Artifact Evaluated NDSS
Available
Functional
Reproduced

BKPIR: Keyword PIR for Private Boolean Retrieval

Jie Song*†‡, Zhen Xu*✉, Yan Zhang*‡, Pengwei Zhan§, Mingxuan Li¶, Shuai Ma‖, Ru Xie*‡

*Institute of Information Engineering, Chinese Academy of Sciences
†Intelligent Policing Key Laboratory of Sichuan Province, Sichuan Police College
‡School of Cyber Security, University of Chinese Academy of Sciences
§Sangfor Technologies Inc.
¶School of Criminal Investigation, People's Public Security University of China
‖SKLCCSE Lab, Beihang University
songjie@scpolicec.edu.cn, {xuzhen, zhangyan, xieru}@iie.ac.cn, zhanpengwei@sangfor.com.cn,
limingxuan@ppsuc.edu.cn, mashuai@buaa.edu.cn

***Abstract*—Keyword Private Information Retrieval (Keyword PIR) enables users to retrieve data associated with specific keywords from a database while keeping their queries private. However, existing Keyword PIR schemes struggle to support the boolean retrieval model, which is essential for practical applications that require logical combinations of terms. This paper proposes a novel keyword PIR scheme leveraging advancements in homomorphic equality operations. It enables privacy-preserving retrieval over databases with many-to-many keyword-value mappings while supporting boolean operators for expressive search logic. Importantly, this extension preserves the core security guarantees of classical PIR. To the best of our knowledge, this is the first work to integrate keyword PIR with the boolean retrieval model.**

Experimental evaluation shows that our scheme achieves a communication cost reduction proportional to the total number of values in the many-to-many keyword-value database, along with aggregate query processing performance gains that scale linearly with the number of values. These improvements enhance its feasibility for real-world applications such as privacy-preserving web search and patent retrieval.

I. Introduction

Private Information Retrieval (PIR) [1] is a cryptographic method that allows users to query databases without revealing their search terms. Keyword PIR [2], for instance, can retrieve keywords while keeping them private, which is critical for sensitive tasks like patent searches. However, to handle complex search conditions in practical applications, keyword PIR must support boolean operators to combine multiple keywords. Without this capability, PIR cannot deliver the precise search results needed for many real-world tasks.

✉Zhen Xu is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230536
www.ndss-symposium.org

Boolean retrieval [3], [4] is a classic model in information retrieval, allowing users to form complex queries by combining keywords with operators like AND, OR, and NOT. This greatly improves the expressiveness and precision of search results. One important application of boolean retrieval in privacy-preserving contexts is web search, where users wish to query search engines without revealing their interests. Web search engines typically rely on an inverted index, where each webpage contains multiple keywords, and each keyword may be associated with multiple webpages, forming a many-to-many relationship. To achieve both privacy and accuracy, a keyword PIR scheme must effectively handle such many-to-many mappings while supporting boolean logic to refine search results. However, current keyword PIR [5]–[8] schemes provide limited support for boolean retrieval. The challenge arises from an implicit assumption that these schemes handle databases with a one-to-one relationship between keywords and values, where querying a keyword yields only a single associated value rather than a set. Boolean retrievals, however, are based on many-to-many relationships. As a result, existing keyword PIR schemes struggle to handle these relationships, complicating boolean operations and reducing their usefulness in practical, real-world scenarios.

One intuitive approach to make keyword PIR applicable to many-to-many databases is to replicate keywords across multiple entries in a linear structure, associating each keyword with different values. While this strategy might allow compatibility with some existing keyword PIR schemes, it introduces significant drawbacks, such as increased storage requirements due to redundant keyword and value pairs and the generation of multiple redundant response payloads, which may further reduce efficiency or even malfunction.

To address the challenges of privacy-preserving boolean retrieval, we propose Many-to-Many Keyword PIR (MMKPIR) and its extension, Boolean-enhanced Keyword PIR (BKPIR). MMKPIR efficiently supports many-to-many keyword-value relationships in keyword PIR, significantly reducing storage, computation, and communication overhead. Building on this, BKPIR enables logical combinations of queried keywords, enhancing the expressiveness of private search. Together, these schemes make privacy-preserving boolean retrieval practical for real-world database settings.

The main contributions of this paper are as follows:

- We identify the limitations of existing keyword PIR schemes in handling many-to-many relationships and propose MMKPIR, which efficiently supports such retrievals. MMKPIR reduces response communication costs by up to a factor of m and achieves speedups proportional to m under the same database scale, where m denotes the total number of distinct values;
- MMKPIR supports extremely large keyword domains at low cost, enabling the use of standardized hash functions for on-the-fly keyword encoding without requiring pre-negotiated mappings. This design supports arbitrary-length keywords, eliminates coordination overhead, and ensures practical keyword scalability;
- We further propose BKPIR, an extension of MMKPIR that supports expressive boolean queries. BKPIR enables a complete set of logical operators over keywords while maintaining performance comparable to MMKPIR. To the best of our knowledge, it is the first keyword PIR scheme that supports private boolean retrieval;
- Both MMKPIR and BKPIR are single-server, single-round PIR protocols that maintain fixed query communication, support large keyword domains, and offer low preprocessing overhead—making them practical for dynamic, real-world database.

II. Background and Related Work

A. Boolean Retrieval

Boolean retrieval [3], [4] enables flexible keyword-based search by combining search terms with boolean operators: AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). This retrieval model allows users to express complex information needs using logical conditions.

Formally, consider a set of keywords $KW = \{kw_1, \ldots, kw_n\}$ and a set of documents $V = \{v_1, \ldots, v_m\}$, where each document contains a subset of keywords from KW. In this model, a keyword $kw_i \in KW$ may appear in multiple documents $v_j \in V$, and each document v_j may contain multiple keywords, resulting a many-to-many relationship between keywords and documents. A boolean query Q is composed of keywords connected by logical operators. A document satisfies Q if it fulfills all specified logical conditions.

For example, consider an inverted index mapping keywords to documents: $kw_1 \rightarrow \{v_1, v_3, v_5\}, kw_2 \rightarrow \{v_2, v_3, v_6\}, kw_3 \rightarrow \{v_4, v_5, v_6\}$, the boolean query $Q = (kw_1 \vee kw_2) \wedge \neg kw_3$ retrieves documents that contain either kw_1 or kw_2, but not kw_3. The result is: $(\{v_1, v_3, v_5\} \cup \{v_2, v_3, v_6\}) \setminus \{v_4, v_5, v_6\} = \{v_1, v_2, v_3\}$. This demonstrates how boolean operators refine retrieval by including or excluding documents based on keyword presence, making this model effective for structured search tasks.

B. Keyword PIR under Different Relational Structures

Keyword PIR. In a keyword PIR protocol, a client retrieves the value linked to a keyword from a server database without revealing the keyword. Given a query $Q(kw_i)$, the server cannot distinguish it from any other $Q(kw_j)$ with non-negligible probability.

Data relationship. Let $\mathcal{DB}(n, m)$ denote a keyword-value database used in boolean retrieval, where $n = |KW|$ and $m = |V|$ are the numbers of distinct keywords and values, respectively. The database defines a mapping $f : KW \rightarrow \mathcal{P}(V)$, where each keyword $kw \in KW$ is associated with a subset of values $f(kw) \subseteq V$. We categorize the structure of f into four typical relationship types:

- One-to-one: Each keyword maps to a single value, and the mapping is injective, i.e., $|f(kw)| = 1$ and $f(kw_i) \neq f(kw_j)$ for $kw_i \neq kw_j$. The query complexity is $O(n)$, as the retrieval operates over a linear structure. All keyword PIR schemes support this setting.
- Many-to-one: Multiple keywords map to the same value, but each keyword still maps to exactly one value, i.e., $|f(kw)| = 1$, without injectivity. Retrieval remains straightforward with complexity $O(n)$, but duplicate values may appear across different keywords.
- One-to-many: Each keyword maps to a subset of values, and different keywords are associated with disjoint subsets, i.e., $f(kw_i) \cap f(kw_j) = \emptyset$ for all $kw_i \neq kw_j$. A query retrieves all values in $f(kw)$, with a complexity of $O(m)$, since the response size grows with the number of possible values. Most existing schemes (e.g., [6]–[9]) assume one-to-one or many-to-one mappings and fail in this setting—either by encountering conflicts or by returning only a single matching value. Modifying the response mechanism in [5] to output ciphertexts based on the products of selection vectors and payloads (instead of their sum) can ensure correctness, but increases communication overhead and risks structural leakage.
- Many-to-many: Keywords may map to arbitrary subsets of values, allowing overlaps, i.e., no constraint is imposed on f. This general structure induces a worst-case retrieval complexity of $O(n \cdot m)$, as all combinations between n keywords and m values may need to be evaluated. Even adaptable schemes like CwPIR [5] face scalability issues in this setting—challenges include database storage expansion (from size n up to $n \cdot m$), duplicate value returns across queries, and the risk of privacy leakage due to structure-preserving encodings.

To support efficient and private retrieval over such general relationship structures, we first introduce the Many-to-Many Keyword PIR (MMKPIR) scheme in Section IV, and extend it with boolean operator support to form the Boolean-Enhanced Keyword PIR (BKPIR) protocol in Section V.

C. Searchable Encryption

Keyword PIR shares similarities with Searchable Encryption (SE) techniques [10]–[12], which allow users to search over encrypted databases. A special class of structured SE schemes supports boolean queries using multiple keywords and logical operators [13]–[16]. However, SE schemes may leak search and access patterns [17]–[19], unless mitigated by techniques such as forward/backward privacy or Oblivious RAM (ORAM) [20], [21]. These mechanisms significantly increase complexity: clients must manage stateful trapdoors, perform re-encryption during updates, and often bear the encryption workload for both data and index.

In contrast, PIR—particularly computational PIR—operates over plaintext databases while offering stronger protection of user access and query patterns. Queries are stateless, support a single round of interaction, and impose minimal burden on clients. PIR can also support dynamic databases without requiring client-side coordination during updates.

Existing SE schemes face significant challenges in simultaneously achieving single-round queries, strong access/search pattern privacy, forward/backward privacy, and low client-side complexity. These trade-offs highlight fundamental distinctions between SE and PIR: SE prioritizes data confidentiality with structured query capabilities, while PIR emphasizes query privacy with minimal leakage and client burden.

D. Labeled PSI

Labeled PSI [22], [23] extends the standard Private Set Intersection (PSI) protocol [24], which enables two parties to compute the intersection of their private sets without revealing additional information. In Labeled PSI, each item in the sender's set is associated with a label, and the receiver obtains the labels corresponding to the intersection elements. Functionally, this can be viewed as a form of multi-query PIR, where the receiver privately retrieves multiple keywords along with their associated labels, without revealing the queried keywords.

While labeled PSI resembles PIR, its homomorphic evaluation of a high-degree polynomial that interpolates the payloads makes its database preprocessing stage highly time-consuming for large payloads, limiting its suitability for frequently updated databases. Additionally, due to the uniqueness of the interpolating polynomial, labeled PSI is constrained to one-to-one databases.

E. State-of-the-art keyword PIR

Recent advances in PIR have led to significant efficiency gains. SimplePIR [25] achieves sublinear communication and high throughput via dummy subsets. KsPIR [26] boosts FHE-based PIR throughput by over $10\times$ through parallelism and batched evaluations. Piano [27] further reduces server-side computation using a lightweight design with sublinear complexity. While these protocols excel in index-based PIR settings, they do not natively support keyword-based queries, and adapting them for keyword retrieval remains non-trivial.

CwPIR [5] was the first PIR scheme to use constant-weight coding for equality operations, reducing the multiplicative depth required for equality circuits to a practical level. Due to its straightforward design—generating selection vectors based on equality operations and obtaining results by multiplying these vectors with the payload inner products—CwPIR is workable for linear structures where keywords and values are repeated, as is common in many-to-many relationships. With appropriate modifications to the response method, CwPIR can support retrieval in these settings, although it is not highly efficient.

SparsePIR [6] is a method for building keyword PIR from a standard PIR. Its core technique involves encoding key-value pairs as functions of multiple database entries. Similar to many multi-query PIR schemes [28]–[30], SparsePIR encodes the database as a hypercube and uses homomorphic encryption and multi-dimensional query vectors for efficient querying. However, due to SparsePIR's use of hash functions to partition and encode the database, each keyword corresponds to a row in a matrix within a partition. This can lead to issues when the same keyword is associated with multiple values. As a result, SparsePIR cannot handle many-to-many relationships, as it may only return one value or encounter conflicts during the decoding process.

Oblivious Ciphertext Compression [8] is a technique that allows a server to compress ciphertexts without learning their plaintexts, while the client knows which ciphertexts encrypt zeroes. This technique uses additive homomorphism and random linear systems to achieve near-optimal lossless compression. It significantly reduces communication overhead in batch (keyword) PIR [6] and labeled PSI [23]. However, this technique does not fundamentally alter the underlying principles of the PIR and labeled PSI schemes it builds upon, meaning that the limitations of [6] and [23] in handling many-to-many relationships remain unresolved.

PIRANA [7], which builds on CwPIR [5] and batch code [31], incorporates constant-weight equality operators and batch processing to support multi-query, achieving greater efficiency than predecessors. However, PIRANA does not natively support keyword queries and requires integration with labeled PSI to do so. Consequently, LPSI-PIRANA, which supports keyword queries, inherits the shortcomings of labeled PSI and is limited to one-to-one databases.

Therefore, in the evaluation section VII, we selected CwPIR as the baseline for comparison, as it is currently the state-of-the-art keyword PIR scheme that can support many-to-many retrieval.

III. Preliminaries

A. Homomorphic Encryption

Homomorphic Encryption (HE) allows computation over encrypted data, producing ciphertexts that decrypt to the correct results of the corresponding plaintext operations. HE-based PIR is a well-established direction [29], [30], [32]–[34], where users send encrypted queries to a server, which homomorphically computes the result and returns encrypted responses—enabling single-server deployment and reducing communication costs.

Fully homomorphic encryption. A homomorphic encryption scheme that supports both addition and multiplication is known as Fully Homomorphic Encryption (FHE). FHE for arbitrary computation depths is expensive. To achieve a trade-off between computation depth and cost, leveled FHE (also known as Somewhat Homomorphic Encryption) schemes with fixed computation depths are commonly employed in practice. Typical leveled FHE schemes include FV [35], BGV [36], and CKKS [37].

Most leveled FHE schemes are based on a hard lattice problem called Learning With Errors(LWE) or its ring variant (RLWE) [38], [39]. Let $L = \mathbb{Z}[x]/(x^N + 1)$ be the polynomial ring, where N is the polynomial modulus degree, a power of 2. A plaintext message is encoded in $L_t = \mathbb{Z}_t[x]/(x^N + 1)$, where t is the plaintext modulus. The ciphertext is an array of polynomials in $L_g = Z_g[x]/(x^N + 1)$, where g is the coefficient modulus that affects how much noise a ciphertext can contain. The growth of noise during homomorphic operations is constrained by N and g, which determine the scheme's security level and computation depth.

Single instruction multiple data operation. Modern leveled FHE schemes (FV, BGV, CKKS) support Single Instruction Multiple Data (SIMD) operations [40], which allow encrypting a vector of values into a single ciphertext. Homomorphic operations are then performed slot-wise across these vector elements, enabling efficient parallel computation. A ciphertext encodes up to N slots, where N is the polynomial modulus degree. Supported SIMD operations include homomorphic addition and multiplication (ciphertext-ciphertext and ciphertext-plaintext), negation, and cyclic rotation of slots. These operations allow element-wise computation across all slots without additional cost.

In this paper, we adopt a SIMD-based leveled HE scheme $\mathcal{HE}$ to encode many-to-many keyword-value relationships as a one-to-one structure within SIMD slots. This design improves computation efficiency and facilitates PIR on complex relational data.

The SIMD homomorphic operation primitives used are:

- $CtCtAdd(\boldsymbol{c}_1, \boldsymbol{c}_2)$: Adds two ciphertexts slot-wise and returns the encrypted sum;
- $CtPtAdd(\boldsymbol{c}, \boldsymbol{p})$: Adds a ciphertext and a plaintext slot-wise, returning the encrypted result;
- $CtCtMul(\boldsymbol{c}_1, \boldsymbol{c}_2)$: Multiplies two ciphertexts slot-wise and returns the encrypted product;
- $CtPtMul(\boldsymbol{c}, \boldsymbol{p})$: Multiplies a ciphertext and a plaintext slot-wise, returning the encrypted product;
- $CtRotate(\boldsymbol{c}, l)$: Cyclically rotates the encrypted vector in $\boldsymbol{c}$ by l slots;
- $CtNegate(\boldsymbol{c})$: Returns the ciphertext of the slot-wise negation.

Except for multiplication operations, most of the above (addition, negation, rotation) incur minimal noise growth, ensuring the ciphertext remains decryptable after multiple operations.

B. Constant-weight Code

Constant-weight encoding. Constant-weight codes are binary vectors of fixed Hamming weight, meaning each codeword contains exactly k ones in a binary string of length w. A constant-weight code with parameters w and k is denoted as CW(w, k). The number of distinct such codewords is $\binom{w}{k}$, which determines the maximum number of values that can be uniquely represented. This number defines the codeword domain size h, i.e., the total number of distinct values encodable by the code. To ensure h encodable values, the condition $\binom{w}{k} \geq h$ must be satisfied.

Given parameters w and h, the smallest feasible k is selected such that all values in $\{1, \ldots, h\}$ can be uniquely mapped. The encoding function $CWEncode(h, w, k)$, described in Algorithm 1, produces a constant-weight vector $\boldsymbol{y} \in \{0, 1\}^w$ representing the input.

Algorithm 1 Function CWEncode(h, w, k)

```
Input: $h, w, k \in \mathbb{N}$ with $\binom{w}{k} \geq h$
 1: $\boldsymbol{y} \leftarrow 0^w$
 2: for $w' = w - 1, w - 2, \ldots, 0$ do
 3:     if $h \geq \binom{w'}{k}$ then
 4:         $\boldsymbol{y}[w'] = 1$
 5:         $h = h - \binom{w'}{k}$
 6:         $k = k - 1$
 7:     end if
 8:     if $k = 0$ then
 9:         break
10:     end if
11: end for
Output: $\boldsymbol{y} \in$ CW$(w, k)$
```

Constant-weight equality operator. CwPIR [5] is a keyword PIR scheme based on constant-weight codes. They propose a constant-weight equality operator to assess the equality of two constant-weight codes within a homomorphic encryption context. The basic idea is to multiply the bits of value 1 in one constant-weight code with the corresponding bits in the other constant-weight code, and then multiply all the intermediate results. If the final result is 1, the two constant-weight codes are deemed equal; otherwise, they differ. Alternatively, arithmetic methods, as demonstrated in Algorithm 2, are used for comparing two constant-weight codes over a field with a multiplicative inverse of $k!$. The constant-weight equality operator, which makes previously impractical homomorphic equality checks feasible, is still in its early stages of development.

Algorithm 2 Constant-weight Equality Operator

```
Input: $\boldsymbol{a}, \boldsymbol{b} \in CW(w, k)$
 1: $k' \leftarrow 0$ , $eq \leftarrow 1$
 2: for $i = 0$ to $w - 1$ do
 3:     $k' \leftarrow k' + \boldsymbol{a}[i] \cdot \boldsymbol{b}[i]$
 4: end for
 5: for $i = 0$ to $k - 1$ do
 6:     $eq \leftarrow eq \cdot (k' - i)$
 7: end for
 8: $eq \leftarrow eq/k!$
Output: $eq \in \{0, 1\}$
```

IV. Keyword PIR for Many-to-many Relationships

To implement keyword PIR on databases with many-to-many relationships, we introduce the Many-to-Many Keyword PIR (MMKPIR) scheme, which efficiently supports such relationships. In this section, we outline the design of this scheme.

A. Construction Overview

MMKPIR focuses on the foundational task of privately retrieving values associated with a single keyword kw^* from a many-to-many database. Real-world applications often require handling many-to-many databases $\mathcal{DB} = \{(kw_i, V_{kw_i})\}_{i=1}^n$, where $V_{kw_i} \subseteq V$ is a set of values linked to kw_i.

Existing keyword PIR schemes face two key limitations in this setting, structural incompatibility and efficiency bottlenecks. For example, linear database structures used in prior work (e.g., [6]) cannot natively represent many-to-many relationships, leading to conflicts when multiple values map to the same keyword. Naive extensions (e.g., [5]) incur $O(n \cdot m)$ complexity due to redundant payload storage and processing.

To resolve these challenges, MMKPIR restructures the original many-to-many database $\mathcal{DB}(n, m)$ into a keyword-to-valueset mapping $\mathcal{DB}_{\text{MMKPIR}} = \{(\mathbf{y}_i, \boldsymbol{p}_i)\}_{i=1}^n$, where each keyword kw_i is encoded into a constant-weight codeword $\mathbf{y}_i \in \{0,1\}^w$ with Hamming weight k, and its associated values $V_{kw_i} \subseteq V$ are encoded into a payload plaintext $\boldsymbol{p}_i$.

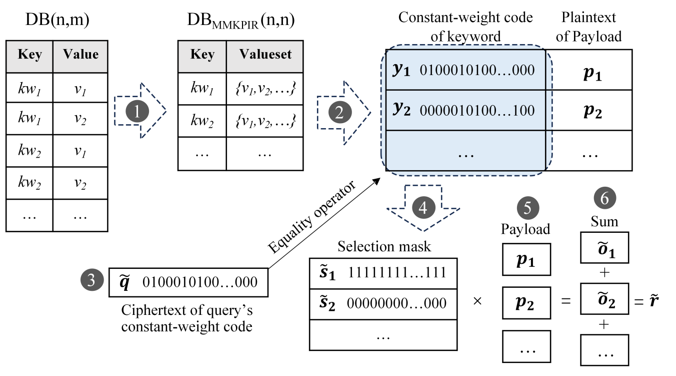
DB(n,m)
Key
Value
kw_1
v_1
kw_1
v_2
kw_2
v_1
kw_2
v_2
…
…
1
DB$_{\text{MMKPIR}}$(n,n)
Key
Valueset
kw_1
$\{v_1,v_2,\ldots\}$
kw_2
$\{v_1,v_2,\ldots\}$
…
…
2
Constant-weight code of keyword
Plaintext of Payload
$\boldsymbol{y}_1$ 0100010100…000
$\boldsymbol{p}_1$
$\boldsymbol{y}_2$ 0000010100…100
$\boldsymbol{p}_2$
…
…
Equality operator
4
5
Payload
6
Sum
3
$\tilde{\boldsymbol{q}}$ 0100010100…000
Ciphertext of query's constant-weight code
Selection mask
$\tilde{\boldsymbol{s}}_1$ 11111111…111
$\tilde{\boldsymbol{s}}_2$ 00000000…000
…
×
$\boldsymbol{p}_1$
$\boldsymbol{p}_2$
…
=
$\tilde{\boldsymbol{o}}_1$
+
$\tilde{\boldsymbol{o}}_2$
+
…
$= \tilde{\boldsymbol{r}}$

Fig. 1. Simple workflow of MMKPIR. Step 1: Convert $\mathcal{DB}(n, m)$ into $\mathcal{DB}_{\text{MMKPIR}}(n, n)$, where each row corresponds to a keyword and its value set. Step 2: Encode all keywords kw_i into constant-weight codewords $\boldsymbol{y}_i$, represent each value set V_{kw_i} as a payload vector $\boldsymbol{p}_i$ and pack it into an RLWE plaintext. Step 3: The client encodes the query keyword into constant-weight codeword and encrypts it into a ciphertext $\tilde{\boldsymbol{q}}$, and sends it to the server. Step 4: The server computes the selection mask and obtains the ciphertexts $(\tilde{\boldsymbol{s}}_1, \ldots, \tilde{\boldsymbol{s}}_n)$. Step 5: Conduct homomorphic multiplication between the selection ciphertext and the payload. Step 6: The server homomorphically aggregates the product $\tilde{\boldsymbol{o}}_i$ to obtain the corresponding payload $\tilde{\boldsymbol{r}}$ for the queried keyword and returns it to the client.

Inspired by BatchPIR's bucketization approach [29], MMKPIR groups each keyword with its value set into a linear structure of n rows. This avoids the redundancy of storing $n \times m$ entries while preserving the many-to-many mapping. Each payload $\boldsymbol{p}_i$ is encoded as an RLWE plaintext using SIMD packing, allowing batch homomorphic operations over the entire value set V_{kw_i}. Figure 1 illustrates the workflow of MMKPIR. This design achieves $O(n)$ structural storage complexity in RLWE plaintext rows, preserves $O(1)$ response size, and facilitates efficient SIMD parallelism.

B. Details of MMKPIR Protocol

Setup. This phase configures the database by converting the many-to-many structure of $n \times m$ keyword-value pairs into a keyword-valueset linear structure of size n.

Algorithm 3 Setup
```
Input: $\mathcal{DB}$, $w, k \in \mathbb{N}$
 1: Initialize an empty temporary storage $TMP$
 2: for each $(kw, v) \in \mathcal{DB}$ do
 3:     $h \leftarrow$ int of HashFunction$(kw)$
 4:     $TMP[h]$.append$(v)$
 5: end for
 6: for each $(h, V_{kw}) \in TMP$ do
 7:     $\boldsymbol{y} \leftarrow$ CWEncode$(h, w, k)$
 8:     $\boldsymbol{y} \leftarrow$ EncodeToPt$(\boldsymbol{y})$
 9:     $\mathcal{DB}_{MMKPIR}[\boldsymbol{y}] = V_{kw}$
10:     $\boldsymbol{p} \leftarrow$ EncodeToPt$(V_{kw})$
11:     $\mathcal{DB}_{MMKPIR}[\boldsymbol{y}] = \boldsymbol{p}$
12: end for
Output: $(\boldsymbol{y}, \boldsymbol{p}) \in \mathcal{DB}_{MMKPIR}$
```

Choose integers w and k such that $\binom{w}{k} \geq n$. Typically, w is set as the polynomial modulus degree N in our scheme. Given the many-to-many database $\mathcal{DB}$, each keyword in $\mathcal{DB}$ is treated as a bucket, with its associated values stored within. A hash function is chosen to compute the hash of each keyword, which is then encoded as a constant-weight code. This creates a new database, $\mathcal{DB}_{MMKPIR}$, where each keyword is uniquely represented by a constant-weight code. Each code maps to a value set V_{kw}, containing all values associated with that keyword in the original $\mathcal{DB}$. Algorithm 3 outlines the specific steps in this process. Here, $CWEncode(h, w, k)$ implements Algorithm 1 to generate a constant-weight code, while $EncodeToPt()$ encodes a vector into an RLWE plaintext.

It should be mentioned that the Setup stage mainly involves modifying the database structure and encoding the keywords and payloads, while the database remains in plaintext. This means that if the database is initially structured as $\mathcal{DB}_{MMKPIR}$, no structural transformation is needed; only the encoding of the database is required.

Algorithm 4 QueryGen
```
Input: $w, k \in \mathbb{N}$, $kw^*$
 1: $h \leftarrow$ int of HashFunction$(kw^*)$
 2: $\boldsymbol{q} \leftarrow$ CWEncode$(h, w, k)$
 3: $\tilde{\boldsymbol{q}} \leftarrow \mathcal{HE}$(EncodeToPt$(\boldsymbol{q})$)
Output: $\tilde{\boldsymbol{q}}$
```

QueryGen. This stage is executed on the client side, using the same constant-weight coding scheme, parameters, and hash function as the Setup stage to encode the queried keyword into a $CW(w, k)$ coding vector (as described in lines 1-2 of Algorithm 4). The vector is then encoded and encrypted into an RLWE ciphertext $\tilde{\boldsymbol{q}}$. Additionally, the query vector can be encrypted by symmetric-key encryption mode to reduce the upload communication cost of the query vector [33]. Algorithm 4 provides a detailed overview of the process.

Response. In this stage, the query ciphertext is homomorphically compared against the constant-weight encodings of all keywords in the database, producing a list of selection ciphertexts $\tilde{\boldsymbol{s}}$ of size n. For a matching keyword, the resulting ciphertext encodes a vector of all 1s; otherwise, it encodes a vector of 0s. Each selection ciphertext is then homomorphically multiplied with the corresponding value payload, yielding a list of ciphertexts $\tilde{\boldsymbol{p}}$, also of size n, where only the ciphertext associated with the queried keyword contains a valid payload; others encode all 0s. The final result ciphertext $\tilde{\boldsymbol{r}}$ is obtained by homomorphically aggregating the products of $\tilde{\boldsymbol{s}}$ and $\tilde{\boldsymbol{p}}$, i.e., computing $\tilde{\boldsymbol{o}}_i = \tilde{\boldsymbol{s}}_i \cdot \tilde{\boldsymbol{p}}_i$, then summing all $\tilde{\boldsymbol{o}}_i$. The server returns $\tilde{\boldsymbol{r}}$ to the client, who decrypts it to obtain the query result.

Algorithm 5 outlines the detailed steps. Note that this is a simplified baseline version. In practice, when $m < N$, ciphertext slots can be proportionally allocated based on the maximum value set size to improve packing efficiency. Specifically, if a keyword kw has $d \in [1, m]$ values, each value can occupy $\lfloor N/d \rfloor$ slots. Assuming each slot holds a payload of t_b bits, the total capacity per ciphertext is $N \cdot t_b$. When $m > N$, each value set requires $\lceil m/N \rceil$ ciphertexts. For simplicity, we assume m is a multiple of N when $m \geq N$, and defer detailed handling to Section IV-C.

Algorithm 5 Response

```
Input: $\tilde{\boldsymbol{q}}$, $k$, $(\boldsymbol{y}, \boldsymbol{p}) \in \mathcal{DB}_{MMKPIR}$, $N$, $n$
 1: for each $(\boldsymbol{y}, \boldsymbol{p}) \in \mathcal{DB}_{MMKPIR}$ do
 2:     $\tilde{\boldsymbol{u}} \leftarrow$ CtPtMul$(\tilde{\boldsymbol{q}}, \boldsymbol{y})$
 3:     $l \leftarrow 1$
 4:     while $l < N$ do
 5:         $\tilde{\boldsymbol{tmp}} \leftarrow$ CtRotate$(\tilde{\boldsymbol{u}}, l)$
 6:         $\tilde{\boldsymbol{u}}' \leftarrow$ CtCtAdd$(\tilde{\boldsymbol{u}}, \tilde{\boldsymbol{tmp}})$
 7:         $l \leftarrow l * 2$
 8:     end while
 9:     $\tilde{\boldsymbol{s}} \leftarrow$ CtPtMul$\left(\prod_{i=0}^{k-1}{}_{\text{CtCtMul}}\text{CtPtAdd}(\tilde{\boldsymbol{u}}', -i), k!^{-1}\right)$
10:     $\tilde{\boldsymbol{o}} \leftarrow$ CtPtMul$(\tilde{\boldsymbol{s}}, \boldsymbol{p})$
11:     $\tilde{\boldsymbol{r}} \leftarrow \sum_{i=1}^{n}{}_{\text{CtCtAdd}}\tilde{\boldsymbol{o}}_i$
12: end for
Output: $\tilde{\boldsymbol{r}}$
```

Lines 3–8 of Algorithm 5 perform cumulative slot-wise addition of ciphertext $\tilde{\boldsymbol{u}}$ using $\lceil \log_2 N \rceil$ calls to $CtRotate$ and $CtCtAdd$, yielding a ciphertext with identical cumulative sums in each slot. Line 9 is the implementation of Algorithm 2, which subtracts vectors of length N and each element is $i \in [0, k-1]$ from $\tilde{\boldsymbol{u}}$. It then executes $CtCtMul$ on the resulting k vectors, followed by $CtPtMul$ to multiply the result with the multiplicative inverse of $k!$ and get the selection ciphertext $\tilde{\boldsymbol{s}}$. Notably, large k may amplify ciphertext noise during repeated multiplications, leading to potential decryption errors. To mitigate this, a divide-and-conquer approach can reduce multiplication depth to $\lceil \log_2 k \rceil$.

C. MMKPIR with Large m and Payloads

The workflow of MMKPIR with large m and payloads (hereafter referred to as MMKPIRL) largely mirrors that of standard MMKPIR, with modifications limited to three components: value set payload encoding during setup, inner product between selection ciphertexts and payloads, and aggregation of results during the response phase. The workflow is illustrated in Figure 2.

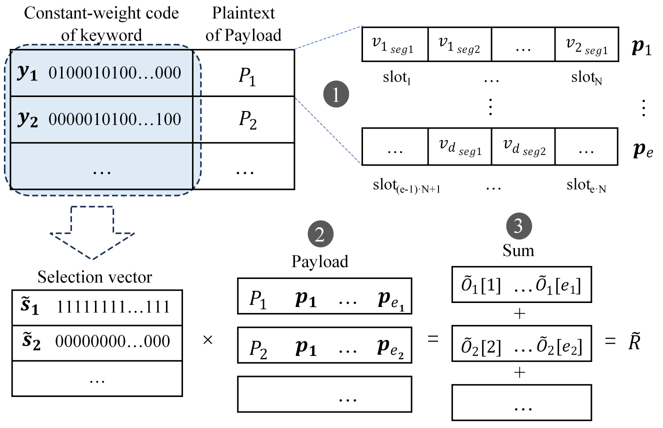
Constant-weight code of keyword
Plaintext of Payload
$\boldsymbol{y_1}$ 0100010100…000
P_1
$\boldsymbol{y_2}$ 0000010100…100
P_2
…
…
$v_{1\ seg1}$
$v_{1\ seg2}$
…
$v_{2\ seg1}$
$\boldsymbol{p}_1$
slot$_1$
…
slot$_N$
1
…
$v_{d\ seg1}$
$v_{d\ seg2}$
…
$\boldsymbol{p}_e$
slot$_{(e-1)\cdot N+1}$
…
slot$_{e\cdot N}$
Selection vector
$\tilde{\boldsymbol{s}}_1$ 11111111…111
$\tilde{\boldsymbol{s}}_2$ 00000000…000
…
×
2
Payload
P_1 $\boldsymbol{p_1}$ … $\boldsymbol{p}_{e_1}$
P_2 $\boldsymbol{p_1}$ … $\boldsymbol{p}_{e_2}$
…
=
3
Sum
$\tilde{O}_1[1]$ …$\tilde{O}_1[e_1]$
+
$\tilde{O}_2[2]$ …$\tilde{O}_2[e_2]$
+
…
= $\tilde{R}$

Fig. 2. Workflow of MMKPIRL. Key adaptations from MMKPIR include: Difference 1: When the number of values in a set exceeds m, or when the total payload size (number of values × payload size) surpasses the capacity of a single RLWE plaintext, multiple RLWE plaintexts are required. Specifically, payloads from d values are partitioned across e plaintexts, utilizing $e \times N$ slots for distributed storage. Difference 2: Since payloads are now represented as lists rather than single plaintexts, the selection ciphertext interacts with these payload lists via inner product computations. Difference 3: To reconcile lists $\tilde{O}$ of differing lengths during homomorphic aggregation, the framework pads shorter lists with zero-encoded plaintexts until all lists match the maximum length e^*. This process is detailed in line 5 of Algorithm 7. The final aggregated result is a ciphertext list $\tilde{R}$ of length $\max(e^*)$.

Setup. Since each slot in a RLWE ciphertext has a finite capacity for carrying payloads, certain retrieval tasks with large payloads may not be feasible to be accommodated by a single slot or ciphertext. Consequently, it becomes necessary to adjust the encoding scheme of the database during the setup stage to facilitate MMKPIR in retrieving content with large payloads. Algorithm 6 outlines the encoding strategy for databases with large m and payloads.

For each row $(\boldsymbol{y}, V_{kw})$ in the $\mathcal{DB}_{MMKPIR}$ derived from executing lines 1-9 of Algorithm 3, an appropriate number of RLWE plaintexts are allocated based on the payload size of each value in V_{kw}. Specifically, the aggregate payload size of the values in V_{kw} is divided by the maximal payload capacity $N \cdot t_b$ of an RLWE plaintext, which t_b is the bit length of each slot, and this quotient is rounded up to ascertain the requisite number of plaintexts e for the value set V_{kw}. d is the size of the value set V_{kw} for each keyword. Let $V_{kw} = \{v_1, \ldots, v_d\}$ denote the set of values associated with keyword kw, where each v_i is a value of predefined length. Initialize a payload array $[pl_1, pl_2, \ldots, pl_e]$ of length e, segment the payload of V_{kw} into e vectors pl of size $N \cdot t_b$, and encode these vectors as RLWE plaintexts. The resulting $\mathcal{DB}_{MMKPIRL}$ is encoding each keyword's associated value set V_{kw} in $\mathcal{DB}_{MMKPIR}$ as $P = [\boldsymbol{p}_1, \ldots, \boldsymbol{p}_e]$, that is, $(\boldsymbol{y}, P) \in \mathcal{DB}_{MMKPIRL}$.

Algorithm 6 Setup for Large m and Payloads

```
Input: $\mathcal{DB}, N, t_b$
 1: $\mathcal{DB}_{MMKPIR} \leftarrow$ Run lines 1-9 of Algorithm 3
 2: Init $E \leftarrow []$ to store the required amount of plaintext for each valueset
 3: for each $(\boldsymbol{y}, V_{kw}) \in \mathcal{DB}_{MMKPIR}$ do
 4:     $d \leftarrow |V_{kw}|$
 5:     $e \leftarrow \left\lceil \frac{1}{N \cdot t_b} \sum_{i=1}^{d} |v_i| \right\rceil$
 6:     $E$.append($e$)
 7:     Init $[pl_1, pl_2, \ldots, pl_e], |\boldsymbol{pl}| = N \cdot t_b$
 8:     Split $V = [v_1, \ldots, v_d]$ into $pl_1 \| \ldots \| pl_e$
 9:     for $i = 1, 2, \ldots, e$ do
10:         $\boldsymbol{p}_i \leftarrow$ EncodeToPt($pl_i$)
11:     end for
12:     $P = [\boldsymbol{p}_1, \ldots, \boldsymbol{p}_e]$
13:     $\mathcal{DB}_{MMKPIRL}[\boldsymbol{y}] = P$
14: end for
Output: $(\boldsymbol{y}, P) \in \mathcal{DB}_{MMKPIRL}$ , $E$
```

Response. In order to handle queries with large m and payloads, we have adapted Algorithm 5 to develop Algorithm 7. For each entry $(\boldsymbol{y}, P)$ in $\mathcal{DB}_{MMKPIRL}$, executing the equality operation as outlined in lines 2-9 of Algorithm 5 to derive the selection ciphertext $\tilde{\boldsymbol{s}}$, followed by $CtPtMul$ with the e plaintext payloads in P to get $\tilde{O}$. Due to the varying number of RLWE plaintexts required for the payloads of each entry in $\mathcal{DB}_{MMKPIRL}$, certain measures are necessary when summing the response ciphertexts (as specified in lines 4-5) to ensure that the length of the final returned ciphertext vector $\tilde{R}$ is fixed and encapsulates all retrieved content.

Algorithm 7 Response for Large m and Payloads

```
Input: $\tilde{\boldsymbol{q}}$, $N$, $k \in \mathbb{N}$, $n$, $E = [e_1, \ldots, e_n]$, $(\boldsymbol{y}, P) \in \mathcal{DB}_{MMKPIRL}$
 1: for each $(\boldsymbol{y}, P) \in \mathcal{DB}_{MMKPIRL}$ do
 2:     $\tilde{\boldsymbol{s}} \leftarrow$ Run lines 2-9 of Algorithm 5
 3:     $\tilde{O} \leftarrow$ CtPtMul($\tilde{\boldsymbol{s}}, P$)
 4:     for $x = 1, 2, \ldots, \max(E)$ do
 5:         $\tilde{R}[x] = \sum_{i=1}^{n}{}_{\text{CtCtAdd}} \begin{cases} \tilde{O}_i[x], & \text{if } x \leq e_i \\ 0, & \text{else} \end{cases}$
 6:     end for
 7: end for
Output: $\tilde{R}$
```

Algorithm 7 shares the same selection mask computation cost as Algorithm 5, since the input structure and equality test remain unchanged. The main difference lies in the inner product phase (lines 3–6), where each of the n selection ciphertexts is applied to a list of e plaintexts, resulting in an increased cost of $O(ne)$ from the corresponding $CtPtMul$ and $CtCtAdd$ operations.

MMKPIR efficiently retrieves multiple values per keyword, but real-world applications often require combining results across keywords using boolean logic. This motivates our extension to Boolean-enhanced Keyword PIR (BKPIR).

V. Boolean-enhanced Keyword PIR

Building on MMKPIR's support for many-to-many relationship, BKPIR introduces formal semantics for boolean operators, enabling queries of the form: $Q = B(kw_1, kw_2, \ldots, kw_z)$, where B represents boolean operators (AND/OR/NOT) applied to z keywords. BKPIR preserves the efficiency and many-to-many retrieval capabilities of MMKPIR while introducing formal semantics for logical composition of query terms.

A. Setup for BKPIR

Understanding BKPIR becomes straightforward once MMKPIR is grasped. During the Setup phase, the database $\mathcal{DB}(n, m)$ is processed to generate a set VM (representing the distinct values in $\mathcal{DB}$). Similar to how MMKPIR reshapes the keyword-value structure into $\mathcal{DB}_{MMKPIR}$, BKPIR reshapes $\mathcal{DB}$ into a keyword-valuemap structure, $\mathcal{DB}_{\text{BKPIR}} = \{(\mathbf{y}_i, \boldsymbol{vm}_i)\}_{i=1}^{n}$. In this structure, $\mathbf{y}_i$ represents constant-weight code for kw_i, each valuemap $\boldsymbol{vm}_i \in \{0,1\}^m$ is a multi-hot encoded vector representing values from VM, where multi-hot encoding represents the presence of multiple values in a document using a binary vector (e.g., [1, 0, 1] indicates values 1 and 3 are present). Each $\boldsymbol{vm}_i$ is packed into an RLWE plaintext for SIMD operations. Algorithm 8 outlines the BKPIR setup procedure, assuming $m \leq N$ for simplicity.

Algorithm 8 Setup for BKPIR

```
Input: $\mathcal{DB}$, $w, k \in \mathbb{N}$
 1: $VM \leftarrow []$
 2: for each $(kw, v) \in \mathcal{DB}$ do
 3:     if $value \notin VM$ then
 4:         $VM$.append($v$)
 5:     end if
 6: end for
 7: $(\boldsymbol{y}, V_{kw}) \in \mathcal{DB}_{MMKPIR} \leftarrow$ Run lines 1-9 of Algorithm 3
 8: for each $(\boldsymbol{y}, V_{kw}) \in \mathcal{DB}_{MMKPIR}$ do
 9:     $\boldsymbol{vm} \leftarrow []$
10:     for each $v \in VM$ do
11:         if $v \in V_{kw}$ then
12:             $\boldsymbol{vm}$.append($1$)
13:         else
14:             $\boldsymbol{vm}$.append($0$)
15:         end if
16:     end for
17:     $\mathcal{DB}_{BKPIR}[\boldsymbol{y}] = \boldsymbol{vm}$
18:     $\boldsymbol{p}_{vm} \leftarrow$ EncodeToPt($\boldsymbol{vm}$)
19:     $\mathcal{DB}_{BKPIR}[\boldsymbol{y}] = \boldsymbol{p}_{vm}$
20: end for
Output: $(\boldsymbol{y}, \boldsymbol{p}_{vm}) \in \mathcal{DB}_{BKPIR}, VM$
```

When $m > N$, the valuemap VM of length m is divided into $f = \lceil m/N \rceil$ vectors, each of length N. These vectors are then encoded into RLWE plaintexts using $EncodeToPt$. The resulting database, $\mathcal{DB}_{BKPIRL}$, associates each keyword with f such plaintexts, which are concatenated to form P_{vm}, representing the complete valuemap for the keyword. Algorithm 9 provides the details.

Algorithm 9 Setup for BKPIR with large m

```
Input: $\mathcal{DB}$, $w, k \in \mathbb{N}$
1: $\boldsymbol{vm}, VM \leftarrow$ Run lines 1-17 of Algorithm 8
2: $f \leftarrow \lceil m/N \rceil$
3: for each $(\boldsymbol{y}, \boldsymbol{vm}) \in \mathcal{DB}_{BKPIRL}$ do
4:     Split $\boldsymbol{vm}$ into $\boldsymbol{vm}'_1 \| \dots \| \boldsymbol{vm}'_f$
5:     for $i = 1, 2, \dots, f$ do
6:         $\boldsymbol{p}_i \leftarrow$ EncodeToPt($\boldsymbol{vm}'_i$)
7:     end for
8:     $P_{vm} \leftarrow [\boldsymbol{p}_1, \dots, \boldsymbol{p}_f]$
9:     $\mathcal{DB}_{BKPIRL}[\boldsymbol{y}] \leftarrow P_{vm}$
10: end for
Output: $(\boldsymbol{y}, P_{vm}) \in \mathcal{DB}_{BKPIRL}, VM, f$
```

B. BKPIR with Logical Operator AND

Response for BKPIR-AND. Once the database $\mathcal{DB}_{BKPIR}$ is set up, BKPIR enables advanced querying with logical operators that combine multiple keywords. For instance, to retrieve values associated with both $keyword_1$ and $keyword_2$, two query ciphertexts, $\tilde{\boldsymbol{q}}_1$ and $\tilde{\boldsymbol{q}}_2$, are generated using Algorithm 4. The server processes these ciphertexts using the BKPIR response algorithm supporting the AND operator, producing the response ciphertexts $\tilde{\boldsymbol{r}}_{AND}$. Details are provided in Algorithm 10.

Note that queried keywords can be combined from different databases, such as $\mathcal{DB}_{BKPIR_1}$ and $\mathcal{DB}_{BKPIR_2}$, provided they share the same valuemap VM. This functionality can also be extended to more than two keywords and databases, though for simplicity, this paper assumes all query keywords come from the same $\mathcal{DB}_{BKPIR}$.

Algorithm 10 Response for BKPIR-AND

```
Input: $\tilde{\boldsymbol{q}}_1, \tilde{\boldsymbol{q}}_2$, $\mathcal{DB}_{BKPIR}$, $N$, $k \in \mathbb{N}$, $n$, $VM$
1: for each $(\boldsymbol{y}, \boldsymbol{p}_{vm}) \in \mathcal{DB}_{BKPIR}$ do
2:     $\tilde{\boldsymbol{r}}_1 \leftarrow$ Run Algorithm 5 lines 2-11    (input $\tilde{\boldsymbol{q}}_1$)
3:     $\tilde{\boldsymbol{r}}_2 \leftarrow$ Run Algorithm 5 lines 2-11    (input $\tilde{\boldsymbol{q}}_2$)
4:     $\tilde{\boldsymbol{r}}_1 \wedge \tilde{\boldsymbol{r}}_2 \leftarrow$ CtCtMul($\tilde{\boldsymbol{r}}_1, \tilde{\boldsymbol{r}}_2$)
5:     $\boldsymbol{p} \leftarrow$ EncodeToPt($VM$)
6:     $\tilde{\boldsymbol{r}}_{AND} \leftarrow$ CtPtMul($\tilde{\boldsymbol{r}}_1 \wedge \tilde{\boldsymbol{r}}_2, \boldsymbol{p}$)
7: end for
Output: $\tilde{\boldsymbol{r}}_{AND}$
```

Algorithm 10 is derived from Algorithm 5 with minor adjustments. In this version, parameters such as $\mathcal{DB}_{BKPIR}$, along with the query ciphertexts $\tilde{\boldsymbol{q}}_1$ and $\tilde{\boldsymbol{q}}_2$, are input into lines 2-11 of Algorithm 5, producing $\tilde{\boldsymbol{r}}_1$ and $\tilde{\boldsymbol{r}}_2$. These ciphertexts correspond to the valuemaps for $keyword_1$ and $keyword_2$, respectively. Since the valuemap employs a multi-hot encoding, the $CtCtMul$ operation on $\tilde{\boldsymbol{r}}_1$ and $\tilde{\boldsymbol{r}}_2$ results in the ciphertext of the multi-hot encoded intersection of the queried keywords. This represents the positions in VM where values corresponding to both $keyword_1$ AND $keyword_2$ exist. Finally, applying $CtPtMul$ to $\tilde{\boldsymbol{r}}_1 \wedge \tilde{\boldsymbol{r}}_2$ and the payload RLWE plaintext of VM yields the response ciphertext $\tilde{\boldsymbol{r}}_{AND}$.

Algorithm 10 doubles the selection cost of Algorithm 5 due to two keyword queries, while maintaining the same $O(n)$ inner product cost, plus one extra $CtCtMul$ for intersection.

Response for BKPIR-AND with Large m and Payloads. BKPIR also supports queries where both m and the payload are large. As described in Algorithm 9, the valuemap of length $m > N$ is encoded into f separate plaintexts. Similarly, the payload for each value in VM is divided and encoded into the corresponding number of payload plaintexts. The intersection between the valuemaps for the queried keywords $keyword_1$ and $keyword_2$ is computed by performing $CtCtMul$ on their ciphertexts. Next, $CtPtMul$ is applied between this intersection and the payload plaintexts, producing a ciphertext matrix $\tilde{\boldsymbol{R}}$ that contains the matching data.

Algorithm 11 Response for BKPIR-AND with Large m and Payloads

```
Input: $\tilde{\boldsymbol{q}}_1, \tilde{\boldsymbol{q}}_2, N, k \in \mathbb{N}, n, t_b, VM, \mathcal{DB}_{BKPIRL}$
1: $f \leftarrow \lceil m/N \rceil, m \leftarrow |VM|, l \leftarrow \lceil \frac{\max(|v_1|,\dots,|v_m|)}{t_b} \rceil$
2: Init $[V_1, V_2, \dots, V_f]$: Reallocate $VM = [v_1, \dots, v_m]$ into
   $[V_1, \dots, V_f]$
3: for $i = 1, 2, \dots, f$ do
4:     Init $[PL_{i,1} \| \dots \| PL_{i,l}]$: Split $V_i$'s payload into
       $[PL_{i,1} \| \dots \| PL_{i,l}]$
5:     for $j = 1, 2, \dots, l$ do
6:         $P_{i,j} \leftarrow$ EncodeToPt($PL_{i,j}$)
7:     end for
8: end for
9: for each $(\boldsymbol{y}, P_{vm} = [\boldsymbol{p}_1, \dots, \boldsymbol{p}_f]) \in \mathcal{DB}_{BKPIRL}$ do
10:     $\tilde{\boldsymbol{s}}_1 \leftarrow$ Run Algorithm 5 lines 2-9    (input $\tilde{\boldsymbol{q}}_1$)
11:     $\tilde{\boldsymbol{s}}_2 \leftarrow$ Run Algorithm 5 lines 2-9    (input $\tilde{\boldsymbol{q}}_2$)
12:     for $i = 1, 2, \dots, f$ do
13:         $\tilde{O}_{1_{i,z}} \leftarrow$ CtPtMul($\tilde{\boldsymbol{s}}_1, \boldsymbol{p}_i$)    $(z \in [1, n])$
            $\tilde{O}_{2_{i,z}} \leftarrow$ CtPtMul($\tilde{\boldsymbol{s}}_2, \boldsymbol{p}_i$)    $(z \in [1, n])$
14:     end for
15: end for
16: for $i = 1, 2, \dots, f$ do
17:     $\tilde{M}_{1_i} \leftarrow \sum_{z=1}^{n}{}_{\text{CtCtAdd}} \tilde{O}_{1_{i,z}}$,   $\tilde{M}_{2_i} \leftarrow \sum_{z=1}^{n}{}_{\text{CtCtAdd}} \tilde{O}_{2_{i,z}}$
18:     $\tilde{S}_i \leftarrow$ CtCtMul($\tilde{M}_{1_i}, \tilde{M}_{2_i}$)
19:     for $j = 1, 2, \dots, l$ do
20:         $\tilde{W}_{i,j} \leftarrow$ CtPtMul($\tilde{S}_i, P_{i,j}$)
21:     end for
22: end for
23: $\tilde{\boldsymbol{R}} \leftarrow \begin{bmatrix} \tilde{W}_{1,1} & \dots & \tilde{W}_{1,l} \\ \vdots & \ddots & \vdots \\ \tilde{W}_{f,1} & \dots & \tilde{W}_{f,l} \end{bmatrix}$
Output: $\tilde{\boldsymbol{R}}$
```

In Algorithm 11, the m values in VM are divided into $f = \lceil m/N \rceil$ vectors V_f, where each vector is of length N (assuming, for simplicity, that m is a multiple of N). Based on the largest payload size in VM, the payload of each value in V_f is split into $l = \lceil \frac{\max(|v_1|,\dots,|v_m|)}{t_b} \rceil$ segments, where l denotes the number of slots required for the value with the largest payload. These segments, denoted as $PL_{f,l}$, are then encoded into RLWE plaintexts $P_{f,l}$ for subsequent computations. Unlike in Algorithm 7, each plaintext here accommodates only one slot per element in VM to hold its payload, requiring l plaintexts to store the largest value's payload. Therefore, the payload size of each value should be as uniform as possible to minimize overhead.

For each entry in $\mathcal{DB}_{BKPIRL}$, the query ciphertexts $\tilde{\boldsymbol{q}}_1$ and $\tilde{\boldsymbol{q}}_2$ are used as inputs for the equality test, producing ciphertexts $\tilde{\boldsymbol{s}}_1$ and $\tilde{\boldsymbol{s}}_2$, which are then multiplied with the elements of the valuemap list $[\boldsymbol{p}_1, \ldots, \boldsymbol{p}_f]$ using $CtPtMul$. The results $\tilde{O}_1$ and $\tilde{O}_2$ contain the ciphertexts for the valuemap entries that match the query criteria, while the other entries remain as 0 ciphertexts. By separately summing $\tilde{O}_1$ and $\tilde{O}_2$ for all n entries, valid valuemap ciphertext vectors $\tilde{M}_1$ and $\tilde{M}_2$ are obtained. Performing $CtCtMul$ on $\tilde{M}_1$ and $\tilde{M}_2$ produces a selection ciphertext list $\tilde{S}$, which contains the ciphertext of the intersection $keyword_1 \wedge keyword_2$. Subsequently, applying $CtPtMul$ to this valuemap and the payload plaintexts $P_{f,l}$ yields a ciphertext matrix $\tilde{\boldsymbol{R}}$, which represents the response for BKPIR queries where $m > N$ and the maximum payload size exceeds t_b. The client can then decrypt these response ciphertexts and reconstruct the queried data by concatenating the non-zero data from slots with the same index f.

Compared to Algorithm 10, Algorithm 11 retains the same selection complexity but incurs a higher inner product cost of $O(nf + nl)$, due to aggregation across n selection masks over f valuemap segments and subsequent multiplications with l payload segments.

C. BKPIR with Logical Operator NOT

To exclude a specific $keyword_3$, from the query results, the NOT logical operator is employed. The implementation of the NOT operator in BKPIR follows a process similar to that of the AND operator, with the main difference occurring during the valuemap ciphertext calculation in the response stage, as described in Algorithm 12. BKPIR-NOT achieves logical negation by decrementing the ciphertext $\tilde{\boldsymbol{r}}$ by 1 and applying the $CtNegate$ operation. A subsequent $CtPtMul$ operation between the negated valuemap ciphertext and the RLWE plaintext of VM yields the result that excludes $keyword_3$.

Algorithm 12 Response for BKPIR-NOT

```
Input: $\tilde{\boldsymbol{q}}_3$, $\mathcal{DB}_{BKPIR}$, $N$, $k \in \mathbb{N}$, $n$, $VM$
1: for each $(\boldsymbol{y}, \boldsymbol{p}_{vm}) \in \mathcal{DB}_{BKPIR}$ do
2:     $\tilde{\boldsymbol{r}}_3 \leftarrow$ Run Algorithm 5 lines 2-11    (input $\tilde{\boldsymbol{q}}_3$)
3:     $\neg\tilde{\boldsymbol{r}}_3 \leftarrow$ CtNegate(CtPtAdd($\tilde{\boldsymbol{r}}_3, -1$))
4:     $\boldsymbol{p} \leftarrow$ EncodeToPt($VM$)
5:     $\tilde{\boldsymbol{r}}_{NOT} \leftarrow$ CtPtMul($\neg\tilde{\boldsymbol{r}}_3, \boldsymbol{p}$)
6: end for
Output: $\tilde{\boldsymbol{r}}_{NOT}$
```

It should be highlighted that both the NOT and the forthcoming OR operators in BKPIR are designed to accommodate queries with large m and payloads. Their underlying principles are similar to those of BKPIR-AND when dealing with large m and payloads, and thus are not reiterated here.

D. BKPIR with Logical Operator OR

To support queries such as $keyword_4 \vee keyword_5$, BKPIR-OR leverages an arithmetic construction of the OR operation. Algorithm 13 details the implementation. BKPIR-OR begins by applying a NOT operation on each keyword, achieved by decrementing the ciphertext $\tilde{\boldsymbol{r}}$ by 1 and applying the $CtNegate$ operation, thus negating the valuemap. The OR operation is then performed by multiplying the two negated valuemap ciphertexts using $CtCtMul$, followed by applying a NOT operation to the result.

Algorithm 13 Response for BKPIR-OR

```
Input: $\tilde{\boldsymbol{q}}_4, \tilde{\boldsymbol{q}}_5$, $\mathcal{DB}_{BKPIR}$, $N$, $k \in \mathbb{N}$, $n$, $VM$
1: for each $(\boldsymbol{y}, \boldsymbol{p}_{vm}) \in \mathcal{DB}_{BKPIR}$ do
2:     $\tilde{\boldsymbol{r}}_4 \leftarrow$ Run Algorithm 5 lines 2-11    (input $\tilde{\boldsymbol{q}}_4$)
3:     $\tilde{\boldsymbol{r}}_5 \leftarrow$ Run Algorithm 5 lines 2-11    (input $\tilde{\boldsymbol{q}}_5$)
4:     $\tilde{\boldsymbol{r}}'_4 \leftarrow$ CtNegate(CtPtAdd($\tilde{\boldsymbol{r}}_4, -1$))
5:     $\tilde{\boldsymbol{r}}'_5 \leftarrow$ CtNegate(CtPtAdd($\tilde{\boldsymbol{r}}_5, -1$))
6:     $\tilde{\boldsymbol{r}}_4 \vee \tilde{\boldsymbol{r}}_5 \leftarrow$ CtNegate(CtPtAdd(CtCtMul($\tilde{\boldsymbol{r}}'_4, \tilde{\boldsymbol{r}}'_5$), $-1$))
7:     $\boldsymbol{p} \leftarrow$ EncodeToPt($VM$)
8:     $\tilde{\boldsymbol{r}}_{OR} \leftarrow$ CtPtMul($\tilde{\boldsymbol{r}}_4 \vee \tilde{\boldsymbol{r}}_5, \boldsymbol{p}$)
9: end for
Output: $\tilde{\boldsymbol{r}}_{OR}$
```

In summary, the logical AND, NOT, and OR operations on the valuemap structure are realized through simple $CtPtAdd$, $CtNegate$, and $CtCtMul$ operations, enabling BKPIR to flexibly set query criteria for private boolean retrievals.

VI. Analysis

A. Security Analysis

Threat Model. Our PIR protocols are designed to ensure strong privacy guarantees in the presence of an honest-but-curious server, which may attempt to infer information from the encrypted queries, intermediate computations or returned ciphertexts. We analyze security from the perspectives of query and response privacy under the RLWE assumption [38].

Query Privacy. In both MMKPIR and BKPIR protocols, all keyword queries are encoded and encrypted under an RLWE-based homomorphic encryption scheme. The structure of the ciphertexts and the uniform evaluation of homomorphic operations ensure that the server cannot distinguish between queries for different keywords, even across multiple rounds. For boolean queries in BKPIR, as long as the logical structure of two queries is the same, their evaluation trace remains indistinguishable to the server. This achieves security analogous to indistinguishability under chosen keyword or boolean query attacks (IND-CKA/IND-CBQA), formal definitions and proofs of which are given in Appendix A.

Response Privacy. In our schemes, all encrypted responses are constructed from RLWE-based ciphertexts, which are semantically secure under the RLWE assumption. In the base MMKPIR and BKPIR protocols, each query response consists of a single ciphertext. Due to the fixed dimension of RLWE ciphertexts and the use of modulus switching to normalize noise, response ciphertexts are indistinguishable.

Security of MMKPIRL and BKPIRL. MMKPIRL and BKPIRL follow the same query construction and interaction patterns as their respective base protocols MMKPIR and BKPIR, inheriting their query privacy guarantees. That is, query ciphertexts remain semantically secure and do not leak the queried keyword.

To support large payloads and large m, both MMKPIRL and BKPIRL return multi-ciphertext responses. To prevent leakage from variable response sizes, MMKPIRL partitions each value set into fixed-size plaintext chunks and pads all responses to a common maximum length $\max(e^*)$, producing a fixed-length ciphertext list. BKPIRL similarly returns a fixed-size matrix based on the number of items m and the maximum value size $\max(|v_1|, \ldots, |v_m|)$. These normalizations ensure that response size and structure are independent of the queried keyword and its associated values. These design choices guarantee that ciphertext length and structure leak no information, thereby preserving privacy. Since ciphertexts remain semantically secure and the response pattern leaks no information, MMKPIRL retains IND-CKA security and BKPIRL preserves IND-CBQA security.

Symmetric Security. MMKPIR and BKPIR can be extended to symmetric PIR [41], protecting both query and database privacy. A secure symmetric PIR must ensure query indistinguishability between any two databases. However, BKPIR's multi-hot encoding may inadvertently expose structural information through payload ciphertexts, which return either an encrypted value or zero. For applications requiring dual privacy protection, our protocol can be extended to achieve symmetric privacy. We describe this adaptation for MMKPIR and BKPIR in Appendix B.

B. Correctness Analysis

The correctness of MMKPIR and BKPIR is primarily influenced by the collision rate of the hash function, the noise budget of ciphertexts, and the multiplicative depth of homomorphic operations. The hash function offers the basis for codewords in constant-weight code, and hash collisions can result in incorrect retrieval. The likelihood of these collisions depends on the hash distribution, the size of the keyword domain, and the size of the codeword domain h. Assuming a uniform hash distribution, the probability of a collision follows the birthday paradox. As described in Algorithm 1, the size of the codeword domain is determined by the parameters w and k. When $w = N$ is fixed, increasing k enlarges the codeword domain, thus reducing the collision probability. Figure 3 shows the relationship between k and codeword domain size, as well as k and multiplication depth. When k is a power of 2, the maximum codeword domain is achieved at the corresponding multiplication depth.

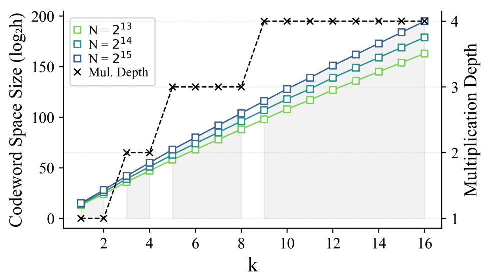

Fig. 3. Codeword domain size ($log_2 h$) and multiplication depth vs k.

Notably, the use of hash functions in our protocol is not for enhancing cryptographic security, but for enabling practical, on-the-fly keyword encoding without pre-negotiated mappings. For example, with $w = N = 2^{14}$ and $k = 11$, the codeword domain reaches $\approx 2^{128}$, matching a 128-bit truncated SHA-3 hash. This allows standardized hash functions to be directly used in both query and database processing, offering low collision risk and eliminating coordination overhead.

Additionally, as shown in Algorithm 2, k influences the multiplicative depth of homomorphic operations. A larger k increases depth, potentially exceeding the ciphertext's noise budget and causing decryption failure. The noise budget itself is determined by N, the plaintext modulus, and the coefficient modulus. Hence, maintaining correctness requires balancing k, N, and other RLWE parameters to ensure that the noise budget supports the required computation depth.

C. Security and Correctness Under Database Dynamics

The design of MMKPIR and BKPIR prioritizes privacy-preserving retrieval while allowing independent database evolution (e.g., adding keywords or modifying value sets). These updates are decoupled from query execution, ensuring security and correctness remain intact as long as public parameters remain stable.

Decoupling Updates from Protocol Execution. Database updates, such as introducing a new keyword or modifying a value set V_{kw}, occur exclusively in the preprocessing phase. The server re-encodes affected entries using unchanged public parameters, ensuring that homomorphic operations remain consistent. Clients can query the latest database without altering their algorithms, while the server processes queries uniformly, preserving both efficiency and simplicity.

Security Under Updates. The security of MMKPIR and BKPIR relies on IND-CKA and IND-CBQA security, which holds as long as public parameters remain fixed. The server, despite observing plaintext-encoded database elements (constant-weight codewords $\boldsymbol{y}_i$, RLWE-packed payloads $\boldsymbol{p}_i$, or valuemaps $\boldsymbol{vm}_i$), gains no advantage in reversing keyword or value relationships due to RLWE's cryptographic hardness. Since all operations on queries and intermediate results occur in the encrypted domain, updates introduce no new attack vectors.

If parameter expansion is required (e.g., increasing w and k for a larger keyword domain), a full re-encoding under new parameters is necessary. While computationally expensive, this maintains security, as each parameter set remains independent. The server cannot leverage knowledge from one parameter set to attack another, as the encoding schemes and ciphertexts are mathematically disjoint.

Correctness Under Updates. Correctness is preserved if new keywords are assigned unique constant-weight codewords $\boldsymbol{y}_{\text{new}}$ via a collision-resistant hash function. Ensuring $\binom{w}{k} \geq |KW|$ prevents codeword collisions and maintains accurate value mappings. If new parameters satisfy these conditions, correctness holds identically to the original protocol.

VII. Evaluation

A. Experimental Setup

Implementation. We implement our schemes in C++ using the Microsoft SEAL library (v4.0) [42] with the BFV homomorphic encryption scheme. SHA-3 is used for keyword hashing, with the resulting integers mapped into constant-weight codewords. To reduce communication overhead, the client encrypts query vectors in secret key mode, while the server applies modulus switching on the response ciphertexts to reduce their size without compromising the noise budget.

We run our experiments on an Ubuntu 22.04 virtual machine using 4 CPU cores (with a base frequency of 3.6 GHz and turbo frequency of 4.9 GHz) and 48 GB of memory. Our implementation is single-threaded. All experiments were repeated 5 times and the average results was taken.

Parameters. The degree of polynomial modulus $N \in \{2^{13}, 2^{14}, 2^{15}\}$, and the coefficient modulus uses the default configuration in SEAL to achieve 128-bit security. In order to obtain a higher noise budget in a freshly encrypted ciphertext and a lower noise budget consumption in a homomorphic multiplication, we choose t as a prime number as small as possible, which is 65537. As for the plaintext, the maximum bit size of each slot t_b is set to 16-bit. For convenience, in our experiment, the bit size of the value payload for $\mathcal{DB}_{MMKPIRL}$ and $\mathcal{DB}_{BKPIRL}$ is multiple of t_b, and m is a multiple of N.

B. Benchmarking MMKPIR

Baseline. As discussed in Section II-E, CwPIR [5] is the best available keyword PIR scheme capable of supporting many-to-many relationships, although its efficiency is not optimal. More recent schemes [6]–[8] provide higher efficiency in one-to-one settings, but they are unable to function properly in many-to-many settings due to fundamental design limitations. Thus, we adopt CwPIR as the primary baseline for evaluating.

To ensure a fair comparison, we normalize the evaluation by focusing on the number of keyword–value combinations, the common retrieval unit across schemes. In CwPIR and similar designs, each database row stores a single combination, whereas in MMKPIR, each row holds multiple combinations due to its ability to associate a keyword with multiple values. Consequently, comparisons must account not just for the number of rows but for the total number of stored combinations ($n \times m$) and the volume of the values (also referred to as the item size).

Performance of MMKPIR. The experimental results in Table I show a linear relationship between the number of database rows (n) and computation/communication costs. In boolean models that typically use inverted indexes, where the retrieval scale is defined as $M = n \times m$, the complexity of MMKPIR remains $O(n)$. Given that m and n are often of similar magnitude—with m typically larger in practice—it follows that $M \approx n^2$. As a result, MMKPIR achieves sublinear complexity with respect to M, specifically $O(n) = O(\sqrt{M})$. In addition, we intentionally set $N = m$ in this table to maximize SIMD slot utilization—when m is divisible by N, payload packing becomes more efficient, and each ciphertext can be fully utilized without padding overhead.

TABLE I
Performance of MMKPIR. ($k = 4$, item size is 2 bytes.)

N	Combin. ($n \times m$)	DB Size (MB)	Prep. time(s)	Server time(s)	Query com.(KB)	Resp. com.(KB)
2^{13}	128×2^{13}	2	0.18	15.90	211	100
	256×2^{13}	4	0.38	32.02		
	512×2^{13}	8	0.73	63.56		
2^{14}	128×2^{14}	4	0.61	91.71	891	218
	256×2^{14}	8	1.30	213.46		
	512×2^{14}	16	2.39	375.86		
2^{15}	128×2^{15}	8	2.10	533.81	3615	482
	256×2^{15}	16	4.03	1076.83		
	512×2^{15}	32	8.37	2166.67		

The choice of the polynomial modulus N represents a trade-off between functionality and efficiency. A larger N increases the ciphertext capacity, allowing more slots for value packing and a higher noise budget for deeper homomorphic multiplication. In our setting, this enables support for larger k and codeword spaces (see Figure 3) and more complex boolean logic over BKPIR (as evaluated in Table VI. For instance, a higher noise budget can accommodate a greater number of multiplication layers, which in turn supports larger codeword sizes and more expressive query capabilities. However, increasing N also incurs higher computational cost and communication overhead, as ciphertext sizes and operation times scale accordingly. Thus, we select N values that balance between acceptable performance and the desired retrieval expressiveness.

Notably, the communication costs reported throughout this paper refer to raw ciphertext sizes exchanged during a query round, and are independent of specific network conditions. Since all our proposed protocols support single-round query execution, their performance remains relatively robust to variations in latency or bandwidth.

TABLE II
Comparison of performance in single row databases.

N	Prep. time(s)		Server time(s)		Query com.(KB)		Resp. com.(KB)	
	CwPIR	MMKPIR	CwPIR	MMKPIR	CwPIR	MMKPIR	CwPIR	MMKPIR
2^{13}	0.04	0.001	0.41	0.14	216	211	103	100
2^{14}	0.04	0.004	2.26	0.82	913	891	224	218
2^{15}	0.05	0.014	12.81	4.11	3702	3615	493	482

Compare MMKPIR with Baseline. MMKPIR natively supports many-to-many retrieval and inverted index queries using an $O(n)$ SIMD-parallel structure, which structurally distinguishes it from CwPIR's row-wise design that requires $n \times m$ rows to support n keywords each mapping to m values. This structural difference underlies the asymptotic and empirical performance gap.

To ensure comparability, we first compare MMKPIR and CwPIR in a single-row setting with $n = 1$. With a database size of 16 KB, $k = 4$, an item size of 2 bytes for MMKPIR, and $m = 2^{13}$, the results in Table II demonstrate that MMKPIR achieves roughly 3× lower total runtime (preprocessing time + server time) and 3.5×–40× faster preprocessing than CwPIR. However, it is important to note that MMKPIR retrieves all m values associated with the keyword in a single query, while CwPIR retrieves only one value per query. Thus, to retrieve the complete set of m values for the same keyword, CwPIR requires m queries—multiplying its effective cost by a factor of m.

To further validate this scalability, we compare both schemes under increasing $n \in \{128, 256, 512\}$ and fixed $m = 2^{16}$, with a constant database size of 128 MB. Our experimental results reveal fundamental efficiency advantages in MMKPIRL's architecture. As shown in Table III, MMKPIRL achieves preprocessing times of (9.58, 19.04, 37.07) seconds and total runtimes of (29.81, 61.18, 120.14) seconds. In comparison, CwPIR's preprocessing times are (84.63, 167.67, 339.13) seconds, and total runtimes are (96.47, 192.89, 389.72) seconds. This translates to approximately 8.9× speedup in preprocessing and 3.2× speedup in total runtime for MMKPIRL. Based on this structural difference, we extrapolate the effective speedup as $3.2 \times m$. For example, in the setting with $m = 2^{16}$, this results in a theoretical $3.2 \times 2^{16} = 209,715\times$ speedup.

TABLE III
COMPARE CwPIR WITH MMKPIRL.

	Combin. ($n \times m$)	DB Size (MB)	Prep. time(s)	Server time(s)	Query com.(KB)	Resp. com.(KB)
MMKPIRL (N=2^{13}, k=4)	$128{\times}2^{16}$	16	1.23	17.23	211	805
		32	2.40	17.88		1610
		64	4.82	18.68		3220
		128	9.58	20.23		6441
	$256{\times}2^{16}$	32	2.36	33.82	211	805
		64	4.71	34.79		1610
		128	9.46	36.63		3220
		256	19.04	42.14		6442
	$512{\times}2^{16}$	64	4.50	64.71	211	805
		128	9.09	68.66		1610
		256	18.28	73.41		3221
		512	37.07	83.07		6441
CwPIR (N=2^{13}, k=4)	$128{\times}1$	16	10.92	9.03	216	721
		32	21.88	9.30		1339
		64	42.51	10.43		2678
		128	84.63	11.84		5358
	$256{\times}1$	32	20.81	17.30	216	721
		64	42.86	18.01		1339
		128	84.21	20.22		2678
		256	167.67	25.22		5362
	$512{\times}1$	64	41.83	34.08	216	721
		128	84.47	36.92		1340
		256	169.47	40.02		2679
		512	339.13	50.59		5359

On the other hand, the performance gap persists even when preprocessing costs are fully amortized over repeated queries in stable database environments. MMKPIRL demonstrates server times of (20.23, 42.14, 83.07) seconds, whereas CwPIR achieves (11.84, 25.22, 50.59) seconds. Although MMKPIRL exhibits approximately 0.6× the per-row efficiency of CwPIR in server time, this does not imply inferiority. Considering MMKPIRL's structural advantage in handling query keyword-value combinations, it theoretically achieves a speedup factor of approximately $0.6 \times 2^{16} = 39,321\times$ in server time compared to CwPIR.

This is consistent with the linear scaling behavior reported in Table 7 of [5], which confirms that CwPIR's cost grows nearly linearly with the number of combinations. We emphasize that this extrapolation serves as a theoretical projection, not a direct experimental result. In practice, as supported by Table 7 and Table 13 in [5], CwPIR becomes computationally impractical under large $n \times m$ settings, e.g., 128×2^{16}. In contrast, MMKPIR remains practical.

In addition, Table II and Table III show MMKPIR's server time improves in single-row but degrades in multi-row databases. This occurs because CwPIR employs oblivious expansion [29], whose expansion time scales with the codeword size w—independent of database scale—whereas MMKPIRL fixes $w = N$, the cost increases linearly with the row count. Consequently, keyword length does not affect MMKPIR's performance, but CwPIR requires fixed bit-length keywords due to its design constraints. We therefore standardized on 16-bit keywords throughout the tests.

In terms of communication, MMKPIR(L) uses a fixed-size query encapsulated in a single ciphertext, independent of the database size or keyword domain. In contrast, CwPIR's query size grows with the keyword domain, scaling as $O(k\sqrt{k!|S|} + k)$. On the response side, MMKPIR(L) aggregates the payloads row-wise, and returns a small, fixed-length vector of ciphertexts per row, determined by the maximum item size. This is significantly more efficient than CwPIR, which may return up to $n \times m$ ciphertexts in many-to-many scenarios, one per combination. As a result, MMKPIR(L) achieves substantial communication savings, especially when the item size is small and multiple values can be packed into a single ciphertext.

C. Benchmarking BKPIR

Performance of BKPIR. BKPIR is a variant of the MMKPIR scheme, introducing an additional inner product computation between the selection vector and the valuemap before calculating the payload inner product. This adjustment allows for logical operations between multiple queries. For single-keyword retrieval, BKPIR requires only one additional $CtPtMul$ operation compared to MMKPIR. Since $CtPtMul$ operations are very fast, this extra step has a negligible impact on performance. As a result, the server-side computational performance of BKPIR and MMKPIR is nearly identical. Therefore, the evaluation results for MMKPIR in Table II and Table III are also applicable to BKPIR for single-keyword retrieval (represented as BKPIR-Single).

Table IV provides a detailed breakdown of the database preprocessing time, server time, and communication overhead for BKPIR under various combinations ($n \times m$) with $k = 4$, $N = 2^{13}$, and an item size of 2 bytes. The performance of BKPIR-NOT is nearly identical to BKPIR-Single, as it only adds lightweight $CtPtAdd$ and $CtNegate$ operations.

TABLE IV
PERFORMANCE OF BKPIR WITH $k = 4$, $N = 2^{13}$ AND ITEM SIZE IS 2 BYTES.

Combinations $(n \times m)$			Prep. Time (s)	Server Run Time (s)									Query com.(KB)	Resp. com.(KB)
				Selection Vector		Logical Operation			Inner Product	Total				
			ALL	NOT	AND OR	NOT	AND	OR	ALL	NOT	AND	OR	ALL	ALL
128			0.002	16.36	32.84					16.37	32.87	32.87		
256	$\times$	2^{13}	0.002	32.60	65.53	0.0002	0.028	0.029	0.006	32.60	65.56	65.57	211	100
512			0.002	64.73	130.12					64.74	130.16	130.16		
128			0.004	16.36	32.84					16.37	32.91	32.91		
256	$\times$	2^{14}	0.004	32.60	65.53	0.0005	0.057	0.059	0.013	32.61	65.60	65.60	211	201
512			0.004	64.73	130.12					64.74	130.19	130.19		
128			0.008	16.36	32.84					16.39	32.97	32.98		
256	$\times$	2^{15}	0.009	32.60	65.53	0.0010	0.108	0.119	0.026	32.62	65.66	65.57	211	402
512			0.009	64.73	130.12					64.76	130.26	130.27		
128			0.018	16.36	32.84					16.41	33.11	33.12		
256	$\times$	2^{16}	0.018	32.60	65.53	0.0019	0.221	0.239	0.050	32.65	65.80	65.82	211	805
512			0.018	64.73	130.12					64.78	130.39	130.41		

The results indicate that both the database preprocessing time and the size of the selection vector increase linearly with n. The time spent computing the selection vector represents the most significant portion of the total server time and is primarily influenced by the parameters N and k. Table V demonstrates how different values of N and k impact server time, particularly for selection vector computation, with an item size of 2 bytes, $n = 128$, and $m = 2^{13}$.

TABLE V
SERVER RUNTIME UNDER DIFFERENT N AND k.

k	$N = 2^{13}$		$N = 2^{14}$		$N = 2^{15}$	
	MMKPIR	BKPIR-Single	MMKPIR	BKPIR-Single	MMKPIR	BKPIR-Single
2	11.10	11.32	65.36	65.89	411.35	410.15
4	16.64	16.36	91.15	91.11	534.69	535.91
8	-	-	143.49	142.71	775.61	775.13
16	-	-	248.38	249.51	1280.22	1278.90

Logical Operation Evaluation. BKPIR achieves functional completeness by supporting logical operators such as AND, OR, and NOT, enabling flexible boolean keyword retrieval. However, like other RLWE-based PIR schemes, BKPIR faces inherent limitations in computational efficiency and noise budget. Each homomorphic operation introduces additional noise, and decryption fails once the accumulated noise exceeds the permissible bound. While bootstrapping techniques [43]–[45] could in theory refresh ciphertexts and enable arbitrarily deep logic circuits, their computational cost remains prohibitively high for practical use.

To explore the practical limits of logic support under realistic conditions (i.e., without bootstrapping), we evaluated BKPIR under various (N, k) configurations and $m = N$, each item occupies 2 bytes. The parameter N controls the RLWE ciphertext size and thus the initial noise budget: increasing N results in larger ciphertexts and higher overhead, but also allows for more complex computation due to a greater noise margin. Meanwhile, the parameter k affecting how much budget is consumed before any logical operation can be applied.

In each test setting, we first determine the initial noise budget and measure the noise consumption incurred by keyword selection and inner-product computation. The difference represents the remaining budget available for logic gates. We then benchmark the noise cost of applying repeated logical operations over the same ciphertexts and record the maximum number of gates executable before decryption fails. This is reported in the "Max Ops" column in Table VI. For consistency, we report per-operation noise consumption using OR gates as a reference. Though OR operations are theoretically slightly more expensive than AND (involving additional negations and additions), our empirical results indicate that the difference is negligible. Therefore, the "Max Ops" count applies equally to AND and OR in practice. NOT operations, on the other hand, introduce minimal noise and can effectively be applied an unlimited number of times.

TABLE VI
LOGICAL OPERATION CAPACITY AND NOISE BUDGET

N	Init. Budget(bits)	k	Sel+IP Budget	Per-Op Budget	Max Ops	Per-Op Latency
		[1,2]	92		2	
2^{13}	153	[3,4]	118	28	1	0.03s
		[5,8]	-		-	
		[9,16]	-		-	
		[1,2]	93		9	
2^{14}	368	[3,4]	122	29	8	0.14s
		[5,8]	153		7	
		[9,16]	182		6	
		[1,2]	94		23	
2^{15}	804	[3,4]	126	30	22	0.71s
		[5,8]	157		21	
		[9,16]	187		20	

In addition to noise-related limits, we also profile the computational cost of logical operations. Table VI reports the average latency per logic gate evaluated in isolation. Results show that logic gates themselves incur very little overhead—ranging from 0.03 to 0.71 seconds per operation depending on N. This confirms that logical composition is not the primary performance bottleneck. Rather, the overall latency is dominated by the number of keywords involved in a boolean expression, since each keyword incurs a costly selection mask computation (as reflected in Algorithm 10 and Algorithm 11). Therefore, the true scalability constraint lies not in the number of logic gates, but in the number of distinct keywords participating in the expression.

D. Benchmarking under Large m and Payloads

We evaluated the variation in response communication overhead for BKPIR under different settings. The response communication volume of BKPIR depends on both m and the item size. Increasing m raises the value of f, leading to more ciphertexts and greater communication volume. Similarly, a larger item size increases l, also resulting in more ciphertexts. For both BKPIR and MMKPIR, query communication volume remains constant at one ciphertext, regardless of the number of keywords or database size. Importantly, the response communication volume is unaffected by n, even for very large values of n. Figure 4 illustrates the relationship between response communication costs, m, and item size under different N settings.

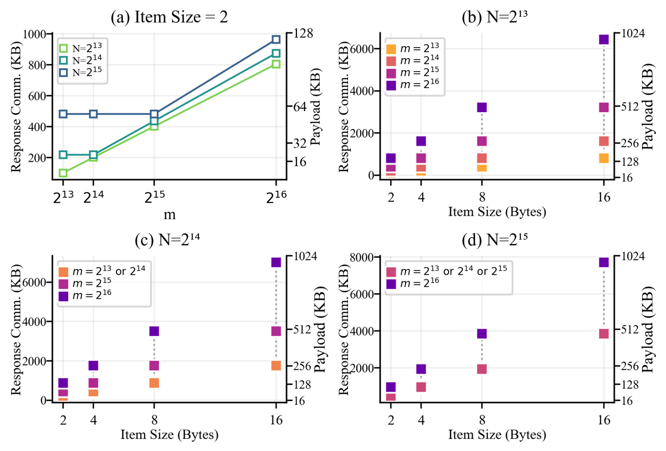

Fig. 4. Response Communication vs (a) m (b-d) item size.

BKPIR reshapes the database into a valuemap form, requiring only a single encoding of the payload, unlike MMKPIR, which needs to encode the valueset for each keyword. This makes BKPIR more efficient during the database preparation phase, especially for large m and payloads. Table VII compares MMKPIR and BKPIR for a single-row database with very large m and item sizes.

The response communication cost is directly influenced by the number of response ciphertexts, as their size remains constant once parameters like the polynomial modulus degree and coefficient modulus are set. Thus, the number of ciphertexts becomes the main factor determining communication cost.

In this context, $e = \left\lceil \frac{1}{N \cdot t_b} \sum_{i=1}^{d} |v_i| \right\rceil$ represents the number of ciphertexts required to store the payload in MMKPIRL, $f = \lceil m/N \rceil$ and $l = \lceil \max(|v_1|, \ldots, |v_m|)/t_b \rceil$ define the size of the response ciphertext matrix in BKPIRL. Here, f denotes the number of ciphertexts that need to be allocated for m values, and l indicates the number of ciphertexts needed to store the payload based on the size of each value. Assuming the valueset size is m and each v_i is of equal size, then $e = f \cdot l$.

TABLE VII
PERFORMANCE UNDER LARGE m AND PAYLOADS
$(n = 1, N = 2^{13}, k = 4)$

Method	Prep. Time(s)	Server time(s) Selection Vector	Logical Operation	Inner Product	Total	Query com.(KB)	Resp. com.(KB)
(m=2^{14}, item size is 4 bytes, row payload is 64KB, $e = f \times l = 2 \times 2$)							
MMKPIRL	0.004	0.13	-	0.02	0.15	211	402
BKPIRL-AND	0.005	0.31	0.04	0.02	0.37		
BKPIRL-OR	0.005	0.31	0.05	0.02	0.38		
BKPIRL-NOT	0.005	0.15	0.0005	0.02	0.17		
(m=2^{16}, item size is 16 bytes, row payload is 1MB, $e = f \times l = 8 \times 8$)							
MMKPIRL	0.08	0.12	-	0.32	0.44	211	6442
BKPIRL-AND	0.07	0.42	0.16	0.27	0.85		
BKPIRL-OR	0.07	0.42	0.22	0.27	0.91		
BKPIRL-NOT	0.07	0.20	0.002	0.27	0.47		
(m=2^{18}, itemsize is 64 bytes, row payload is 16MB, $e = f \times l = 32 \times 32$)							
MMKPIRL	1.14	0.13	-	4.45	4.58	211	103073
BKPIRL-AND	1.16	0.82	0.82	4.01	5.65		
BKPIRL-OR	1.16	0.82	0.95	4.01	5.78		
BKPIRL-NOT	1.16	0.39	0.008	4.01	4.41		
(m=2^{20}, item size is 256 bytes, row payload is 256MB, $e = f \times l = 128 \times 128$)							
MMKPIRL	19.83	0.11	-	74.07	74.18	211	1649193
BKPIRL-AND	19.43	2.49	3.6	64.54	70.63		
BKPIRL-OR	19.43	2.49	3.78	64.54	70.81		
BKPIRL-NOT	19.43	1.24	0.03	64.54	65.81		

Table VIII compares the asymptotic cost of major homomorphic operations—including $CtCtMul$, $CtPtMul$, $CtRotate$—as well as the multiplicative depth and number of response ciphertexts for CwPIR, MMKPIRL (Algorithm 7), and a single-keyword query version of BKPIRL. Specifically, the BKPIRL-Single row corresponds to a simplified version of Algorithm 11 that handles single-keyword queries without logical operations.

TABLE VIII
ASYMPTOTIC COMPLEXITY OF CWPIR, MMKPIRL, AND BKPIRL IN LARGE MANY-TO-MANY DATABASES

Method	CtCtMul	CtPtMul	CtRotate	Depth	Resp. cts.
CwPIR	nmk	$nm \cdot \left\lceil \frac{l}{N} \right\rceil$	w	$\lceil \log_2 k \rceil$	$nm \cdot \left\lceil \frac{l}{N} \right\rceil$
MMKPIRL	nk	ne	$nlog_2N$	$\lceil log_2k \rceil$	e
BKPIRL-Single	nk	$nf + nl$	$nlog_2N$	$\lceil log_2k \rceil$	$f \cdot l$

It should be noted that while the original CwPIR returns $\left\lceil \frac{l}{N} \right\rceil$ ciphertexts in response, adapting it for a many-to-many database retrieval necessitates returning all $n \times m$ ciphertexts to the client without aggregation to ensure correctness. Consequently, the total number of response ciphertexts becomes $nm \cdot \left\lceil \frac{l}{N} \right\rceil$.

VIII. Conclusion

In this work, we present MMKPIR, an efficient keyword PIR scheme tailored for many-to-many relationships, addressing limitations in prior works. Compared to existing state-of-the-art schemes, MMKPIR achieves significantly lower communication and computation overhead under the same database scale. Building on this, we propose BKPIR, the first keyword PIR scheme to support expressive boolean queries while maintaining performance comparable to MMKPIR. Together, these schemes advance practical privacy-preserving search by supporting large keyword domains, dynamic databases, and boolean logic in a single-server, single-round setting. They hold promise for real-world applications such as private web search and confidential patent retrieval. Future work may explore further optimizations and broader integration into privacy-preserving search frameworks.

Acknowledgment

This work was supported by the National Key Research and Development Program of China under Grant 2023YFB3106302. The authors are grateful to the anonymous reviewers for their valuable comments.

References

[1] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *Proceedings of IEEE 36th Annual Foundations of Computer Science*, Oct. 1995, pp. 41–50.

[2] B. Chor, N. Gilboa, and M. Naor, "Private information retrieval by keywords," 1997.

[3] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[4] G. Salton and M. McGill, *Introduction to modern information retrieval*. New York, NY: McGraw-Hill, 1983.

[5] R. A. Mahdavi and F. Kerschbaum, "Constant-weight {PIR}: Single-round Keyword {PIR} via Constant-weight Equality Operators," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1723–1740.

[6] S. Patel, J. Y. Seo, and K. Yeo, "{Don't} be dense: Efficient keyword {PIR} for sparse databases," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3853–3870.

[7] J. Liu, J. Li, D. Wu, and K. Ren, "Pirana: Faster multi-query pir via constant-weight codes," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4315–4330.

[8] A. Bienstock, S. Patel, J. Y. Seo, and K. Yeo, "Batch PIR and labeled PSI with oblivious ciphertext compression," in *33rd USENIX Security Symposium (USENIX Security 24)*, Aug. 2024, pp. 5949–5966.

[9] J. Groth, A. Kiayias, and H. Lipmaa, "Multi-query Computationally-Private Information Retrieval with Constant Communication Rate," in *Public Key Cryptography – PKC 2010*, ser. Lecture Notes in Computer Science, P. Q. Nguyen and D. Pointcheval, Eds. Berlin, Heidelberg: Springer, 2010, pp. 107–123.

[10] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, May 2000, pp. 44–55.

[11] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public Key Encryption with Keyword Search," in *Advances in Cryptology - EUROCRYPT 2004*, ser. Lecture Notes in Computer Science, C. Cachin and J. L. Camenisch, Eds. Berlin, Heidelberg: Springer, 2004, pp. 506–522.

[12] N. Wang, W. Zhou, J. Wang, Y. Guo, J. Fu, and J. Liu, "Secure and efficient similarity retrieval in cloud computing based on homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, 2024.

[13] P. Golle, J. Staddon, and B. Waters, "Secure Conjunctive Keyword Search over Encrypted Data," in *Applied Cryptography and Network Security*, ser. Lecture Notes in Computer Science, M. Jakobsson, M. Yung, and J. Zhou, Eds. Berlin, Heidelberg: Springer, 2004, pp. 31–45.

[14] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries," in *Advances in Cryptology – CRYPTO 2013*, ser. Lecture Notes in Computer Science, R. Canetti and J. A. Garay, Eds. Berlin, Heidelberg: Springer, 2013, pp. 353–373.

[15] T. Moataz and A. Shikfa, "Boolean symmetric searchable encryption," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013, pp. 265–276.

[16] B. Ferreira, B. Portela, T. Oliveira, G. Borges, H. Domingos, and J. Leitão, "Boolean searchable symmetric encryption with filters on trusted hardware," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 1307–1319, 2020.

[17] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: ramification, attack and mitigation." in *Ndss*, vol. 20. Citeseer, 2012, p. 12.

[18] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-Abuse Attacks Against Searchable Encryption," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 668–679.

[19] S. Oya and F. Kerschbaum, "Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption," in *30th USENIX security symposium (USENIX Security 21)*, 2021, pp. 127–142.

[20] P. Rizomiliotis and S. Gritzalis, "ORAM Based Forward Privacy Preserving Dynamic Searchable Symmetric Encryption Schemes," in *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 65–76.

[21] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption," in *Advances in Cryptology – CRYPTO 2016*, ser. Lecture Notes in Computer Science, M. Robshaw and J. Katz, Eds. Berlin, Heidelberg: Springer, 2016, pp. 563–592.

[22] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled PSI from Fully Homomorphic Encryption with Malicious Security," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 1223–1237.

[23] K. Cong, R. C. Moreno, M. B. da Gama, W. Dai, I. Iliashenko, K. Laine, and M. Rosenberg, "Labeled PSI from Homomorphic Encryption with Reduced Computation and Communication," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 1135–1150.

[24] H. Chen, K. Laine, and P. Rindal, "Fast Private Set Intersection from Homomorphic Encryption," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1243–1255.

[25] L. Ren, M. H. Mughees, and I. Sun, "Simple and practical amortized sublinear private information retrieval using dummy subsets," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1420–1433.

[26] M. Luo, F.-H. Liu, and H. Wang, "Faster fhe-based single-server private information retrieval," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1405–1419.

[27] M. Zhou, A. Park, W. Zheng, and E. Shi, "Piano: extremely simple, single-server pir with sublinear server computation," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4296–4314.

[28] M. H. Mughees and L. Ren, "Vectorized Batch Private Information Retrieval," in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 437–452.

[29] S. Angel, H. Chen, K. Laine, and S. Setty, "PIR with Compressed Queries and Amortized Query Processing," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 962–979.

[30] C. Aguilar Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, "XPIR : Private Information Retrieval for Everyone," *Proceedings on Privacy Enhancing Technologies*, vol. avril 2016, pp. 155–174, Apr. 2016.

[31] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Batch codes and their applications," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 2004, pp. 262–271.

[32] M. H. Mughees, H. Chen, and L. Ren, "OnionPIR: Response Efficient Single-Server PIR," in *Proceedings of the 2021 ACM SIGSAC Confer-*

ence on Computer and Communications Security, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 2292–2306.

[33] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo, "{Communication–Computation} Trade-offs in {PIR}," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1811–1828.

[34] S. J. Menon and D. J. Wu, "SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 930–947.

[35] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[36] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[37] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Advances in Cryptology – ASIACRYPT 2017*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.

[38] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*. Springer, 2010, pp. 1–23.

[39] Z. Brakerski and V. Vaikuntanathan, "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages," in *Advances in Cryptology – CRYPTO 2011*, ser. Lecture Notes in Computer Science, P. Rogaway, Ed. Berlin, Heidelberg: Springer, 2011, pp. 505–524.

[40] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Designs, codes and cryptography*, vol. 71, pp. 57–81, 2014.

[41] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin, "Protecting Data Privacy in Private Information Retrieval Schemes," *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 592–629, Jun. 2000.

[42] "Microsoft SEAL (release 4.0)," https://github.com/Microsoft/SEAL, Mar. 2022, microsoft Research, Redmond, WA.

[43] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.

[44] R. Geelen and F. Vercauteren, "Bootstrapping for bgv and bfv revisited," *Journal of Cryptology*, vol. 36, no. 2, p. 12, 2023.

[45] A. Kim, M. Deryabin, J. Eom, R. Choi, Y. Lee, W. Ghang, and D. Yoo, "General bootstrapping approach for rlwe-based homomorphic encryption," *IEEE Transactions on Computers*, 2023.

Appendix A
Security Proofs

Definition 1 (IND-CKA): A keyword PIR protocol is *IND-CKA secure* (indistinguishable under chosen keyword attack) if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$, and for any two keywords kw_0, kw_1, it holds that:

$$|\Pr[\mathcal{A}(\text{View}(kw_0)) = 1] - \Pr[\mathcal{A}(\text{View}(kw_1)) = 1]| \leq \text{negl}(\lambda),$$

where λ is the security parameter, $\mathsf{View}(kw)$ contains all encrypted queries, homomorphic computation traces, and response ciphertexts generated during protocol execution for keyword kw.

Definition 2 (IND-CBQA): A PIR protocol supporting boolean keyword queries is *IND-CBQA secure* (indistinguishable under chosen boolean query attack) if, for any PPT adversary $\mathcal{A}$, and for any two boolean queries:

$$Q_0 = B(kw_1^{(0)}, \dots, kw_z^{(0)}), \quad Q_1 = B(kw_1^{(1)}, \dots, kw_z^{(1)}),$$

with the same logical structure B, it holds that:

$$|\Pr[\mathcal{A}(\text{View}(Q_0)) = 1] - \Pr[\mathcal{A}(\text{View}(Q_1)) = 1]| \leq \text{negl}(\lambda).$$

Theorem 1: Assuming the RLWE problem is hard, MMKPIR is IND-CKA secure.

Proof of Theorem 1. Construct a sequence of hybrid games:

Game 0: Real protocol execution with query kw_b.

Game 1: Replace query ciphertexts $\tilde{\boldsymbol{q}}_b$ with fresh RLWE encryptions under the same noise distribution. Indistinguishability follows from RLWE-based IND-CPA security.

Game 2: Replace equality test results with RLWE ciphertexts encrypting random bits (0/1) under the appropriate noise level for their position in the computation graph. This preserves distributional equivalence as per RLWE error propagation.

Game 3: Replace the final response ciphertexts with random ciphertexts of the same dimension and noise level, which remain indistinguishable under RLWE-based encryption.

Game 4: Switch kw_b to kw_{1-b}. The computational indistinguishability between adjacent games implies that any non-negligible distinguishing advantage by the adversary would yield an efficient algorithm for solving RLWE, thereby completing the proof.

□

Theorem 2: Assuming the RLWE problem is hard, BKPIR is IND-CBQA secure.

Proof of Theorem 2. Extend the hybrid argument with boolean operator preservation:

Game 0: Real execution of boolean query Q_b.

Game 1: Replace all keyword ciphertexts $\{\tilde{\boldsymbol{q}}_j^{(b)}\}$ with fresh RLWE encryptions, maintaining consistent noise growth across logical operations.

Game 2: For each boolean operation, replace intermediate results with properly noised RLWE ciphertexts encrypting either:

- The real computed bit, when following the correct computation path
- A random bit with matching noise distribution, when on dummy paths

Game 3: Replace the final response ciphertexts with random ciphertexts of the same dimension and noise level.

Game 4: Switch Q_b to Q_{1-b}. Structural equivalence of boolean circuits and noise consistency maintain indistinguishability.

The argument reduces boolean query distinguishability to RLWE hardness. □

Appendix B
Symmetric Security Extensions

The basic MMKPIR and BKPIR protocols can be extended to achieve *symmetric PIR* security as defined by Gertner et al. [41], which ensures privacy for both the client query and the database content. We describe how to adapt BKPIR to satisfy this stronger privacy notion, focusing on key challenges and corresponding countermeasures.

Two databases $\mathcal{DB}$ and $\mathcal{DB}'$ with m entries per record are structurally equivalent if:

$$\forall i \in [m] : \mathcal{DB}[*, i] = \mathcal{DB}'[*, i] \vee \big(\mathcal{DB}[*, i] = \perp \wedge \mathcal{DB}'[*, i] = \perp\big).$$

A protocol achieves symmetric security if for any PPT adversary $\mathcal{A}$:

$$\big|\Pr[\mathcal{A}(\mathsf{View}(\mathcal{DB})) = 1] - \Pr[\mathcal{A}(\mathsf{View}(\mathcal{DB}')) = 1]\big| \leq \text{negl}(\lambda),$$

denoted as $\mathsf{View}(\mathcal{DB}) \overset{c}{\approx} \mathsf{View}(\mathcal{DB}')$ under the RLWE assumption.

However, the baseline BKPIR protocol may leak structural information due to its use of multi-hot encoding. Specifically, the response

$$\boldsymbol{r} = \sum_{i=1}^{m} \mathsf{mask}_i \cdot \mathsf{Enc}(v_i),$$

where $\mathsf{mask}_i \in \{0, 1\}$, allows the server to distinguish between $\mathsf{Enc}(v_i)$ and encryptions of zero. This leakage arises because the presence of a non-zero ciphertext magnitude indicates that $v_i \neq \perp$.

To mitigate this leakage, the following countermeasures can be implemented:

First, introduce *uniform response obfuscation*. Each response is padded with an encryption of zero using fresh noise:

$$\boldsymbol{r}^* = \boldsymbol{r} + \mathsf{Enc}(0; \eta_{\mathsf{obfs}}),$$

where η_{obfs} is a noise bound that matches or exceeds the maximum noise level $\eta_{\max}$ across all valid ciphertexts. This ensures that all responses are statistically indistinguishable in magnitude. This countermeasure incurs minimal computational overhead, requiring only a single ciphertext addition per query and slightly increasing noise growth.

Second, inject random placeholder entries into the database to obscure the true distribution of values. Specifically, Expand the database to $m' = m + \Delta_m$ entries with:

$$\mathcal{DB}' = \mathcal{DB} \cup \left\{ (\mathsf{kw}_{\mathsf{ph}}^{(j)}, \mathsf{Enc}(0)) \right\}_{j=1}^{\Delta_m},$$

where $\Delta_m = \omega(\log \lambda)$ ensures dummy entries dominate adversarial advantage. This introduces $\lceil \Delta_m / N \rceil$ additional plaintext rows and ciphertext-plaintext multiplications per row, resulting in moderate overhead.

Third, apply periodic randomization to the value map. After every $K = \mathsf{poly}(\lambda)$ queries, the server permutes the value mapping:

$$\mathsf{VM}' = \pi(\mathsf{VM}), \quad \pi \in_R S_{m'},$$

using a cryptographically secure permutation π. This prevents the server from associating specific ciphertext positions with query patterns over time. This incurs an amortized preprocessing cost equivalent to re-encoding and shuffling m' entries every K queries.

Theorem 3: Let the underlying encryption scheme be RLWE-based with noise parameter $\eta_{\mathsf{obfs}} \geq \eta_{\max}$. Then, the modified BKPIR protocol satisfies $\mathsf{View}(\mathcal{DB}) \overset{c}{\approx} \mathsf{View}(\mathcal{DB}')$ for structurally equivalent databases.

Proof Sketch. Construct a sequence of hybrid games to argue indistinguishability: (1) Replace real responses with padded ones $\boldsymbol{r}^*$, which are indistinguishable due to RLWE; (2) Replace dummy entries with real entries incrementally, noting that all are encrypted and indistinguishable; (3) Apply permutations to the value map and argue that pseudorandomness hides any correlation. Each hybrid transition is indistinguishable under standard RLWE-based security assumptions. □

Appendix C
Artifact Appendix

A. Description & Requirements

This artifact provides the reference implementation of the MMKPIR and BKPIR protocols proposed in our paper. The implementation includes:

- Core protocol implementation in C++ using Microsoft SEAL for homomorphic encryption.
- Scripts to reproduce results for MMKPIR and BKPIR in Tables I–VII.

1) How to access: The artifact is publicly available at https://github.com/gimmehug/bkpir. The repository contains build instructions, experiment scripts, and a detailed README. We have also uploaded the artifact package to the Zenodo repository at https://doi.org/10.5281/zenodo.16962425.

2) Hardware dependencies: The implementation has no specialized hardware requirements. For optimal performance during large-scale experiments (particularly Table III and IV with large keyword-value combinations), we recommend 64GB RAM for stable execution.

3) Software dependencies:

- OS: Ubuntu 20.04+ recommended
- C++ compiler: g++ $\geq$ 6.0
- CMake: $\geq$ 3.13
- Microsoft SEAL $\geq$ 4.0

4) Benchmarks: All experiments use synthetic data generated in-memory. No external datasets are required.

B. Artifact Installation & Configuration

The artifact requires standard development tools and follows conventional build processes: install prerequisite packages using the system package manager, build and install Microsoft SEAL following official documentation, configure the build environment using CMake, and compile the source code.

Detailed, step-by-step installation instructions are provided in the repository's README file.

C. Experiment Workflow

Scripts for reproducing Tables I–VII are located in the `scripts/` directory. Each script sets parameters, runs the protocol, and outputs CSV files with performance results.

D. Major Claims

- (C1): MMKPIR supports native $n \times m$ keyword-value mappings using $O(n)$ structure by packing m values per ciphertext row. This enables scalable support for many-to-many inverted index structures with sublinear communication complexity. Demonstrated in E1–E3 (Tables I–III).
- (C2): BKPIR extends MMKPIR to support boolean keyword combinations (AND/OR/NOT) with complete logical expressiveness. It is the first PIR construction to natively support boolean retrieval logic. Logical operations introduce only small additional cost (milliseconds). Demonstrated in E4 and E7 (Tables IV and VII).
- (C3): MMKPIR and BKPIR show comparable performance in single-keyword settings, indicating that boolean support in BKPIR introduces negligible overhead when unused. Demonstrated in E5 and E7 (Tables V and VII).
- (C4): BKPIR supports up to 23 boolean logic depth (e.g., nested AND/OR) under practical parameter settings, with tunable noise growth and latency tradeoffs. Demonstrated in E6 (Table VI).

E. Evaluation

We provide shell scripts to reproduce all tables in our paper. Each script is located under the `scripts/` directory and automatically runs the corresponding experiment.

1) Experiment E1: MMKPIR Single-keyword Retrieval: [5 human-minutes + 2 compute-hours]

[How to, Preparation & Execution]: Refer to the README file.

[Result]: This script runs the MMKPIR protocol under varying (N, n, m) parameters and saves output as `results/table1.csv`.

2) Experiment E2: Microbenchmark on Single-row Database: [5 human-minutes + 5 compute-minutes]

[How to, Preparation & Execution]: Refer to the README file.

[Result]: Shows microbenchmark under a single-row database and saves output as a CSV file.

3) Experiment E3: MMKPIR Scalability: [5 human-minutes + 40 compute-minutes]

[How to, Preparation & Execution]: Refer to the README file.

[Result]: This script evaluates MMKPIR under growing (n, m) while fixing k and N, and stores results in a CSV file.

4) Experiment E4: BKPIR Boolean Query Performance: [5 human-minutes + 2 compute-hours]

[How to, Preparation & Execution]: Refer to the README file.

[Result]: CSV file demonstrating latency and communication costs for BKPIR. See Table IV.

5) Experiment E5: MMKPIR and BKPIR(Single-keyword) Test: [5 human-minutes + 4 compute-hours]

[How to, Preparation & Execution]: Refer to the README file.

[Result]: CSV file demonstrating latency of MMKPIR and BKPIR (with a single keyword input) under varying N and k.

6) Experiment E6: Noise Budget, Logic Depth, and Boolean Circuit Latency: [5 human-minutes + 20 compute-minutes]

[How to, Preparation & Execution]: Refer to the README file.

[Result]: CSV file with noise budget, cost per logic gate, and maximum supported logic depth under different N and k.

7) Experiment E7: Item Number and Payload Size Scaling: [5 human-minutes + 20 compute-minutes]

[How to, Preparation & Execution]: Refer to the README file.

[Result]: CSV file with communication and runtime cost of MMKPIRL and BKPIRL at expanded item counts and size.

F. Customization

The implementation supports extensive configuration, users can customize key parameters such as:

- Polynomial modulus degree (N): 2^{13}, 2^{14} and 2^{15}
- Number of keywords `(-n)` (recommended: $\leq$ 512 for 64GB RAM)
- Number of items `(-m)` (should be a multiple of N for better packing)
- Hamming weight `(-k)` (typical range: 2–16)
- Item size `(-il)` (recommended: multiple of 2 bytes) and keyword length `(-kl)` (flexible)

G. Notes

All experimental scripts are self-contained. For large-scale runs (e.g., Table III-IV), ensure adequate memory is available. Execution logs are saved for all script runs.