$$\mathsf{Recover}(\mathsf{st}, \mathsf{hint}_C, \mathsf{Answer}(\mathsf{qu}, \mathsf{hint}_S)) = v_i, \tag{1}$$

where $(\mathsf{hint}_S, \mathsf{hint}_C) \leftarrow \mathsf{Setup}(D)$ and $(\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(k_i)$. For k that is not in the key set of D, the output of Recover is $\perp$ with probability of $1 - \mathsf{negl}(\lambda)$.

Security. A keyword PIR scheme is (T, ε)*-secure* if, for all adversaries $\mathcal{A}$ running in time T and for all $k_i, k_j \in \mathcal{K}$, it holds

$$\begin{aligned} &|\Pr[\mathcal{A}(\mathsf{qu}) = 1 \mid (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(k_i)] \\ &-\Pr[\mathcal{A}(\mathsf{qu}) = 1 \mid (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(k_j)]| \leq \varepsilon. \end{aligned} \tag{2}$$

2.3 Learning With Errors

The security of our keyword PIR schemes relies on the decision version of the Learning With Errors (LWE) assumption [42].

Definition 2. *Given the LWE secret dimension N, the number of samples M, the ciphertext modulus q, and the error distribution χ, the (N, M, q, χ)-LWE problem is (T, ε)-hard if all adversaries running in time T have advantage at most ε in distinguishing the two distributions of $(A, \mathbf{s} \cdot A + \mathbf{e})$ and $(A, \mathbf{r})$, where the matrix $A \leftarrow \mathbb{Z}_q^{N \times M}$, the secret $\mathbf{s} \leftarrow \mathbb{Z}_q^N$, an error vector $\mathbf{e} \leftarrow \chi^M$, and the random vector $\mathbf{r} \leftarrow \mathbb{Z}_q^M$.*

Additively Homomorphic Encryption from LWE. Regev [42] gives a secret-key additive-homomorphic encryption scheme that is secure under the LWE assumption. With LWE parameters (N, M, q, χ) and a plaintext modulus p, the secret key is $\mathbf{s} \leftarrow \mathbb{Z}_q^N$. The encryption of a message $\mu \in \mathbb{Z}_p^M$ is

$$(A, \mathbf{b}) := (A, \mathbf{s} \cdot A + \mathbf{e} + \mu) \in \mathbb{Z}_q^{N \times M} \times \mathbb{Z}_q^M, \tag{3}$$

where $A \leftarrow \mathbb{Z}_q^{N \times M}$ and $e \leftarrow \chi^M$. With the secret key $\mathbf{s}$ and the scalar $\Delta := \lfloor q/p \rfloor$, the decryption of a ciphertext $(A, \mathbf{b})$ is

$$\mu := \mathsf{Round}_\Delta(\mathbf{b} - \mathbf{s} \cdot A \bmod q) \in \mathbb{Z}_p^M, \tag{4}$$

where $\mathsf{Round}_\Delta(\cdot)$ rounds the input to the nearest multiple of Δ and then divides it by Δ. Decryption succeeds as long as the absolute value of the error sampled from χ is smaller than $\Delta/2$. Additive homomorphism means that given two ciphertexts $(A_1, \mathbf{b}_1)$ and $(A_2, \mathbf{b}_2)$, their sum $(A_1 + A_2, \mathbf{b}_1 + \mathbf{b}_2)$ decrypts to the sum of the plaintexts, provided again that the error remains sufficiently small.

3 Review of Kushilevitz-Ostrovsky PIR

In this section, we review the row retrieval ability of Kushilevitz-Ostrovsky PIR (KOPIR) [24], which will be used in our keyword PIR constructions.

Protocol Row-KOPIR

Parameters: A database represented as a matrix in $\mathbb{Z}_p^{r \times c}$, LWE parameters (N, r, q, χ), a random LWE matrix $A \in \mathbb{Z}_q^{N \times r}$, and a scalar $\Delta := \lfloor q/p \rfloor$.

Protocol execution:

$\mathsf{Setup}(D \in \mathbb{Z}_p^{r \times c}) \to \mathsf{hint}$:

- Compute and return $\mathsf{hint} := A \cdot D \in \mathbb{Z}_q^{N \times c}$.

$\mathsf{Query}(i \in [r]) \to (\mathsf{st}, \mathsf{qu})$:

- Sample $\mathbf{s} \leftarrow \mathbb{Z}_q^N, \mathbf{e} \leftarrow \chi^r$ and define $\mathsf{st} := \mathbf{s}$.
- Compute $\mathsf{qu} := \mathbf{s} \cdot A + \mathbf{e} + \Delta \cdot \mathbf{u}_i \in \mathbb{Z}_q^r$, where $\mathbf{u}_i$ is the unit vector with a single 1 at index i.
- Return $(\mathsf{st}, \mathsf{qu})$.

$\mathsf{Answer}(\mathsf{qu} \in \mathbb{Z}_q^r, D \in \mathbb{Z}_p^{r \times c}) \to \mathsf{ans}$:

- Compute and return $\mathsf{ans} := \mathsf{qu} \cdot D \in \mathbb{Z}_q^c$.

$\mathsf{Recover}(\mathsf{st} \in \mathbb{Z}_q^N, \mathsf{hint} \in \mathbb{Z}_q^{N \times c}, \mathsf{ans} \in \mathbb{Z}_q^c) \to \mathsf{vec}$:

- Compute and return $\mathsf{vec} := \mathsf{Round}_\Delta(\mathsf{ans} - \mathsf{st} \cdot \mathsf{hint} \bmod q) \in \mathbb{Z}_p^c$.

Figure 2: The construction of Row-KOPIR from SimplePIR [23]

3.1 Definition of Row-KOPIR

We observe that in the basic scheme of KOPIR [24], the database is represented as a matrix, and then *a row* of the matrix database, instead of a single element, is privately retrieved each time. We abstract PIR with the row retrieval property as Row-KOPIR, which facilitates our constructions of $\mathsf{KPIR}^{\mathsf{hash}}$ and $\mathsf{KPIR}^{\mathsf{index}}$. Similar to recent PIR works [14, 23, 26, 31], we allow the server to pre-process the database before a client makes its query and to output a hint to each client. This pre-processing pushes most of the server's computation into a setup phase [14, 23]. It is worth noting that the hint should have a sublinear size relative to the database and be used by all clients for all of their queries. Below, we give formal syntax and security definitions of Row-KOPIR with pre-processing. For simplicity, we use Row-KOPIR to refer to Row-KOPIR with pre-processing in the subsequent sections.

Definition 3 (Row-KOPIR [24])**.** *Row-KOPIR with pre-processing consists of four routines, which all take the security parameter κ as an implicit input.*

- $\mathsf{Setup}(D) \to \mathsf{hint}$*: On input a matrix database D, output a pre-processed* hint *to the client.*

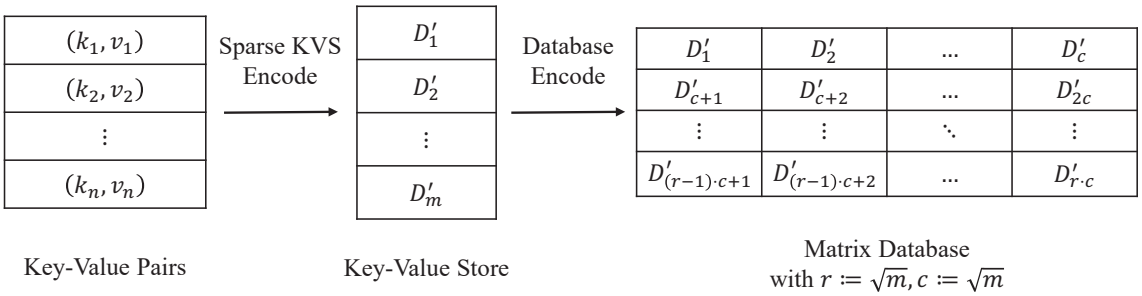


Figure 3: Database Encoding for $\mathsf{KPIR}^{\mathsf{kvs}}$

- $\mathsf{Query}(i) \to (\mathsf{st}, \mathsf{qu})$*: On input an index i, output a secret client's state* st *and a query* qu.
- $\mathsf{Answer}(D, \mathsf{qu}) \to \mathsf{ans}$*: On input the matrix database D and the query* qu*, output an answer* ans.
- $\mathsf{Recover}(\mathsf{st}, \mathsf{hint}, \mathsf{ans}) \to \mathsf{vec}$*: On input the state* st*, the hint* hint *and the answer* ans*, output a vector* vec.

A Row-KOPIR scheme should satisfy the following correctness and security properties.

Correctness. A Row-KOPIR scheme is correct if for any matrix database D of size $r \times c$ and all $i \in [r]$, it holds that

$$\mathsf{Recover}(\mathsf{st}, \mathsf{hint}, \mathsf{Answer}(D, \mathsf{qu})) = D[i, \cdot], \tag{5}$$

where $(\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(i)$ and $\mathsf{hint} \leftarrow \mathsf{Setup}(D)$.

Security. The client's query should reveal no information about its desired vector of the matrix database. Formally, a Row-KOPIR scheme is (T, ε)-secure if, for all adversaries $\mathcal{A}$ running in time T, on matrix database size $r \times c$, and for all $i, j \in [r]$, it holds that

$$\begin{aligned} &\Pr[\mathcal{A}(\mathsf{qu}) = 1 \mid (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(i)] \\ &- \Pr[\mathcal{A}(\mathsf{qu}) = 1 \mid (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(j)] \leq \varepsilon. \end{aligned} \tag{6}$$

For Row-KOPIR schemes to be non-trivial, the total communication between the client and server should be much smaller than the size of the matrix database. That is $|\mathsf{hint}| + |\mathsf{qu}| + |\mathsf{ans}| = o(|D|)$. It thus excludes trivial constructions, e.g., the client directly downloads the whole public database.

3.2 Instantiation from SimplePIR

Figure 2 shows an efficient instantiation of Row-KOPIR with pre-processing from SimplePIR [23]. The construction of SimplePIR is that the server arranges the database as a square matrix D of size $r \times c$, where $r \approx c$. To retrieve i-th element, (1) the client parses i as $(i_{\mathsf{row}}, i_{\mathsf{col}})$, which corresponds to the position in the matrix D. The client sends to the server ciphertext $\mathsf{ct} := (A, \mathbf{b} := \mathbf{s} \cdot A + \mathbf{e} + \Delta \cdot \mathbf{u}_i)$, where plaintext $\mathbf{u}_i$ is the dimension-r unit vector that has a single 1 at index i_{row}. (2) The server performs homomorphic evaluation on ct to compute[1] $\mathsf{ct}' := (A' := A \cdot D, \mathbf{b}' := \mathbf{b} \cdot D)$ and returns ct' to the client. (3) The client decrypts ct' to the vector $\mathsf{vec} := \mathsf{Round}_\Delta(\mathbf{b}' - \mathbf{s} \cdot A')$, which corresponds to the i_{row} row of matrix D, and then extracts $\mathsf{vec}[i_{\mathsf{col}}]$. Therefore, without the last extraction, SimplePIR is already an efficient construction of Row-KOPIR. We provide additional details and security proof in Appendix B.

4 Keyword PIR from Sparse Key-Value Stores

4.1 Sparse Key-Value Store

A Key-Value Store (KVS) [18, 37] is a data structure that maps a set of keys to the corresponding values. Below, we present a new notion, called sparse KVS.

Definition 4. *An α-sparse Key-Value Store (α-sparse KVS) is parameterized by a key space $\mathcal{K}$, a value space $\mathcal{V}$, input length n and output length m, and consists of two algorithms:*

- $\mathsf{Encode}(L) \to (D, H)$*: On input a set of key-value pairs $L := \{(k_i, v_i)\}_{i \in [n]} \in (\mathcal{K} \times \mathcal{V})^n$, output a vector D and a set of hash functions $H := \{h_i\}_{i \in [\alpha]}$.*
- $\mathsf{Decode}(H, k, D[H(k)]) \to v$*: On input a set of hash functions $H := \{h_i\}_{i \in [\alpha]}$, a key $k \in \mathcal{K}$ and entries $D[H(k)] := \{D[h_i(k)]\}_{i \in [\alpha]}$, output a value $v \in \mathcal{V} \cup \{\perp\}$.*

The α-sparse property allows Decode to access only a constant number α of elements in the vector D. Below, we show the correctness for inclusion and non-inclusion.

Correctness of inclusion. An α-sparse KVS is *correct for inclusion* if, for all $L \in (\mathcal{K} \times \mathcal{V})^n$ with distinct keys, it holds that for any $(k, v) \in L$, $\mathsf{Decode}(H, k, D[H(k)]) = v$ with probability 1, where $(D, H) \leftarrow \mathsf{Encode}(L)$.

Correctness of non-inclusion. An α-sparse KVS is *correct for non-inclusion* if, for all $L := (K, V) \in (\mathcal{K} \times \mathcal{V})^n$ with distinct keys, it holds that for any $k \notin K$,

[1] SimplePIR further pushes the dominant part, i.e., evaluating and sending $A \cdot D$, into the pre-processing phase.

$\mathsf{Decode}(H,k,D[H(k)]) = \perp$ with overwhelming probability, where $(D,H) \leftarrow \mathsf{Encode}(L)$.

Instantiation from Binary Fuse Filters. In Appendix C, we instantiate α-sparse KVS using Binary Fuse Filters (BFF) [20], which constructs the vector D with $\alpha := 3$ hash functions. Therefore, the decoding only accesses 3 positions of the encoding D. In our evaluated settings, m does not exceed $1.156n$, resulting in a small expansion. This property will be leveraged in our keyword PIR construction.

We emphasize the differences between our α-sparse KVS and sparse-OKVS used in circuit-based private set intersection [21] and private set union [43]. Our α-sparse KVS has a smaller encoding size, more efficient decoding, and a more succinct definition. Specifically, the encoding output D of sparse-OKVS is structured as $D := D_0 \| D_1$ with $|D_0| = \omega(|D_1|)$. The decoding requires access to a constant number of elements in large D_0 and a large number of elements in small D_1. In contrast, the encoding of our α-sparse KVS only comprises a single structure D with a similar size as D_0 of sparse-OKVS, and the decoding only accesses a constant number of elements on D. In addition to the efficiency advantages, our definition is more succinct and clear. These improvements mainly come from that sparse-OKVS is used in private set operations and requires the obliviousness property to protect the privacy of set elements. However, the input of α-sparse KVS is key-value pairs of a public database, which leads to fewer limitations for KVS constructions. Therefore, we explicitly consider the set of hash functions H as encoding output, which could depend on the input key-value pairs, while in sparse-OKVS, H must be independent of the input.

4.2 Construction $\mathsf{KPIR}^{\mathsf{kvs}}$

We present the construction $\mathsf{KPIR}^{\mathsf{kvs}}$, which is a generic transformation from standard PIR to keyword PIR using sparse KVS. We note that although both $\mathsf{KPIR}^{\mathsf{kvs}}$ and ChalametPIR use Binary Fuse Filters, ChalametPIR involves a single invocation of linear-communication FrodoPIR as shown in Appendix A. Our key idea is to utilize its *sparsity* property to invoke a constant number of sublinear-communication index PIR, offering significant communication advantages.

We present the construction as follows. Figure 3 graphically shows the database encoding of $\mathsf{KPIR}^{\mathsf{kvs}}$ and Figure 4 presents the detailed construction. Specifically, the server first exploits sparse KVS to encode the database D of size n into a vector D' of size $m := (1+\varepsilon)\cdot n$. We utilize the BFF scheme as our sparse KVS instantiation such that for each $(k,v) \in D$, $v = \sum_{i\in[\alpha]} D'[h_i(k)]$, where $h_i : \{0,1\}^* \to [m]$ is a random hash function. Then, for each query k, the client employs a standard PIR to secretly retrieve the entries in the positions $\{h_1(k),\ldots,h_\alpha(k)\}$ from the encoded database D'. Finally, the client decodes the output by computing $\sum_{i\in[\alpha]} D'[h_i(k)]$. Due to the sparsity, it only involves a constant number of standard PIR on a slightly expanded database. We note that $\mathsf{KPIR}^{\mathsf{kvs}}$ can be constructed from any index PIR, because it retrieves the entry of interest, instead of the whole row from the matrix database. To achieve high throughput and maintain constructions' consistency, we still use SimplePIR-based Row-KOPIR and then extract the entry of interest from the retrieved row. Essentially, $\mathsf{KPIR}^{\mathsf{kvs}}$ uses the default SimplePIR.

Protocol $\mathsf{KPIR}^{\mathsf{kvs}}$

Parameters: A database size n. An α-sparse KVS $(\mathsf{Encode},\mathsf{Decode})$ with output size m. A Row-KOPIR scheme $(\mathsf{Setup},\mathsf{Query},\mathsf{Answer},\mathsf{Recover})$.

Protocol execution:

$\mathsf{Setup}(D \in (\mathcal{K}\times\mathcal{V})^n) \to \mathsf{hint}_C, \mathsf{hint}_S$:

- Compute $(D', H) \leftarrow \mathsf{KVS.Encode}(D) \in \mathbb{Z}_p^m$, where $H := \{h_i : \{0,1\}^* \to [m]\}_{i\in[\alpha]}$.
- Compute $\mathsf{hint} \leftarrow \mathsf{Row\text{-}KOPIR}.\mathit{Setup}(D')$, where D' is represented as a matrix in $\mathbb{Z}_p^{\sqrt{m}\times\sqrt{m}}$.
- Return $(\mathsf{hint}_C := (H,\mathsf{hint}), \mathsf{hint}_S := D')$.

$\mathsf{Query}(k \in \mathcal{K}, \mathsf{hint}_C) \to (\mathsf{st},\mathsf{qu})$:

- Parse $\mathsf{hint}_C = (H,\mathsf{hint})$. For $i \in [\alpha]$, compute $h_i(k)$ written as a pair $(\mathsf{row}_i,\mathsf{col}_i)$, and $(\mathsf{st}_i,\mathsf{qu}_i) \leftarrow \mathsf{Row\text{-}KOPIR.Query}(\mathsf{row}_i)$.
- Return $\mathsf{st} := (k,\{\mathsf{st}_i,\mathsf{col}_i\}_{i\in[\alpha]}), \mathsf{qu} := (\{\mathsf{qu}_i\}_{i\in[\alpha]})$.

$\mathsf{Answer}(\mathsf{qu},\mathsf{hint}_S) \to \mathsf{ans}$:

- Parse $\mathsf{qu} = (\mathsf{qu}_1,\ldots,\mathsf{qu}_\alpha)$. For $i \in [\alpha]$, compute $\mathsf{ans}_i \leftarrow \mathsf{Row\text{-}KOPIR.Answer}(\mathsf{qu}_i,\mathsf{hint}_S)$.
- Return $\mathsf{ans} := (\{\mathsf{ans}_i\}_{i\in[\alpha]})$.

$\mathsf{Recover}(\mathsf{st},\mathsf{hint}_C,\mathsf{ans}) \to v$:

- Parse $\mathsf{st} = (k,\{\mathsf{st}_i,\mathsf{col}_i\}_{i\in[\alpha]})$, $\mathsf{hint}_C = (H,\mathsf{hint})$. and $\mathsf{ans} = (\{\mathsf{ans}_i\}_{i\in[\alpha]})$. For $i \in [\alpha]$, compute $\mathsf{vec}_i \leftarrow \mathsf{Row\text{-}KOPIR.Recover}(\mathsf{st}_i,\mathsf{hint},\mathsf{ans}_i)$.
- Reture $v := \mathsf{KVS.Decode}(H,k,\{\mathsf{vec}_i[\mathsf{col}_i]\}_{i\in[\alpha]})$.

Figure 4: Keyword PIR construction $\mathsf{KPIR}^{\mathsf{kvs}}$

Computation and communication costs. The primary cost is dominated by a constant number α of invocations of Row-KOPIR. The database size is $r \times c$ with $r := \sqrt{(1+\varepsilon)\cdot n}$ and $c := \sqrt{(1+\varepsilon)\cdot n}$, where n is the number of key-value pairs. In our evaluation, the parameter α is set to 3 and the parameter ε is a small constant between $0.125 \sim 0.156$.

Supporting databases with larger key-value sizes. Our pro-

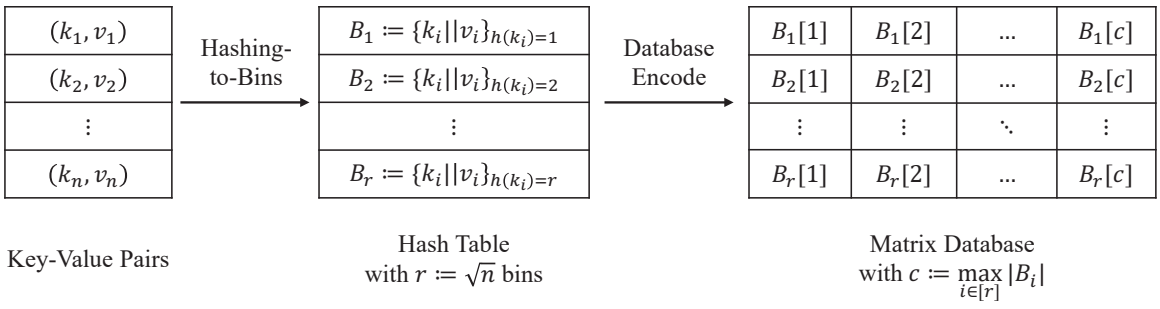


Figure 5: Database Encoding for $\mathsf{KPIR}^{\mathsf{hash}}$

tocol supports key-value pairs of arbitrary lengths. (1) For large keys, we can utilize a collision-resistant hashing function to reduce the length of keys. Specifically, given a hash function $h(\cdot) : \{0,1\}^* \to \{0,1\}^{\mu}$, we can compress a large key into a new smaller key of μ bits. The correctness requires that the hashing outputs for different keys are different except with negligible probability. Therefore, μ should be sufficiently long such that these collisions do not occur in practice. (2) To choose efficient parameters for underlying homomorphic encryption schemes, recent efficient PIR constructions [14, 23] usually support a database in which each entry is within a single element of the plaintext space $\mathbb{Z}_p$, e.g., 8-10 bits in the parameter settings of SimplePIR. To handle large key-value pairs, we encode each key-value pair as multiple elements of $\mathbb{Z}_p$, and store these elements consecutively in the same row of the encoded matrix. After reconstructing the corresponding row of elements, the client recovers any record of its choosing.

Theorem 1. *Assuming the underlying Row-KOPIR protocol* $(\mathsf{Setup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$ *provides correctness and security and KVS satisfies the correctness for inclusion and non-inclusion, the construction* $\mathsf{KPIR}^{\mathsf{hash}}$ *in Figure 6 achieves correctness and security.*

Proof. There are two cases for the correctness analysis. In the case of $k \in \{k_1, k_2, \ldots, k_n\}$ and $k = k_j$, according to the inclusion correctness of sparse KVS and the correctness of Row-KOPIR, $v = \mathsf{KVS.Decode}(H, k, \{\mathsf{vec}_i[\mathsf{col}_i]\}_{i \in [\alpha]}) = \mathsf{KVS.Decode}(H, k_j, \{\mathsf{vec}_i[\mathsf{col}_i]\}_{i \in [\alpha]}) = v_j$. Hence, the output $v = v_j$. In the case of $k \notin \{k_1, k_2, \ldots, k_n\}$, according to the non-inclusion correctness of sparse KVS and the correctness of Row-KOPIR, $\mathsf{KVS.Decode}(H, k, \{\mathsf{vec}_i[\mathsf{col}_i]\}_{i \in [\alpha]}) = \perp$ with overwhelming probability. The security is straightforwardly guaranteed by the security of Row-KOPIR and the security of a constant number α of Row-KOPIR invocations through standard hybrid argument.

□

5 Keyword PIR from Hashing-to-Bins

We present a keyword PIR construction called $\mathsf{KPIR}^{\mathsf{hash}}$ using hashing-to-bins techniques. Similar to MulPIR [5], the main idea is that the server uses hashing to map key-value pairs into a hash table, in which each bin is viewed as a single element, and with the same hashing function, the client then retrieves an entire bin using standard PIR. Different from MulPIR, we instantiate PIR with SimplePIR-based Row-KOPIR and provide parameter analysis in our construction to ensure practical efficiency.

Figure 5 graphically shows the database encoding of $\mathsf{KPIR}^{\mathsf{hash}}$ and Figure 6 presents the construction. Specifically, the two parties agree on a publicly known random hash function h in advance. Besides, an empty hash table with r bins is initialized, where r is set as $\sqrt{n}$ for database size n. The setting is to minimize the overhead of Row-KOPIR, which has better communication costs for a square matrix. For each key-value pair (k, v), the server first inserts $k||v$ into the $h(k)$-th bin. These inserted bins will be combined into a matrix, where each row corresponds to the corresponding bin of the hash table. Note that due to the randomness of hashing, the bins have different numbers of entries. To set all bins to be of the same size, we compute the size of the most populated bin and pad dummy entries in bins that are not fully occupied. Finally, the two parties invoke the Row-KOPIR protocol, which enables efficient retrieval of the $h(k)$-th bin corresponding to the client's query k. If the query is in the server's key-value pairs, both parties map it to the same bin and hence the client only needs to lookup the corresponding value from this bin.

Computation and communication costs. The primary cost is dominated by a single invocation of Row-KOPIR. The size of the matrix database for Row-KOPIR is $r \times c$ with $r := \sqrt{n}$ and $c := (1+\varepsilon) \cdot \sqrt{n}$, where ε is a small constant that depends on the relation between the number of bins and the size of the database [19, 40]. In the evaluation, ε is set to $1.142 \sim 1.873$ for our experimental settings. We note that we can use the same solution as the construction from KVS in Section 4 to handle large key-value pairs.

Optimization from permutation-based hashing. We can employ the permutation-based hashing techniques [7, 38] to reduce the bit-length of the stored items in each bin. Specifically, given an element x and the number r of bins, we partition x into two segments: x_1 and x_2, where x_1 comprises

Protocol $\mathsf{KPIR}^{\mathsf{hash}}$

Parameters: A database size n and $r := \sqrt{n}$. A hash function $h : \{0,1\}^* \to [r]$. A Row-KOPIR scheme $(\mathsf{Setup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$.

Protocol execution:

$\mathsf{Setup}(D \in (\mathcal{K} \times \mathcal{V})^n) \to \mathsf{hint}_C, \mathsf{hint}_S$:

- Initialize r empty bins $B_1, \ldots, B_r$.
- For each $(k, v) \in D$, compute $i := h(k) \in [r]$ and append $k \| v$ into B_i.
- Combine $B_1, \ldots, B_r$ as rows into a matrix D' of size $r \times c$, where $c := \max_{i \in [r]} \{|B_i|\}$.
- Compute $\mathsf{hint}_C \leftarrow \mathsf{Row\text{-}KOPIR.Setup}(D')$.
- Return $(\mathsf{hint}_C, \mathsf{hint}_S := D')$.

$\mathsf{Query}(k \in \mathcal{K}) \to (\mathsf{st}, \mathsf{qu})$:

- Compute $(\mathsf{st}', \mathsf{qu}) \leftarrow \mathsf{Row\text{-}KOPIR.Query}(i)$, where $i := h(k) \in [r]$.
- Return $(\mathsf{st} := (k, \mathsf{st}'), \mathsf{qu})$

$\mathsf{Answer}(\mathsf{qu}, \mathsf{hint}_S) \to \mathsf{ans}$:

- Compute $\mathsf{ans} \leftarrow \mathsf{Row\text{-}KOPIR.Answer}(\mathsf{qu}, \mathsf{hint}_S)$.
- Return ans.

$\mathsf{Recover}(\mathsf{st}, \mathsf{hint}_C, \mathsf{ans}) \to v$:

- Parse $\mathsf{st} := (k, \mathsf{st}')$. Compute $\{k_1^* \| v_1^*, \ldots, k_c^* \| v_c^*\} \leftarrow \mathsf{Row\text{-}KOPIR.Recover}(\mathsf{st}', \mathsf{hint}_C, \mathsf{ans})$.
- Return $v := v_i^*$ if $k = k_i^*$ exists for $i \in [c]$, $v := \perp$ otherwise.

Figure 6: Keyword PIR construction $\mathsf{KPIR}^{\mathsf{hash}}$

the first $\log r$ bits of x and x_2 comprises the remaining bits of x. Subsequently, we map the element x_2 to a bin indexed by $x_1 \oplus h(x_2)$. Notably, the value stored in the bin, namely x_2, is $\log r$ bits shorter than the original element x, thereby leading to a reduction in the overhead of Row-KOPIR.

Theorem 2. *Assuming the underlying Row-KOPIR protocol* $(\mathsf{Setup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$ *provides correctness and security, the construction* $\mathsf{KPIR}^{\mathsf{hash}}$ *in Figure 6 achieves correctness and security.*

Proof. There are two cases for the correctness analysis. In the case of $k \in \{k_1, k_2, \ldots, k_n\}$ and $k = k_j$, according to the correctness of Row-KOPIR, $\{k_1^* \| v_1^*, \ldots, k_c^* \| v_c^*\}$ is equal to the bin $B_{h(k_j)}$ and $k_j \| v_j \in B_{h(k_j)}$. Therefore, the output $v = v_j$. In the case of $k \notin \{k_1, k_2, \ldots, k_n\}$, there does not exist $k_i^* = k$ for each $k_i^* \in \{k_1^* \| v_1^*, \ldots, k_c^* \| v_c^*\}$. Therefore, $v = \perp$. Note that $\{k_1^* \| v_1^*, \ldots, k_c^* \| v_c^*\}$ includes dummy points. We set these as rare values to avoid colliding with k. The security is straightforwardly guaranteed by the security of Row-KOPIR. □

6 Keyword PIR from Approximate Key-to-Index Mapping

We formalize approximate key-to-index mappings, which is a critical building block of the following keyword PIR construction. Approximate key-to-index mappings are widely used in the database field [17], and to the best of our knowledge, this is the first time using them in the cryptographic field.

6.1 Approximate Key-to-Index Mapping

An approximate key-to-index mapping is a data structure parametric in an integer ε. Given a sorted key set $K \subset \mathcal{K}$ as input, it outputs a mapping between keys from K and their approximate positions in the set K with error ε.

Definition 5. *An approximate key-to-index mapping is parameterized by an error* ε*, key space* $\mathcal{K}$*, input set size n, and consists of two algorithms:*

- $\mathsf{Map}_\varepsilon(K \subset \mathcal{K}) \to \mathsf{map}$*: On input a sorted set* $K \subset \mathcal{K}$ *of size n, output a structure* map.
- $\mathsf{Extract}_\varepsilon(k \in \mathcal{K}, \mathsf{map}) \to i_k$*: On input a key* $k \in \mathcal{K}$ *and the structure* map*, output a position* $i_k \in [n]$.

Approximate key-to-index mappings should satisfy the following correctness.

Correctness. An ε-approximate key-to-index mapping is correct if, for any set $K \subset \mathcal{K}$ and every key $k \in K$, the Extract output i_k satisfies $i_k \in [\mathsf{Pos}(k) - \varepsilon, \mathsf{Pos}(k) + \varepsilon]$, where $\mathsf{Pos}(k)$ is the correct position of k within K.

Instantiation from Piece-wise Linear Approximation. Paolo Ferragina and Giorgio Vinciguerra [17] proposed an efficient construction of approximate key-to-index mapping, called Piece-wise Linear Approximation (PLA). Specifically, given a sorted set K of size n, PLA outputs a piece-wise linear approximation function. It includes d segments, where each segment consists of a triple $S := (t, a, b)$. Here, t denotes the initial position of this segment, and (a, b) defines a linear function $f_S(x) = a \cdot x + b$. Building on earlier works [35], their method reduces the ε-approximate PLA problem to constructing the convex hull of a set of points $(k_i, \mathsf{Pos}(k_i))$ for $i \in [n]$. We give a high-level overview of their method and refer to the works [17, 35] for more details. The procedure involves the following steps:

1. Compute the convex hull for the current set of points.

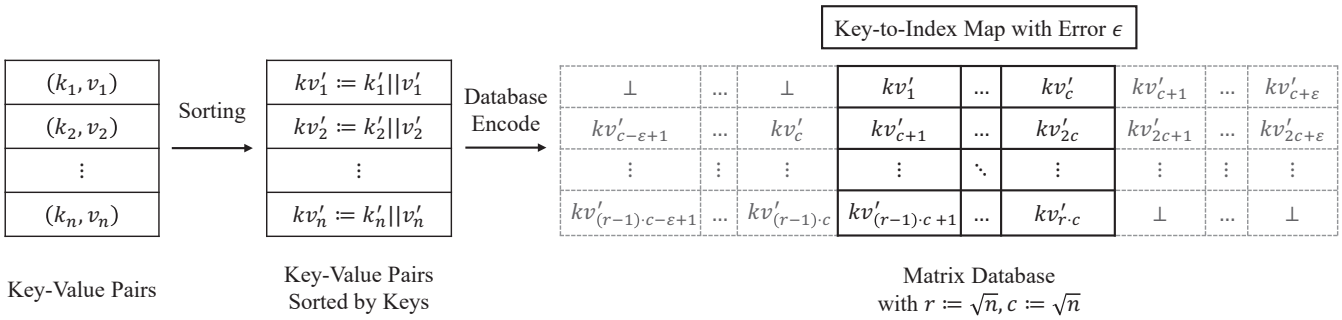


Figure 7: Database Encoding for $\mathsf{KPIR}^{\mathsf{index}}$

2. While the hull remains within a (potentially rotated) rectangle with height no greater than 2ε, increment the index i and add the next point to the set.

3. If the enclosing rectangle exceeds the height threshold of 2ε, terminate the extension. At this stage, define a segment of the PLA model by choosing a line that bisects the rectangle.

4. Clear the set of processed points and continue processing the remaining points from the input.

As demonstrated below, PLA formally establishes the feasibility of this construction and the relationship between the error parameter ε and the resulting number of segments d.

Lemma 1 (PLA [17, 35])**.** *Given a sorted set $K \subset \mathcal{K}$ of n entries that are ascending, there exists a linear-time and linear-space algorithm to compute a PLA mapping with error ε and the number of segments d, such that $d \leq n/2\varepsilon$.*

This work focuses on the setting where the number of segments d is sublinear with the key size n, because in our keyword PIR construction, the output of PLA will be downloaded by the client. With appropriate parameters, d could be sublinear with n. For example, we can set $\varepsilon := \sqrt{n}$, and hence d is less than $\sqrt{n}$. We will give more practical parameter settings in the following keyword PIR construction.

6.2 Construction $\mathsf{KPIR}^{\mathsf{index}}$

We present an efficient keyword PIR construction called $\mathsf{KPIR}^{\mathsf{index}}$ using approximate key-to-index mappings. This requires only a single invocation of Row-KOPIR on an encoded database with almost the same size as the input database. The main idea is to construct an approximate key-to-index mapping with error ε and then retrieve all 2ε entries around the approximate position of the query key, by utilizing a repetition-based matrix encoding combined with Row-KOPIR.

We present the construction as follows. Figure 7 graphically shows the database encoding of $\mathsf{KPIR}^{\mathsf{index}}$ and Figure 8 presents the detailed construction. Specifically, our solution is first to compute an approximate key-to-index mapping map with error ε by invoking the PLA method [17]. This means that for any key k, the mapping map returns a position i_k, which is at most ε away from the correct one $\mathsf{pos}(k)$ in the input database, namely $i_k \in [\mathsf{pos}(k) - \varepsilon, \mathsf{pos}(k) + \varepsilon]$. Therefore, for each query, we need to retrieve continuous 2ε entries around i_k. To achieve a single invocation of Row-KOPIR, we propose a repetition-based matrix encoding method with the following steps. (1) We sort the key-value database D of size n according to the keys and represent the sorted database as a square matrix D' of size $\sqrt{n} \times \sqrt{n}$ in a row-wise manner. (2) To enable retrieving continuous 2ε entries in one row, we repeatedly append ε entries at the beginning and ε entries at the end of the row in the matrix, resulting the encoded matrix D' of size $\sqrt{n} \times (\sqrt{n} + 2\varepsilon)$. (3) For each query, the client determines the row position of its query and invokes Row-KOPIR to obtain an entire row and extract the final result. We note that larger ε reduces the size of the key-to-index mapping (i.e., the client's memory) but increases the size of the encoded matrix (i.e., the server's memory), causing higher PIR costs.

Computation and communication costs. Similar to the construction for hashing-to-bins, the primary cost is dominated by a single invocation of Row-KOPIR. The database size is $r \times c$ with $r := \sqrt{n}$ and $c := \sqrt{n} + 2\varepsilon$, where n is the number of key-value pairs. We note that we can use the same solution as the construction from KVS in Section 4 to handle large key-value pairs.

Improved parameters for approximate key-to-index mappings. As shown in Lemma 1, to achieve the mapping size d is sublinear (e.g., square root) to the database size n, it requires setting large ε (e.g., $\sqrt{n}$), which results in large encoded matrix D' and high overhead of Row-KOPIR. We propose practical parameter settings, namely ε is set to a small constant, e.g., 2 in our evaluation. To address the large size of the mapping, we hash the input keys into uniformly random values. Such a nearly uniform distribution makes it easy to learn a piecewise linear approximation function with only a small number of segments. Note that using hashing functions is compatible with our constructions because hashing is inherently used to compress the size of keys. Although the number of segments

Protocol $\mathsf{KPIR}^{\mathsf{index}}$

Parameters: A database size n. A Row-KOPIR scheme $(\mathsf{Setup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$. An approximate key-to-index mapping AKIM $(\mathsf{Map}_\varepsilon, \mathsf{Extract}_\varepsilon)$ with error ε.

Protocol execution:

$\mathsf{Setup}(D \in (\mathcal{K} \times \mathcal{V})^n) \to \mathsf{hint}_C, \mathsf{hint}_S$:

- Compute $\{(k'_1 \| v'_1), \ldots, (k'_n \| v'_n)\}$ by sorting D in ascending order based on the keys.
- Compute $\mathsf{map} \leftarrow \mathsf{AKIM.Map}_\varepsilon(k'_1, \ldots, k'_n)$.
- Define a matrix D' of size $r \times (c + 2\varepsilon)$ with $r := \sqrt{n}$ and $c := \sqrt{n}$, where the i-th row of D' includes $\{k'_{(i-1)\cdot c-\varepsilon+1} \| v'_{(i-1)\cdot c-\varepsilon+1}, \ldots, k'_{i\cdot c+\varepsilon} \| v'_{i\cdot c+\varepsilon}\}$.
- Compute $\mathsf{hint} \leftarrow \mathsf{Row\text{-}KOPIR.Setup}(D')$.
- Return $(\mathsf{hint}_C := (\mathsf{map}, \mathsf{hint}), \mathsf{hint}_S := D')$.

$\mathsf{Query}(k \in \mathcal{K}, \mathsf{hint}_C) \to (\mathsf{st}, \mathsf{qu})$:

- Parse $\mathsf{hint}_C = (\mathsf{map}, \mathsf{hint})$ and compute $\mathsf{pos} \leftarrow \mathsf{AKIM.Extract}_\varepsilon(\mathsf{map}, k) \in [m]$.
- Compute $i := \lceil \mathsf{pos}/c \rceil$ and $(\mathsf{st}', \mathsf{qu}) \leftarrow \mathsf{Row\text{-}KOPIR.Query}(i)$.
- Return $(\mathsf{st} := (k, \mathsf{st}'), \mathsf{qu})$

$\mathsf{Answer}(\mathsf{qu}, \mathsf{hint}_S) \to \mathsf{ans}$:

- Compute $\mathsf{ans} \leftarrow \mathsf{Row\text{-}KOPIR.Answer}(\mathsf{qu}, \mathsf{hint}_S)$.
- Return ans.

$\mathsf{Recover}(\mathsf{st}, \mathsf{hint}_C, \mathsf{ans}) \to v$:

- Parse $\mathsf{hint}_C = (\mathsf{map}, \mathsf{hint})$, $\mathsf{st} := (k, \mathsf{st}')$. Compute $\{(k^*_1, v^*_1), \ldots, (k^*_{c+2\varepsilon}, v^*_{c+2\varepsilon})\} \leftarrow \mathsf{Row\text{-}KOPIR.Recover}(\mathsf{st}', \mathsf{hint}, \mathsf{ans})$.
- Return $v := v^*_i$ if $k = k^*_i$ exists for $i \in [c]$, $v := \perp$ otherwise.

Figure 8: Keyword PIR construction $\mathsf{KPIR}^{\mathsf{index}}$

is theoretically linear with the size of the database, the actual number of segments is only about 1% of the database size in our evaluation. We give detailed results in Appendix D.

Theorem 3. *Assuming the underlying Row-KOPIR protocol* $(\mathsf{Setup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$ *provides correctness and security, the construction* $\mathsf{KPIR}^{\mathsf{index}}$ *in Figure 8 achieves correctness and security.*

Proof. There are two cases for the correctness analysis. In the case of $k \in \{k_1, k_2, \ldots, k_n\}$ and $k = k_j$, according to the correctness of Row-KOPIR and approximate key-to-index map- pings, $\{(k^*_1, v^*_1), \ldots, (k^*_{c+2\varepsilon}, v^*_{c+2\varepsilon})\}$ is equal to the bin $B_{\lceil \mathsf{pos}/c \rceil}$ and $k_j \| v_j \in B_{\lceil \mathsf{pos}/c \rceil}$. Therefore, the output $v = v_j$. In the case of $k \notin \{k_1, k_2, \ldots, k_n\}$, it does not exist k^*_i in $\{(k^*_1, v^*_1), \ldots, (k^*_{c+2\varepsilon}, v^*_{c+2\varepsilon})\}$ that is equal to k. Therefore, $v = \perp$ in this case. The security is straightforwardly guaranteed by the security of Row-KOPIR. □

7 Implementation and Evaluation

We implement our protocols in Java and conduct all experiments on two machines, each equipped with an Intel Core i9-9900K processor running at 3.6 GHz and 128 GB of memory. All evaluations are performed using a single thread. These machines simulate the client and server, respectively, and are connected via network cards, forming a realistic LAN network with a bandwidth of 2.5 Gbps and an RTT latency of 0.4 ms. Our source code is available at https://github.com/alibaba-edu/mpc4j.

7.1 Implementation Details

For comprehensive and fair comparisons, we conduct a unified platform to implement both our proposed constructions and the state-of-the-art keyword PIR, ChalametPIR [10]. We introduce the implementation details below.

Implementation details of our protocols. We set the computational security parameter $\kappa := 128$ and the statistical security parameter $\lambda := 40$. Our three protocols are built on SimplePIR. We implement SimplePIR in our platform and adhere to their original parameter settings. Specifically, we set the LWE dimension to $N := 2^{10}$, the ciphertext modulus $q := 2^{32}$, and the error distribution χ to the discrete Gaussian distribution with standard deviation $\sigma := 6.4$. In addition, we set plaintext modulus $p := 2^8$. With such parameters, plaintexts and ciphertexts are supported and operated by native data types, byte and int, respectively. This also simplifies the conversions between $\mathbb{Z}_p$ and $\mathbb{Z}_q$ since they only involve shifting operations, instead of division and modular. Besides, we also set larger LWE parameters with $N := 1408$ as recommended by HintlessPIR [26] and provide experimental results in Appendix D for completeness. We note that larger N slightly increases the communication and computation overhead, but the additional overhead is marginal.

We use Binary Fuse Filters (BFF) [20] to instantiate our sparse KVS in the $\mathsf{KPIR}^{\mathsf{kvs}}$ construction. Following the open-source implementation [1, 20], we set the number of hash functions to $k := 3$ and the expansion rate $1 + \varepsilon := \max\left(\left\lfloor\left(0.875 + 0.25 \cdot \max\left(1, \frac{\log(10^6)}{\log(n)}\right)\right) \cdot n\right\rfloor, \lfloor 1.125n \rfloor\right)$, where n is the database size. Empirically, ε is set to $1.125 \sim 1.156$ in our evaluation. We change the operations

Table 1: Communication and runtime comparison of our protocols with the state-of-the-art ChalametPIR [10]. The best result is marked in green, and the second best result is marked in blue.

Parameter		Protocol	Setup Phase		Online Phase	
DB Size	Entry Len.		Comm (MB)	Time (s)	Comm (MB)	Time (ms)
2^{18}	32B	ChalametPIR [10]	0.27	47.78	1.16	39.41
		$\mathsf{KPIR}^{\mathsf{kvs}}$	13.75	6.03	0.08	26.86
		$\mathsf{KPIR}^{\mathsf{hash}}$	19.22	7.71	0.03	19.08
		$\mathsf{KPIR}^{\mathsf{index}}$	14.46	6.78	0.03	14.32
	64B	ChalametPIR [10]	0.49	56.61	1.16	44.15
		$\mathsf{KPIR}^{\mathsf{kvs}}$	18.28	12.19	0.11	41.82
		$\mathsf{KPIR}^{\mathsf{hash}}$	24.47	13.28	0.04	28.94
		$\mathsf{KPIR}^{\mathsf{index}}$	20.33	13.01	0.04	25.66
	128B	ChalametPIR [10]	0.92	80.50	1.16	53.61
		$\mathsf{KPIR}^{\mathsf{kvs}}$	25.50	21.21	0.15	78.16
		$\mathsf{KPIR}^{\mathsf{hash}}$	37.19	28.84	0.06	54.50
		$\mathsf{KPIR}^{\mathsf{index}}$	29.30	23.57	0.05	48.51
	256B	ChalametPIR [10]	1.79	126.06	1.16	81.31
		$\mathsf{KPIR}^{\mathsf{kvs}}$	35.06	42.10	0.21	139.93
		$\mathsf{KPIR}^{\mathsf{hash}}$	56.72	64.58	0.09	117.51
		$\mathsf{KPIR}^{\mathsf{index}}$	44.42	51.04	0.08	98.43
2^{20}	32B	ChalametPIR [10]	0.27	188.58	4.50	116.52
		$\mathsf{KPIR}^{\mathsf{kvs}}$	26.88	24.95	0.16	93.44
		$\mathsf{KPIR}^{\mathsf{hash}}$	33.59	29.04	0.06	60.34
		$\mathsf{KPIR}^{\mathsf{index}}$	27.35	23.97	0.05	46.97
	64B	ChalametPIR [10]	0.49	217.76	4.50	117.52
		$\mathsf{KPIR}^{\mathsf{kvs}}$	36.00	46.38	0.21	159.16
		$\mathsf{KPIR}^{\mathsf{hash}}$	46.97	55.22	0.08	106.43
		$\mathsf{KPIR}^{\mathsf{index}}$	37.44	43.53	0.07	79.28
	128B	ChalametPIR [10]	0.92	317.15	4.50	188.37
		$\mathsf{KPIR}^{\mathsf{kvs}}$	49.94	85.50	0.29	264.49
		$\mathsf{KPIR}^{\mathsf{hash}}$	67.47	113.08	0.11	170.54
		$\mathsf{KPIR}^{\mathsf{index}}$	52.91	89.50	0.10	122.98
	256B	ChalametPIR [10]	1.79	473.13	4.50	256.43
		$\mathsf{KPIR}^{\mathsf{kvs}}$	69.09	162.05	0.40	474.74
		$\mathsf{KPIR}^{\mathsf{hash}}$	99.00	224.45	0.16	292.48
		$\mathsf{KPIR}^{\mathsf{index}}$	77.66	171.92	0.14	262.66
2^{22}	32B	ChalametPIR [10]	0.27	741.23	18.00	306.47
		$\mathsf{KPIR}^{\mathsf{kvs}}$	53.75	103.05	0.31	303.34
		$\mathsf{KPIR}^{\mathsf{hash}}$	60.94	109.33	0.11	173.86
		$\mathsf{KPIR}^{\mathsf{index}}$	53.61	97.22	0.10	156.48
	64B	ChalametPIR [10]	0.49	876.66	18.00	371.49
		$\mathsf{KPIR}^{\mathsf{kvs}}$	72.00	187.17	0.42	519.98
		$\mathsf{KPIR}^{\mathsf{hash}}$	85.22	213.15	0.15	290.46
		$\mathsf{KPIR}^{\mathsf{index}}$	72.42	177.96	0.14	259.71
	128B	ChalametPIR [10]	0.92	1226.85	18.00	587.21
		$\mathsf{KPIR}^{\mathsf{kvs}}$	99.34	371.80	0.58	975.03
		$\mathsf{KPIR}^{\mathsf{hash}}$	120.06	429.84	0.21	528.97
		$\mathsf{KPIR}^{\mathsf{index}}$	100.61	341.82	0.19	449.16
	256B	ChalametPIR [10]	1.79	1875.72	18.00	939.90
		$\mathsf{KPIR}^{\mathsf{kvs}}$	138.19	739.71	0.81	1940.87
		$\mathsf{KPIR}^{\mathsf{hash}}$	176.34	901.53	0.30	1031.57
		$\mathsf{KPIR}^{\mathsf{index}}$	143.58	702.99	0.27	844.02

of BFF from XOR to modular addition in the LWE plaintext space for ensuring compatibility with SimplePIR. We use the PLA method [17] to instantiate our approximate key-to-index mapping in the $\mathsf{KPIR}^{\mathsf{index}}$ construction. We implement the PLA scheme in our platform, following the reference implementation [3]. We set the error $\varepsilon := 4$ and the number

Table 2: Performance of $\mathsf{KPIR}^{\mathsf{index}}$ on difference approximation error ε. 'Client-Map Size' means the size of the client's key-to-index mapping and 'Server-DB Exp. Rate' means the expansion rate between the server's original and encoded databases. The best result is marked in green.

Parameter		Setup Phase		Online Phase		Memory	
Entry Len.	Epsilon ε	Comm (MB)	Time (s)	Comm (MB)	Time (ms)	Client-Map Size (KB)	Server-DB Exp. Rate
32B	4	27.35	23.97	0.05	46.97	324.22	1.069
	8	28.37	24.72	0.05	83.61	94.95	1.118
	16	30.81	27.33	0.05	79.45	25.59	1.217
	32	35.79	30.88	0.06	104.24	7.05	1.415
64B	4	37.44	43.53	0.07	79.28	324.22	1.094
	8	39.47	47.50	0.07	109.38	94.95	1.160
	16	43.90	51.76	0.08	107.47	25.59	1.293
	32	52.88	63.45	0.08	132.05	7.05	1.558
128B	4	52.91	89.50	0.10	122.98	324.22	1.127
	8	56.94	95.97	0.10	181.46	94.95	1.219
	16	65.37	112.20	0.11	203.70	25.59	1.401
	32	82.35	138.99	0.13	241.96	7.05	1.765
256B	4	77.66	171.92	0.14	262.66	324.22	1.190
	8	85.69	189.22	0.15	291.12	94.95	1.317
	16	102.12	246.84	0.16	326.87	25.59	1.571
	32	135.10	334.75	0.20	450.40	7.05	2.079

of segments is $0.013n$ for our settings, where n is the input database size. In Appendix D, we also show the detailed parameters of our three constructions in Tables 3, 4, and 5, including the size of the encoded matrix database and expansion rate.

Implementation details of ChalametPIR [10]. For a fair comparison, we implement ChalametPIR in our platform, following the parameter settings of their source code [2]. We compare with FrodoPIR-based ChalametPIR because (1) their open-source code supports only FrodoPIR and (2) as shown in Appendix A, ChalametPIR lacks a theoretically analyzed or empirically validated SimplePIR-based construction. Specifically, ChalametPIR sets the LWE dimension to $N := 1774$, ciphertext modulus $q := 2^{32}$, and the error distribution χ to the uniformly random ternary distribution. In addition, they use KVS and instantiate it with BFF. For the BFF implementation, we use the same parameters as our $\mathsf{KPIR}^{\mathsf{kvs}}$ construction, which is also in line with their implementation.

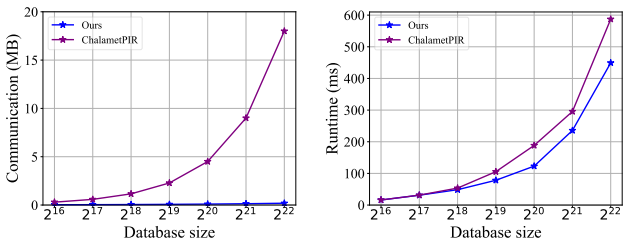
7.2 Evaluation Results

We evaluate our three constructions, i.e., $\mathsf{KPIR}^{\mathsf{kvs}}$, $\mathsf{KPIR}^{\mathsf{hash}}$, and $\mathsf{KPIR}^{\mathsf{index}}$, and compare with the state-of-the-art keyword PIR, ChalametPIR [10]. We show the overheads of the setup and online phases. It is worth noting that the setup phase is only executed once by the server and hence the communication and computational costs can be amortized over multiple clients for multiple queries. Therefore, we mainly focus on the online costs below.

Performance of our constructions. Table 1 shows the runtime and communication costs of our three constructions. All of these constructions achieve practical overheads. For example, for a database containing 1 million 256-byte entries, the runtime of all our schemes is less than 0.5 seconds and the communication is less than 0.4 MB. Among three constructions, $\mathsf{KPIR}^{\mathsf{index}}$ is the most efficient because it only invokes Row-KOPIR once on an encoded database with almost the same size as the original database. The performance of $\mathsf{KPIR}^{\mathsf{hash}}$ is slightly worse than that of $\mathsf{KPIR}^{\mathsf{index}}$. The reason is that $\mathsf{KPIR}^{\mathsf{index}}$ expands the original database. $\mathsf{KPIR}^{\mathsf{kvs}}$ requires more communication and computational costs, since it requires three invocations of Row-KOPIR. In addition, our three keyword PIR schemes introduce small additional memory costs. In particular, the server's memory is caused by storing the encoded database. As shown in Tables 3, 4, 5, $\mathsf{KPIR}^{\mathsf{kvs}}$, $\mathsf{KPIR}^{\mathsf{hash}}$ and $\mathsf{KPIR}^{\mathsf{index}}$ increase at most $1.156\times$, $1.873\times$, and $1.365\times$ memory costs, respectively. Moreover, except for storing the hint of underlying index PIR, the additional client memory from key-to-index mappings is still small, e.g., three and one hash functions in $\mathsf{KPIR}^{\mathsf{kvs}}$ and $\mathsf{KPIR}^{\mathsf{hash}}$, respectively. The memory of $\mathsf{KPIR}^{\mathsf{index}}$ is given in detail below.

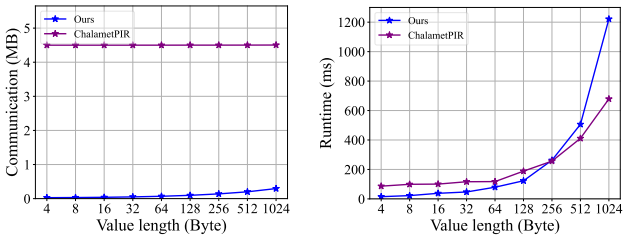
In Table 2, we further report detailed performance of $\mathsf{KPIR}^{\mathsf{index}}$ for different values of ε, i.e., error of approximate key-to-index mapping. We observe that as ε increases, the online computation and communication overhead will correspondingly increase. Similarly, larger ε will increase the size of the encoded database, which is the server's memory cost. However, larger ε will decrease the size of the approximate key-to-index mapping, which is the client's memory cost. As shown in Table 2, the key-to-index mapping size is small even when $\varepsilon := 4$, particularly 324 KB. Therefore, in our following evaluation, we set $\varepsilon := 4$ for comparisons.

Communication comparison with ChalametPIR. We show the communication comparison in Table 1. All our three con-



(a) Communication (b) Runtime

Figure 9: Online communication and runtime comparisons on different database sizes. We fix the value length as 128 bytes.



(a) Communication (b) Runtime

Figure 10: Online communication and runtime comparisons on different value lengths. We fix the database size as 2^{20}.

structions achieve $15 \sim 178\times$ online communication improvement over ChalametPIR. The reason is that our online communication is sublinear to the database size, rather than linear correlation in ChalametPIR. A drawback of our constructions is large setup communication, which is caused by sending hints in the invocation of SimplePIR. However, this communication can be amortized over multiple queries, since the hint is only downloaded by the client once and for all. We also show the online communication comparisons between ChalametPIR and the most efficient construction KPIR$^{\text{index}}$ in Figures 9(a) and 10(a). We observe that as the database size increases, the communication of ChalametPIR increases linearly, but KPIR$^{\text{index}}$ introduces a slight increase and hence has better advantages. They are in line with the theoretical analysis. Moreover, value lengths have small effects on the communication costs for both schemes.

Runtime comparison with ChalametPIR. We show the runtime comparison in Table 1. Our constructions achieve $1.1 \sim 2.4\times$ runtime improvement, depending on database size and entry length. For example, for a database containing 1 million 32-byte entries, KPIR$^{\text{index}}$ only requires 47 ms, outperforming ChalametPIR by $2.4\times$. Essentially, both our schemes and ChalametPIR are efficient, because they only invoke lightweight LWE-based homomorphic encryption without heavy operations. We also show the online runtime comparisons between ChalametPIR and the most efficient KPIR$^{\text{index}}$ in Figures 9(b) and 10(b). We observe that as the database size increases, we have larger advantages of online runtime. In addition, KPIR$^{\text{index}}$ has runtime advantages when the value length is smaller than 256 bytes, while ChalametPIR shows slightly better performance for large value lengths.

8 Conclusion

In this work, we introduce three practical keyword PIR constructions. Our core insight lies in constructing a key-to-index mapping and reducing the keyword PIR problem to standard PIR. To this end, we encode the server's key-value database into an indexable database along with three different key-to-index mappings. After that, we can achieve keyword PIR by invoking standard PIR on the encoded database to retrieve specific positions according to the mappings. We fully implement our constructions and state-of-the-art keyword PIR scheme ChalametPIR in a unified platform for fair comparisons. We evaluate the efficiency of our constructions and the results show that our scheme achieves $15 \sim 178\times$ communication improvement and $1.1 \sim 2.4\times$ runtime improvement, depending on database size and entry length.

An interesting research direction is to support database updates and authentication. For database updates, when only database values change, using the update method of SimplePIR, KPIR$^{\text{hash}}$ modifies the hash table's bins that have value updates. Therefore, the overhead is only proportional to the number of changed bins. Nevertheless, modifying both keys and values remains challenging. Moreover, KPIR$^{\text{kvs}}$ and KPIR$^{\text{index}}$ still require re-executing the whole setup. We leave efficient database updates as future work. On the other hand, extending our keyword PIR to authenticated variants may leverage VeriSimplePIR [15]. VeriSimplePIR produces a provable digest as the database commitment and verifies responses for consistency with the commitment. However, unlike index PIR, our keyword PIR framework introduces two challenges: (1) verifying the encoded database D' is correctly derived from the committed D; (2) ensuring consistency between D' used in PIR and the encoding output. Designing authenticated keyword PIR is also an interesting future work.

Acknowledgments

We would like to express our deepest gratitude for the invaluable help provided by our shepherd as well as all the reviewers for their constructive comments. This work is supported by the AXA research fund, the National Natural Science Foundation of China (Grant No. 62402272) and the Major Programs of the National Social Science Foundation of China (Grant No. 22&ZD147).

Ethics Considerations

This work provides effective solutions for keyword private information retrieval tasks, encouraging researchers to pay more attention to the privacy of database queries. The experiments in this paper are all based on public datasets and do not contain any personal or illegal information. We firmly believe that our research was done ethically.

Open Science

We fully support the principles of the Open Science Policy. We have open-sourced our implementation at https://github.com/alibaba-edu/mpc4j. Besides, our artifact is available at https://zenodo.org/records/14722434.

References

[1] Binary fuse filter. https://github.com/FastFilter/fastfilter_java.

[2] Chalamet. https://github.com/claucece/chalamet.

[3] Pgm-index. https://github.com/gvinciguerra/PGM-index.

[4] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *Proceedings of USNIEX OSDI*, 2021.

[5] Asra Ali, Tancrede Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–computation trade-offs in pir. In *Proceedings of USENIX Security*, 2021.

[6] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *Proceedings of IEEE S&P*, 2018.

[7] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proceedings of IEEE FOCS*, pages 787–796, 2010.

[8] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Near-optimal oblivious key-value stores for efficient psi, psu and volume-hiding multi-maps. In *Proceedings of UESNIX Security*, 2023.

[9] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Proceedings of CRYPTO*, 2012.

[10] Sofía Celi and Alex Davidson. Call me by my name: Simple, practical private information retrieval for keyword queries. In *Proceedings of ACM CCS*, 2024.

[11] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. 1997.

[12] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, 1998.

[13] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *Proceedings of EUROCRYPT*, pages 3–33, 2022.

[14] Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. *Proceedings on Privacy Enhancing Technologies*, 2023.

[15] Leo de Castro and Keewoo Lee. Verisimplepir: verifiability in simplepir at no online cost for honest servers. In *Proceedings of USENIX Security*, 2024.

[16] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive 2012/144*, 2012.

[17] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020.

[18] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In *Proceedings of CRYPTO*, 2021.

[19] Gaston H Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM (JACM)*, 28(2):289–304, 1981.

[20] Thomas Mueller Graf and Daniel Lemire. Binary fuse filters: Fast and smaller than xor filters. *Journal of Experimental Algorithmics*, 27(1):1–15, 2022.

[21] Meng Hao, Weiran Liu, Liqiang Peng, Hongwei Li, Cong Zhang, Hanxiao Chen, and Tianwei Zhang. Unbalanced circuit-psi from oblivious key-value retrieval. In *Proceedings of UESNIX Security*, 2024.

[22] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *Proceedings of SOSP*, pages 396–416, 2023.

[23] Alexandra Henzinger, Matthew M Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In *Proceedings of USENIX Security*, 2023.

[24] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of FOCS*, pages 364–373, 1997.

[25] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh. In *Proceedings of CRYPTO*, pages 284–314, 2023.

[26] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. Hintless single-server private information retrieval. In *Proceedings of CRYPTO*, 2024.

[27] Jian Liu, Jingyu Li, Di Wu, and Kui Ren. Pirana: Faster multi-query pir via constant-weight codes. In *Proceedings of IEEE S&P*, 2024.

[28] Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight pir: Single-round keyword pir via constant-weight equality operators. In *Proceedings of UESNIX Security*, 2022.

[29] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, pages 155–174, 2016.

[30] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *Proceedings of IEEE S&P*, 2022.

[31] Samir Jordan Menon and David J Wu. Ypir: High-throughput single-server pir with silent preprocessing. In *Proceedings of USENIX Security*, 2024.

[32] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. In *Proceedings of ACM CCS*, 2021.

[33] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. In *Proceedings of IEEE S&P*, 2023.

[34] Ofri Nevo, Ni Trieu, and Avishay Yanai. Simple, fast malicious multiparty private set intersection. In *Proceedings of ACM CCS*, 2021.

[35] Joseph O'Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, 24(9):574–578, 1981.

[36] Sarvar Patel, Joon Young Seo, and Kevin Yeo. Don't be dense: Efficient keyword pir for sparse databases. In *Proceedings of UESNIX Security*, 2023.

[37] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Psi from paxos: fast, malicious private set intersection. In *Proceedings of EUROCRYPT*, 2020.

[38] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *Proceedings of USENIX Security*, 2015.

[39] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Transactions on Privacy and Security*, 21(2):1–35, 2018.

[40] Martin Raab and Angelika Steger. "balls into bins"—a simple and tight analysis. In *Proceedings of RANDOM*, 1998.

[41] Srinivasan Raghuraman and Peter Rindal. Blazing fast psi from improved okvs and subfield vole. In *Proceedings of ACM CCS*, 2022.

[42] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):1–40, 2009.

[43] Cong Zhang, Yu Chen, Weiran Liu, Liqiang Peng, Meng Hao, Anyu Wang, and Xiaoyun Wang. Unbalanced private set union with reduced computation and communication. In *Proceedings of ACM CCS*, pages 1434–1447, 2024.

[44] Cong Zhang, Yu Chen, Weiran Liu, Min Zhang, and Dongdai Lin. Linear private set union from multi-query reverse private membership test. In *Proceedings of USENIX Security*, 2023.

[45] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. In *Proceedings of EUROCRYPT*, pages 395–425, 2023.

[46] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server pir with sublinear server computation. In *Proceedings of IEEE S&P*, 2024.

A Incompatibility of ChalametPIR and SimplePIR

In this section, we illustrate the incompatibility of ChalametPIR and SimplePIR.

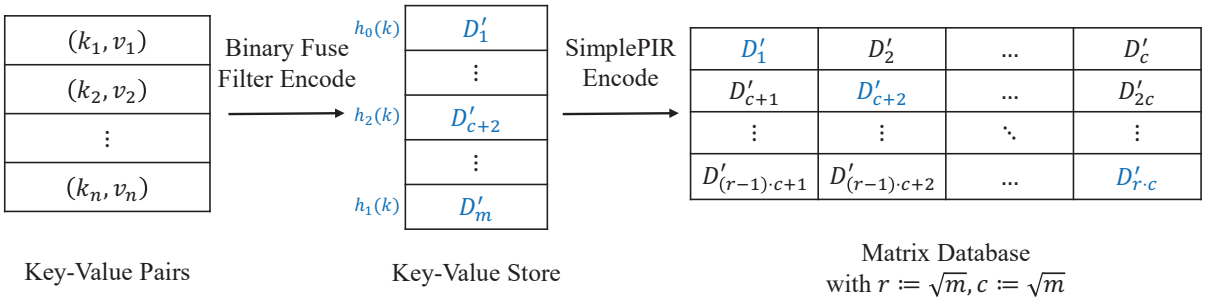


Figure 11: An illustrative example of failure retrieval using SimplePIR-instantiated ChalametPIR. The client retrieves the value corresponding to k. Following the decoding of Binary Fuse Filters, it requires the entries D'_1, D'_m, D'_{c+2} at the positions $h_0(k), h_1(k), h_2(k)$, respectively. However, these three entries are not in the same row with the matrix encoding of SimplePIR. With a single invocation of SimplePIR, it can not obtain the sum of D'_1, D'_m, D'_{c+2}.

ChalametPIR encodes the key-value database D of size n into a vector D' of size $m := O(n)$ via Binary Fuse Filters [20] with α functions $\{h_i : \{0,1\}^* \to [m]\}_{i\in[\alpha]}$. To retrieve the corresponding value of the query k, it constructs a size-m vector qu with all zeros except that $\mathsf{qu}[h_i(k)] := 1$ satisfying $v = \mathsf{qu}^\top \cdot D'$ if $(k,v) \in D$. ChalametPIR instantiates this functionality by utilizing a single invocation of a modified FrodoPIR [14]. Specifically, the client homomorphically encrypts the vector qu, rather than one-hot vectors as in original FrodoPIR, and sends the ciphertext ct of size $O(m)$ and the server evaluates the inner-product $\mathsf{qu}^\top \cdot D'$ in ciphertext.

However, this framework is incompatible with SimplePIR, where the encoded vector D' is transformed into a matrix of size $\sqrt{m} \times \sqrt{m}$. Specifically, to successfully retrieve the corresponding value of k, all $D'[h_i(k)]$ for $i \in [\alpha]$ should be in the same row, such that a FrodoPIR-analogical variant of SimplePIR with encrypted multi-one query is invoked. Unfortunately, due to the randomness of hash functions, it is challenging to construct Binary Fuse Filters satisfying the above requirements. These additional constraints will cause a high failure probability during the encoding process. Therefore, it requires theoretically analyzed or empirically validated Binary Fuse Filters satisfying the above requirements. In Figure 11, we show a failure retrieval when simply replacing FrodoPIR with SimplePIR in the original ChalametPIR framework.

B Details and Security of Row-KOPIR

In this section, we provide additional details and security proof of Row-KOPIR in Figure 2.

Concrete costs. We give concrete computational and communication costs with no hidden constants. In the one-time setup phase, Row-KOPIR requires (1) the server to execute $N \cdot r \cdot c$ multiplications in $\mathbb{Z}_q$ and (2) the client to download $N \cdot c$ elements in $\mathbb{Z}_q$. Then, for each query, Row-KOPIR requires (1) the client to execute $(N + c + 1) \cdot r$ multiplications in $\mathbb{Z}_q$, upload r elements in $\mathbb{Z}_q$, and download c elements in $\mathbb{Z}_q$, and (2) the server to execute $r \cdot c$ multiplications in $\mathbb{Z}_q$. Therefore, the one-time setup communication is $N \cdot c$, and the online communication is $c + r$ for each query. Therefore, to minimize the online communication, given a database containing n elements, the encoded matrix should have rows and columns of similar sizes, namely $\sqrt{n} \times \sqrt{n}$. This achieves $2\sqrt{n}$ communication cost, which is sublinear to the database size n. In our keyword PIR schemes, we will follow this principle to set the rows and columns of the encoded matrix to optimize the communication cost.

Reducing communication overhead. Recent works [22, 26] proposed solutions to reduce the total communication overhead of SimplePIR at the expense of increased computation. These optimizations can be directly employed in Row-KOPIR. We explain the high-level idea. Specifically, the communication is dominated by the one-time communication of $A \cdot D$ in the setup phase. As observed in these works, $A \cdot D$ is only used for the client in the decryption to compute $\mathbf{s} \cdot A \cdot D$. Therefore, to avoid sending $A \cdot D$, an alternative solution is to send encrypted $\mathbf{s}$ and perform vector-matrix multiplication on the server side by using using Ring-LWE-based homomorphic encryption such as BFV [9, 16]. This makes the communication cost independent of the total size of $D \cdot A$. However, this approach increases the online computation overhead. In our implementation, we do not apply this optimization to maintain high throughput.

Security. Below, we prove the security based on the hardness of the LWE problem.

Theorem 4. *Assume (N, r, q, χ) be LWE parameters, and $A \in \mathbb{Z}_q^{N\times r}$ be the random LWE matrix used in Row-KOPIR. If the (N, r, q, χ)-LWE problem is (T, ε)-hard, then the Row-KOPIR in Figure 2 is $(T - O(r), 2\varepsilon)$-secure.*

Proof. We reduce the security of Row-KOPIR to the hardness of LWE. For any $i \in [r]$, we define the distribution

$$Q := \{(A, qu) : (st, qu) \leftarrow \mathrm{Query}(i)\}.$$

Let $\mathbf{u}_i$ be the unit vector with a single 1 at index i. We define the following distributions:

- $\mathcal{D}_1 = \{(A, \mathbf{s}\cdot A + \mathbf{e}) : \mathbf{s} \leftarrow \mathbb{Z}_q^N, \mathbf{e} \leftarrow \chi^r\}$
- $\mathcal{D}_2 = \{(A, \gamma) : \gamma \leftarrow \mathbb{Z}_q^r\}$

Now, consider the simulator $\mathcal{S}$ that, given as input $(A, \mathbf{v}) \in \mathbb{Z}_q^{N\times r} \times \mathbb{Z}_q^r$, computes and outputs $(A, \mathbf{v} + \Delta \cdot \mathbf{u}_i)$. We have

- when $(A, \mathbf{v})$ is sampled from $\mathcal{D}_1$, the distribution of simulator's output is identical to Q.
- when $(A, \mathbf{v})$ is sampled from $\mathcal{D}_2$, the distribution of simulator's output is identical to $\mathcal{D}_2$.

Since the (N, r, q, χ)-LWE problem is (T, ε)-hard, any algorithm running in time T has advantage at most ε in distinguishing between $\mathcal{D}_1$ and $\mathcal{D}_2$. Note that the simulator $\mathcal{S}$ runs in time $O(r)$, we know that any algorithm distinguishing between Q and $\mathcal{D}_2$ in time at most $T - O(r)$ can have success probability at most ε. By the triangle inequality, we have

$$
\begin{aligned}
&|\Pr[\mathcal{A}(\mathsf{qu}) = 1 \,|\, (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(i)] \\
&- \Pr[\mathcal{A}(\mathsf{qu}) = 1 \,|\, (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(j)]| \\
\leq &|\Pr[\mathcal{A}(\mathsf{qu}) = 1 \,|\, (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(i)] - \Pr[\mathcal{A}(\gamma) = 1 \,|\, \gamma \leftarrow \mathbb{Z}_q^r]| \\
&+ |\Pr[\mathcal{A}(\gamma) = 1 \,|\, \gamma \leftarrow \mathbb{Z}_q^r] - \Pr[\mathcal{A}(\mathsf{qu}) = 1 \,|\, (\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Query}(j)]| \\
\leq &2\varepsilon.
\end{aligned}
$$

□

C Instantiation of α-sparse KVS

In this section, we instantiate α-sparse KVS using Binary Fuse Filters (BFF) [20] with $\alpha := 3$. The Encode and Decode algorithms are presented in Algorithms 1 and 2, respectively. The correctness of inclusion follows immediately given the construction of Binary Fuse Filters. Below, we analyze the correctness of non-inclusion. Given $(D, H) \leftarrow \mathrm{Encode}(L)$ with $L := (K, V)$ and a key $k \notin K$, the decode output is $k^h \| v \leftarrow \mathrm{Decode}(H, k, D[H(k)])$. We consider two possible cases. (1) There exists some $k^* \in K$ such that $H(k^*) = H(k)$, namely $\{h_1(k^*), h_2(k^*), h_3(k^*)\} = \{h_1(k), h_2(k), h_3(k)\}$. According to the collision-resistant property of $\mathsf{hash} : \{0,1\}^* \to \{0,1\}^{2\lambda}$, the probability of $\mathsf{hash}(k^*) = \mathsf{hash}(k)$ is $2^{-\lambda}$. (2) For any $k^* \in K$, it holds $H(k^*) \neq H(k)$, namely $\{h_1(k^*), h_2(k^*), h_3(k^*)\} \neq \{h_1(k), h_2(k), h_3(k)\}$. The decoding procedure inputs three independent and uniform entries of D and outputs their sum $k^h \| v$, where k^h is a uniform element in $\{0,1\}^{2\lambda}$. The probability of $k^h = \mathsf{hash}(k)$ is at most $2^{-\lambda}$. Therefore, for any $k \notin K$, the decoding output is $\perp$ except with the probability $\mathsf{negl}(\lambda)$.

Algorithm 1 Encode of Binary Fused Filters [20]

Parameter: A collision-resistant hash function $\mathsf{hash} : \{0,1\}^* \to \{0,1\}^{2\lambda}$. Range of encoded vector $\mathcal{D}$.
Input: A set of key-value pairs $L := \{(k_i, v_i)\}_{i\in[n]} \in (\mathcal{K} \times \mathcal{V})^n$ with distinct keys.

repeat
  Sample hash functions $h_1, h_2, h_3 : \mathcal{K} \to [m]$.
  Initialize empty vector C of size m.
  For $i \in [n]$ and $j \in [3]$, push k_i to $C[h_j(k_i)]$.
  Initialize empty stack Q.
  For $i \in [m]$, push i into Q if $|C[i]| = 1$.
  Initialize empty stack P.
  while $|Q| > 0$ **do**
    Pop i from Q.
    if $|C[i]| = 1$ **then**
      Set $k := C[i]$ and push (k, i) to P.
      For $j \in [3]$, remove k from $C[h_j(k)]$.
      For $j \in [3]$, push $h_j(k)$ into Q if $|C[h_j(k)]| = 1$.
    end if
  end while
until $|P| = n$
Initialize empty vector D of size m.
while $|P| > 0$ **do**
  Pop (k, i) from P and set $H := \{h_j(k)\}_{j\in[3]} \setminus \{i\}$.
  For $x \in H$, sample $D[x] \leftarrow \mathcal{D}$ if $D[x]$ is empty.
  Set $D[i] := \mathsf{hash}(k) \| v - \sum_{x\in H} D[x]$, where $(k, v) \in L$.
end while
For $i \in [m]$, sample $D[i] \leftarrow \mathcal{D}$ if $D[i]$ is empty.
return D and h_1, h_2, h_3.

Algorithm 2 Decode of Binary Fused Filters [20]

Parameter: A collision-resistant hash function $\mathsf{hash} : \{0,1\}^* \to \{0,1\}^{2\lambda}$. Range of encoded vector $\mathcal{D}$.
Input: A set of hash functions h_1, h_2, h_3, a key k, and entries $D[h_1(k)], D[h_2(k)], D[h_3(k)]$.

Set $k^h \| v := D[h_1(k)] + D[h_2(k)] + D[h_3(k)] \in \mathcal{D}$.
return v if $k^h = \mathsf{hash}(k)$, or $\perp$ otherwise.

D Additional Experiments

In this section, we present additional experimental settings and results for completeness.

In Tables 3, 4, and 5, we provide detailed parameters of our three constructions, including the size of the encoded matrix database and the expansion rate (i.e., the rate of the encoded database size and the original database size) for different database sizes and entry lengths. For clarity, we show the number of rows and columns of the encoded matrix database. These results are in line with theoretical analysis in our constructions.

We evaluate our constructions on larger LWE parameters

with $N := 1408$ as recommended by HintlessPIR [26]. We provide the experimental results in Table 6, showing the runtime and communication costs of our three constructions and comparisons with ChalametPIR. We observe that larger N slightly increases the communication and computation overhead, but the additional overhead is marginal. Compared to ChalametPIR, we also maintain a significant advantage on communication costs with a comparable and even better computational costs, depending on database size and entry length.

Table 3: Parameters of the matrix database in $\mathsf{KPIR}^{\mathsf{kvs}}$

DB Size	Entry Len.	# Row	# Column	Exp. Rate
2^{18}	32	3,445	3,520	1.156
	64	4,664	4,680	1.156
	128	6,315	6,528	1.156
	256	8,915	8,976	1.156
2^{20}	32	6,859	6,880	1.125
	64	9,216	9,216	1.125
	128	12,550	12,784	1.125
	256	17,607	17,688	1.125
2^{22}	32	13,717	13,760	1.125
	64	18,432	18,432	1.125
	128	25,234	25,432	1.125
	256	35,214	35,376	1.125

Table 4: Parameters of the matrix database in $\mathsf{KPIR}^{\mathsf{hash}}$

DB Size	Entry Len.	# Row	# Column	Exp. Rate
2^{18}	32	3,239	5,600	1.730
	64	4,345	6,408	1.475
	128	5,971	9,673	1.620
	256	8,320	15,576	1.873
2^{20}	32	6,477	9,240	1.427
	64	8,689	10,152	1.168
	128	11,942	19,176	1.606
	256	16,639	23,760	1.428
2^{22}	32	12,953	16,040	1.238
	64	17,378	20,304	1.170
	128	23,884	31,960	1.338
	256	33,277	38,016	1.142

Table 5: Parameters of the matrix database in $\mathsf{KPIR}^{\mathsf{index}}$

DB Size	Entry Len.	# Row	# Column	Exp. Rate
2^{18}	32	3,239	3,680	1.137
	64	4,345	5,184	1.193
	128	5,971	7,480	1.253
	256	8,320	11,352	1.365
2^{20}	32	6,477	6,920	1.069
	64	8,689	9,504	1.094
	128	11,942	13,464	1.127
	256	16,639	19,800	1.190
2^{22}	32	12,953	13,400	1.035
	64	17,378	18,216	1.048
	128	23,884	25,432	1.065
	256	33,277	36,432	1.095

Table 6: Communication and runtime comparison of our protocols with the state-of-the-art ChalametPIR [10] with LWE secret key dimension $N := 1408$. The best result is marked in green, and the second-best result is marked in blue.

Parameter		Protocol	Setup Phase		Online Phase	
DB Size	**Entry Size**		**Comm (MB)**	**Time (s)**	**Comm (MB)**	**Time (ms)**
2^{18}	32B	ChalametPIR [10]	0.27	47.78	1.16	39.41
		KPIR^{kvs}	18.91	8.07	0.08	26.77
		$\text{KPIR}^{\text{hash}}$	24.49	11.04	0.03	16.92
		$\text{KPIR}^{\text{index}}$	19.85	16.14	0.04	25.86
	64B	ChalametPIR [10]	0.49	56.61	1.16	44.15
		KPIR^{kvs}	25.14	14.76	0.11	40.66
		$\text{KPIR}^{\text{hash}}$	34.81	20.37	0.04	29.37
		$\text{KPIR}^{\text{index}}$	27.92	14.83	0.04	25.05
	128B	ChalametPIR [10]	0.92	80.50	1.16	53.61
		KPIR^{kvs}	35.06	28.41	0.15	75.60
		$\text{KPIR}^{\text{hash}}$	50.40	39.11	0.06	55.30
		$\text{KPIR}^{\text{index}}$	40.26	30.86	0.05	52.83
	256B	ChalametPIR [10]	1.79	126.06	1.16	81.31
		KPIR^{kvs}	48.21	55.64	0.21	138.17
		$\text{KPIR}^{\text{hash}}$	73.73	79.50	0.08	105.85
		$\text{KPIR}^{\text{index}}$	61.05	66.70	0.08	104.29
2^{20}	32B	ChalametPIR [10]	0.27	188.58	4.50	116.52
		KPIR^{kvs}	36.95	33.71	0.16	93.80
		$\text{KPIR}^{\text{hash}}$	44.04	36.88	0.06	54.08
		$\text{KPIR}^{\text{index}}$	37.49	34.53	0.05	52.55
	64B	ChalametPIR [10]	0.49	217.76	4.50	117.52
		KPIR^{kvs}	49.50	60.79	0.21	156.94
		$\text{KPIR}^{\text{hash}}$	65.36	73.48	0.08	105.80
		$\text{KPIR}^{\text{index}}$	51.36	57.90	0.07	86.60
	128B	ChalametPIR [10]	0.92	317.15	4.50	188.37
		KPIR^{kvs}	68.66	115.71	0.29	259.25
		$\text{KPIR}^{\text{hash}}$	92.04	145.40	0.11	166.61
		$\text{KPIR}^{\text{index}}$	72.63	115.89	0.10	142.56
	256B	ChalametPIR [10]	1.79	473.13	4.50	256.43
		KPIR^{kvs}	95.00	227.76	0.40	468.22
		$\text{KPIR}^{\text{hash}}$	138.96	311.68	0.16	308.56
		$\text{KPIR}^{\text{index}}$	106.67	240.41	0.14	257.93
2^{22}	32B	ChalametPIR [10]	0.27	741.23	18.00	306.47
		KPIR^{kvs}	73.91	134.84	0.31	310.28
		$\text{KPIR}^{\text{hash}}$	84.43	150.74	0.11	193.94
		$\text{KPIR}^{\text{index}}$	73.24	129.69	0.10	159.59
	64B	ChalametPIR [10]	0.49	876.66	18.00	371.49
		KPIR^{kvs}	99.00	236.67	0.42	515.49
		$\text{KPIR}^{\text{hash}}$	120.27	274.83	0.15	285.28
		$\text{KPIR}^{\text{index}}$	99.10	234.00	0.14	251.92
	128B	ChalametPIR [10]	0.92	1226.85	18.00	587.21
		KPIR^{kvs}	136.60	444.62	0.58	972.58
		$\text{KPIR}^{\text{hash}}$	167.28	547.84	0.21	541.51
		$\text{KPIR}^{\text{index}}$	137.86	443.20	0.19	438.43
	256B	ChalametPIR [10]	1.79	1875.72	18.00	939.90
		KPIR^{kvs}	190.01	935.83	0.81	1913.79
		$\text{KPIR}^{\text{hash}}$	252.40	1143.17	0.31	1101.73
		$\text{KPIR}^{\text{index}}$	196.94	890.47	0.27	827.82