Mk-PIR: Multi-Keyword Private Information Retrieval

Shengnan Zhao[1], Junyu Lu[2], Yuchen Huang[1], Dongdong Miao[2], and Chuan Zhao[2]

[1] Quan Cheng Laboratory, Jinan, China
[2] University of Jinan, Jinan, China
`zsn.sdu@gmail.com`, `ljyflyje3@foxmail.com`, `cn_yuchenhuang@163.com`, `m_dong30@163.com`, `ise_zhaoc@ujn.edu.cn`

Abstract. Private information retrieval (PIR) enables a client to fetch a record from databases held by untrusted servers while hiding the access pattern (index or keyword) from the servers. In practical settings, however, data objects (*e.g.,* articles, videos) are commonly tagged with multiple identifiers, which can be structured as {index, value, keywords}. Current PIR schemes are constrained to retrieving records based on a single index *or* a single keyword, and cannot efficiently handle conjunctive queries requiring multiple keywords. To address this limitation, we propose Mk-PIR, a PIR scheme that enables a client to retrieve records that match *all* specified keywords simultaneously. We propose two distinct constructions: $\mathsf{MkPIR}^{\mathbf{I}}$, an inverted-index-based solution built upon our Oblivious Set Intersection (OSI) primitive, which enables private intersection computation on the server side without revealing client queries; and $\mathsf{MkPIR}^{\mathbf{F}}$, a forward-index-based solution utilizing our Private Subset Determination (PSD), which privately outputs matching indices by verifying subset relationships. Two constructions adapt to diverse database configurations where keywords are not required to be the primary key. Experimental results show that the average time to determine whether an index satisfies multiple keywords ranges from 0.5 to 332 ms, demonstrating that Mk-PIR achieves flexible query capabilities with modest performance overhead.

Keywords: Private information retrieval · Privacy protection.

1 Introduction

Private Information Retrieval (PIR) [10] enables a client to retrieve records from a server's database while the server is unaware of which records are queried. Traditional PIR schemes can be categorized into single-server [11], [37], [30], [33], [24] and multi-server [5], [4], [13], [23], [31], [14], [20] architectures, with the latter providing stronger privacy guarantees through distributed query processing across non-colluding servers. However, given the challenges in ensuring non-collusion among multiple servers in the real world, computationally secure single-server PIR schemes have emerged as more practical solutions. This work advances the field by introducing a novel multi-keyword PIR designed specifically for the single-server setting, addressing critical limitations in existing approaches.

According to the client query structure, current PIR research has developed along two main lines of investigation: Index PIR and Keyword PIR. The index PIR schemes [3], [32], [12], [22], [38] operate under the assumption that the client knows the exact location (usually the physical address of the requested value). Researchers focus their optimization efforts on reducing communication and computation overhead through techniques such as homomorphic encryption (HE) and batch query processing. Keyword PIR approaches [2], [27], [7], [21], [26] consider more realistic data structures where keywords follow natural sparsity patterns. That is, the data has a unique identifier and the format can be described as $\{Keyword, Value\}$, which is also called a *key-value* store.

However, both classical Index PIR and Keyword PIR share a fundamental constraint: neither inherently supports multi-keyword queries. This limitation severely restricts practical deployment in document retrieval systems requiring specific keyword combinations, as well as applications like targeted legal searches and sensor data filtering – all demanding private retrieval of records satisfying multiple identifiers simultaneously. For confidential databases such as medical records, clients must not obtain extraneous information during retrieval. While Symmetrical PIR (SPIR) schemes [16] [17] [25] [36] ensure server privacy during response phases, which means that clients only obtain target information, they exhibit fundamental limitations at the input phase: index acquisition on the client side still requires strong assumptions through prior knowledge (such as mapping tables). Most significantly, these SPIR schemes cannot support multi-keyword queries, an essential capability for real-world applications. We therefore address the following core challenge:

How can clients privately retrieve records matching multiple identifiers without pre-storing prior knowledge of the database and without compromising server privacy?

Given structured data as $\{Index, Value, Keywords\}$, this paper introduces Mk-PIR, a novel PIR scheme that allows clients to privately retrieve records matching *all* specified keywords simultaneously (Figure 1). In particular, Mk-PIR does not require keywords as primary keys, enabling the retrieval of indices that match multiple non-primary-key keywords in the database. Furthermore, Mk-PIR delivers *on-tap* availability to clients without any pre-downloads and hint storage requirements while maintaining continuous update compatibility at the server.

Mk-PIR operates through two privacy-preserving phases: a novel “Keywords-to-Indices" transformation and a standard “Index-to-Value" retrieval step. The core design focuses on implementing a functional Keyword PIR for the first phase. That is, the client needs to privately retrieve indices associated with specific keywords, representing a specialized execution of classical Keyword PIR with two key distinguishing characteristics. First, the algorithm returns a set of indices rather than the actual database records. Second, it must handle a many-to-many relationship between keywords and indices, unlike classical Keyword PIR systems

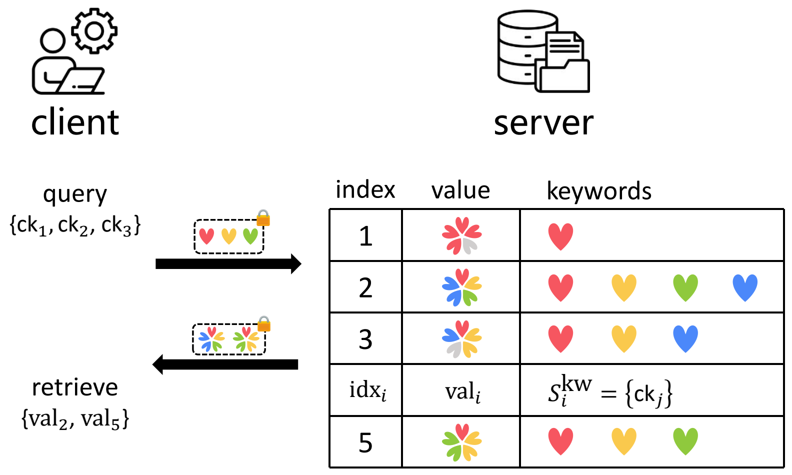


Fig. 1: An example of Mk-PIR

that maintain a strict one-to-one mapping (a prior knowledge of the index or keyword serving as the primary key for retrieval). In our proposed Mk-PIR, one index may be associated with various keywords (many-to-one), and meanwhile, a single keyword may correspond to multiple indices (one-to-many).

1.1 Our Contributions

For the case where keywords are non-unique attributes rather than primary keys, we introduce two multi-keyword PIR constructions. In summary, we make the following contributions:

- We propose Mk-PIR, a novel multi-keyword private information retrieval that enables clients to privately retrieve records matching *all* specified keywords simultaneously. We offer two distinct constructions, $\mathsf{MkPIR}^{\mathbf{I}}$ and $\mathsf{MkPIR}^{\mathbf{F}}$, specifically designed to handle database configurations where indices exhibit many-to-many relationships with keywords. Two constructions provide comprehensive coverage for various data organization scenarios while maintaining query privacy.
- In $\mathsf{MkPIR}^{\mathbf{I}}$, an efficient inverted-index-based construction, we introduce Oblivious Set Intersection (OSI), which enables clients to privately retrieve only those indices matching all queried keywords while maintaining server obliviousness. Furthermore, the performance of $\mathsf{MkPIR}^{\mathbf{I}}$ is further optimized through batching techniques that significantly reduce computational complexity.
- In $\mathsf{MkPIR}^{\mathbf{F}}$, we design Private Subset Determination (PSD), which performs encrypted comparisons between client queries and server-side keyword sets. In addition, we develop efficient binary encoding combined with SIMD ciphertext packing, achieving both privacy and efficiency in keyword-rich environments.

1.2 Related Works

The Keyword PIR can be regarded as a form of Index PIR operating on sparse index structures. We present the most related works about Index PIR and Keyword PIR, which correspond to two distinct data storage modes in a database.

Index PIR. In this framework, data is stored as $(\mathit{Index}, \mathit{Data})$ pairs, where clients possess prior knowledge of target indices before querying, thus termed Index PIR. Chor *et al.* [8] significantly reduced the communication complexity of PIR, providing a crucial insight for the subsequent research on CPIR. Melchor *et al.* [28] leveraged Ring Learning with Errors (RLWE) homomorphic encryption to achieve sublinear communication costs, avoiding the need to transfer the entire database while simultaneously reducing server-side computation overhead. Angel *et al.* [3] proposed a novel approach where clients transmit only encrypted indices rather than full query vectors, enabling servers to reconstruct the encrypted query vectors locally, thereby improving communication efficiency.

Recent advances have focused on optimizing homomorphic operations in Index PIR. Mughees *et al.* [32] developed a novel approach combining BFV [15] ciphertexts for initial database multiplication and RGSW [18] ciphertexts for subsequent operations, effectively controlling noise growth during homomorphic computation. This dual-ciphertext technique was further refined by Menon *et al.* [29] and de Castro *et al.* [6], who improved format conversions and computational efficiency while building upon the core methodology established in [32]. These innovations collectively address the critical challenge of computational overhead in practical Index PIR implementations.

Keyword PIR. First introduced by Chor *et al.* [9], the scheme enables clients to privately retrieve data associated with a specific keyword without revealing query intent. Ali *et al.* [2] employed cuckoo hashing to map sparse keywords to dense indices, while Mahdavi *et al.* [27] avoided false positives using constant weight equality operators whose depth depends only on the keyword domain size. Patel *et al.* [34] further optimized costs by database partitioning and linear combination encoding, though with some performance trade-offs.

The Keyword PIR has seen significant theoretical and practical developments, including ChalametPIR by Celi and Davidson [7], which provides a generic framework for converting Index PIR over flat arrays to Keyword PIR. Very recently, Hao *et al.* [21] developed three practical schemes with costs scaling only with database size and PIR invocations, demonstrating superior performance to ChalametPIR in real-world scenarios. These innovations collectively address the core challenges of communication efficiency and computational overhead.

Challenge. In practical applications, multiple keywords may reference the same data item (*e.g.,* keywords in an academic paper). Classical Keyword PIR imposes a strict constraint that each keyword must uniquely identify data (one-to-one mapping). However, data are commonly tagged with multiple keywords such as $\{\mathit{Index}, \mathit{Value}, \mathit{Keywords}\}$. Our work addresses the more realistic and challenging case of retrieving records matching *all* queried keywords simultaneously. Mk-PIR handles true multi-keyword conjunctions through two sequential executions. This carefully designed separation of concerns provides both efficiency and privacy guarantees. However, we emphasize that the existence of a communication-efficient single-interaction Mk-PIR construction remains an open question.

The rest of the paper is organized as follows. In Section 2, we describe the essential preliminaries used in Mk-PIR. In Section 3, we briefly describe the method for constructing Mk-PIR, followed by two detailed constructions in Section 4 and Section 5, respectively. Extensive experiments and performance comparisons are provided in Section 6. Finally, Section 7 concludes the study.

2 Preliminaries

2.1 Notations

Table 1 summarizes the notations used in the rest of the paper.

Table 1: notations

Notation	Description
m	Number of client's query keywords
n	Number of server database indices
t	Number of server database keywords
p_i	The i-th prime number
(pk, sk)	Key pair of the client
D	Server database
$\mathsf{idx}_i, i \in [n]$	The i-th index in D
$\mathsf{val}_i, i \in [n]$	The value corresponding to idx_i in D
$S_i^{\mathsf{kw}}, i \in [n]$	The keyword set to $\{\mathsf{idx}_i, \mathsf{val}_i\}$ in D
$S_{\mathsf{ck}} = \{\mathsf{ck}_i\}_{i\in[m]}$	The keyword set queried by the client
λ	Statistical security parameter
κ	Computational security parameter
$\mathcal{K}$	Domain of keywords defined by the server
$\deg(P)$	Degree of the polynomial

2.2 Gödel Coding

Gödel number [19] is used in proving the incompleteness theorem, which establishes a mapping relationship from an integer sequence to a number. By the fundamental theorem of arithmetic, each Gödel number corresponds to a unique sequence of symbols, which ensures the determinacy of encoding and decoding.

Given n different integers $[x_1, x_2, \ldots, x_n]$ and n different primes $[p_1, p_2, \ldots, p_n]$, we can encode these integers by the calculation:

$$\mathsf{G.coding}(x_1, x_2, \ldots, x_n) := \prod_{i=1}^{n} p_i, \tag{1}$$

where $\mathsf{G.coding}$ denotes Gödel coding algorithm.

2.3 Constant-weight Code

A constant-weight code (CW) is a type of error-correcting code where every codeword has the same Hamming weight (the same number of non-zero symbols). In a binary constant-weight code, every codeword is a binary string with a fixed number of ones (m), exemplified by the one-hot code ($m = 1$) and balanced codes where codewords contain an equal number of ones and zeros.

Given codewords $(x, y) \in CW(m, k)$ encoded by the binary constant-weight codes of length m and Hamming weight of k, the plain equality operator [27] can be used as follows:

$$f_{PCW}(x, y) = \prod_{y[i]=1} x[i], \tag{2}$$

where the operator is oblivious to the first operand but depends on the second.

2.4 Fully Homomorphic Encryption (FHE)

Fully homomorphic encryption enables ciphertext computations but faces noise accumulation challenges during operations. Current solutions employ bootstrapping or leveled designs to control noise growth and maintain decryption capability. In this paper, we use the FHE scheme based on the Ring Learning With Error (RLWE) problem. The scheme is defined by the following three algorithms:

- $\mathsf{FHE.KeyGen}(1^\lambda)$: Sample $a \leftarrow R_q$, $s \leftarrow \{-1, 0, 1\}^n$, and error $e \leftarrow \chi$; output public key $pk = (pk_0, pk_1) = (a, -(as + e))$ and private key $sk = s$.
- $\mathsf{FHE.Enc}(m, pk)$: Sample $u, e_0, e_1 \leftarrow \chi$; compute $\Delta = \lfloor q/t \rfloor$ and output $c = (c_0, c_1) = (pk_0 \cdot u + e_0, pk_1 \cdot u + e_1 + \Delta m)$.
- $\mathsf{FHE.Dec}(c, sk)$: Compute $m' = (s \cdot c_0 + c_1) \mod q$; output $m = \lfloor (t \cdot m')/q \rceil \mod t$.

FHE enables homomorphic addition and multiplication operations between ciphertexts, as well as between ciphertexts and plaintexts. These operations inevitably increase the noise in the ciphertext, and decryption will fail if the noise exceeds the budget. Therefore, it is very important to choose sufficient parameters and control the growth of noise in homomorphic computations.

SIMD (Single Instruction Multiple Data) is a parallel computing technology, which aims to process multiple data elements through one instruction. RLWE-based FHE supports the ciphertext is decomposed into many slots, and then parallel computing if the polynomial modulus degree $\deg(P)$ and the plaintext modulus m_p satisfy: $m_p \equiv 1 \ (mod\ 2\deg(P))$.

2.5 Hash-to-bin Techniques

Hash-to-bin techniques offer an effective approach for reducing computational overhead in batch query processing [3], [35]. As illustrated in Figure 2, our construction employs these techniques to efficiently map and distribute query items across bins, thereby optimizing the overall performance.

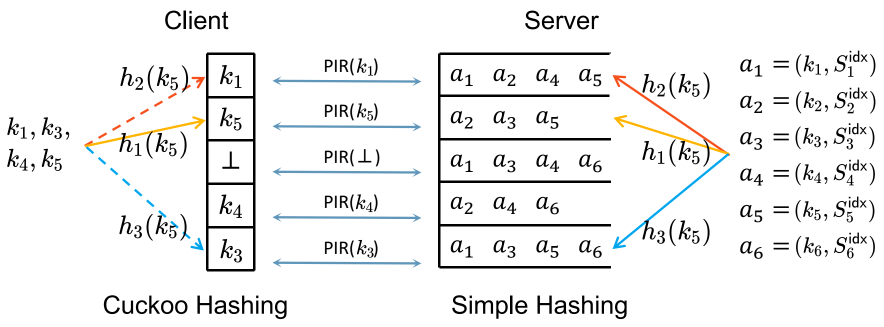


Fig. 2: Hash-to-bin in PIR

Simple Hashing. The server randomly samples b independent hash functions $H_b = \{h_1, h_2, \ldots, h_b\}$ and utilizes them to map t items into $c = (1+\varepsilon)m$ bins where $\varepsilon > 0$ is a constant. Specifically, the server places one item (k_i, S_i^{idx}) into bins $h_1(k_i), h_2(k_i), \ldots, h_b(k_i)$. A single bin may contain multiple items. Eventually, there are $b \times t$ entries across all bins.

Cuckoo Hashing. For a query keyword k, the client identifies its candidate storage positions within bins $h_1(k), h_2(k), \cdots, h_b(k)$. The core principle of Cuckoo Hashing is that keywords are dynamically inserted through iterative displacement. If all candidate bins of the keyword k are occupied, it displaces the existing keyword to its alternative bins, repeating until all keywords stabilize or a reboot occurs. This creates a static structure where each keyword resides in exactly one of its b candidate bins.

Based on hash-to-bin techniques, PIR operates exclusively on the specific bin containing the client's keyword. This synchronized bin-level access enables efficient batch query processing and significantly reduces computational overhead.

3 Technical Overview

Mk-PIR conceptually executes a *functional* Keyword PIR followed by an Index PIR, operating through two privacy-preserving phases: a "Keywords-to-Indices" transformation and an "Index-to-Value" retrieval phase. The core innovation lies in the first phase, where the client privately obtains all indices matching query keywords through our OSI and PSD algorithms. These indices then enable standard Index PIR retrieval of the corresponding message contents.

The OSI and PSD address the core challenge of Mk-PIR by combining efficient index structures with specialized batch processing and retrieval algorithms, while maintaining the required privacy guarantees throughout the query process. Given that existing Keyword PIR schemes with the form of $\{(k_i||\mathsf{val}_i)\}_{i\in[n]}$ cannot support multi-keyword queries since keywords stored in the database are the primary key, we introduce new algorithms. Basically, the client has a set of keywords $S_{\mathsf{ck}} = \{\mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m\} \subseteq \{0,1\}^*$, and the database server has $D = \{(\mathsf{idx}_i, \mathsf{val}_i, S_i^{\mathsf{kw}})\}_{i\in[n]}$, where $S_i^{\mathsf{kw}} \subseteq \mathcal{K}$, $\mathcal{K}$ denotes the keyword space defined by the database server. Notably, the client does not necessarily have a priori knowledge of the keyword set. After the OSI or PSD, the client obtains:

$$S_{\mathsf{idx}} = \bigcup_{i=1}^{n} \big\{\mathsf{idx}_i \mid S_{\mathsf{ck}} \subseteq S_i^{\mathsf{kw}}\big\}.$$

The OSI is integrated into $\mathsf{MkPIR}^{\mathbf{I}}$ to handle scenarios where indices significantly outnumber keywords (*i.e.,* one keyword corresponds to multiple indices). We design a new mapping structure according to the server database $D = \{(\mathsf{idx}_i, \mathsf{val}_i, S_i^{\mathsf{kw}})\}_{i\in[n]}$. The server first prepares two subdatabases $D_1 = \{(\mathsf{idx}_i, S_i^{\mathsf{kw}})\}_{i\in[n]}$ and $D_2 = \{(\mathsf{idx}_i, \mathsf{val}_i)\}_{i\in[n]}$. Then, the server generates an inverted-index table $D_1' = \{\mathsf{kw}_j, S_j^{\mathsf{idx}}\}_{j\in[|\mathcal{K}|]}$, where $S_j^{\mathsf{idx}} = \bigcup \big\{\mathsf{idx}_j | \mathsf{kw}_j \in S_i^{\mathsf{kw}}\big\}_{i\in[n], j\in[|\mathcal{K}|]}$. This inverted-index structure offers query efficiency advantages by scaling with the keyword count rather than the index count. For multi-keyword queries, OSI employs homomorphic operations to perform secure set computations, ultimately yielding the intersection:

$$S_{\mathsf{idx}} = \bigcap_{j=1}^{|\mathcal{K}|} \{S_j^{\mathsf{idx}} \mid \mathsf{kw}_j = \mathsf{ck}_i, i \in [m]\}.$$

PSD-based $\mathsf{MkPIR}^{\mathbf{F}}$ is specifically designed for scenarios where the total number of keywords is much larger than that of the indices (*i.e.,* one index corresponds to multiple keywords). Building upon the server's forward-indexed database *i.e.,* $D_1 = \{(\mathsf{idx}_i, S_i^{\mathsf{kw}})\}$ with constant-weight code encoding, we develop a privacy-preserving subset determination method. The method PSD enables encrypted evaluation of whether the client's query set $S\mathsf{ck}$ is contained within S_i^{kw} through the function $f_{PSD}(S_{\mathsf{ck}}, S_i^{\mathsf{kw}})$, which outputs 1 if $S_{\mathsf{ck}} \subseteq S_i^{\mathsf{kw}}$ and 0 otherwise, while ensuring the server learns nothing about the query or matching results. The client ultimately receives the matching index set:

$$S_{\mathsf{idx}} = \bigcup_{i=1}^{n} \big\{\mathsf{idx}_i \mid f_{PSD}(S_{\mathsf{ck}}, S_i^{\mathsf{kw}}) = 1\big\}.$$

To enhance efficiency, we utilize hashing-to-bin techniques to structure both the database and index table, combined with optimizations including Batch-PIR, SIMD, and other tricks. These approaches enable effective retrieval of the index set for targeted keywords, with several techniques being equally applicable to the subsequent "Index-to-Value" retrieval phase.

4 Inverted-Index-based Mk-PIR

In this section, we present $\mathsf{MkPIR}^{\mathbf{I}}$, an inverted-index-based construction that builds upon our proposed Oblivious Set Intersection (OSI) algorithm. We also enhance $\mathsf{MkPIR}^{\mathbf{I}}$ by integrating batching techniques to improve its overall computational complexity.

4.1 High-Level Idea of $\mathsf{MkPIR}^{\mathbf{I}}$ and OSI

The scheme $\mathsf{MkPIR}^{\mathbf{I}}$ executes a functional Keyword PIR (*i.e.,* OSI-based PIR) followed by an Index PIR in sequence. Through the OSI, the client privately obtains the complete set of indices matching all query keywords.

The OSI algorithm essentially performs two privacy-preserving operations on the inverted-index table: index sets corresponding to query keywords and the intersection among these sets. While the first operation may appear relatively straightforward and could be implemented using conventional PIR, the key challenge lies in the storage format of the inverted-index table. Standard encryption schemes prevent subsequent intersection computations on index sets. To address this, we employ Gödel coding to represent index sets, enabling intersection operations to be performed directly on ciphertexts of PIR. Furthermore, we incorporate the optimization technique (hash-to-bin) to enhance the overall efficiency.

Apart from the OSI algorithm, $\mathsf{MkPIR}^{\mathbf{I}}$ requires additional preprocessing steps and optimization techniques. We present the principal components from both server-side and client-side perspectives:

Server Side There are three key technical components: inverted-index construction, bucket-based batch optimization, and a specialized retrieval algorithm.

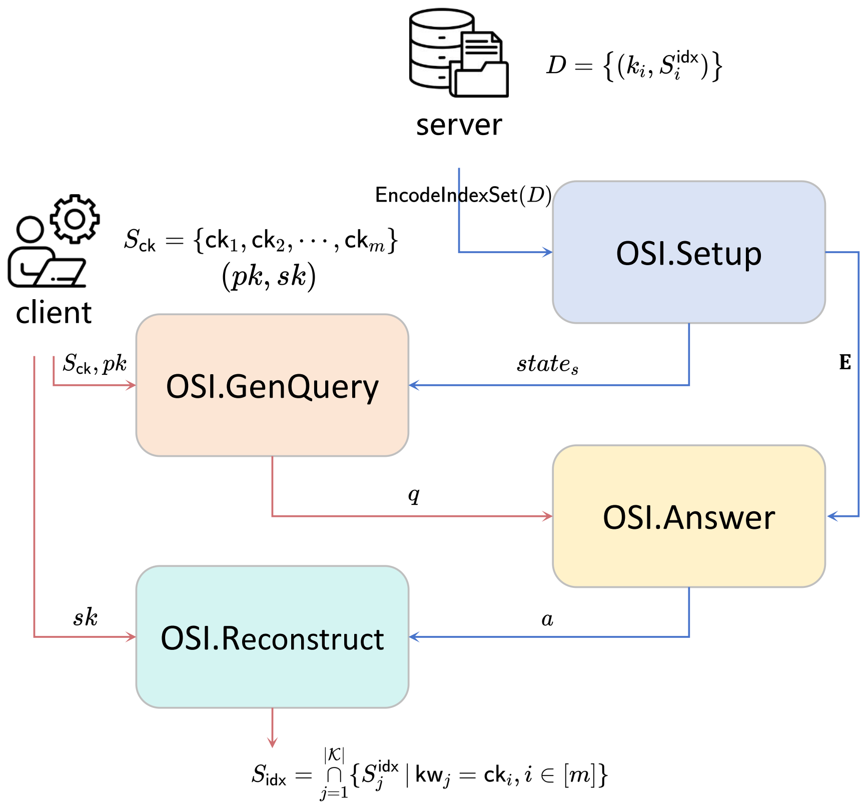
The server must first construct an inverted-index table from the database, which serves as a necessary preprocessing step for keyword-to-index searches. Since the query of the client may contain multiple keywords, the system employs hashing-to-bin and batch techniques to organize both the database and the index table. This approach enables efficient retrieval of the corresponding index sets for the targeted keywords.

Client Side The implementation centers around the bucket-based batch technique for an efficient multi-keyword query.

The batch technique operates throughout two phases: in the “Keywords-to-Indices” phase, clients employ cuckoo hashing to reorganize query keywords, complementing the server’s hashing-to-bin organization of the inverted-index table, enabling efficient OSI across all bins. The design naturally extends to the “Index-to-Value" phase, where batch Index PIR techniques are applied to optimize the retrieval of multiple indices obtained from the first phase.

4.2 Detailed OSI Algorithms

We first present the encoding algorithm, which operates exclusively on the server side as a preprocessing step for the subsequent OSI execution between server and client. Following this, we provide a detailed description of the four core algorithms constituting the OSI scheme in Figure 3: Setup, GenQuery, Answer, and Reconstruct.



server
$D = \{(k_i, S_i^{\mathsf{idx}})\}$
$\mathsf{EncodeIndexSet}(D)$
OSI.Setup
client
$S_{\mathsf{ck}} = \{\mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m\}$
(pk, sk)
S_{ck}, pk
OSI.GenQuery
$state_s$
$\mathbf{E}$
q
OSI.Answer
sk
OSI.Reconstruct
a
$S_{\mathsf{idx}} = \bigcap_{j=1}^{|\mathcal{K}|} \{S_j^{\mathsf{idx}} \mid \mathsf{kw}_j = \mathsf{ck}_i, i \in [m]\}$

Fig. 3: Oblivious Set Intersection

Given a database $D = \{(\mathsf{idx}_i, \mathsf{val}_i, S_i^{\mathrm{kw}})\}_{i\in[n]}$, the server performs the following decomposition: $D_1 = \left\{(k_j, S_j^{\mathsf{idx}})\right\}_{j\in[t]}$ and $D_2 = \{(\mathsf{idx}_i, \mathsf{val}_i)\}_{i\in[n]}$, where k_j is the keyword and S_j^{idx} is the index set consisting of all indices associated with the keyword k_i in D.

Encode Index Sets of Database The Algorithm 1 EncodeIndexSet shows that encodes the inverted-index set via Gödel coding.

Algorithm 1 EncodeIndexSet algorithm

```
Input: $\{(k_1, S_1^{\mathsf{idx}}), (k_2, S_2^{\mathsf{idx}}), \ldots, (k_t, S_t^{\mathsf{idx}})\}$
Output: $\{(k_1, S_1^{\mathsf{idx}\prime}), (k_2, S_2^{\mathsf{idx}\prime}), \ldots, (k_t, S_t^{\mathsf{idx}\prime})\}$
1: for $i = 1$ to $t$ do
2:     Initialize $S_i^{\mathsf{idx}\prime} := 1$
3:     Establish a one-to-one mapping: $j \to p_j, j \in [n]$
4:     for $j = 1$ to $n$ do
5:         $r \leftarrow Z_l$
6:         $S_i^{\mathsf{idx}\prime} := S_i^{\mathsf{idx}\prime} \cdot y_j^r$
7:     end for
8: end for
9: return $\{(k_1, S_1^{\mathsf{idx}\prime}), (k_2, S_2^{\mathsf{idx}\prime}), \ldots, (k_t, S_t^{\mathsf{idx}\prime})\}$
```

The server establishes a bijective mapping $\phi : i \to p_i$ between all indices and prime numbers, where p_i denotes the i-th prime (with $p_1 = 2$). Notably, this mapping operates on the original database structure $\{(\mathsf{idx}_i, \mathsf{val}_i)\}_{i\in[n]}$ rather than inverted-index sets, implementing a global encoding scheme.

To facilitate representation and calculation, the mapping can be explicitly defined as:

$$[1, 2, \ldots, n] \to [2, 3, \ldots, p_n], \tag{3}$$

where n is the number of data items in the database. Using Gödel coding, the server transforms any index set S into a compact integer representation:

$$S' = \prod_{i=1}^{n} y_i^{r_i}, \tag{4}$$

where

$$y_i = \begin{cases} 1, & \text{if} \quad i \in S \\ p_i, & \text{else} \end{cases}, \tag{5}$$

and $r_i \leftarrow Z_l$ for a small integer l. This encoding scheme effectively converts set operations into arithmetic computations.

OSI Setup Phase The Algorithm 2 shows Setup that performs server-side database preprocessing and prepares for subsequent client queries.

Algorithm 2 OSI.Setup algorithm

```
Input: datebase $D = \{(k_1, S_1^{\mathsf{idx}}), (k_2, S_2^{\mathsf{idx}}), \ldots, (k_t, S_t^{\mathsf{idx}})\}$.
Output: coded matrix $\mathbf{E}$ and state parameter $state_s$.
 1: $\{SubD_1, \ldots, SubD_c\} \leftarrow \mathsf{SimpleHash}(D, H_b)$
 2: Generate unit element $u = (k_u, S_u^{\mathsf{idx}})$
 3: $u' \leftarrow \mathsf{EncodeIndexSet}(\{u\})$
 4: for $i = 1$ to $c$ do
 5:     $SubD'_i \leftarrow \mathsf{EncodeIndexSet}(SubD_i)$
 6:     Insert $u'$ into $SubD'_i$
 7:     $(e_{1i}, e_{2i}) \leftarrow \mathsf{LinearCode}(SubD'_i, H_l)$
 8: end for
 9: $\mathbf{E} = \{(e_{11}, e_{21}), \ldots, (e_{1c}, e_{2c})\}$
10: $state_s = \{(|SubD_1|, \ldots, |SubD_c|), H_b, H_l\}$
11: return $\mathbf{E}$ and $state_s$
```

The SimpleHash algorithm distributes the database D into $c = (1 + \varepsilon)m$ subdatabases $\{SubD_1, SubD_2, \cdots, SubD_c\}$, where ε is a small constant and m represents the batch query size. The server randomly samples b independent hash functions $H_b = \{h_1, h_2, \ldots, h_b\}$ and utilizes these to map each value into the corresponding buckets, creating an efficient partitioning of the data for subsequent processing.

To handle the redundant buckets inherent in cuckoo hashing during client query processing, we incorporate a specially designed unit element $u = (k_u, S_u^{\mathsf{idx}})$ into our OSI algorithms. Here, k_u serves as a publicly known dummy keyword absent from all actual data items, while S_u^{idx} represents the complete index set of the database. This solution effectively addresses the structural requirements of cuckoo hashing and prevents potential correctness issues during homomorphic intersection operations on filtered results. Through Gödel coding (Eq. 4), the unit element becomes $u' = (k_u, 1)$, where the multiplicative identity "1" preserves algebraic structure while remaining neutral in set operations.

The LinearCode algorithm is applied to the privacy-preserving retrieval for our functional Keyword PIR. The algorithm takes as input a database D and three independent hash functions $H_l = h_1, h_2, h_3$, producing encoded output pairs $(e_{11}, e_{21}), \ldots, (e_{1c}, e_{2c})$. Here, h_1 handles data partitioning, h_2 generates short random-position basis vectors, and h_3 computes authentication tags to verify if client queries ck belong to the server's keyword set. Unlike the LinearCode approach in [34], our version processes the structure $SubD'_i = (k_i, S_i^{\mathsf{idx}\prime})$ by maintaining separate representations for keywords k_i and encoded index sets $S_i^{\mathsf{idx}\prime}$. This is achieved through two independent linear equations within the algorithm, yielding paired coding results $\{(e_{1i}, e_{2i})\}$ that preserve the structural separation between keywords and corresponding index sets throughout computation.

OSI Query Phase The Algorithm 3 shows GenQuery that enables the client encode keywords for querying.

Algorithm 3 OSI.GenQuery algorithm

```
Input: a set of keywords $S_{\mathsf{ck}} = \{\mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m\}$, $state_s$ and public-key $pk$
Output: query $q$
1: Parse $state_s = \{(|SubD_1|, \ldots, |SubD_b|), H_b, H_l\}$
2: $A \leftarrow \mathsf{CuckooHash}(S_{\mathsf{ck}}, H_b)$    ▷ Initialize a cuckoo hash table $A$ of size $c$
3: Generate $k_u$ to pad $A$
4: for $i = 1$ to $c$ do
5:     $(v_{1i}, v_{2i}) \leftarrow \mathsf{QueryVector}(A[i], H_l)$    ▷ $v_{1i}$ is the short-band vector, $v_{2i}$ is the select vector.
6: end for
7: $V = \{(v_{11}, v_{21}), \ldots, (v_{1c}, v_{2c})\}$
8: $q \leftarrow \mathsf{FHE.Enc}(V, pk)$
9: return $q$
```

Given a keyword set $S_{\mathsf{ck}} = \{\mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m\}$ and $state_s$ from the server, the client employs hash functions H_b to map keywords to a Cuckoo hash table A. Fill the blank positions in A with k_u. The QueryVector algorithm takes the first two hash functions in LinearCode and the bucket $A[i]$ of the Cuckoo hash table as input, and outputs two components: a short-band vector v_{1i} used to recover the final results and a selection vector v_{2i} for block filtering. Finally, the client encrypts all paired vectors utilizing the BFV homomorphic encryption scheme.

OSI Answer Phase The Algorithm 4 shows Answer where the server responds to the query in a bucket-by-bucket fashion.

Algorithm 4 OSI.Answer algorithm

```
Input: coded matrix $\mathbf{E}$ and query $q$
Output: answer $a$
1: Decomposition query $q$ into $q_1, q_2, \ldots, q_c$
2: Parse $\mathbf{E} = [(e_{11}, e_{21}), \ldots, (e_{1c}, e_{2c})]$
3: for $i = 1$ to $c$ do
4:     $a_{1i}, a_{2i} \leftarrow \mathsf{Answer}(q_i, \mathbf{E}[i])$
5: end for
6: $a_1 = \prod_{i=1}^{c} a_{1i}$    ▷ Homomorphic multiplication
7: $a_2 = [a_{21}, a_{22}, \ldots, a_{2c}]$
8: $a = (a_1, a_2)$
9: return $a$
```

The server first decomposes the query into corresponding buckets and subsequently processes each query per bucket. Within each bucket, given that the query comprises a short-band vector and a selection vector, the Answer algorithm - referencing the approach in [34] - executes in two stages: first computing intermediate results via inner product between the short-band vector and encoding, then deriving final encrypted results through inner product between the selection vector and intermediate results. The server then homomorphically multiplies the results across all buckets before returning the final result to the client.

OSI Reconstruction Phase The Algorithm 5 shows Reconstruction that the client gets the set of indices that satisfy all the keywords.

Algorithm 5 OSI.Reconstruct algorithm

```
Input: $a$ and $sk$
Output: $S_{\mathsf{idx}}$ :Set of multi-keyword matching indices
 1: Parse $a = (a_1, a_2)$
 2: Parse $a_2 = [a_{21}, a_{22}, \ldots, a_{2b}]$
 3: $\{y_1, y_2, \ldots, y_b\} \leftarrow \mathsf{FHE.Dec}(a_2, sk)$
 4: if $\{h(\mathsf{ck}_1), h(\mathsf{ck}_2), \ldots, h(\mathsf{ck}_m)\} \not\subseteq \{y_1, y_2, \ldots, y_c\}$ then
 5: return $\perp$
 6: end if
 7: $x \leftarrow \mathsf{FHE.Dec}(a_1, sk)$
 8: initialize $S_{\mathsf{idx}} := \emptyset$
 9: for $i = 1$ to $n$ do
10:     if $x \neq 0 \mod p_i$ then
11:         $S_{\mathsf{idx}} := S_{\mathsf{idx}} \cup \{i\}$
12:     end if
13: end for
14: return $S_{\mathsf{idx}}$
```

The client first checks all second ciphertexts to verify keywords upon receiving the encrypted responses. Only when all keywords are successfully matched does the client decrypt all first ciphertexts. The final intersection can then be easily obtained through a modulo operation, which benefits from the property of Gödel coding. This process relies on the separate encoding of indices and $h_3(k)$ values in OSI.Setup, which causes the server response to contain two paired values. A correct authentication produces valid outputs, whereas failed authentication leads to incorrect results.

4.3 $\mathsf{MkPIR}^{\mathbf{I}}$ Construction

Building upon the preceding OSI algorithm, the implementation of Mk-PIR becomes straightforward, with its architecture illustrated in Figure 4.

$\mathsf{MkPIR}^{\mathbf{I}}$ operates through an efficient two-phase construction comprising offline preprocessing and online query processing. In the offline phase, the server performs critical preparations by decomposing the original database D into two structured sub-databases: $D_1 = \{(\mathsf{idx}_i, S_i^{\mathsf{kw}})\}_{i\in[n]}$ maintaining index-to-keyword mappings, and $D_2 = \{(\mathsf{idx}_i, \mathsf{val}_i)\}_{i\in[n]}$ storing the actual values. This phase further includes inverted index construction through D_1' generation and specialized encodings via OSI.Setup and BatchCode operations. The online phase executes two privacy-preserving interactions: first, clients securely retrieve target indices matching query keywords through our OSI algorithms (involving $\mathsf{GenQuery}$, Answer, and $\mathsf{Reconstruct}$), then efficiently obtain corresponding values via optimized batched $\mathsf{IndexPIR}$ on D_2'. This designed separation of offline preprocessing from online operations ensures both computational efficiency and privacy guarantees, with the OSI enabling secure keyword-to-index resolution while standard PIR techniques handle the subsequent index-to-value retrieval.

Parameter:

– The client holds a set of keywords $S_{\mathsf{ck}} = \{\mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m\} \subseteq \{0,1\}^*$ and a key pair (pk, sk).
– The server holds a database $D = \{(\mathsf{idx}_i, \mathsf{val}_i, S_i^{\mathsf{kw}})\}_{i\in[1,n]}$, where S_i^{kw} is a set of val_i-related keywords and $S_i^{\mathsf{kw}} \subseteq \mathcal{K}$.

Construction:

Offline Phase

1. For $i \in [n]$, the server decomposes the database D into $D_1 = \{(\mathsf{idx}_i, S_i^{\mathsf{kw}})\}$ and $D_2 = \{(\mathsf{idx}_i, \mathsf{val}_i)\}$.
2. The server encodes D_1 to $D_1' = \{(k_j, S_j^{\mathsf{idx}})\}_{j\in[|\mathcal{K}|]}$, here $k_j \in \mathcal{K}$ and $S_j^{\mathsf{idx}} = \bigcup_{i=1}^{n} \left\{\mathsf{idx}_i | k_j \in S_i^{\mathsf{kw}}\right\}$.
3. The server obtains $\{\mathbf{E}, state_s\} \leftarrow \mathsf{OSI.Setup}(D_1')$ and publishes $state_s$.
4. The server executes batch encoding on D_2 and obtains $D_2' \leftarrow \mathsf{BatchCode}(D_2, H_b)$, here $H_b \in state_s$.

Online Phase

1. The client obtains $state_s$ and sends to the server the query q, where $q \leftarrow \mathsf{OSI.GenQuery}(S_{\mathsf{ck}}, state_s, pk)$.
2. The server receives q and sends to the client the answer a, where $a \leftarrow \mathsf{OSI.Answer}(q, \mathbf{E})$.
3. The client receives a and reconstructs the index set S_{idx}, where $S_{\mathsf{idx}} \leftarrow \mathsf{OSI.Reconstruct}(a, sk)$ essentially satisfing $S_{\mathsf{idx}} = \bigcup_{i=1}^{n} \left\{\mathsf{idx}_i | S_{\mathsf{ck}} \subseteq S_i^{\mathsf{kw}}\right\}$ or $\emptyset$.
4. The client and server execute batch Symmetrical PIR schemes. The client receives $S_{\mathsf{val}} \leftarrow \mathsf{PIR}(S_{\mathsf{idx}}, D_2')$.

Fig. 4: $\mathsf{MkPIR}^{\mathbf{I}}$ Construction

4.4 Complexity Analysis

Our $\mathsf{MkPIR}^{\mathbf{I}}$ construction operates through two communication rounds, with the second round implementing a conventional Index PIR scheme. Our primary innovation lies in the first round. For this reason, we provide detailed computational and communication costs specifically for this phase. The detailed complexity is shown in the Table 2.

Table 2: OSI algorithm complexity

	Phase	Computation	Communication
Offline	Setup	$O(wt)$	–
Online	Query	$O(m)$	$O(\sqrt{t})$
	Answer	$O(wt)$	$O(n)$
	Reconstruction	$O(n)$	–

During the offline phase, the server processes the database by first partitioning it into multiple tables organized by keywords k and corresponding values val. The OSI algorithms are then applied, where t keywords are distributed across c bins through w independent hash functions, forming distinct sub-databases. Each sub-database undergoes two key operations: insertion of a unit element u followed by specific encoding transformations. Finally, the server transmits the processed state value $state_s$ to the client. This comprehensive offline preparation maintains an overall computational complexity of $O(wt)$.

During the online phase, the client processes m keywords by distributing them across c bins using w hash functions, while padding empty queries for the remaining bins. These c queries are then encrypted and batched using the BFV scheme with SIMD before transmission to the server. Upon receiving the encrypted query q, the server decomposes it into components $q_1, q_2, \cdots, q_c$ and computes responses using the preprocessed matrix E. The client subsequently decrypts and authenticates the server's response against the original keywords before performing modular operations to recover the final index set. This phase maintains a computational complexity of $O((m+w)t)+O(n)$ and communication complexity of $O(\sqrt{t}+n)$.

4.5 More Discussions of OSI

An important observation emerges from the server's use of Gödel coding: by simply inverting the encoding rules, we can obtain an encoding scheme that naturally supports union operations rather than intersections. This modified encoding, where we assign:

$$y_i = \begin{cases} p_i, \text{if} \quad i \in S \\ 1, \ \text{else} \end{cases},$$

effectively converts the multiplicative structure to support union operations through the same fundamental arithmetic properties.

While classical PIR schemes typically do not consider privacy on the server side, the original intention of OSI is to facilitate the client to learn the intersection (*i.e.,* indices matching all queried keywords) without obtaining additional information about the server. In the $\mathsf{OSI.Reconstruct}$ algorithm, however, by analyzing the exponents of prime factors, the client learns the number of unmatched keywords for each index. We discuss two solutions: inserting random dummy elements into buckets, or performing bit-wise AND operations on BFV ciphertext with auxiliary bit-extraction mechanisms. The latter approach, for example, polynomial interpolation, enables bit extraction but increases computation and noise budget consumption.

5 Forward-Index-based Mk-PIR

Section 4 presents Mk-PIR built upon an inverted index table, which proves theoretically efficient when the server's keyword count is smaller than its index count, as this architecture reduces the dimensionality of the problem. However, $\mathsf{MkPIR}^{\mathbf{I}}$ becomes suboptimal when the keyword cardinality exceeds the number of indices. In such scenarios, a forward-index-based construction becomes necessary to enable the client to privately retrieve indices corresponding to keywords.

In this section, we introduce our second construction $\mathsf{MkPIR}^{\mathbf{F}}$ based on our proposed Private Subset Determination (PSD). $\mathsf{MkPIR}^{\mathbf{F}}$ is suitable when the number of total keywords is greater than the number of indices in the server.

5.1 Design Intuition of PSD

Given the server's database $\{(\mathsf{idx}_i, \mathsf{val}_i, S_i^{\mathsf{kw}})\}_{i\in[n]}$ and the client's keyword set S_{ck}, the PSD should output 1 if $S_{\mathsf{ck}} \subseteq S_i^{\mathsf{kw}}$ and 0 otherwise. The server encodes all keywords using Constant-weight codes. Building upon the bit string equality comparison defined in Eq. 2, we develop a subset determination method that evaluates whether a set S_x is contained within another set S_y. This subset relationship is formally defined by:

$$f_{PSD}(S_x, S_y) = \prod_{i=1}^{|S_x|} \left(\sum_{j=1}^{|S_y|} f_{PCW}(x_i, y_j) \right), \tag{6}$$

yielding a binary result:

$$f_{PSD}(S_x, S_y) = \begin{cases} 1, \text{ if} \quad S_x \subseteq S_y \\ 0, \text{ else} \end{cases}. \tag{7}$$

It is trivial to demonstrate the correctness. The inner sum verifies its presence in S_y through f_{PCW}, producing 1 if found and 0 otherwise. The product of these values yields 1 only when all elements of S_x exist in S_y, confirming the subset relation. Formally, if $S_a \subseteq S_b$, all $f_{PCW}(x, y) = 1$, hence $f_{PSD}(S_a, S_b) = 1$. If any $x \in S_a$ is not in S_b, the corresponding $f_{PCW}(x, y) = 0$, making $f_{PSD}(S_a, S_b) = 0$.

In the implementation of PSD algorithms, the client first homomorphically encrypts S_{ck} before sending it to the server. Then, the server performs holomorphic operations to evaluate the subset condition between the received encrypted S_{ck} and each S_i^{kw} for $i \in [n]$. The server generates an n-length vector through secure computation of $f_{PSD}(\cdot,\cdot)$, where each element is a homomorphic ciphertext of '0' or '1'. While similar to classical PIR query vectors (composed of '0's with a single '1' at the target position), our scheme differs crucially: the subset judgment produces multiple '1's in the output vector. This creates two implementation challenges: (1) direct inner product computation would yield summed results that clients cannot uniquely decompose, and (2) element-wise multiplication preserves matching positions but incurs $O(n)$ communication overhead, violating PIR efficiency requirements.

We compress the n index results into a single ciphertext through careful encoding. Mapping indices to primes via $\prod_{i=1}^{n}[(p_i - 1)x_i + 1])$ would work, though its multiplicative depth $O(1 + \log_2 n)$ causes unacceptable noise growth. Therefore, we employ binary weight encoding that maps each index idx_i to 2^i and computes:

$$s = \sum_{i=1}^{n} x_i \cdot 2^i, \tag{8}$$

where x_i represents the subset judgment output for position i. This scheme requires only one level of homomorphic multiplication and $\log N$ levels of homomorphic addition, maintaining acceptable noise accumulation. Upon receiving and decrypting s, the client performs binary decomposition to identify matching indices through their bit positions.

5.2 Detailed PSD algorithms

As shown in Figure 5, we divide the PSD scheme into four phases: setup, query, answer, and reconstruction. To maintain homomorphic computations within the plaintext domain constraints, we implement index mapping with binary coding segmentation, employing SIMD to efficiently pack the results into compact ciphertext representations (either single or multiple ciphertexts as required by the computational parameters).

PSD Setup Phase The algorithm implementation is as described in Algorithm 6.

Mahdavi *et al.* [27] introduced the PerfectMapping algorithm, which establishes a bijective mapping between elements in $[n]$ and constant-weight codes. In our scheme, the server maintains the database $D = (\mathsf{idx}_i, \mathsf{val}_i, S_i^{\mathsf{kw}})_{i\in[n]}$ and applies PerfectMapping to encode all keywords. The resulting parameters *parm* are subsequently transmitted to the client for query preparation.

PSD Query Phase The algorithm implementation is as described in Algorithm 7.

The client maintains a keyword set $S_{\mathsf{ck}} = \mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m$ and a public-private key pair (pk, sk). After receiving the parameters *parm* from the server, the client performs the following operations: first, it encodes S_{ck} using the PerfectMapping algorithm to obtain the constant-weight code representation $c = (c_1, c_2, \cdots, c_m)$; then, it encrypts this coded vector c using the BFV homomorphic encryption scheme under the public key pk.

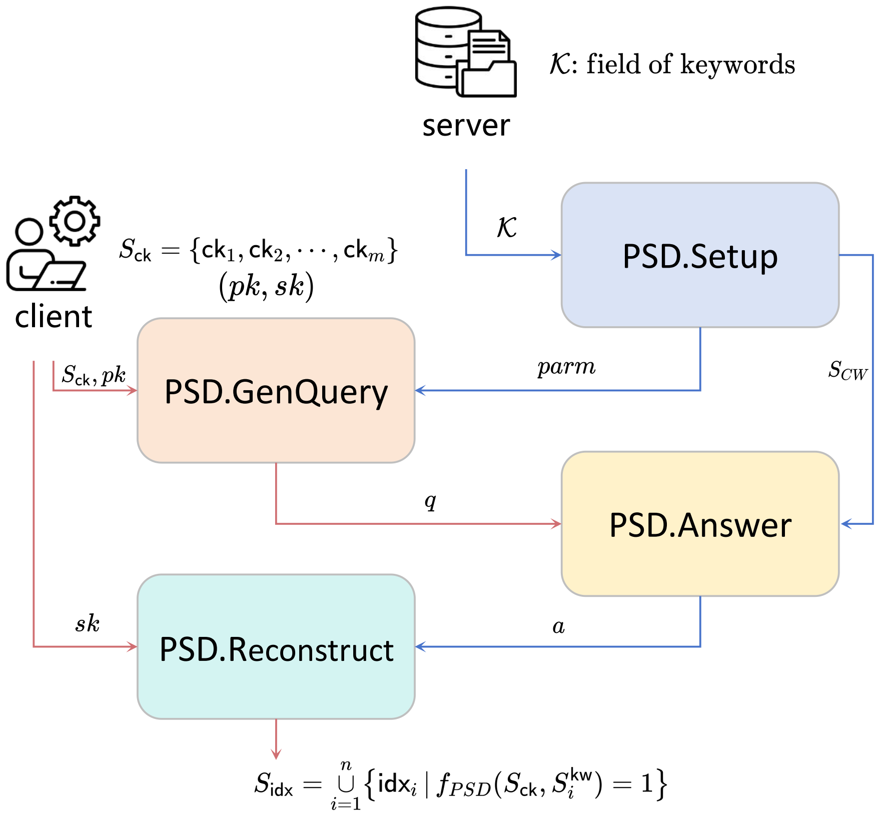

server
$\mathcal{K}$: field of keywords
client
$S_{\mathsf{ck}} = \{\mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m\}$
(pk, sk)
$\mathcal{K}$
PSD.Setup
S_{ck}, pk
PSD.GenQuery
$parm$
S_{CW}
q
PSD.Answer
sk
PSD.Reconstruct
a
$S_{\mathsf{idx}} = \bigcup_{i=1}^{n} \{\mathsf{idx}_i \mid f_{PSD}(S_{\mathsf{ck}}, S_i^{\mathsf{kw}}) = 1\}$

Fig. 5: Private Subset Determination

Algorithm 6 PSD.Setup algorithm

Input: a field of keywords $\mathcal{K}$
Output: encoded keywords S_{CW} and parameter $parm$

```
1: Select $parm = (c, w)$ and $\binom{c}{w} \geq |\mathcal{K}|$ ▷ $c$ is the codeword length, $w$ is the number of 1.
2: initialize $S_{CW} = \emptyset$
3: Parse $(\mathsf{kw}_1, \mathsf{kw}_2, \ldots, \mathsf{kw}_t) = \bigcup_{i=1}^{n} S_i^{\mathsf{kw}}$
4: for $i = 1$ to $t$ do
5:     $cw_i \leftarrow \mathsf{PerfectMapping}(\mathsf{kw}_i, c, w)$
6: end for
7: $S_{CW} = (cw_1, \ldots, cw_t)$
8: return $(S_{CW}, parm)$
```

Algorithm 7 PSD.GenQuery algorithm

```
Input: a set of keywords $S_{\mathsf{ck}} = \{\mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m\}$, parameter $parm$ from setup phase
    and public-key $pk$
Output: the query of keywords $q$
 1: Parse $parm = (c, w)$
 2: for $i = 1$ to $m$ do
 3:     $c_i \leftarrow \mathsf{PerfectMapping}(\mathsf{kw}_i, c, w)$
 4: end for
 5: $c = (c_1, c_2, \ldots, c_m)$
 6: $q \leftarrow \mathsf{FHE.Enc}(c, pk)$
 7: return $q$
```

PSD Answer Phase The algorithm implementation is as described in Algorithm 8.

Algorithm 8 PSD.Answer algorithm

```
Input: the query $q$ from query phase and encoded keywords $S_{CW}$ from setup phase
Output: the answer $a$
 1: Decomposition $q$ into $q_1, q_2, \ldots, q_m$
 2: Parse $S_{CW} = (cw_1, \ldots, cw_t)$
 3: for $i = 1$ to $m$ do
 4:     for $j = 1$ to $t$ do
 5:         $eq_{ij} \leftarrow f_{PCW}(q_i, cw_j)$
 6:     end for
 7: end for
 8: Initialize the vector $v$ of length $n$, $a = 0$
 9: for $i = 1$ to $n$ do
10:     $v[i] := \prod_{j=1}^{m}(\sum_{k=1}^{|S_i^{\mathsf{kw}}|} eq_{jk})$ ▷ $eq_{jk} \leftarrow f_{PCW}(q_j, cw_k)$, where $cw_k$ is the codeword
    of the element in $S_i^{\mathsf{kw}}$
11:     $a := a + v[i] \times 2^{i-1}$
12: end for
13: return $a$
```

Upon receiving the query q from the client, the server performs the following operations: First, it decomposes q into its components $(q_1, q_2, \cdots, q_m)$. Then, for each $i \in [m]$ and $j \in [t]$, the server executes the comparison function $f_{PCW}(q_i, cw_j)$ to evaluate the relationship between the client's query terms and the server's coded keywords. To optimize communication efficiency, the server performs vector compression by first forming a ciphertext vector v of length n, where each entry $v[i]$ is the ciphertext of 1 if $S_{\mathsf{ck}} \subseteq S_i^{\mathsf{kw}}$ and the ciphertext of 0 otherwise. The server then computes the compressed representation through bitwise accumulation, initializing $a = 0$ and iteratively updating it as $a = a + v[i] \cdot 2^{i-1}$ for all $i \in [n]$, effectively encoding the matching indices into a compact integer format.

PSD Reconstruction Phase The algorithm implementation is as described in Algorithm 9.

Algorithm 9 PSD.Reconstruct algorithm

```
Input: the answer $a$ from answer phase and private-key $sk$
Output: a set of indices $S_{\mathsf{idx}}$
1: $x \leftarrow \mathsf{FHE.Dec}(a, sk)$
2: Factorization $x \rightarrow \sum_{i=1}^{n}(\mathsf{bit}_i \times 2^{i-1})$
3: initialize $S_{\mathsf{idx}} = \emptyset$
4: for $i = 1$ to $n$ do
5:     if $\mathsf{bit}_i == 1$ then
6:         $S_{\mathsf{idx}} := S_{\mathsf{idx}} \bigcup \{i\}$
7:     end if
8: end for
9: return $S_{\mathsf{idx}}$
```

In the final phase, the client decrypts the server's answer a to recover the integer value x. Using the binary weight encoding scheme, the client performs unique factorization of x through the relation $x = \sum_{i=1}^{n}(\mathsf{bit}_i \times 2^{i-1})$. This decomposition directly reveals the matching indices, as each nonzero bit $\mathsf{bit}_i = 1$ corresponds to a valid index i where the subset condition $S_{\mathsf{ck}} \subseteq S_i^{\mathsf{kw}}$ holds true.

5.3 $\mathsf{MkPIR}^{\mathbf{F}}$ Construction

Figure 6 shows the PSD-based construction $\mathsf{MkPIR}^{\mathbf{F}}$. The server maintains a database $D = \{(\mathsf{idx}_i, \mathsf{val}_i, S_i^{\mathsf{kw}})\}_{i\in[n]}$. The client has a set of keywords $S_{\mathsf{ck}} = \{\mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m\}$.

The construction can be divided into an offline phase and an online phase. In the offline phase, the server samples $parm$ and encodes all keywords $\mathcal{K}$ to get a set S_{CW}. The online phase is divided into two rounds of communication. In the first round, the client obtains the index corresponding to the query. First, the server sends $parm$ to the client. The client encodes the query q by $\mathsf{PSD.GenQuery}$ algorithm and sends the query q to the server. The server replies with an answer a to the client. Then the client decrypts a to obtain a set of indices S_{idx}. In the second round, both parties execute a normal Index PIR to get the information corresponding to the obtained indices.

5.4 Complexity Analysis

As stated in Section 4.4, we just provide the complexity analysis for the first communication round, with complete metrics provided in Table 3.

During offline preparation, the server initializes encoding parameters c and w, ensuring their combinatorial capacity exceeds the keyword domain $\mathcal{K}$ size. Each keyword $\mathsf{kw} \in \mathcal{K}$ undergoes constant-weight encoding, requiring $O(t \cdot \log|\mathcal{K}|)$ computational complexity.

Parameter:

- The client holds a set of keywords $S_{\mathsf{ck}} = \{\mathsf{ck}_1, \mathsf{ck}_2, \cdots, \mathsf{ck}_m\} \subseteq \{0,1\}^*$ and a key pair (pk, sk).
- The server holds a database $D = \{(\mathsf{idx}_i, \mathsf{val}_i, S_i^{\mathsf{kw}})\}_{i \in [1,n]}$, where S_i^{kw} is a set of val_i-related keywords and $S_i^{\mathsf{kw}} \subseteq \mathcal{K}$.

Construction:

Offline Phase

The server obtains $(S_{CW}, parm) \leftarrow \mathsf{PSD.Setup}(\mathcal{K})$ and publishes $parm$.

Online Phase

1. The client receives $parm$ and sends to the server the encoded query q, where $q \leftarrow \mathsf{PSD.GenQuery}(S_{\mathsf{ck}}, parm, pk)$.
2. The server receives q and sends to the client the generated answer a, where $a \leftarrow \mathsf{PSD.Answer}(q, S_{CW})$.
3. The client receives the answer a and reconstructs the index set S_{idx}, where $S_{\mathsf{idx}} \leftarrow \mathsf{PSD.Reconstruct}(a, sk)$ that essentially satisfies $S_{\mathsf{idx}} = \bigcup_{i=1}^{n} \left\{\mathsf{idx}_i | S_{\mathsf{ck}} \subseteq S_i^{\mathsf{kw}}\right\}$ or $\emptyset$.
4. The client and server execute batch Symmetrical PIR schemes. The client receives $S_{\mathsf{val}} \leftarrow \mathsf{PIR}(S_{\mathsf{idx}}, D)$.

Fig. 6: $\mathsf{MkPIR}^{\mathbf{F}}$ Construction

Table 3: PSD algorithm complexity

Phase		Computation	Communication
Offline	Setup	$O(t \cdot \log \lvert\mathcal{K}\rvert)$	–
Online	Query	$O(m \cdot \log \lvert\mathcal{K}\rvert)$	$O(m \cdot \log \lvert\mathcal{K}\rvert)$
	Answer	$O(mt \cdot \log \lvert\mathcal{K}\rvert + n)$	$O(n)$
	Reconstruction	$O(n)$	–

The online phase begins with the client encoding and encrypting m query keywords using constant-weight coding before transmission. The server performs encrypted equality checks between the m ciphertext encodings and t plaintext keyword encodings. Through selective summation and multiplicative aggregation, it generates subset determination ciphertexts a for client retrieval. Final index recovery involves n binary decompositions of decrypted results, yielding $O(mt \cdot \log|\mathcal{K}|) + O(n)$ computational complexity and $O(m \cdot \log|\mathcal{K}|) + O(n)$ communication complexity.

6 Performance

In this section, we present the experimental setup, optimization, and detailed results to evaluate the efficiency and feasibility of our proposed Mk-PIR. Specifically, we focus on measuring the computational cost and communication overhead of the core algorithm (OSI in $\mathsf{MkPIR}^{\mathbf{I}}$ and PSD in $\mathsf{MkPIR}^{\mathbf{F}}$). An implementation of our Mk-PIR is available at https://github.com/Brave11again/mkpir.

6.1 Settings

We implemented Mk-PIR using the BFV homomorphic encryption scheme from Microsoft's SEAL library (https://github.com/microsoft/SEAL). Our experiments used statistical security parameter $\lambda = 40$ and computational security parameter $\kappa = 128$. Encryption parameters included polynomial degrees of 8196 or 16392, a coefficient modulus exceeding 256 bits, and a plaintext modulus over 20 bits. All tests ran on an AMD 7940HS CPU with 16GB RAM under Windows 11.

6.2 Optimizations

We adopt the BFV homomorphic encryption scheme, leveraging its SIMD capabilities provided by the SEAL library. To minimize computation and communication overhead, during the query generation phase, we pack b bits (each being 0 or 1) into a single ciphertext, following the technique proposed by [3]. Subsequently, the server executes an oblivious expansion on this packed ciphertext to produce b separate ciphertexts, each encrypting a single 0 or 1. This approach is employed in both the OSI and PSD implementations, as the plaintext queries consist of vectors of 0s and 1s.

To prevent the resultant inner product from exceeding the plaintext domain and compromising correctness, we implement distinct optimizations for each algorithm:

Server-side OSI We encode indices in small batches (*e.g.,* 4 indices per batch). Leveraging the SIMD slot technique, we sequentially place each batch's encoding into consecutive slots ranging from 1 to N, where N is the polynomial degree.

This approach achieves acceleration by processing $4N$ indices simultaneously. To reduce the communication overhead of PIR, a common approach is to represent the database in a multi-dimensional manner. In our OSI experiment, a two-dimensional representation scheme is adopted: first, the data within each bucket is linearly encoded in blocks of 128 items, forming the first dimension; then, all encoding results within a single bucket are integrated to form the second dimension. Thus, during the $\mathsf{OSI.Answer}$ phase, the server decodes each block first, and then performs a secondary inner product on all block results to select the correct outcome, thereby optimizing both communication and computation processes.

Server-side PSD We partition the inner product computations into blocks. It can be set to 20 blocks per piece. The results from each block are sequentially placed into slots 1 to N, reducing the number of returned ciphertexts.

Additionally, the modulus reduction technique can be applied prior to the server's response to reduce the ciphertext modulus while maintaining correct decryption. The SEAL library adopted in our implementation conveniently supports all aforementioned optimizations.

6.3 Theoretical Comparisons

To verify the performance advantages of $\mathsf{MkPIR}^{\mathbf{I}}$ and $\mathsf{MkPIR}^{\mathbf{F}}$, we compared them with state-of-the-art schemes in terms of computational complexity, communication complexity, and others. The specific comparisons are shown in Table 4.

Table 4: Comparison of different schemes

Method	Computation	Communication	Query Keyword	$f : A \leftrightarrow B$	Server pirvacy
[27]	$O(mt \cdot \log\lvert\mathcal{K}\rvert))$	$O(m \cdot \log\lvert\mathcal{K}\rvert)$	Single	✗	✗
[34]	$O(mn)$	$O(m\sqrt{n})$	Single	✗	✗
[1]	$O(mt \cdot \log\lvert\mathcal{K}\rvert)$	$O(n)$	Single	✗	✗
Baseline	$O(2^t)$ + SPIR	$O(\sqrt{2^t})$ + SPIR	Multiple	✗	✗
$\mathsf{MkPIR}^{\mathbf{I}}$	$O(wt)$ + SPIR	$O(\sqrt{t}+n)$ + SPIR	Multiple	✓	✓
$\mathsf{MkPIR}^{\mathbf{F}}$	$O(mt \cdot \log\lvert\mathcal{K}\rvert + n)$ + SPIR	$O(m \cdot \log\lvert\mathcal{K}\rvert + n)$ + SPIR	Multiple	✓	✓

$f : A \leftrightarrow B$ denotes a many-to-many mapping between the indices and the keywords. SPIR denotes Symmetrical PIR.

Schemes [27] [34] [1] only support single-keyword queries. $\mathsf{MkPIR}^{\mathbf{I}}$ and $\mathsf{MkPIR}^{\mathbf{F}}$ enable multi-keyword queries, offering superior functional scalability. Regarding index-keyword mapping, these schemes require a strict one-to-one correspondence, which significantly broadens their application scope. The integration of Index PIR increases the computational and communication complexity for $\mathsf{MkPIR}^{\mathbf{I}}$

and $\mathsf{MkPIR}^{\mathbf{F}}$. This trade-off renders them particularly suitable for scenarios demanding flexible multi-keyword search capabilities.

We define a baseline method in the table, which supports direct multi-keyword retrieval. To support arbitrary keyword combinations, the server computes the power set for all possible keyword combinations. Specifically, the server prepares all subsets derived from the inverted-index table, sorts them, and establishes mappings (*e.g.,* through interpolation). However, this baseline approach suffers two critical limitations: (1) For n keywords, it requires $O(2^n)$ intersection operations and storage, and (2) database updates necessitate full recomputation of the power set, rendering the method computationally infeasible.

6.4 Experimental Costs

Data of $\mathsf{MkPIR}^{\mathbf{I}}$ appears in Table 5. Computation is divided into four phases: oblivious ciphertext expansion (T1), block-wise linear decoding (T2), secondary selection of decoding results (T3), and the intersection operation on the selected results (T4). Computational costs increase with larger n, t, m, and polynomial degree $\deg(P)$. Growth in n causes exponential total time increases, with T2 as the computational bottleneck. Batch processing shows minimal impact on cost when m rises: increasing queries from 4 to 8 reduces cost growth from 30% to 0.92% as n scales. For small m, batch processing incurs higher costs than the non-batched mode due to hashing-to-bin overhead. This inefficiency reverses at larger m where batching significantly reduces costs. We compared the communication costs of the non-batch mode ($m = 2$) with those of the batch mode ($m = 4, 8$). Non-batch transmission costs are 423 KB, while batch costs are 1784 KB. Receive costs depend on $\deg(P)$: the server packs ciphertext indices into polynomial-degree-sized slots (4 indices per slot). When m exceeds available slots, additional transmission occurs.

Based on the experimental data in Table 6, computational costs of $\mathsf{MkPIR}^{\mathbf{F}}$ scale substantially with larger parameters. For example, at $m = 2$ and $t = 2^{10}$, total time increases from $824s$ ($n = 2^{12}$) to $2,268s$ ($n = 2^{14}$). T3 consistently consumes over 50% of processing time across all configurations, confirming its role as the computational bottleneck. Communication costs remain fixed at 1784 KB sent and 1558 KB received due to ciphertext packing in 20-index slots, regardless of parameter variations. This demonstrates stable transmission overhead within slot capacity constraints.

We presented the overall computational and communication costs for the two constructions of Mk-PIR. Figure 7 illustrates the costs of $\mathsf{MkPIR}^{\mathbf{I}}$ when $t = 2^{12}$. In terms of computational cost, when the database size n increases from 2^{14} to 2^{18}, for different numbers of keywords ($m = 4$, 8), the growth rates of their computational costs were similar by using batch processing operations. Regarding communication overhead, it can be divided into two types: one is queries without batch processing operations, and the other is queries with batch processing operations. The total communication overhead in both types is close to that of the first round, while the overhead of the second round of communication is negligible. Figure 8 illustrates the overall computational and communication

Table 5: Computational and communication costs of OSI

m	deg(P)	t	Computation (s)														
			$n = 2^{14}$					$n = 2^{16}$					$n = 2^{18}$				
			T1	T2	T3	T4	Total	T1	T2	T3	T4	Total	T1	T2	T3	T4	Total
2	8196	2^{12}	1.72	6.06	0.52	0.03	8.33	1.64	23.90	0.98	0.03	26.55	1.66	96.26	4.18	0.04	102.14
		2^{14}	1.64	23.34	2.02	0.03	27.03	1.64	94.58	4.12	0.03	100.37	1.66	373.72	16.06	0.04	391.48
		2^{16}	1.66	95.62	8.06	0.03	105.37	1.66	375.16	15.88	0.03	392.73	1.62	1506.76	63.62	0.04	1572.04
4	16392	2^{12}	23.96	20.53	3.32	0.32	48.13	26.27	78.97	3.26	0.33	108.83	26.38	332.98	13.14	1.31	373.81
		2^{14}	28.77	77.49	14.15	0.36	120.77	26.36	318.53	13.06	0.33	358.28	26.54	1274.93	54.12	1.33	1356.92
		2^{16}	28.03	313.69	55.73	0.35	397.80	26.43	1257.78	52.51	0.33	1337.05	26.27	5094.11	212.55	1.32	5334.25
8	16392	2^{12}	67.19	21.81	3.94	0.73	93.67	64.67	76.80	3.72	0.70	145.89	69.84	326.21	15.12	2.89	414.06
		2^{14}	65.96	76.23	13.54	0.76	156.49	68.81	327.65	13.64	0.68	410.78	67.04	1331.56	58.85	2.90	1460.35
		2^{16}	66.01	322.84	56.3	0.71	445.86	66.51	1314.96	58.08	0.73	1440.28	70.54	5077.50	232.32	2.97	5383.33
Comm. (KB)	$m = 2$ (No Batch)	Send	423					423					423				
		Receive	317					633					2531				
	$m = 4$, 8 (Batch)	Send	1784					1784					1784				
		Receive	1558					1558					6232				

Table 6: Computational and communication costs of PSD

m	t	Computation (s)														
		$n = 2^{12}$					$n = 2^{13}$					$n = 2^{14}$				
		T1	T2	T3	T4	Total	T1	T2	T3	T4	Total	T1	T2	T3	T4	Total
2	2^{10}	43.19	222.13	507.02	52.11	824.45	40.70	212.28	911.52	98.72	1263.22	41.59	164.56	1866.07	196.11	2268.33
	2^{11}	43.22	354.47	492.25	51.02	940.96	42.24	363.05	904.38	103.10	1412.68	40.27	336.13	1787.26	191.03	2354.69
	2^{12}	42.82	734.58	476.18	47.40	1300.98	40.07	696.63	912.52	94.61	1743.83	42.48	707.49	2041.28	202.18	2993.43
3	2^{10}	65.98	307.24	656.28	50.42	1079.92	67.41	313.39	1241.95	98.89	1721.64	65.04	317.41	2339.78	200.56	2922.79
	2^{11}	63.87	696.53	672.41	54.34	1487.15	65.04	707.43	1404.77	93.45	2270.69	64.77	603.64	2330.21	192.04	3190.66
	2^{12}	64.13	1243.54	642.44	52.39	2002.50	69.89	1060.2	1243.46	190.97	2478.28	68.03	1187.09	2389.51	190.97	3835.60
4	2^{10}	193.23	428.25	901.31	49.23	1572.02	198.72	394.68	1509.83	95.17	2198.40	191.16	469.21	3463.90	208.73	4333.00
	2^{11}	192.12	784.63	757.71	47.24	1781.70	191.67	819.01	1521.69	101.76	2634.13	195.36	794.57	2983.63	198.94	4172.50
	2^{12}	203.22	1681.51	810.02	51.47	2746.22	198.35	1673.13	1756.77	105.32	3733.57	203.46	1715.58	3319.84	215.13	5454.01
Comm. (KB)	Send	1784														
	Receive	1558														

Polynomial degree deg(P) = 16392.

T1: oblivious ciphertext expansion; T2: equality comparison between the constant-weight code sent by the client and all constant-weight codes in the server; T3: generation of a '0/1' vector based on the comparison results; T4: packaging the final results.

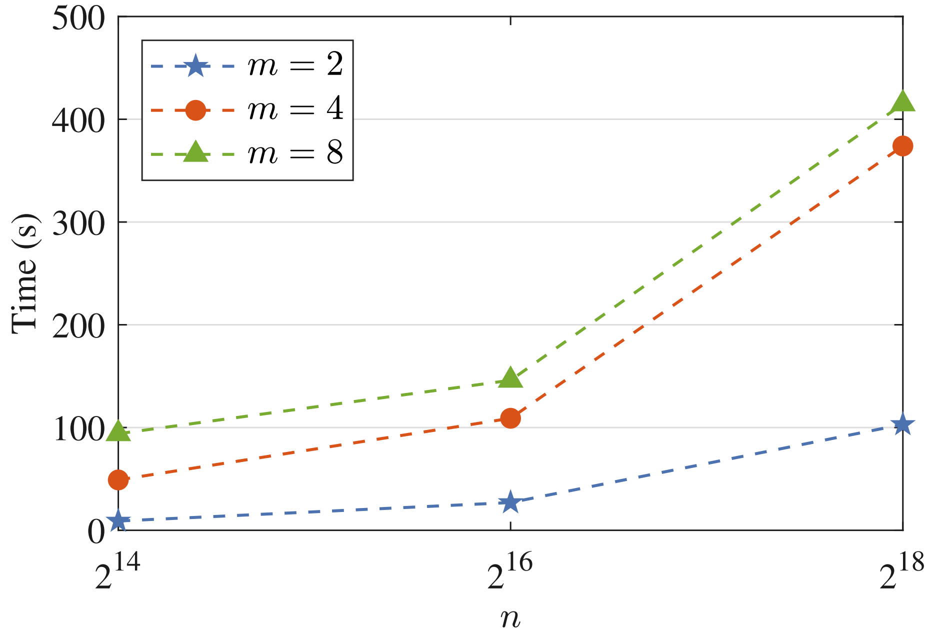
costs of $\mathsf{MkPIR}^{\mathbf{F}}$ when $t = 2^{10}$. Similar to the algorithm scheme based on OSI, the bottlenecks in our scheme regarding computational and communication costs are all related to the first round of communication. Since we employed polynomials of a sufficiently high order, with a degree sufficient to support computations of size up to 2^{14} and below, no additional communication operations were required, and the communication costs remained consistent.

7 Conclusion

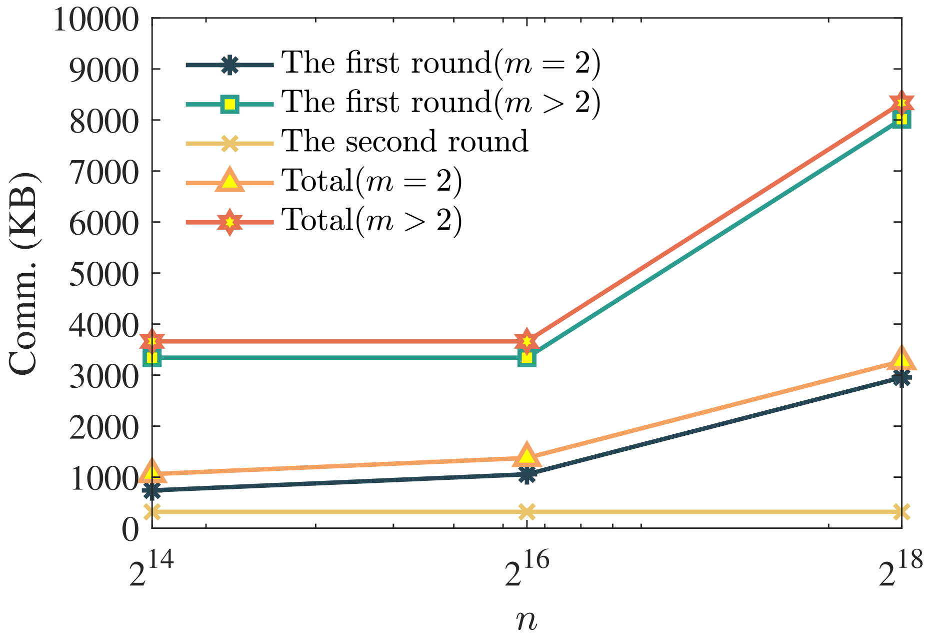
Orienting to data format $\{Index,\ Value,\ Keywords\}$, we propose Mk-PIR, a private information retrieval (PIR) enabling a client to retrieve records matching all queried keywords. Mk-PIR operates via two phases using functional Keyword PIR for keyword-to-index transformation and Index PIR for value retrieval. We introduce two constructions: $\mathsf{MkPIR}^{\mathbf{I}}$ (based on oblivious set intersection) and $\mathsf{MkPIR}^{\mathbf{F}}$ (based on private subset determination). The existence of a single-interaction Mk-PIR construction remains an open problem.

References

1. Ahmad, I., Agrawal, D., Abbadi, A.E., Gupta, T.: Pantheon: Private retrieval from public key-value store. Proceedings of the VLDB Endowment **16**(4), 643–656 (2022)
2. Ali, A., Lepoint, T., Patel, S., Raykova, M., Schoppmann, P., Seth, K., Yeo, K.: Communication–computation trade-offs in pir. In: 30th USENIX security symposium (USENIX Security 21). pp. 1811–1828 (2021)
3. Angel, S., Chen, H., Laine, K., Setty, S.: Pir with compressed queries and amortized query processing. In: 2018 IEEE symposium on security and privacy (SP). pp. 962–979. IEEE (2018)
4. Beimel, A., Ishai, Y., Kushilevitz, E., Orlov, I.: Share conversion and private information retrieval. In: 2012 IEEE 27th Conference on Computational Complexity. pp. 258–268. IEEE (2012)
5. Beimel, A., Stahl, Y.: Robust information-theoretic private information retrieval. Journal of Cryptology **20**(3), 295–321 (2007)
6. de Castro, L., Lewi, K., Suh, E.: Whispir: Stateless private information retrieval with low communication. Cryptology ePrint Archive (2024)
7. Celi, S., Davidson, A.: Call me by my name: Simple, practical private information retrieval for keyword queries. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 4107–4121 (2024)
8. Chor, B., Gilboa, N.: Computationally private information retrieval. In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. pp. 304–313 (1997)
9. Chor, B., Gilboa, N., Naor, M.: Private information retrieval by keywords (1997)
10. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. Journal of the ACM (JACM) **45**(6), 965–981 (1998)
11. Corrigan-Gibbs, H., Henzinger, A., Kogan, D.: Single-server private information retrieval with sublinear amortized time. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 3–33. Springer (2022)

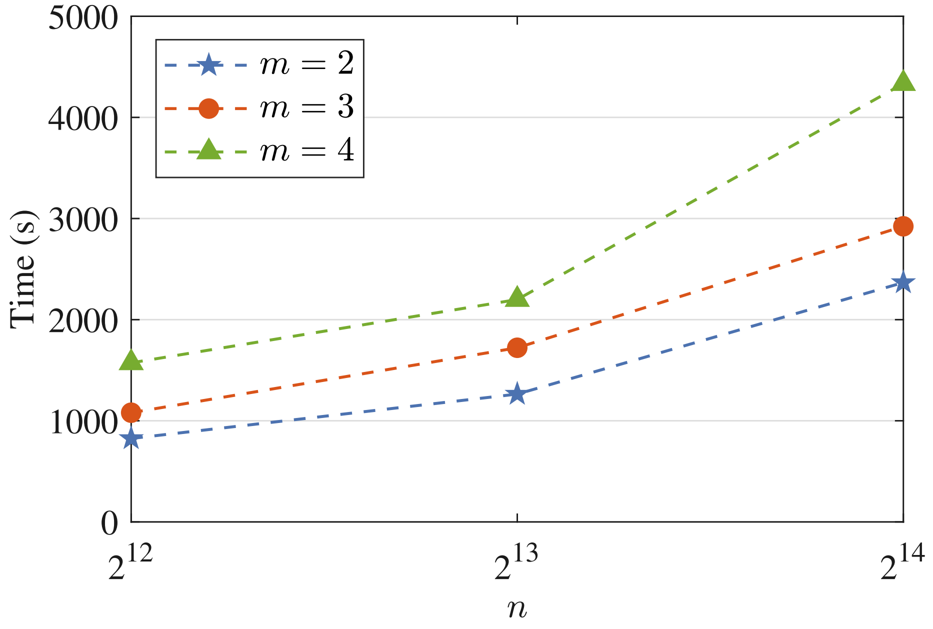


(a) Computation

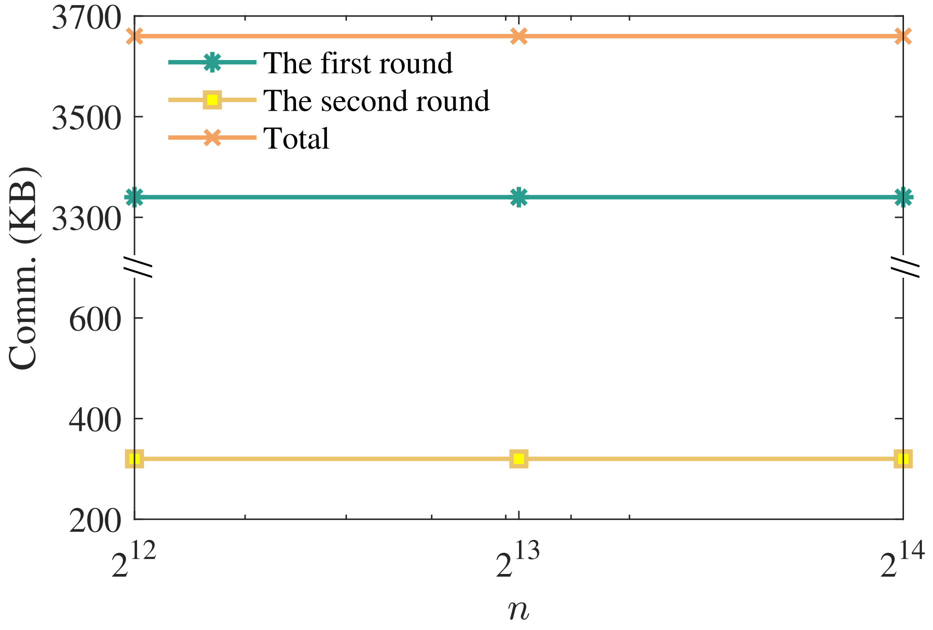


(b) Communication

Fig. 7: Overall costs of $\mathsf{MkPIR}^{\mathbf{I}}$



(a) Computation



(b) Communication

Fig. 8: Overall costs of $\mathsf{MkPIR}^{\mathbf{F}}$

12. Davidson, A., Pestana, G., Celi, S.: Frodopir: Simple, scalable, single-server private information retrieval. Proceedings on Privacy Enhancing Technologies (2023)
13. Dvir, Z., Gopi, S.: 2-server pir with subpolynomial communication. Journal of the ACM (JACM) **63**(4), 1–15 (2016)
14. Eriguchi, R., Kurosawa, K., Nuida, K.: Multi-server PIR with full error detection and limited error correction. In: 3rd Conference on Information-Theoretic Cryptography, ITC 2022, Cambridge, USA. pp. 1:1–1:20 (2022)
15. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive (2012)
16. Freedman, M.J., Ishai, Y., Pinkas, B., Reingold, O.: Keyword search and oblivious pseudorandom functions. In: Theory of Cryptography Conference. pp. 303–324. Springer (2005)
17. Garg, S., Hajiabadi, M., Ostrovsky, R.: Efficient range-trapdoor functions and applications: Rate-1 ot and more. In: Theory of Cryptography Conference. pp. 88–116. Springer (2020)
18. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Annual cryptology conference. pp. 75–92. Springer (2013)
19. Gödel, K.: Some metamathematical results on completeness and consistency, on formally undecidable propositions of principia mathematica and related systems i, and on completeness and consistency. J. van Heijenoort, From Frege to Godel, A source book in mathematical logic **1931**, 592–618 (1879)
20. Gong, T., Henry, R., Psomas, A., Kate, A.: More is merrier: Relax the non-collusion assumption in multi-server pir. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 4348–4366. IEEE (2024)
21. Hao, M., Liu, W., Peng, L., Zhang, C., Wu, P., Zhang, L., Li, H., Deng, R.H.: Practical keyword private information retrieval from key-to-index mappings. Cryptology ePrint Archive (2025)
22. Henzinger, A., Hong, M.M., Corrigan-Gibbs, H., Meiklejohn, S., Vaikuntanathan, V.: One server for the price of two: Simple and fast single-server private information retrieval. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 3889–3905 (2023)
23. Kales, D., Omolola, O., Ramacher, S.: Revisiting user privacy for certificate transparency. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 432–447. IEEE (2019)
24. Lazzaretti, A., Papamanthou, C.: Treepir: sublinear-time and polylog-bandwidth private information retrieval from ddh. In: Annual International Cryptology Conference. pp. 284–314. Springer (2023)
25. Lin, C., Liu, Z., Malkin, T.: Xspir: Efficient symmetrically private information retrieval from ring-lwe. In: European Symposium on Research in Computer Security. pp. 217–236. Springer (2022)
26. Liu, J., Li, J., Wu, D., Ren, K.: Pirana: Faster multi-query pir via constant-weight codes. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 4315–4330. IEEE (2024)
27. Mahdavi, R.A., Kerschbaum, F.: Constant-weight pir: Single-round keyword pir via constant-weight equality operators. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 1723–1740 (2022)
28. Melchor, C.A., Barrier, J., Fousse, L., Killijian, M.O.: Xpir: Private information retrieval for everyone. Proceedings on Privacy Enhancing Technologies pp. 155–174 (2016)

29. Menon, S.J., Wu, D.J.: Spiral: Fast, high-rate single-server pir via fhe composition. In: 2022 IEEE symposium on security and privacy (SP). pp. 930–947. IEEE (2022)
30. Menon, S.J., Wu, D.J.: Ypir: High-throughput single-server pir with silent preprocessing. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 5985–6002 (2024)
31. Mozaffari, H., Houmansadr, A.: Heterogeneous private information retrieval. In: Network and Distributed Systems Security (NDSS) Symposium 2020 (2020)
32. Mughees, M.H., Chen, H., Ren, L.: Onionpir: Response efficient single-server pir. In: Proceedings of the 2021 ACM SIGSAC conference on computer and communications security. pp. 2292–2306 (2021)
33. Mughees, M.H., Ren, L.: Vectorized batch private information retrieval. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 437–452. IEEE (2023)
34. Patel, S., Seo, J.Y., Yeo, K.: Don't be dense: Efficient keyword pir for sparse databases. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 3853–3870 (2023)
35. Pinkas, B., Schneider, T., Zohner, M.: Scalable private set intersection based on ot extension. ACM Transactions on Privacy and Security (TOPS) **21**(2), 1–35 (2018)
36. Wu, Z., Zhang, D., Li, Y., Han, X.: Pskpir: Symmetric keyword private information retrieval based on psi with payload. Cryptology ePrint Archive (2023)
37. Zhou, M., Lin, W.K., Tselekounis, Y., Shi, E.: Optimal single-server private information retrieval. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 395–425. Springer (2023)
38. Zhou, M., Park, A., Zheng, W., Shi, E.: Piano: extremely simple, single-server pir with sublinear server computation. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 4296–4314. IEEE (2024)