LetoPIR: Fast Keyword Private Information Retrieval with Logarithmic Communication

Baoyu Li[1,2], Kang Yang[2], Qi Liu[1], Binwu Xiang[3], Xiaogang Zhou[4], Xiang Xie[3], and Yu Yu[1,3]

[1] Shanghai Jiao Tong University
[2] State Key Laboratory of Cryptology
[3] East China Normal University
[4] China Telecom Quantum Information Technology Group Co., Ltd.

Abstract. Keyword private information retrieval (PIR) allows a client to retrieve a record associated with a keyword from a database without revealing any information about the keyword. In the standard single-server setting, existing hintless keyword PIR protocols incur substantial communication and computation costs. In this paper, we propose an efficient approach to generate k-hot vectors (i.e., vectors with exactly k non-zero components) in homomorphic-encryption form, and present a bucket-merging technique to decrease the maximum size of buckets. Based on these techniques, we construct LetoPIR, a hintless keyword PIR protocol that outperforms previous PIR protocols in the same setting. Compared to the state-of-the-art hintless keyword PIR scheme, SparsePIR (USENIX'23), LetoPIR achieves a $12.4\times \sim 17.0\times$ improvement in communication cost for databases ranging from 256 MB to 4 GB with records of 256 bytes, and more than $3.0\times$ improvement in computation cost for the 256 MB database. Compared to the state-of-the-art keyword PIR scheme with client hint, KPIR (USENIX'25), LetoPIR reduces the communication cost by $51.4\times \sim 184.8\times$, while achieving a similar (even better) computation cost.

1 Introduction

Private Information Retrieval (PIR), introduced by Chor et al. [17], allows a client to retrieve a record from a remote database without revealing which record is accessed. PIR has been widely applied in a variety of applications such as contact tracing [51, 1], private web search [28, 7], and anonymous communication [30, 45, 32]. Furthermore, Apple has already deployed PIR for privacy-preserving real-time caller ID lookup [5].

By requiring clients to interact with only a single server, the single-server setting avoids the need for aided servers and the associated non-collusion assumption, making it more suitable for the above applications. In this paper, we focus on the single-server setting. According to whether a query is an index or a keyword, PIR is typically categorized into index PIR and keyword PIR. In index PIR, a client must know the exact index i of a desired record [17]. Keyword

PIR allows a client to query a keyword $\hat{k}$, without knowing whether a matching entry $\mathsf{DB}[\hat{k}]$ exists or where it is located in the database [16]. Since keywords are more user-friendly than indices, many real-world applications prefer keyword PIR rather than index PIR.

A standard paradigm of constructing keyword PIR is to convert a keyword-value database into an index-value database and then invoke index PIR on the transformed database. In this paradigm, keyword PIR constructions based on Key-Value Store (KVS), such as SparsePIR [47], ChalametPIR [14], and KPIR [27], use schemes such as RB-OKVS [9] or Binary Fuse Filter (BFF) [25] to obtain dense representations with small expansion factors [48, 22]. Separately, Ali et al. [3] present a generic, simple transformation for sparse databases, which can adopt any index PIR scheme as a building block. Alternatively, a few keyword-PIR schemes [38, 6] use constant-weight codes rather than KVS, with TCAKPIR+ [6] being the state-of-the-art protocol in this line. However, existing concretely efficient single-server keyword PIR protocols have communication complexity of at least $O(\sqrt{N})$, where N is the database size, resulting in large costs for low-bandwidth networks or multiple-keyword retrieval. In addition, ChalametPIR and KPIR require the server to precompute a large number of client hints [29] for each client, which are downloaded and stored by the client. For resource-constrained clients, this storage cost is undesirable.

The goal of this paper is to design a concretely efficient keyword PIR protocol with logarithmic communication. Very recently, the index PIR scheme (called VIA) [34] achieves $O(\log N)$ communication by generating one-hot vectors from logarithmic-sized ciphertexts. By combining VIA with the generic transformation [3] based on Cuckoo hashing, one can construct a hintless keyword PIR scheme (named Cuckoo+VIA) with $O(\nu \log N)$ communication, where ν is the number of hash functions. However, it requires ν independent index-PIR executions, which limits the concrete efficiency.

1.1 Our Contributions

In terms of techniques, we first propose an efficient approach to generate k-hot vectors with logarithmic communication complexity, which can be seamlessly integrated into the KVS-based keyword PIR paradigm. Then, we present a bucket-merging technique to reduce the maximum bucket size in the recent hash-to-bucket framework [27], where both the communication and computation costs of keyword PIR are significantly affected by the maximum bucket size. In particular, when hashing n^2 entries into n buckets, and n ranges from 2^8 to 2^{12}, our technique reduces the maximum bucket size by $13.34\% \sim 5.02\%$. Our techniques to generate k-hot vectors and merge buckets may be of independent interest for applications beyond keyword PIR. In addition, we propose a bucket packing technique that packs multiple encoded buckets into one polynomial. It not only reduces coefficient waste and enables SIMD-style decoding, but also allows us to compress response size with ring switching, while preserving compatibility with homomorphic KVS decoding.

By combining our techniques with the index PIR scheme Respire [13], we design a hintless keyword PIR protocol (referred to as LetoPIR) with logarithmic communication. We give a theoretical comparison between LetoPIR with the recent keyword PIR protocols in Table 1. In particular, our LetoPIR protocol achieves both communication and client-computation complexities of $O(\log N)$. While the combined keyword PIR protocol Cuckoo+VIA (Cuc-VIA in short) [34, 3] requires ν independent index-PIR executions, LetoPIR needs only a single index-PIR execution. While ChalametPIR and KPIR require clients to store database-dependent hints, other PIR schemes do not.

We implemented LetoPIR in C++, and evaluated its performance. Our source code will be made publicly available upon acceptance.

Compared to the state-of-the-art hintless keyword PIR scheme SparsePIR [47], for databases ranging from 256 MB to 4 GB with records of 256 bytes, LetoPIR improves the communication cost by a factor of $12.4\times \sim 17.0\times$ and reduces the computational cost by more than $3.0\times$ for the 256 MB database. [5] Compared to the best-known keyword PIR scheme with client hint, i.e., KPIR [27]: (1) for a database containing $2^{20} \sim 2^{24}$ records with each 256 bytes, LetoPIR reduces the communication cost by a factor of $51.4\times \sim 184.8\times$, while its computation time ranges from $0.9\times$ to $1.0\times$ that of KPIR; (2) for a database containing 2^{18} records with each 1024 bytes (resp., 4096 bytes), LetoPIR not only improves the communication cost by a factor of $36.6\times$ (resp., $26.7\times$), but also speeds up the computation time by $3.5\times$ (resp., $6.5\times$).

While no implementation is available for the keyword PIR scheme Cuckoo+VIA [3, 34], we estimate the performance of Cuckoo+VIA by computing the communication cost and conservatively estimating the computation time. In particular, based on the VIA implementation [34], we use three independent index-PIR executions to estimate the computational cost of Cuckoo+VIA, and ignore the extra cost of Cuckoo hashing. Compared to Cuckoo+VIA, for databases from 256 MB to 4 GB, LetoPIR reduces the communication cost by $2.3\times \sim 2.4\times$ and computation cost by $12.9\times \sim 13.9\times$, even with the conservative estimate. Besides, while Cuckoo+VIA only achieves a 110-bit security level (as the parameters of VIA [34] have only 110-bit security), LetoPIR satisfies the 128-bit security level.

1.2 Technical Overview

Prior Solution Concretely efficient keyword PIR schemes in the single-server setting are mainly designed with Homomorphic Encryption (HE). They generally follow two paradigms: (1) one combines HE with KVS; (2) the other combines HE with constant-weight codes. However, keyword PIR schemes with constant-weight codes, e.g., CWPIR [38] and KPIR-C [6], are much more expensive than

[5] As the source code of SparsePIR is *not* publicly available, its computation time is extracted from the original publication [47]. We report the setting with 2^{20} records of 256 bytes each for consistency with the evaluation settings of the other protocols. While SparsePIR uses 3.7 GHz Intel Xeon W-2135, our implementation uses 2.6 GHz Intel Xeon Platinum 8358. Thus, the estimation for computation-time improvement is conservative.

Table 1: Comparison between our keyword PIR protocol and the recent protocols, where N denotes the database size and ν is the number of hash functions used in Cuckoo hashing. In this table, "Comm." and "Comp." denote the communication and computation complexities, respectively, without involving the setup phase that is run only once and can be reused for multiple queries.

Keyword PIR Scheme	**Total Comm.**	**Client Comp.**	**Server Comp.**
ChalametPIR [14]	$O(N)$	$O(N)$	$O(N)$
KPIR [27]	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(N)$
TCAKPIR+ [6]	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(N)$
SparsePIR [47]	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(N)$
Cuc-VIA [34, 3]	$O(\nu \log N)$	$O(\nu \log N)$	$O(\nu N)$
LetoPIR (ours)	$O(\log N)$	$O(\log N)$	$O(N)$

KVS-based keyword PIR in terms of both computational and communication costs. Therefore, we focus on the first paradigm combining HE with KVS. The keyword PIR schemes in this paradigm are typically divided into a setup phase (independent of queries) and an online phase (i.e., queries are known). In the setup phase (a.k.a., the offline phase in prior works), which is run only once and can be reused for a polynomial number of queries, the server encodes the database with a KVS. Depending on the underlying index PIR scheme, the server sends record-dependent hints to the client, who stores them for PIR schemes with client hints, while there is no additional storage for hintless PIR in this phase. In the online phase, the client chooses a keyword and generates its corresponding KVS decoding vector, and then sends the encrypted vector to the server, which uses it to homomorphically compute the corresponding record. Finally, the server sends the resulting record in the encryption form to the client, who decrypts the ciphertext to obtain the record associated with the keyword.

Below, we discuss the basic approach behind the hintless keyword PIR schemes in the first paradigm (typically, SparsePIR [47]). [6] Specifically, in SparsePIR (built on Spiral [41]), a client packs the decoding vector into polynomial coefficients and encrypts the polynomial as a query sent to a server. The server homomorphically expands the query, evaluates KVS decoding over the encoded buckets, and sends back a response in the form of HE ciphertexts to the client. Nevertheless, due to the expanded decoding representation, the communication complexity of SparsePIR remains $O(\sqrt{N})$. Moreover, its concrete efficiency is constrained by three bottlenecks: (1) the cost of generating an $O(\sqrt{N})$-length decoding vector, (2) the decoding cost determined by the largest bucket, and (3) coefficient waste for small records. SparsePIR applies a naive hash-to-bucket approach [49], which yields unbalanced loads due to the inherent variance of the

[6] Since we focus on hintless keyword PIR, we here ignore the keyword PIR schemes with client hints, e.g., ChalametPIR [14] and KPIR [27] that are based on FrodoPIR [18] and SimplePIR [29] respectively.

balls-into-buckets process. Even under uniform hashing, the maximum bucket size can be noticeably larger than the average, which becomes a bottleneck since the concrete decoding cost is dominated by the largest bucket. To reduce the coefficient waste for small records, the index PIR scheme Respire [13] packs multiple messages into one polynomial and utilizes ring switching to reduce the communication cost on the response. However, it must ensure that the packed layout still supports the correct homomorphic decoding and allows the server to isolate the target slot before applying ring switching. Therefore, it is infeasible to directly apply the techniques of Respire to keyword PIR.

Our Solution We address the efficiency bottlenecks as described above by presenting three new techniques to (1) generate k-hot vectors in logarithmic communication, (2) perform bucket merging, and (3) realize bucket packing with ring-switch-based response compression. As a result, we reduce the communication cost relative to the recent concretely efficient keyword PIR protocols discussed above, while incurring only a small computational overhead. Specifically, our techniques are shown as follows.

To reduce the cost of generating a decoding vector, we produce a k-hot vector in an encrypted form from the encrypted bit decompositions of k-hot positions rather than directly transmitting an expanded decoding representation. Specifically, for each k-hot position $u \in [\ell] = \{0, \ldots, \ell - 1\}$, where $\ell = O(\sqrt{N})$ is the length of a k-hot vector, the client computes the bit decomposition $u = \sum_{j=0}^{\lceil \log \ell \rceil - 1} u_j \cdot 2^j$, and then encrypts each bit u_j as an RGSW ciphertext. For each k-hot position, the server initializes an accumulator $\mathsf{acc} \leftarrow \mathsf{RLWE}(f, s)$, where f denotes a constant polynomial and s is a secret key. For each $j \in [\lceil \log \ell \rceil]$, the server homomorphically selects between acc and $\mathsf{acc} \cdot X^{2^j}$ using the encrypted bit u_j. After processing all bits, the accumulator encodes a one-hot vector with its nonzero entry encoded in a polynomial coefficient. After summing the accumulators for all k-hot positions and then applying homomorphic expansion, we yield an RLWE ciphertext of encrypting a k-hot vector. Through the above technique, for generating a length-ℓ k-hot vector in an encrypted form, we are able to reduce the communication cost on a single query from $O(\ell)$ to $O(k \log \ell)$ at the cost of an additional computational overhead of $O(k \log \ell)$. To further reduce the communication cost, we let the client compress the RGSW ciphertexts into more compact RLWE ciphertexts, which are then homomorphically converted back to the RGSW ciphertexts by the server.

In keyword PIR, the server uses a hash-to-bucket [27] technique to partition all key-value pairs into buckets. As discussed above, the maximum bucket size is the bottleneck in keyword PIR. To address this, we introduce bucket merging. The server first hashes all key-value pairs into initial buckets, then uses First Fit Decreasing (FFD) [31] to merge them into a smaller set of balanced buckets. It sends the resulting lookup table to the client, which uses it to locate the merged bucket and generate the corresponding decoding ciphertext. Bucket merging preserves correctness and privacy. The only extra cost is a lookup table stored by the client.

We propose bucket packing to mitigate the coefficient-waste bottleneck for small records, which packs several encoded buckets into one polynomial while preserving compatibility with homomorphic KVS decoding and ring switching. At a high level, the server groups ρ encoded buckets together and places their coefficients into different slots of a single polynomial, so that KVS decoding can be evaluated in parallel across these slots. The client then uses public metadata to determine the target packed bucket, the target slot, and the local decoding positions corresponding to the queried keyword. The client encrypts this information and sends it to the server as a query. After homomorphic decoding, the server uses encrypted selectors to extract the target slot and finally applies ring switching to compress the response. This reduces the number of packed ciphertexts processed during decoding, the coefficient waste, and the response size. Because bucket packing is fixed and query-independent, correctness and privacy are preserved. The only extra cost is that the client stores a small mapping rule.

2 Preliminaries

2.1 Notation

Throughout this paper, we use λ to denote the security parameter. We use $\mathsf{negl}(\cdot)$ to denote an unspecified negligible function such that $\mathsf{negl}(\lambda) = o(\lambda^{-c})$ for every constant c. For $n \in \mathbb{Z}_{>0}$, denote $[n] = \{0, \ldots, n-1\}$. Unless otherwise specified, all vector, bucket, and slot indices in this paper are zero-based. We use a bold lowercase (resp., uppercase) letter such as $\mathbf{a}$ (resp., $\mathbf{A}$) to denote a vector (resp., matrix), and write $\mathbf{a}_i$ (resp., $\mathbf{A}_{i,j}$) to denote the i-th component of $\mathbf{a}$ (resp., the entry in the i-th row and j-th column of $\mathbf{A}$). For a string x, we use $|x|$ to denote its length. We write $[a]_q = a \bmod q$. For $n \in \mathbb{Z}_{>0}$, a vector $\mathbf{v} \in \{0,1\}^n$ is called *one-hot* if $\sum_{i\in[n]} v_i = 1$. More generally, $\mathbf{v}$ is called k-*hot* if $\sum_{i\in[n]} v_i = k$.

2.2 Keyword Private Information Retrieval

Following prior works such as [16, 27], we divide a keyword PIR scheme into a $\mathsf{Setup_C}$ /$\mathsf{Setup_S}$ phase and an online phase. In the $\mathsf{Setup_C}$/$\mathsf{Setup_S}$ phase, the protocol generates client-side parameters, server-side parameters, and a query key. In the online phase, the client and the server perform an interactive retrieval protocol. A keyword PIR protocol consists of five algorithms $(\mathsf{Setup_C}, \mathsf{Setup_S}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$ and satisfies correctness and query privacy. The formal syntax and security definitions are given in Appendix B.

2.3 Homomorphic Encryption

Homomorphic encryption (HE) [23, 11, 21, 12, 20, 15] enables computation over ciphertexts while preserving the underlying plaintext computation after decryption. Practical HE schemes are typically based on lattice assumptions, in particular the Learning with Errors (LWE) assumption and its ring variant (RLWE) [50,

37]. We use an RLWE-based HE scheme throughout the paper. The formal definitions of LWE and RLWE are deferred to Appendix F. We define the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^d+1)$, where d is a power of two and $\mathbb{Z}_q = \{0, \ldots, q-1\}$. Let $\mathcal{X}$ denote the discrete Gaussian error distribution used in encryption. For a decomposition basis $\gamma \in \{\mathsf{gadget}, \mathsf{RNS}\}$, let $\mathbf{g}_\gamma$ be its gadget vector and Decomp_γ the corresponding decomposition; they satisfy $\langle \mathbf{g}_\gamma, \mathsf{Decomp}_\gamma(a) \rangle = a$ over the underlying ciphertext ring. The two decompositions are defined in Appendix F.

We use three ciphertext types from TFHE [15]. An $\mathsf{RLWE}(m, s)$ ciphertext encrypts m under s; $\mathsf{Rlev}^\gamma(m, s)$ encrypts the gadget multiples of m in basis γ; and $\mathsf{RGSW}^\gamma(m, s)$ supports external products with RLWE ciphertexts. We omit the basis superscript when it is clear from context. Their construction and the BFV-style encoding are given in Appendix F. We use standard homomorphic addition, plaintext-ciphertext multiplication, automorphism, and key switching [11, 21, 12]. Their interfaces are summarized in Appendix F.

External Product The external product, denoted by $\boxtimes$, enables homomorphic multiplication between an $\mathsf{RGSW}(a, s)$ ciphertext and an $\mathsf{RLWE}(b, s)$ ciphertext, resulting in $\mathsf{RLWE}(a \cdot b, s)$ [15].

CMUX Given $\mathsf{RGSW}(b, s)$ for $b \in \{0, 1\}$ and two RLWE ciphertexts ct_0, ct_1, we can select ct_b by computing Eq.1 [15].

$$\begin{aligned} ct_b &= \mathsf{CMUX}(\mathsf{RGSW}(b, s), ct_0, ct_1) \\ &= ct_0 + \mathsf{RGSW}(b, s) \boxtimes (ct_1 - ct_0) \end{aligned} \tag{1}$$

Expand Given a ciphertext $\mathsf{RLWE}(m, s)$, coefficients $\mathsf{RLWE}(m_i, s)$ can be extracted homomorphically [4]. If we stop after the r-th round, the output is $\{\mathsf{RLWE}(\sum_{2^r \mid i} m_{i \cdot 2^r + j} X^i, s)\}_{j \in [2^r]}$. We describe this process in Alg. 13 of Appendix F.4.

RingSwitch Ring switching [24] maps an RLWE ciphertext over a larger ring to one over a smaller ring while preserving the underlying plaintext under an appropriate embedding. We describe this process in Alg. 14 of Appendix F.5.

RLWE2RGSW Given $t = \lceil \log_\beta q \rceil$ RLWE ciphertexts of the form $\mathsf{RLWE}(\beta^i \cdot m, s)$ and a conversion key $\mathsf{Rlev}(s^2, s)$, we can transform these ciphertexts into a RGSW ciphertext $\mathsf{RGSW}(m, s)$ [44]. We describe this process in Alg. 15 of Appendix F.6.

2.4 Key-Value Store

A Key-Value Store (KVS) [48, 22] encodes a set of key–value pairs into an array D together with public hash functions H. For an inserted key $\hat{k}$, $\mathsf{Decode}(D, \hat{k}, H)$ recovers its associated record. The formal syntax is given in Appendix D.

Binary Fuse Filter [25] A Binary Fuse Filter (BFF) yields a compact KVS whose decoder accesses a small number of positions determined by public hash functions. Its construction is given in Appendix D.

Random Band Oblivious Key-Value Stores [9] Random Band Oblivious Key-Value Stores (RB-OKVS) use short random bands to obtain low encoding overhead. Its construction is given in Appendix D.

2.5 First Fit Decreasing

First Fit Decreasing (FFD) [31] is a standard heuristic for bin packing. It sorts items in descending order of size and places each item into the first bucket with sufficient remaining capacity. If none exists, it opens a new bucket. We describe this procedure in Alg. 12 of Appendix E.

3 Generating K-hot Vectors

We present a faster k-hot vector generation method with logarithmic communication and apply this technique to some key-value stores.

We start from a faster one-hot ciphertext vector generation with logarithmic communication and extend it to a k-hot vector. Compared with VIA [34], our one-hot vector generation method restricts the vector length, but achieves higher computational efficiency: for a vector of length n, it requires $n-1$ key switches and $\log n$ external products, whereas VIA requires $n-1$ external products. At the same decomposition length, an external product requires twice as many Hadamard products as a key switch.

Algorithm 1: $\mathsf{OneHotVecGen}(\hat{C}, c_f)$

Input: RGSW ciphertexts $\hat{C} \leftarrow \{C_j \leftarrow \mathsf{RGSW}(u_j, s)\}_{j\in[\log n]}$
Output: A one-hot vector $\mathbf{v}$ of length n

$acc \leftarrow c_f$
for $i \in [\log n]$ **do**
  $acc \leftarrow \mathsf{CMUX}(C_i, acc, acc \cdot X^{2^i})$
end
$\{c_j\}_{j\in[n]} \leftarrow \mathsf{Expand}(acc, \lceil \log n \rceil)$
return $\{c_j\}_{j\in[n]}$

Consider the one-hot ciphertext vector $\mathbf{v}$ of length n, where $\mathbf{v}_u = \mathsf{RLWE}(f, s)$, where f is an arbitrary constant polynomial and s is the secret key, and all other components of $\mathbf{v}$ are zero ciphertexts. Let $\{u_j\}$ be the binary representation of u. We initialize the RGSW ciphertexts $\hat{C} \leftarrow \mathsf{RGSW}(\{u_j\}_{j\in[\log n]}, s)$ and an accumulator $acc \leftarrow \mathsf{RLWE}(f, s)$. To obtain $\mathsf{RLWE}(f \cdot X^u, s)$, we update acc by conducting CMUX on $ct_0 = acc$ and $ct_1 = acc \cdot X^{2^j}$ for each $\mathsf{RGSW}(u_j, s)$ where $j \in [\log n]$. After that, applying the Expand algorithm to $\mathsf{RLWE}(f \cdot X^u, s)$ outputs a one-hot vector where the u-th entry is $\mathsf{RLWE}(f, s)$ and the other entries are $\mathsf{RLWE}(0, s)$. This procedure is described in Alg. 1. We denote $c_f = \mathsf{RLWE}(f, s)$.

We now extend Alg. 1 to support a k-hot ciphertext vector. For the target position $u_i, i \in [k]$, we repeat the accumulation procedure to obtain acc_i, and then sum them together. After applying Expand to the accumulator $acc = \sum_i acc_i$, we can easily get the k-hot ciphertext vector. This procedure is described in Alg. 2.

Algorithm 2: $\mathsf{kHotVecGen}(\{\hat{C}_j, c_f^j\})$

Input: RGSW ciphertexts $\{\hat{C}_j\}_{j\in[k]}$ and their corresponding accumulators c_f^j
Output: A k-hot vector $\mathbf{v}$ of length n

$acc \leftarrow (0, 0)$
for $i \in [k]$ **do**
  $acc_i \leftarrow c_f^i$
  $\{C_j\}_{j\in[\log n]} \leftarrow \hat{C}_i$
  for $j \in [\log n]$ **do**
    $acc_i \leftarrow \mathsf{CMUX}(C_j, acc_i, acc_i \cdot X^{2^j})$
  end
  $acc \leftarrow acc + acc_i$
end
$\{c_j\}_{j\in[n]} \leftarrow \mathsf{Expand}(acc, \lceil \log n \rceil)$
return $\{c_j\}_{j\in[n]}$

We instantiate the above homomorphic k-hot vector generation method for logarithmic-communication KVS decoding in both BFF and RB-OKVS. The two instantiations are described below.

Applying to BFF decoding. As described above, given a vector $\mathbf{v}$ of length n, BFF requires selecting $k = \alpha$ positions $u_0, \ldots, u_{\alpha-1} \in [n]$ such that $\mathbf{v}[u_j] = 1$ for all $j \in [\alpha]$, while all other entries are zero. Assume a family of hash functions $H = \{h_j\}_{j\in[\alpha]}$ is used to map key-value pairs to index-value pairs. To query the record associated with a key $\hat{k}$, the client computes $u_j \leftarrow h_j(\hat{k}), j \in [\alpha]$. For each u_j, the client decomposes it into bits and encrypts them as RGSW ciphertexts and sends them to the server. The server computes $c_f^j \leftarrow (0, \Delta)$ for each $j \in [\alpha]$ locally, then invokes the $\mathsf{kHotVecGen}$ algorithm to obtain the corresponding k-hot vector. Performing the inner product between the k-hot vector and the index-value database completes the BFF decoding.

Applying to RB-OKVS decoding. RB-OKVS requires that given a vector $\mathbf{v}$, a starting position u_{start} is selected such that a contiguous segment of length ω, $\mathbf{v}[u_{\mathsf{start}}], \ldots, \mathbf{v}[u_{\mathsf{start}} + \omega - 1] \in \{0, 1\}^\omega$, while all other positions are zero. We consider a relaxed k-hot vector, where k positions are now set as 0 or 1 instead of only 1. Let $f = \sum_{j=0}^{\omega-1} \mathbf{v}[u_{\mathsf{start}} + j]X^j$. To hide $\mathbf{v}$, the client computes $c_f = \mathsf{RLWE}(f, s)$, generates the encryption of u_{start}, and then sends them to the server. By assigning c_f to acc and invoking $\mathsf{kHotVecGen}$, the server obtains the corresponding ω-hot vector. The ω elements starting from u_{start} are RLWE ciphertexts encrypting $\mathbf{v}[u_{\mathsf{start}}], \ldots, \mathbf{v}[u_{\mathsf{start}}+\omega-1]$, while the remaining elements

are ciphertexts encrypting 0. Performing the inner product between the ω-hot vector and the index-value database completes the RB-OKVS decoding. We illustrate the accumulation process in Fig. 1.

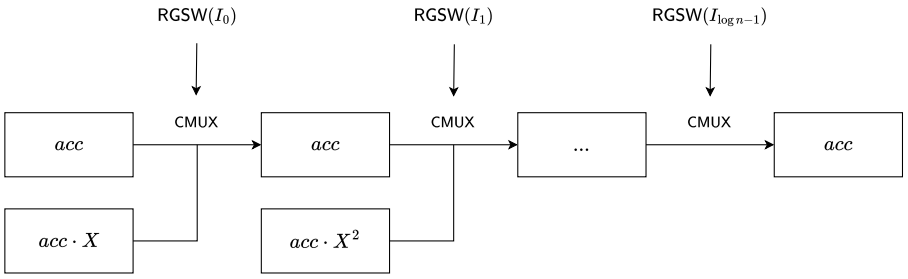

Fig. 1: The relaxed k-hot accumulation process of RB-OKVS. Here, RGSW denotes the bit-select ciphertext, and acc denotes c_f.

4 Bucket Merging and Bucket Packing

As discussed in Section 1.2, current KVS-based constructions have three main bottlenecks: representing and generating an $O(\sqrt{N})$-length decoding vector, decoding cost governed by the largest bucket, and coefficient waste for small records. This section addresses the latter two bottlenecks. We introduce bucket merging to reduce the maximum bucket size and bucket packing to reduce coefficient waste, and then combine them into a unified layout.

4.1 Bucket Merging

Let X_i denote the load of bucket i where $X_i \sim \text{Binomial}\left(N, \frac{1}{T}\right)$ under uniform hashing. For sufficiently large N, we approximate X_i by a Gaussian distribution $X_i \approx \mathcal{X}(\mu, \sigma^2)$, where $\mu = \frac{N}{T}, \sigma^2 = N \cdot \frac{1}{T}\left(1 - \frac{1}{T}\right)$.

Let $B_{\max} = \max_{1 \le i \le T} X_i$ be the maximum bucket load. In order to support homomorphic KVS decoding over all buckets in a uniform manner, the server pads every bucket to the same length $B_{\max}$ before encoding and evaluation. As a result, the concrete decoding cost is governed by the largest bucket rather than the average one. Reducing $B_{\max}$ is therefore critical.

To this end, we introduce an intermediate hashing step. Instead of mapping into T buckets, we first hash into $2T$ buckets. Let Y_i denote the load of the i-th intermediate bucket. Then $Y_i \sim \text{Binomial}\left(N, \frac{1}{2T}\right)$ and, for large N, $Y_i \approx \mathcal{X}(\mu', \sigma'^2)$, where $\mu' = \frac{N}{2T}, \sigma'^2 = N \cdot \frac{1}{2T}\left(1 - \frac{1}{2T}\right)$.

We then merge the $2T$ intermediate buckets into T target buckets. Each target bucket has a capacity $cap = 2\mu' + \epsilon = \frac{N}{T} + \epsilon$, where $\epsilon > 0$ is a slack parameter.

We model this step as a bucket packing problem. Each intermediate bucket is an item with size equal to its load, and each target bucket has capacity *cap*. Let $\mathcal{B} = \{b_1, \ldots, b_{2T}\}$ denote the intermediate buckets with sizes $w_i = |b_i|$, and let $\mathcal{G} = \{G_1, \ldots, G_T\}$ denote the target buckets. The goal is to assign buckets such that $\sum_{b_i \in G_j} w_i \leq cap, \forall j \in [T]$.

We apply First Fit Decreasing (FFD) [31] to pack the intermediate buckets. Sorting buckets in descending order of size and placing each into the first feasible target bucket improves load balance and reduces the maximum load after merging. Finally, each target bucket is padded to capacity *cap*, yielding a matrix representation. We illustrate this process in Fig. 2.

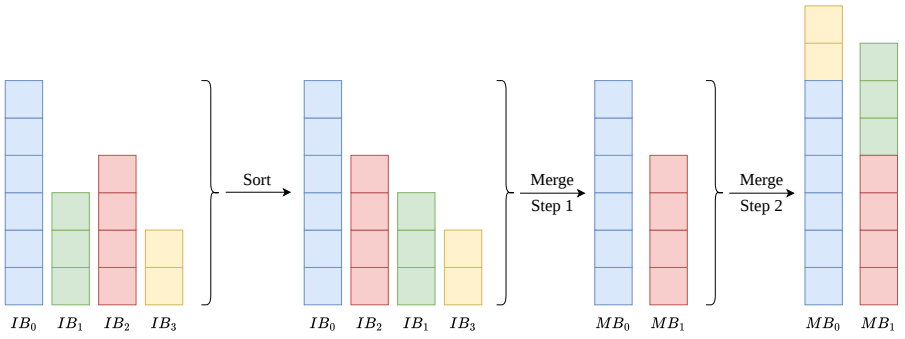

Fig. 2: **Bucket merging**. IB_i denotes the i-th bucket obtained from hash-to-bucket, where each block corresponds to a key-value pair.

4.2 Bucket Packing

After bucket merging, the server obtains a set of balanced buckets and applies KVS encoding to each bucket. When the encoded items are small compared with the polynomial degree, storing each item in a separate polynomial results in coefficient waste and extra communication and computation during homomorphic evaluation. A natural solution is to pack multiple encoded buckets into one polynomial. However, this slot arrangement must still support homomorphic KVS decoding: the server can evaluate decoding correctly on the packed representation, and the client needs to identify and extract the target result from the packed slots. We describe this bucket packing method below.

Assume the polynomial degree is $d = \rho \cdot |r|$ where $|r|$ is the record length. Given ρT initial buckets B_i for $i \in [\rho T]$, each containing the same number of elements. We then apply KVS encoding to each bucket B_i to obtain $\hat{B}_i$. For every group of ρ buckets $\hat{B}_{j\rho}, \ldots, \hat{B}_{(j+1)\rho-1}$, we embed the corresponding encoded records in each bucket following Alg. 16 in Appendix F.7. It essentially places these encoded records into the polynomial slots according to their bucket

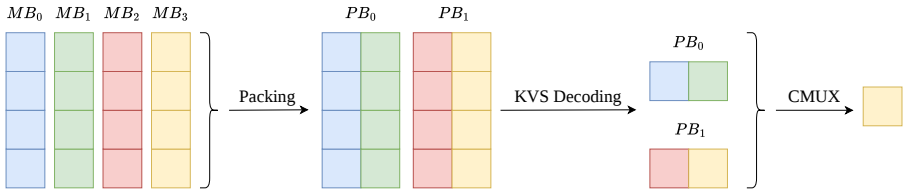

Fig. 3: Bucket packing across multiple encoded buckets. Each block is treated as a polynomial, and the packed concatenation indicates that some blocks are encoded into the same polynomial.

indices. To retrieve a keyword, it suffices to identify the packed bucket that may contain it. We then perform KVS decoding on the target bucket and return the record at the corresponding slot. We illustrate the bucket packing and decoding method in Fig. 3.

4.3 Integration

We now combine bucket merging and bucket packing with homomorphic KVS decoding. Starting from the hash-to-bucket layout, the server first hashes all key-value pairs into $2\rho T$ initial buckets, applies bucket merging to obtain ρT balanced buckets, and sends the lookup table to the client. After KVS encoding, every ρ encoded buckets are packed into one polynomial, resulting in T packed buckets.

For a query keyword $\hat{k}$, the client identifies the merged bucket containing $\hat{k}$ by searching the lookup table, derives the corresponding packed-bucket index, slot index, and decoding positions, and then encrypts the associated decoding information. The server homomorphically evaluates KVS decoding on each packed bucket in parallel. It then uses encrypted selectors to choose the target packed bucket and extract the desired slot, followed by a ring-switch step. This integration reduces the number of packed ciphertexts processed during decoding by a factor of ρ, while preserving the KVS decoding logic inside each slot. The complete protocol is given in Section 5.

5 Concretely Efficient Keyword PIR

In this section, we present LetoPIR, a hintless keyword PIR protocol, which integrates logarithmic-communication homomorphic KVS decoding with bucket merging and bucket packing. We first show the BFF-based instantiation of LetoPIR, which achieves logarithmic communication by compressing the KVS decoding information into a compressed query. We then demonstrate how the same framework can be extended to the RB-OKVS-based variant.

The server holds a key-value database $\{(\hat{k}_i, r_i)\}_{i\in[N]}$. During the $\mathsf{Setup}_\mathsf{C}$ phase, the client samples a query key c_s along with the corresponding switching

Algorithm 3: $\mathsf{Setup}_{\mathsf{C}}(1^{\lambda})$

Input: The security parameter λ
Output: A query key c_s and switching material s_s

$c_s \leftarrow \mathcal{X}(0, \sigma)$
Generate the switching material s_s using c_s.
return (c_s, s_s)

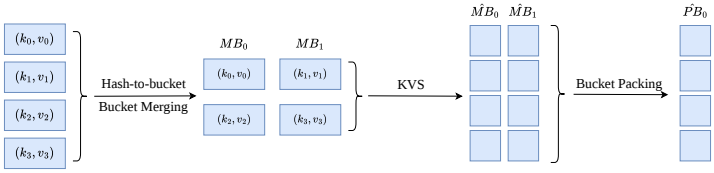

Fig. 4: **Database configuration.** We illustrate the encoding of the database in the setup phase, where MB denotes the merged buckets obtained after performing bucket merging on the initial buckets IB produced by hash-to-bucket. $\hat{MB}$ denotes the merged buckets after KVS encoding, and $\hat{PB}$ denotes the packed buckets.

material s_s, as shown in Alg. 3. Here, c_s is the client-side secret key used in query generation and recovery, while s_s denotes the public switching material derived from c_s, including the key-switching keys used by $\mathsf{KeySwitch}$, the conversion key used by $\mathsf{RLWE2RGSW}$, and the key material used in ring switching. During the $\mathsf{Setup}_{\mathsf{S}}$ phase, the server first rewrites each record as $\tilde{r}_i \leftarrow \mathsf{hash}(\hat{k}_i)||r_i$, where $\mathsf{hash}(\cdot) : \{0,1\}^* \rightarrow \{0,1\}^{2\lambda}$ is a collision-resistant hash function.

As shown in Fig. 4, instead of directly arranging the database into a matrix, the server first uses a hash function $h(\cdot) : \{0,1\}^* \rightarrow [2\rho T]$ to hash all key-value pairs into initial buckets $\{IB_i\}_{i\in[2\rho T]}$. The server then applies the bucket merging technique to balance the loads of the initial buckets, resulting in ρT merged buckets $\{MB_i\}_{i\in[\rho T]}$. Subsequently, each merged bucket is padded to the maximum bucket size. For each merged bucket MB_i, the server samples α hash functions $H_i = \{h_{i,j}(\cdot)\}_{j\in[\alpha]}$ and uses BFF to encode it into an encoded bucket $\hat{MB}_i$. Finally, for each ρ encoded buckets $\hat{MB}_{j\rho}, \ldots, \hat{MB}_{(j+1)\rho-1}$, the server employs the bucket packing procedure described in Section 4.2. This procedure packs $\hat{MB}_{j\rho}, \ldots, \hat{MB}_{(j+1)\rho-1}$ into one packed bucket $\hat{PB}_j$ that can be used by the underlying index PIR protocol. We describe this process in Alg. 4.

In Fig. 5, using Respire [13] as the underlying index PIR protocol, when a client retrieves the record $\mathsf{DB}[\hat{k}]$ for a keyword $\hat{k}$, it first computes the initial bucket index $j \leftarrow h(\hat{k})$, then locates the corresponding merged bucket index $\tilde{b}$ using the lookup table received from the server. Next, the client uses a public packing map π to obtain the packed bucket index, the in-polynomial slot index, and the local packed positions, i.e., $(b_1, b_2) = (\lfloor \tilde{b}/\rho \rfloor, \tilde{b} \ \% \ \rho) \leftarrow \pi(\tilde{b})$, where $b_1 \in [T]$ denotes the packed-bucket index and $b_2 \in [\rho]$ denotes the slot inside the

Algorithm 4: $\mathsf{Setup}_{\mathsf{S}}(1^{\lambda}, \mathsf{DB})$

Input: The security parameter λ, the database DB
Output: Server-side parameters pp_s and client-side parameters pp_c

for $i \in [N]$ **do**
  $\tilde{r}_i \leftarrow \mathsf{hash}(\hat{k}_i)||r_i$
  $b_i^{(\mathrm{db})} \leftarrow h(\hat{k}_i), b_i^{(\mathrm{db})} \in [2\rho T]$
  $IB_{b_i^{(\mathrm{db})}} \leftarrow IB_{b_i^{(\mathrm{db})}} \cup (\hat{k}_i, \tilde{r}_i)$
end
Merge $\{IB_i\}_{i\in[2\rho T]}$ to $\{MB_i\}_{i\in[\rho T]}$, record the table Tab.
for $i \in [\rho T]$ **do**
  $(\hat{MB}_i, H_i) \leftarrow \mathsf{Encode}(MB_i)$
end
for $i \in [T]$ **do**
  Pack $\{\hat{MB}_{i\cdot\rho+j}\}_{j\in[\rho]}$ into $\hat{PB}_i$ by bucket packing.
end
return $pp_s = \hat{DB}, pp_c = (\{H_i\}_{i\in[\rho T]}, \pi, \mathsf{Tab})$, where $\hat{DB} \leftarrow \{\hat{PB}_i\}_{i\in[T]}$

Algorithm 5: $\mathsf{Query}(pp_c, \hat{k}, c_s)$

Input: The query keyword $\hat{k}$, the query key c_s, client-side parameters $pp_c = (\{H_i\}_{i\in[\rho T]}, \pi, \mathsf{Tab})$
Output: The query qu

$\tilde{b} \leftarrow h(\hat{k})$, then locate the corresponding merged bucket index $\tilde{b}$ using the merge lookup table Tab
$\{u_0, ..., u_{\alpha-1}\} \leftarrow H_{\tilde{b},0}(\hat{k}), ..., H_{\tilde{b},\alpha-1}(\hat{k})$
$(b_1, b_2) \leftarrow \pi(\tilde{b})$, where $b_1 = b_{1,0}||...||b_{1,\log T-1}$ and $b_2 = b_{2,0}||...||b_{2,\log\rho-1}$
$qu_{\mathsf{dec}} \leftarrow \mathsf{VecGen}(\{u_0, \ldots, u_{\alpha-1}\}, c_s)$
$f_{\mathsf{sel}} \leftarrow \sum_{i=0}^{\log T-1}\sum_{j=0}^{t-1} g_j \cdot b_{1,i} X^{(i\cdot t+j)} + \sum_{i=0}^{\log\rho-1}\sum_{j=0}^{t-1} g_j \cdot b_{2,i} X^{((i+\log T)\cdot t+j)}$
$qu_{\mathsf{sel}} \leftarrow \mathsf{RLWE}(f_{\mathsf{sel}}, c_s)$
return $qu \leftarrow (qu_{\mathsf{dec}}, qu_{\mathsf{sel}})$

selected packed polynomial. The client then compresses the local decoding positions $\{u_\ell\}_{\ell\in[\alpha]}$ using VecGen and encrypts the selection information for (b_1, b_2). We describe this process in Alg. 5. We describe VecGen in Appendix F.8.

When the server receives the query, it first applies $\mathsf{VecExpand}$ to reconstruct the encrypted bit decompositions of the local decoding positions. We describe $\mathsf{VecExpand}$ in Appendix F.8. It then sets the local accumulators to encryptions of the plaintext constant Δ and invokes $\mathsf{kHotVecGen}$ to derive the corresponding encrypted decoding vector. Using this vector, the server performs homomorphic KVS decoding on each packed bucket, obtaining one ciphertext for each packed polynomial. It next extracts the encrypted bit-slices of b_1 and b_2, converts them into RGSW ciphertexts, and performs a two-level selection. It first uses the bits of b_1 and CMUX gates to select the target packed polynomial. It then uses the bits of b_2 and another CMUX procedure to extract the target slot from the selected

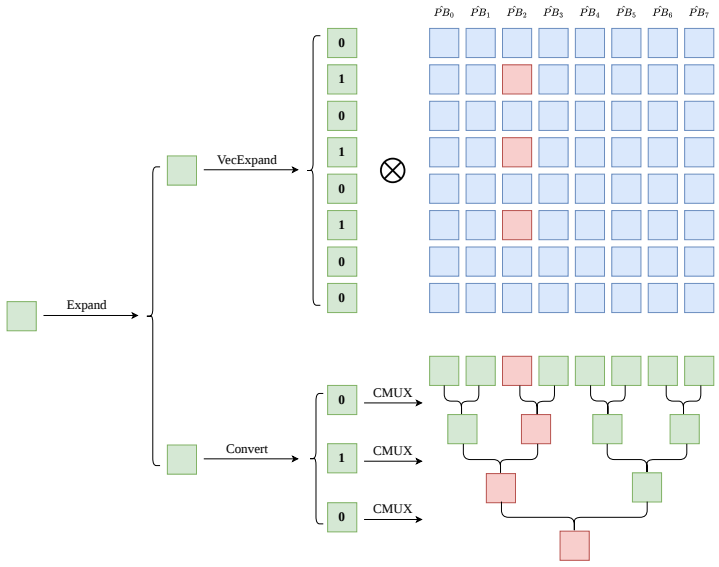

Fig. 5: **Overview of LetoPIR instantiated with Respire over the packed database.** The blue blocks represent the packed database after the setup procedure in Fig. 4, where each column corresponds to a packed bucket $\hat{PB}_i$. We assume that the queried keyword is mapped to the second merged bucket and, after BFF encoding, corresponds to the red positions. Homomorphic KVS decoding produces one ciphertext for each packed bucket by summing the entries at the red positions within that bucket. The server then uses encrypted selectors to choose the target packed bucket and extract the target slot via CMUX, before applying ring switching to compress the response.

packed polynomial. Finally, it applies ring switching to compress the selected ciphertext before returning the final ciphertext res to the client. We describe this process in Alg. 6.

The client decrypts the ciphertext $(\hat{h}||r) \leftarrow \mathsf{Decrypt}(res, c_s)$ and checks whether $\mathsf{hash}(\hat{k})$ equals $\hat{h}$ or not. The client accepts r as $\mathsf{DB}[\hat{k}]$ if they are equal. Otherwise, it outputs $\perp$, indicating that the database does not contain a record corresponding to the keyword $\hat{k}$. We describe this process in Alg. 7.

Theorem 1 (Correctness). *Assuming the underlying index PIR is correct and the tagged KVS satisfies correctness and absent-key soundness, if the ciphertext modulus q keeps decryption correct, then for any query $\hat{k} \in \mathcal{K}$ with $(\hat{k}, r) \in \mathsf{DB}$, the client obtains r. If $(\hat{k}, \cdot) \notin \mathsf{DB}$, the client obtains $\perp$.*

The proof appears in Appendix C.

Theorem 2 (Query Privacy). *Assume that the underlying RLWE-based encryption scheme is circular/KDM secure with respect to the public evaluation ma-*

Algorithm 6: $\mathsf{Answer}(pp_s, s_s, qu)$

Input: The server-side parameters $pp_s = \hat{DB}$, the switching material s_s, and the query qu
Output: The response res

$(qu_{\mathsf{dec}}, qu_{\mathsf{sel}}) \leftarrow qu$
$\{\hat{C}_i\}_{i\in[\alpha]} \leftarrow \mathsf{VecExpand}(qu_{\mathsf{dec}}, s_s)$
for $i \in [\alpha]$ **do**
  $c_f^i \leftarrow (0, \Delta)$
end
$\{c_j\}_{j\in[M_{\mathrm{enc}}]} \leftarrow \mathsf{kHotVecGen}(\{\hat{C}_i, c_f^i\}_{i\in[\alpha]})$
for $i \in [T]$ **do**
  $\hat{c}_i \leftarrow (0, 0)$
  for $j \in [M_{\mathrm{enc}}]$ **do**
    $\hat{c}_i \leftarrow \mathsf{Add}(\hat{c}_i, \mathsf{PCMul}(\hat{DB}_{i,j}, c_j))$
  end
end
$\{c'_i\}_{i\in[t\cdot(\log T+\log\rho)]} \leftarrow \mathsf{Expand}(qu_{\mathsf{sel}}, \lceil \log\left(t\cdot(\log T + \log\rho)\right)\rceil)$
$\ell \leftarrow T$
for $i \in [\log T]$ **do**
  $C \leftarrow \mathsf{RLWE2RGSW}(\{c'_{i\cdot t+j}\}_{j\in[t]}, s_s)$
  $\ell \leftarrow \frac{\ell}{2}$
  for $j \in [\ell]$ **do**
    $\hat{c}_j \leftarrow \mathsf{CMUX}(C, \hat{c}_j, \hat{c}_{j+\ell})$
  end
end
for $i \in [\log\rho]$ **do**
  $C \leftarrow \mathsf{RLWE2RGSW}(\{c'_{(\log T+i)\cdot t+j}\}_{j\in[t]}, s_s)$
  $\hat{c}_0 \leftarrow \mathsf{CMUX}(C, \hat{c}_0, \hat{c}_0 \cdot X^{-2^i})$
end
$\hat{c}_0 \leftarrow \mathsf{RingSwitch}(\hat{c}_0, s_s)$
return $res \leftarrow \hat{c}_0$

Algorithm 7: $\mathsf{Recover}(res, \hat{k}, c_s)$

Input: The response res, the query keyword $\hat{k}$, and the query key c_s
Output: The record m

$\mathsf{hash}(\hat{k}')||r \leftarrow \mathsf{Decrypt}(res, c_s)$
if $\mathsf{hash}(\hat{k}) = \mathsf{hash}(\hat{k}')$ **then**
  return $m \leftarrow r$
end
return $m \leftarrow \perp$

terial generated from the client secret key. Then, for any two keywords $\hat{k}, \hat{k}' \in \mathcal{K}$, even given the setup outputs (pp_s, pp_c) and the switching material s_s, the query

distributions $\mathsf{Query}(pp_c, \hat{k}, c_s)$ *and* $\mathsf{Query}(pp_c, \hat{k}', c_s)$ *are computationally indistinguishable.*

The proof appears in Appendix C.

Extension to RB-OKVS based PIR. The above construction instantiates LetoPIR with BFF, which yields our first variant, denoted LetoPIR-v1. We obtain LetoPIR-v2 by replacing the BFF encoding inside each merged bucket with RB-OKVS encoding. All other components remain unchanged, including bucket merging, bucket packing, packed-bucket selection, and the ring-switch.

The main difference lies in the homomorphic KVS decoding step. For a merged bucket encoded with RB-OKVS, decoding is determined by a start position and a short band vector rather than by α isolated positions. Concretely, for a keyword $\hat{k}$ in a merged bucket, the client computes the RB-OKVS decoding information $u_{\mathsf{start}} \leftarrow h_1(\hat{k}), \mathbf{u} \leftarrow h_2(\hat{k})$, where u_{start} is the start position of the decoding band and $\mathbf{u} \in \mathbb{F}^{\omega}$ is the corresponding coefficient vector. The client then encrypts two objects: the compressed bit decomposition of u_{start}, which specifies the location of the band, and a ciphertext encrypting the polynomial $f = \sum_{j=0}^{\omega-1} \mathbf{u}[j] X^j$, which carries the RB-OKVS decoding payload.

Upon receiving the query, the server first applies VecExpand to the compressed ciphertext for u_{start}, obtaining the encrypted bit decomposition of the band location. It then uses these ciphertexts as the first input to kHotVecGen, while using the encrypted polynomial f as the second input. This produces an ω-slot ciphertext vector corresponding to the band $[u_{\mathsf{start}}, u_{\mathsf{start}}+\omega-1]$. The entries in this vector encrypt the coefficients $\mathbf{u}[0], \ldots, \mathbf{u}[\omega-1]$ on the target band and 0 elsewhere. The server then performs RB-OKVS decoding by taking a homomorphic inner product between this band vector and the encoded database for each bucket. After this step, the protocol proceeds exactly as in LetoPIR-v1: the server performs packed-bucket selection, extracts the target slot, applies ring switching, and returns the resulting ciphertext to the client. Compared with BFF, RB-OKVS typically achieves a smaller expansion factor, which reduces the bucket size and leads to lower upload communication and lower server-side computation in our setting.

Compressed Query Representation. The dominant source of upload communication is the ciphertexts used to encode the bit decompositions in VecGen. To reduce this cost, we use the same compression strategy as in Respire [13]. Instead of sending full RGSW ciphertexts, the client packs the effective coefficients into one RLWE ciphertext at power-of-two-spaced positions and encrypts it as (a, b). The polynomial a is pseudorandom and can therefore be regenerated by the server from a short seed. The client transmits this seed and the occupied coefficients of b. Upon receiving the compressed query, the server reconstructs a from the seed and places the transmitted coefficients of b back into their designated positions, padding the remaining coefficients with zeros. It then applies a reduced-round hybrid extraction procedure: first a shortened HomTrace to collect the nonzero coefficients, then the usual binary-tree-based Expand, and finally

RLWE2RGSW to recover the RGSW ciphertexts required by kHotVecGen. This optimization preserves the protocol flow above while reducing the upload size.

6 Performance Evaluation

We instantiate the underlying index PIR scheme with Respire [13] and the KVS scheme with RB-OKVS and BFF for LetoPIR. As a result, LetoPIR-v1 and LetoPIR-v2 denote the BFF-based and RB-OKVS-based constructions, respectively. We compare the efficiency of LetoPIR with the recent keyword PIR schemes, including ChalametPIR [14], KPIR [27] and SparsePIR [47], on databases containing 2^{20}, 2^{22}, and 2^{24} records, each of size 256 bytes. Additionally, we compare performance on databases with 2^{18} records under record sizes of 64, 1024, and 4096 bytes.

6.1 Experimental Setup

We provide a C++ proof-of-concept implementation of LetoPIR. We use the HEXL library [10] to accelerate Number Theoretic Transform (NTT) operations. All experiments are conducted on a machine equipped with an Intel Xeon Platinum 8358 CPU @ 2.60GHz and 2TB RAM, running Ubuntu 22.04. The peak memory consumption is 165 GB. For timing measurements, we execute each protocol 50 times and report the average runtime. All experiments are performed using *a single thread*.

6.2 Parameter Selection

We choose the HE parameters using the LWE estimator [2] to achieve a 128-bit security level. The resulting parameters are listed in Table 2. We choose the decomposition bases to balance noise growth, computation, and communication. Following prior works [41, 13], our parameters ensure a decryption failure probability of at most 2^{-40}. Inspired by [8], we adopt a more compact parameterization for generating RNS-based RGSW ciphertexts. Our implementation uses both gadget and RNS decomposition bases. The compact-query conversion uses gadget-basis conversion material to reconstruct RNS-basis RGSW ciphertexts. All subsequent key switching, CMUX, external products, and RingSwitch use the RNS basis over the extended modulus $Q = qP$. Here, P is the auxiliary modulus used during RNS-based homomorphic operations, after which the results are modulus switched back to the ciphertext modulus q. The communication in the setup phase includes evaluation keys for both bases.

We set the BFF expansion factor to $\kappa_1 = 1.25$ and use $\alpha = 3$ hash functions. For RB-OKVS, we set the expansion factor to $\kappa_2 = 1.05$ and the band length to $\omega = 32$. Since bucket merging is performed during the database preprocessing, we heuristically choose a sufficiently small ϵ. If FFD fails, the server resamples a hash seed, and repeats the hash-to-bucket procedure until bucket merging succeeds. For each key-value pair (k, r), we encode the record as $\tilde{r} \leftarrow \mathsf{hash}(k) \,\|\, r$,

Table 2: HE and decomposition parameters for LetoPIR, where d is the ring degree, q is the ciphertext modulus, P is the auxiliary modulus, p is the plaintext modulus, σ_1, σ_2 are the standard deviations of the Gaussian error distribution over Q and q, respectively.

d	$\log q$	$\log P$	p	σ_1	σ_2	λ (bit)
4096	56	48	256	3.19	0.82	128

Table 3: Efficiency comparison of keyword PIR protocols in the setup phase, which is run only once and can be reused for any polynomial number of queries. We highlight the best performance in yellow and the second-best one in blue.

Database	Metric	ChalametPIR	KPIR	LetoPIR-v1	LetoPIR-v2
$2^{20} \times 256$ B	Setup Communication	1.74 MB	77.7 MB	14.08 MB	14.08 MB
	Setup Computation	3708.2 s	279.3 s	22.2 s	205.8 s
	Client Storage	1.74 MB	77.3 MB	0.16 MB	0.16 MB
$2^{22} \times 256$ B	Setup Communication	1.74 MB	143.6 MB	14.20 MB	14.20 MB
	Setup Computation	20736.5 s	947.3 s	94.9 s	897.8 s
	Client Storage	1.74 MB	142.3 MB	0.29 MB	0.29 MB
$2^{24} \times 256$ B	Setup Communication	1.74 MB	277.3 MB	14.47 MB	14.47 MB
	Setup Computation	97128.1 s	5440.4 s	388.7 s	4140.1 s
	Client Storage	1.74 MB	272.3 MB	0.55 MB	0.55 MB

where the output length of collision-resistant hash function hash is 256 bits. To simplify the description, we use the record length to represent the length of $\mathsf{hash}(k) \,\|\, r$.

6.3 Performance Comparison

Below, we show the efficiency improvement of hintless keyword PIR with logarithmic-communication k-hot vector generation, bucket merging, and bucket packing.

Performance evaluation in the setup phase. Table 3 compares the performance of keyword PIR protocols in the setup phase, where the database includes 2^{20}, 2^{22}, or 2^{24} entries, each of size 256 bytes. We ignore SparsePIR in this table, since its implementation is not publicly available and the setup efficiency is not reported in [47]. Unlike ChalametPIR [14] and KPIR [27], LetoPIR does not require the client to store a large database-dependent hint. Instead, the client stores only a few public parameters describing the database structure and the hash functions, resulting in minimal client storage. This advantage is useful for resource-constrained clients. Furthermore, LetoPIR achieves the fastest computation time for the setup phase, meaning that eliminating client hints does not require a heavy setup phase.

Performance evaluation in the online phase for different numbers of records. The experimental results in Table 4 demonstrate the efficiency advantage of our keyword PIR scheme LetoPIR in the online phase (excluding the one-time reusable setup phase), compared to the recent keyword PIR schemes.

Table 4: **Efficiency comparison between our protocols (LetoPIR-v1 and LetoPIR-v2) and recent keyword PIR protocols in the online phase (excluding the one-time reusable setup phase).** We highlight the best performance in yellow and the second-best one in blue. Since the implementation of SparsePIR is not publicly available, we estimate the communication cost of SparsePIR, and extract the server computation time for $2^{20} \times 256$ B database from the original publication [47]. While Cuckoo+VIA (Cuc-VIA in short) [3, 34] provides a 110-bit security level, LetoPIR has the 128-bit security level.

Database	Metric	ChalametPIR	KPIR	SparsePIR	Cuc-VIA	LetoPIR-v1	LetoPIR-v2
$2^{20} \times 256$ B	Query Size (KB)	4608	65.6	14	2.1	1.01	0.78
	Response Size (KB)	0.92	77.3	21	4.2	2	2
	Comm. Cost (KB)	4608.92	142.9	35	6.3	3.01	2.78
	Server Comp. (ms)	170	460	1420	6462	530.14	464.52
	Client Comp. (ms)	12812	14	N/A	6	1.46	2.16
	Total Comp. (ms)	12982	474	>1420	6468	531.60	466.68
$2^{22} \times 256$ B	Query Size (KB)	18432	142.4	14	2.4	1.11	0.83
	Response Size (KB)	0.92	130.0	21	4.2	2	2
	Comm. Cost (KB)	18432.92	272.4	35	6.6	3.11	2.83
	Server Comp. (ms)	720	1215	N/A	16557	1522.86	1277.48
	Client Comp. (ms)	142268	15	N/A	6	1.42	2.14
	Total Comp. (ms)	142988	1230	N/A	16563	1524.28	1279.62
$2^{24} \times 256$ B	Query Size (KB)	73728	260.0	28	2.6	1.21	0.88
	Response Size (KB)	0.92	272.3	21	4.2	2	2
	Comm. Cost (KB)	73728.92	532.3	49	6.8	3.21	2.88
	Server Comp. (ms)	3442	4592	N/A	54501	4867.68	4012.36
	Client Comp. (ms)	662554	26	N/A	6	1.65	2.14
	Total Comp. (ms)	665996	4618	N/A	54507	4869.33	4014.50

While no implementation is available for the combined keyword PIR scheme Cuckoo+VIA [3, 34], we conservatively estimate the computation time by using three independent index-PIR executions and ignoring the extra cost of Cuckoo hashing, based on the VIA implementation [34].

Fig. 6 illustrates the trade-off between computation and communication in the online phase among the keyword PIR schemes. Overall, Table 4 and Fig. 6 show that LetoPIR achieves a substantially more communication-efficient regime while maintaining practical computation cost. This advantage is consistent with our design goals: logarithmic-communication k-hot vector generation reduces the query size, bucket merging lowers the decoding cost, and bucket packing improves coefficient utilization while enabling response compression.

The released VIA implementation [35] required minor adjustments in our environment. We applied the minimal implementation corrections required for correct execution and used the corrected version for all VIA measurements. We retained its polynomial degree, error distribution, and ciphertext modulus, which provide approximately 110-bit security level according to the lattice estimator [2]. Then we use the corrected VIA implementation to estimate the computation time of Cuckoo+VIA [3, 34], recalling that no public implementation combines the index PIR scheme VIA [34] with the generic cuckoo-hashing framework [3].

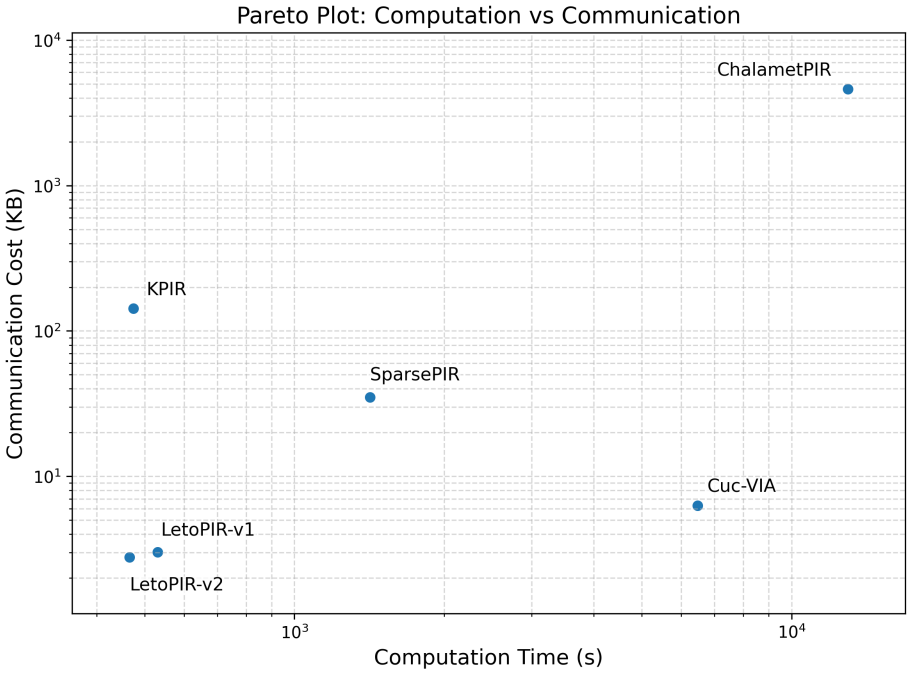

Fig. 6: Pareto plot of computation and communication costs in the online phase for five PIR schemes on a database with 2^{20} rows of 256 bytes each. Both axes are shown on logarithmic scales. The computation cost of SparsePIR is shown as a lower bound because its client computation is unavailable. Schemes closer to the lower-left corner achieve a better trade-off between computation and communication.

Comparison with SparsePIR. Since SparsePIR does not provide an open-source implementation, its running time is extracted from the original publication [47], where only the runtime for the $2^{20} \times 256$ B database is available. While SparsePIR [47] uses a 3.7 GHz Intel Xeon W-2135 CPU, LetoPIR is run on a 2.6 GHz Intel Xeon Platinum 8358 CPU. When retrieving one record from a database containing 2^{20} entries with each record of 256 bytes, LetoPIR achieves a 12.6× reduction in communication and $> 3.0\times$ improvement in computation, compared with the hintless keyword PIR scheme SparsePIR. Our performance advantage comes from addressing the efficiency bottleneck of previous hintless keyword PIR schemes: (1) our logarithmic-communication k-hot vector generation avoids transmitting an expanded decoding vector; (2) our bucket-merging approach shrinks the largest bucket and thus reduces the decoding cost; (3) our bucket packing technique reduces coefficient waste while enabling response compression.

Table 5: **Efficiency comparison of keyword PIR schemes across different record sizes in the online phase (excluding the one-time reusable setup phase).** Since SparsePIR does not provide an open-source implementation, only its communication cost is reported. We highlight the best performance in yellow and the second-best one in blue.

Database	Metric	ChalametPIR	KPIR	SparsePIR	LetoPIR-v1	LetoPIR-v2
$2^{18} \times 64$ B	Query Size (KB)	1152	12.65	14	0.85	0.72
	Response Size (KB)	0.21	14.38	21	2	2
	Communication Cost (KB)	1152.21	27.03	35	2.85	2.72
	Server Computation (ms)	8	28.88	N/A	183.48	168.58
	Client Computation (ms)	2180	4.75	N/A	1.48	2.16
	Total Computation (ms)	2188	33.63	N/A	184.96	170.74
$2^{18} \times 1024$ B	Query Size (KB)	1152	63.24	14	0.96	0.72
	Response Size (KB)	3.21	109.38	21	4	4
	Communication Cost (KB)	1155.21	172.62	35	4.96	4.72
	Server Computation (ms)	127	1622	N/A	540.12	468.78
	Client Computation (ms)	2145	17	N/A	1.46	2.12
	Total Computation (ms)	2272	1637	N/A	541.58	470.90
$2^{18} \times 4096$ B	Query Size (KB)	1152	127.63	14	1.01	0.72
	Response Size (KB)	12.81	318.13	42	16	16
	Communication Cost (KB)	1164.81	445.76	56	17.01	16.72
	Server Computation (ms)	512	7995	N/A	1438.18	1229.88
	Client Computation (ms)	2174	20	N/A	1.38	2.04
	Total Computation (ms)	2686	8015	N/A	1439.56	1231.92

Comparison with Cuckoo+VIA. Compared to the combined keyword PIR scheme Cuckoo+VIA [3, 34], LetoPIR reduces the communication cost by $2.3\times \sim 2.4\times$ and the computation time by $12.9\times \sim 13.9\times$, for databases containing $2^{20} \sim 2^{24}$ records of 256 bytes. While Cuckoo+VIA executes the index-PIR scheme VIA three times independently for each query, LetoPIR uses only a single execution of index PIR. For Cuckoo+VIA, the performance comparison ignores the extra cost of Cuckoo hashing, and uses the parameters for the 110-bit security level, making the reported improvement very conservative.

Comparison with ChalametPIR and KPIR. ChalametPIR [14] and KPIR [27] are built on FrodoPIR [18] and SimplePIR [29], respectively. Both schemes generate database-dependent hints in the setup phase, where each client must download and store the hints. In contrast, LetoPIR is hintless and avoids storing any database-dependent hint on the client side. To generate a query, a client needs to construct a random matrix by performing $O(N)$ encryption evaluations in ChalametPIR. LetoPIR instead generates the query using a logarithmic-communication k-hot vector generation procedure, which incurs only logarithmic client-side computation.

For databases ranging from 2^{20} to 2^{24} entries (each of 256 bytes), LetoPIR reduces the communication cost by a factor of $1{,}658\times \sim 25{,}600\times$ compared to ChalametPIR and by a factor of $51.4\times \sim 184.8\times$ compared to KPIR. This substantial reduction in communication is a key advantage of our protocol, since communication is often the efficiency bottleneck for limited network bandwidths

or issuing a lot of keyword queries. LetoPIR also improves the computation cost by 27.8× ∼ 165.9× compared to ChalametPIR. Compared with KPIR, LetoPIR requires 0.9× ∼ 1.0× computation cost.

Performance evaluation in the online phase for different record sizes. Table 5 compares the performance among different keyword PIR schemes on databases with 2^{18} records and record sizes across 64, 1024, and 4096 bytes. The VIA implementation [35] does not support record sizes different from 256 bytes, and thus this table does not compare LetoPIR with Cuckoo+VIA.

In the online phase, LetoPIR achieves the lowest communication cost across all evaluated record sizes. Compared with SparsePIR, KPIR, and ChalametPIR, it reduces communication by 3.4× ∼ 12.9×, 9.9× ∼ 36.6×, and 69.7× ∼ 423.6×, respectively. These results indicate that the communication advantage of LetoPIR persists across different record sizes.

For record sizes of 1024 and 4096 bytes, LetoPIR also achieves the lowest computation cost. It reduces computation by 3.5× and 6.5× compared with KPIR, and by 2.2× ∼ 4.8× compared with ChalametPIR.

Additional experiments. We defer the evaluation of bucket merging, the setup-phase results under varying record sizes, and the comparison with constant-weight-code-based keyword PIR schemes to Appendix G.

Acknowledgments

We thank OpenAI ChatGPT and Codex for improving writing and optimizing the code. The correctness of all content and code was verified by the authors.

References

1. Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.
2. Martin R. Albrecht, Rachel Player, and Sam Scott. LWE estimator. `https://github.com/malb/lwe-estimator`, 2017. Accessed: 2023-12-01.
3. Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. pages 1811–1828, 2021.
4. Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*, pages 962–979. IEEE, 2018.
5. Apple Inc. What's new in privacy (wwdc24 session 10123). `https://developer.apple.com/videos/play/wwdc2024/10123/`, 2024. Accessed: 2026-04-02.
6. Ali Arastehfard, Weiran Liu, Qixian Zhou, Zinan Shen, Liqiang Peng, Lin Qu, Shuya Feng, and Yuan Hong. Kpir-c: Keyword pir with arbitrary server-side computation. *Cryptology ePrint Archive*, 2025.

7. Hilal Asi, Fabian Boemer, Nicholas Genise, Muhammad Haris Mughees, Tabitha Ogilvie, Rehan Rishi, Kunal Talwar, Karl Tarbe, Akshay Wadia, Ruiyu Zhu, et al. Scalable private search with wally. *arXiv preprint arXiv:2406.06761*, 2024.
8. Olivier Bernard and Marc Joye. Approximate crt-based gadget decomposition and application to tfhe blind rotation. *Cryptology ePrint Archive*, 2024.
9. Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. {Near-Optimal} oblivious {Key-Value} stores for efficient {PSI},{PSU} and {Volume-Hiding}{Multi-Maps}. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 301–318, 2023.
10. Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. Intel HEXL: Accelerating homomorphic encryption with Intel AVX512-IFMA52. Cryptology ePrint Archive, Report 2021/420, 2021. `https://eprint.iacr.org/2021/420`.
11. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual cryptology conference*, pages 868–886. Springer, 2012.
12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
13. Alexander Burton, Samir Jordan Menon, and David J Wu. Respire: High-rate pir for databases with small records. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1463–1477, 2024.
14. Sofía Celi and Alex Davidson. Call me by my name: Simple, practical private information retrieval for keyword queries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4107–4121, 2024.
15. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*, pages 3–33. Springer, 2016.
16. Benny Chor, Niv Gilboa, and Moni Naor. *Private information retrieval by keywords*. Technion-IIT, Department of Computer Science, 1997.
17. Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.
18. Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. *Proceedings on Privacy Enhancing Technologies*, 2023.
19. Leo de Castro, Kevin Lewi, and Edward Suh. Whispir: Stateless private information retrieval with low communication. *Cryptology ePrint Archive*, 2024.
20. Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 617–640. Springer, 2015.
21. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.
22. Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. pages 395–425, 2021.
23. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

24. Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P Smart. Ring switching in bgv-style homomorphic encryption. In *International Conference on Security and Cryptography for Networks*, pages 19–37. Springer, 2012.
25. Thomas Mueller Graf and Daniel Lemire. Binary fuse filters: Fast and smaller than xor filters. *Journal of Experimental Algorithmics (JEA)*, 27(1):1–15, 2022.
26. Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. pages 83–105, 2019.
27. Meng Hao, Weiran Liu, Liqiang Peng, Cong Zhang, Pengfei Wu, Lei Zhang, Hongwei Li, and Robert H Deng. Practical keyword private information retrieval from {Key-to-Index} mappings. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3397–3416, 2025.
28. Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *Proceedings of the 29th symposium on operating systems principles*, pages 396–416, 2023.
29. Alexandra Henzinger, Matthew M Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast {Single-Server} private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3889–3905, 2023.
30. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography from anonymity. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 239–248. IEEE, 2006.
31. David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on computing*, 3(4):299–325, 1974.
32. Albert Hyukjae Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. 2015.
33. Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. Hintless single-server private information retrieval. In *Annual International Cryptology Conference*, pages 183–217. Springer, 2024.
34. Chenyang Liu, Xukun Wang, and Zhifang Zhang. Via: Communication-efficient single-server private information retrieval. In *2026 IEEE symposium on security and privacy (SP)*. IEEE, 2026.
35. Chenyang Liu, Xukun Wang, and Zhifang Zhang. VIA: Communication-efficient single-server private information retrieval. `https://github.com/owniai/VIA`, 2026. Accessed: August 12, 2026.
36. Ming Luo, Feng-Hao Liu, and Han Wang. Faster fhe-based single-server private information retrieval. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1405–1419, 2024.
37. Vadim Lyubashevsky, Chris Peikert, and Oded 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*, pages 1–23. Springer, 2010.
38. Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators. pages 1723–1740, 2022.
39. Rasoul Akhavan Mahdavi, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Inspire: Communication-efficient pir with server-side preprocessing. *Cryptology ePrint Archive*, 2025.
40. Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, pages 155–174, 2016.

41. Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE symposium on security and privacy (SP)*, pages 930–947. IEEE, 2022.
42. Samir Jordan Menon and David J Wu. {YPIR}:{High-Throughput}{Single-Server}{PIR} with silent preprocessing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5985–6002, 2024.
43. Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 700–718. Springer, 2012.
44. Gabrielle De Micheli, Duhyeong Kim, Daniele Micciancio, and Adam Suhl. Faster amortized FHEW bootstrapping using ring automorphisms. Cryptology ePrint Archive, Paper 2023/112, 2023.
45. Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. {PIR-Tor}: Scalable anonymous communication using private information retrieval. In *20th USENIX security symposium (USENIX security 11)*, 2011.
46. Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 2292–2306, 2021.
47. Sarvar Patel, Joon Young Seo, and Kevin Yeo. {Don't} be dense: efficient keyword {PIR} for sparse databases. In *32nd USENIX security symposium (USENIX Security 23)*, pages 3853–3870, 2023.
48. Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. pages 739–767, 2020.
49. Martin Raab and Angelika Steger. "balls into bins"—a simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.
50. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.
51. Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *arXiv preprint arXiv:2004.13293*, 2020.

A Related Work

HE-based Index PIR PIR protocols with client hints require the server to precompute these hints during the setup phase to reduce online computation. However, they incur higher communication overhead than hintless ones. Henzinger et al. introduced SimplePIR [29], which encodes the database as a matrix and retrieves an entire column per query, thereby achieving $O(\sqrt{N})$ communication complexity. This results in high communication costs for large database sizes. Subsequent works allow the client to precisely locate the index in the encoded database to retrieve a single record per query. Meanwhile, some works make full use of homomorphic packing techniques to reduce the total communication cost [33, 18, 42, 39].

Hintless PIR protocols pre-share public parameters derived from the secret key during the setup phase. In the online phase, the client transmits a compressed query, and the server reconstructs the full query homomorphically, thereby reducing online communication [4]. The server computes and returns a

high-rate response. Subsequent works aim to further optimize online communication overhead [40, 46, 41, 13, 34, 36, 19]. Most of them achieve $O(\sqrt{N})$ upload communication complexity with constant download complexity. More recently, VIA proposed a scheme with logarithmic upload complexity and constant download complexity [34].

Keyword PIR Keyword PIR can be constructed by combining an arbitrary search data structure [16, 3] (e.g., hash tables or trees) with any underlying index PIR scheme, thereby enabling private access by keywords without changing how the database is maintained. Another approach constructs keyword PIR using homomorphic encryption and constant weight codes [38, 6]. These constructions suffer from high computational costs and large ciphertext sizes.

Recently, Celi et al. proposed ChalametPIR [14]. ChalametPIR is built on Binary Fuse Filter [25], enabling retrieval with a single FrodoPIR invocation [18] with high throughput. However, its upload communication and client computation remain linear in the number of records. Hao et al. proposed KPIR [27], which improves the construction of ChalametPIR and achieves $O(\sqrt{N})$ communication. Since the keyword position cannot be precisely localized, KPIR still requires downloading an entire column of data. Arastehfard et al. proposed a TFHE-based keyword PIR scheme [6]. Their constructions support arbitrary circuit evaluation on the server side, which resolves the issue of specifying a default value when the keyword is absent. However, the protocol still incurs $O(\sqrt{N})$ download communication. Angel et al. proposed SparsePIR [47], a keyword PIR protocol based on the Spiral [41]. It achieves $O(\sqrt{N})$ upload communication and constant download communication without client hint.

B Definition of Keyword PIR

Let $\mathcal{K}$ and $\mathcal{M}$ denote the keyword space and record space, respectively, and let $\mathsf{DB}[\hat{k}] \in \mathcal{M}$ be the record associated with $\hat{k} \in \mathcal{K}$. A keyword PIR protocol is a tuple $(\mathsf{Setup_C}, \mathsf{Setup_S}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Recover})$ with the following syntax:

- $(c_s, s_s) \leftarrow \mathsf{Setup_C}(1^\lambda)$: On input the security parameter λ, output a query key c_s and switching material s_s.
- $(pp_s, pp_c) \leftarrow \mathsf{Setup_S}(1^\lambda, \mathsf{DB})$: On input the security parameter λ and the database DB, output server-side parameters pp_s and client-side parameters pp_c.
- $qu \leftarrow \mathsf{Query}(pp_c, \hat{k}, c_s)$: On input client-side parameters pp_c, a keyword $\hat{k} \in \mathcal{K}$, and the query key c_s, output a query qu.
- $res \leftarrow \mathsf{Answer}(pp_s, s_s, qu)$: On input server-side parameters pp_s, switching material s_s, and the query qu, output a response res.
- $m \leftarrow \mathsf{Recover}(res, \hat{k}, c_s)$: On input the response res, the keyword $\hat{k}$, and the query key c_s, output a record m or $\perp$.

A keyword PIR protocol satisfies the following properties:

1. **Correctness**. For every λ, database DB, and keyword $\hat{k} \in \mathcal{K}$:
   (i) If $\hat{k} \in \mathsf{DB}$, then

$$\Pr[\mathsf{Recover}(res, \hat{k}, c_s) = \mathsf{DB}[\hat{k}]] \geq 1 - \mathsf{negl}(\lambda),$$

   (ii) If $\hat{k} \notin \mathsf{DB}$, then

$$\Pr[\mathsf{Recover}(res, \hat{k}, c_s) = \perp] \geq 1 - \mathsf{negl}(\lambda),$$

   where

$$\begin{aligned} (c_s, s_s) &\leftarrow \mathsf{Setup}_{\mathsf{C}}(1^\lambda), \\ (pp_s, pp_c) &\leftarrow \mathsf{Setup}_{\mathsf{S}}(1^\lambda, \mathsf{DB}), \\ qu &\leftarrow \mathsf{Query}(pp_c, \hat{k}, c_s), \\ res &\leftarrow \mathsf{Answer}(pp_s, s_s, qu). \end{aligned}$$

2. **Query Privacy**. For any two keywords $\hat{k}, \hat{k}' \in \mathcal{K}$, any probabilistic polynomial-time adversary $\mathcal{A}$, and setup outputs (pp_s, pp_c, s_s), the distributions $\mathsf{Query}(pp_c, \hat{k}, c_s)$ and $\mathsf{Query}(pp_c, \hat{k}', c_s)$ are computationally indistinguishable. In particular, let

$$\begin{aligned} Q &:= \mathsf{Query}(pp_c, \hat{k}, c_s), \\ Q' &:= \mathsf{Query}(pp_c, \hat{k}', c_s), \\ \Delta &:= |\Pr[\mathcal{A}(pp_s, pp_c, s_s, Q) = 1] \\ &\quad - \Pr[\mathcal{A}(pp_s, pp_c, s_s, Q') = 1]|. \end{aligned} \tag{2}$$

   Then $\Delta \leq \mathsf{negl}(\lambda)$.

C Security Proofs

Proof (Proof of Theorem 1). Fix a query keyword $\hat{k}$. Let $\tilde{b}^\star$ be the merged bucket selected by the public hash and merging maps, and let $(i^\star, s^\star)$ be the packed-bucket index and slot index determined by the public packing map. By construction, the query generated by Query encodes exactly the KVS decoding information associated with $\hat{k}$ inside $\tilde{b}^\star$.

By the correctness of the underlying KVS scheme, homomorphic decoding over the encoded contents of $\tilde{b}^\star$ returns $\mathsf{hash}(\hat{k}) \,\|\, r$ if $(\hat{k}, r) \in \mathsf{DB}$. For a keyword not in the database, the decoded value may be arbitrary, but absent-key soundness ensures that it has the required digest only with negligible probability. Since bucket packing only changes the layout of encoded buckets, the decoded value appears in slot $s^\star$ of packed bucket $i^\star$.

The server then applies two levels of homomorphic selection: the first selects packed bucket $i^\star$, and the second extracts slot $s^\star$ from that packed bucket. By the correctness of CMUX and ring switching, these operations preserve the underlying plaintext. Hence, the response encrypts the decoded record corresponding to $\hat{k}$.

The client decrypts the response and verifies the hash digest. If $(\hat{k}, r) \in \mathsf{DB}$, it outputs r; otherwise, it outputs $\perp$, except with negligible probability due to decryption failure or an absent-key soundness error. Therefore, the protocol satisfies correctness.

Proof (Proof of Theorem 2). Let $(c_s, s_s) \leftarrow \mathsf{Setup}_{\mathsf{C}}(1^\lambda)$ and $(pp_s, pp_c) \leftarrow \mathsf{Setup}_{\mathsf{S}}(1^\lambda, \mathsf{DB})$. For a keyword $\hat{k}$, let $m_{\hat{k}}$ denote the plaintext query representation derived from $\hat{k}$, including the encoded decoding information and the encoded selection information used by the client. Then the transmitted query consists of fresh RLWE ciphertexts and compressed representations derived from fresh RLWE ciphertexts. Since the compression procedure is public post-processing, it preserves computational indistinguishability. Likewise, for another keyword $\hat{k}'$, let $m_{\hat{k}'}$ denote the corresponding plaintext query representation. The values (pp_s, pp_c, s_s) are generated before the query and have the same distribution regardless of whether the queried keyword is $\hat{k}$ or $\hat{k}'$. In particular, s_s consists of public evaluation material derived from the client secret key and is independent of the queried keyword. By the assumed security of the underlying RLWE-based encryption scheme in the presence of such evaluation material, encryptions of $m_{\hat{k}}$ and $m_{\hat{k}'}$ are computationally indistinguishable. Therefore,

$$\mathsf{Query}(pp_c, \hat{k}, c_s) \approx_c \mathsf{Query}(pp_c, \hat{k}', c_s),$$

even given (pp_s, pp_c, s_s).

Hence, no PPT adversary can distinguish whether the client queried $\hat{k}$ or $\hat{k}'$ except with negligible probability. This proves query privacy.

D Key-Value Stores

Definition 1 (Key-Value Store). *A key-value store is parameterized by a key space $\mathcal{K}$ and a value space $\mathcal{M}$ and consists of two algorithms:*

- $(D, H) \leftarrow \mathsf{Encode}(L)$ *takes a set of N key–value pairs $L = \{(\hat{k}_i, r_i)\}_{i \in [N]} \in (\mathcal{K} \times \mathcal{M})^N$ and outputs an encoding $D \in \mathcal{M}^{M_{\text{enc}}}$ and public hash information H.*
- $y \leftarrow \mathsf{Decode}(D, \hat{k}, H)$ *takes D, a key $\hat{k} \in \mathcal{K}$, and H, and outputs a value $y \in \mathcal{M}$.*

Correctness requires that, for every $(\hat{k}, r) \in L$,

$$(D, H) \leftarrow \mathsf{Encode}(L),$$
$$\Pr[\mathsf{Decode}(D, \hat{k}, H) = r] \geq 1 - \mathsf{negl}(\lambda).$$

For keys outside L, we additionally require the tagged KVS to satisfy absent-key soundness.

Definition 2 (Absent-Key Soundness). *A tagged KVS satisfies absent-key soundness if, for every $\hat{k} \notin L$,*

$$\Pr\left[\mathsf{Decode}(D, \hat{k}, H) = \mathsf{hash}(\hat{k}) \,\|\, r\right] \leq \mathsf{negl}(\lambda),$$

where $(D, H) \leftarrow \mathsf{Encode}(L)$.

To detect a missing keyword, our protocol encodes $\mathsf{hash}(\hat{k}) \,\|\, r$ rather than r. We require the resulting tagged KVS to satisfy Definition 2. The client checks the digest after decoding and returns $\perp$ if it does not match the queried keyword.

We present the concrete algorithms of Binary Fuse Filters (BFF) [25] and Random Band Oblivious Key-Value Stores (RB-OKVS) [9], which are used as key-to-value encoding primitives in our constructions. For completeness, we give both the encoding and decoding procedures.

We first describe the BFF-based construction. Similar to Appendix C in [27], the encoding stores each key-value pair into α locations determined by public hash functions, and decoding only accesses these α positions. Concretely, for a key-value list

$$L := \{(k_i, r_i)\}_{i \in [n]},$$

the encoder outputs an array D together with α hash functions $h_0, ..., h_{\alpha-1}$. For a queried key k, the decoder reads $D[h_0(k)], ..., D[h_{\alpha-1}(k)]$, and reconstructs the corresponding record by summing these entries. We describe the encoding and decoding procedure in Alg. 8 and Alg. 9, respectively.

The correctness argument is standard. For any inserted key k, the α touched positions satisfy

$$\sum_{i \in [\alpha]} D[h_i(k)] = \mathsf{hash}(k) \,\|\, r.$$

Hence, the decoder recovers r correctly. For a non-inserted key, the decoder outputs $\perp$ except with negligible probability, due to either a collision in the position tuple $(h_0(k), ..., h_{\alpha-1}(k))$ or a collision in the digest $\mathsf{hash}(k)$.

We next describe the RB-OKVS-based construction. We describe the encoding and decoding procedure in Alg. 10 and Alg. 11, respectively. At a high level, RB-OKVS encodes all key-value pairs into a vector $D \in \mathbb{F}^m$ by solving a sparse linear system

$$\mathbf{M} \cdot D = y,$$

where each row of $\mathbf{M}$ corresponds to one key and has non-zero entries only in a short random band. This is the key feature that makes the encoding both compact and efficiently solvable.

For each key k, let

$$\mathsf{pos}(k) \in [m - \omega + 1]$$

denote the starting position of its band, and let

$$\mathsf{coef}(k) = (\gamma_{k,0}, \ldots, \gamma_{k,\omega-1}) \in \mathbb{F}^{\omega}$$

Algorithm 8: $\mathsf{Encode}(L)$

Input: The key-value database $L = \{(\hat{k}_i, r_i)\}_{i\in[n]}$
Output: An encoded database D, hash functions H

while *true* **do**
  generate m buckets C, stack P, Q
  sample α hash functions $h_0, ..., h_{\alpha-1} : \mathcal{K} \to [m]$
  for $i \in [n], j \in [\alpha]$, push $\hat{k}_i$ to $C[h_j(\hat{k}_i)]$
  for $i \in [m]$, push i to Q if $|C[i]| = 1$
  while $|Q| > 0$ **do**
    $i \leftarrow$ pop the top element of Q
    if $|C[i]| = 1$ **then**
      set $\hat{k} \leftarrow C[i]$ and push $(\hat{k}, i)$ to P
      for $j \in [\alpha]$, remove $\hat{k}$ from $C[h_j(\hat{k})]$
      for $j \in [\alpha]$, push $h_j(\hat{k})$ to Q if $|C[h_j(\hat{k})]| = 1$
    end
  end
  break if $|P| = n$
end
generate new database D
while $|P| > 0$ **do**
  pop $(\hat{k}, i)$ from P and set $\hat{H} = \{h_j(\hat{k}) \mid j \in [\alpha]\} \setminus \{i\}$
  for $x \in \hat{H}$, sample $D[x] \leftarrow \mathcal{M}$ if $D[x]$ is empty
  set $D[i] = \mathsf{hash}(\hat{k})||r - \sum_{x\in\hat{H}} D[x]$, where $(\hat{k}, r) \in L$
end
for $i \in [m]$, sample $D[i] \leftarrow \mathcal{M}$ if $D[i]$ is empty
return $D, H = (h_0, ..., h_{\alpha-1})$

Algorithm 9: $\mathsf{Decode}(D, \hat{k}, H)$

Input: The encoded D, the query keyword $\hat{k}$, and the hash functions $H = \{h_i\}_{i\in[\alpha]}$
Output: A record r

$h'||r \leftarrow \sum_{i\in[\alpha]} D[h_i(\hat{k})]$
if $h' = \mathsf{hash}(\hat{k})$ **then**
  return r
end
return $\perp$

Algorithm 10: $\mathsf{Encode}(L)$

Input: The key-value database $L = \{(\hat{k}_i, r_i)\}_{i\in[n]}$
Output: An encoded database D, hash functions H

Initialize a sparse matrix $\mathbf{M} \in \mathbb{F}^{n\times m}$ mapping row indices to non-zero entries.
for $i \in [n]$ **do**
  $a_i \leftarrow h_1(\hat{k}_i), \mathbf{u}_i \leftarrow h_2(\hat{k}_i)$
  for $j \in [\omega]$ **do**
    $\mathbf{M}[i][a_i + j] \leftarrow \mathbf{u}_i[j]$
  end
end
Sort rows of $\mathbf{M}$ according to first non-zero location
Solve the linear system $\mathbf{M}D = (y_0, \ldots, y_{n-1})^T$ using Gaussian elimination.
return $D, H = (h_1, h_2)$

Algorithm 11: $\mathsf{Decode}(D, \hat{k}, H)$

Input: The encoded D, the query keyword $\hat{k}$, and the hash functions $H = (h_1, h_2)$
Output: A decoded encoded record y

$a \leftarrow h_1(\hat{k}), \mathbf{u} \leftarrow h_2(\hat{k})$
$y \leftarrow 0$
for $j \in [\omega]$ **do**
  $y \leftarrow y + (\mathbf{u}[j] \cdot D[a + j])$
end
return y

denote the non-zero coefficients in that band, where both are derived pseudo-randomly from public hash functions. The row associated with k is then

$$\mathbf{m}_k = (0, \ldots, 0, \gamma_{k,0}, \ldots, \gamma_{k,\omega-1}, 0, \ldots, 0) \in \mathbb{F}^m,$$

whose support is exactly the interval

$$[\mathsf{pos}(k),\ \mathsf{pos}(k) + \omega - 1].$$

The target record is set to

$$y_k := \mathsf{hash}(k) \,\|\, r_k.$$

The correctness follows directly from the construction. For any inserted key k_i, the row $\mathbf{m}_{k_i}$ is included in $\mathbf{M}$, and therefore

$$\sum_{t\in[\omega]} \gamma_{k_i,t} \cdot D[\mathsf{pos}(k_i) + t] = \mathbf{m}_{k_i} \cdot D = y_i = \mathsf{hash}(k_i) \,\|\, r_i.$$

Thus, decoding returns the encoded record y_i. For any non-inserted key, the decoding output does not pass the outer hash verification except with negligible probability, assuming that $\mathsf{hash}(\cdot)$ is collision resistant and that the derived band representation does not accidentally decode to a valid digest.

Algorithm 12: $\mathsf{FFD}(\{a_i\}_{i\in[n]})$

Input: Items $\{a_i\}_{i\in[n]}$ with sizes $s(a_i)$ and bin capacity *cap*
Output: A set of bins B

Sort items such that $s(a_1) \geq s(a_2) \geq \cdots \geq s(a_n)$
$B \leftarrow \emptyset$
for $i \in [n]$ **do**
  $\mathsf{placed} \leftarrow \mathsf{false}$
  for *each* $B_j \in B$ **do**
    if $\mathrm{load}(B_j) + s(a_i) \leq cap$ **then**
      place a_i into B_j
      $\mathsf{placed} \leftarrow \mathsf{true}$
      break
    end
  end
  if $\mathsf{placed} = \mathsf{false}$ **then**
    open a new bin B_{new}
    place a_i into B_{new}
    $B \leftarrow B \cup \{B_{\mathrm{new}}\}$
  end
end
return B

E First Fit Decreasing

In this appendix, we describe the First Fit Decreasing (FFD) [31] algorithm, a classical greedy heuristic for the bin packing problem. Given a set of items with sizes and a bin capacity *cap*, FFD first sorts the items in descending order of size, and then iteratively places each item into the first bin that can accommodate it; if no such bin exists, a new bin is opened. Despite its simplicity, FFD achieves good empirical performance and provides a well-known approximation guarantee for bin packing.

Formally, let the item set be

$$I = \{a_1, a_2, \ldots, a_n\},$$

where each item a_i has size $\mathsf{len}(a_i) > 0$. Each bucket has a capacity C.

Here,

$$\mathrm{load}(B_j) = \sum_{x \in B_j} \mathsf{len}(x)$$

denotes the current load of bucket B_j.

F Details of Homomorphic Encryption

Definition 3 (Learning With Errors [50]). *Let $n = n(\lambda)$ be the dimension, $q = q(\lambda)$ be a modulus, and $\mathcal{X} = \mathcal{X}(\lambda)$ denote a small discrete Gaussian distribution over $\mathbb{Z}_q$. The LWE assumption states that the following two distributions*

are computationally indistinguishable:

$$\{(\mathbf{a}_i, \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i)\}_{i \in [n]} \approx \{(\mathbf{a}_i, z_i)\}_{i \in [n]}$$

where $\mathbf{a}_i \leftarrow \mathbb{Z}_q^n$, $\mathbf{s} \leftarrow \mathcal{X}^n$, $e_i \leftarrow \mathcal{X}$, *and* $z_i \leftarrow \mathbb{Z}_q$.

Definition 4 (Ring Learning With Errors [37]). *Let* $d = d(\lambda)$ *be a power of two, and define the ring* $R_q = \mathbb{Z}_q[X]/(X^d + 1)$. *Let* $\mathcal{X} = \mathcal{X}(\lambda)$ *denote a discrete Gaussian distribution over* R_q. *The RLWE assumption states that the following two distributions are computationally indistinguishable:*

$$\{(a_i, a_i \cdot s + e_i)\}_{i \in [n]} \approx \{(a_i, z_i)\}_{i \in [n]}$$

where $a_i \leftarrow R_q$, $s \leftarrow \mathcal{X}$, $e_i \leftarrow \mathcal{X}$, *and* $z_i \leftarrow R_q$.

Definition 5 (Gadget Decomposition [43]). *Let* $q \in \mathbb{Z}$ *be a modulus and* $\beta \in \mathbb{Z}$ *a decomposition base. Define* $\ell = \lceil \log_\beta q \rceil$, *and the* gadget vector *as* $\mathbf{g} = (1, \beta, \beta^2, \ldots, \beta^{\ell-1}) \in \mathbb{Z}_q^\ell$. *For* $a \in R_q$, *its* gadget decomposition *is*

$$\mathsf{Decomp}(a) = \left(\left[\left\lfloor \frac{a}{\beta^i} \right\rceil \right]_\beta \right)_{i=0}^{\ell-1} \in R_\beta^\ell,$$

which satisfies $\langle \mathbf{g}, \mathsf{Decomp}(a) \rangle = [a]_q$.

Definition 6 (RNS Gadget Decomposition [26]). *Let* $Q = \prod_{i=0}^{L-1} q_i$, *where the moduli* q_i *are pairwise coprime. For each* i, *define*

$$\widehat{q}_i = Q/q_i, \qquad g_{\mathsf{RNS},i} = \widehat{q}_i [\widehat{q}_i^{-1}]_{q_i} \bmod Q.$$

The RNS gadget vector is $\mathbf{g}_{\mathsf{RNS}} = (g_{\mathsf{RNS},0}, \ldots, g_{\mathsf{RNS},L-1})$. *For* $a \in R_Q$, *define its coefficient-wise RNS decomposition as*

$$\mathsf{Decomp}_{\mathsf{RNS}}(a) = \big([a]_{q_0}, \ldots, [a]_{q_{L-1}}\big).$$

By the Chinese remainder theorem,

$$\langle \mathbf{g}_{\mathsf{RNS}}, \mathsf{Decomp}_{\mathsf{RNS}}(a) \rangle = [a]_Q.$$

F.1 Ciphertext Structure

We use three ciphertext types RLWE, Rlev, and RGSW from TFHE [15]. Unless otherwise specified, all ciphertexts are encrypted under a secret key s. For a decomposition basis $\gamma \in \{\mathsf{gadget}, \mathsf{RNS}\}$, let $\mathbf{g}_\gamma$ denote its gadget vector and let ℓ_γ denote its length, where $\mathbf{g}_{\mathsf{gadget}} = \mathbf{g}$, $\ell_{\mathsf{gadget}} = \ell$, and $\ell_{\mathsf{RNS}} = L$.

– $\mathsf{RLWE}(m, s)$: Sample $c_0 \leftarrow R_q$, $e \leftarrow \mathcal{X}$, and output

$$ct = (c_0, c_0 \cdot s + e + m) = (c_0, c_1).$$

– $\mathsf{Rlev}^{\gamma}(m, s)$: Output

$$\mathsf{Rlev}^{\gamma}(m, s) = \{\mathsf{RLWE}(\mathbf{g}_{\gamma,i} \cdot m, s)\}_{i=0}^{\ell_\gamma - 1}.$$

– $\mathsf{RGSW}^{\gamma}(m, s)$: Output

$$\mathsf{RGSW}^{\gamma}(m, s) = \big(\mathsf{Rlev}^{\gamma}(s \cdot m, s), \mathsf{Rlev}^{\gamma}(m, s)\big).$$

We omit the basis superscript when it is clear from context.

We adopt BFV-style encoding [21], where ciphertexts are of the form $ct = \mathsf{RLWE}(\Delta \cdot m, s)$ for $\Delta = \lfloor q/p \rfloor$ and plaintext modulus p. Given $ct = (c_0, c_1)$ and s, decryption computes

$$\mathsf{Decrypt}(ct, s) = \left\lfloor \frac{c_1 - c_0 \cdot s}{\Delta} \right\rceil.$$

Decryption is correct if the noise has ℓ_∞-norm below $\Delta/2$.

F.2 Basic Homomorphic Operations

All ciphertexts follow the BFV-style RLWE encryption. We use the following standard operations [11, 21, 12]:

– $\mathsf{Add}(ct_1, ct_2)$: Given $ct_1 = (c_0^1, c_1^1)$ and $ct_2 = (c_0^2, c_1^2)$, return

$$ct = (c_0^1 + c_0^2,\ c_1^1 + c_1^2).$$

– $\mathsf{PCMul}(pt, ct)$: Given $pt \in R_q$ and $ct = (c_0, c_1)$, return

$$ct' = (pt \cdot c_0,\ pt \cdot c_1).$$

– $\mathsf{KeySwitch}(ct, \mathsf{ksk})$: Given $ct = \mathsf{RLWE}(m, s')$ and $\mathsf{ksk} = \mathsf{Rlev}(s', s)$, return $ct' = \mathsf{RLWE}(m, s)$.
– $\mathsf{Auto}(ct, i)$: Given $ct = \mathsf{RLWE}(m, s)$ and $i \in \mathbb{Z}$, return $ct' = \mathsf{RLWE}(\sigma_i(m), s)$, where $\sigma_i : X \to X^i$.

F.3 Modulus Switching

Given an RLWE ciphertext $ct = (c_0, c_1) \in R_{d,Q}^2$ and a smaller modulus $q < Q$, modulus switching [12] outputs

$$\mathsf{ModSwitch}_{Q \to q}(ct) = \left(\left\lfloor \tfrac{q}{Q} c_0 \right\rceil,\ \left\lfloor \tfrac{q}{Q} c_1 \right\rceil\right) \in R_{d,q}^2,$$

where rounding is performed coefficient-wise.

We next describe the Expand procedure, ring switching, the RLWE-to-RGSW transformation ($\mathsf{RLWE2RGSW}$), and polynomial embedding.

F.4 Expand

The Expand [4] procedure takes an RLWE ciphertext and produces a set of ciphertexts that encode its coefficient-wise decomposition. Given a ciphertext ct encrypting a polynomial $m(X)$, the algorithm outputs a collection $\{ct_j\}$ such that each ct_j corresponds to a rotated and scaled component of m. This is achieved by iteratively applying automorphisms and combining the results to isolate different coefficient positions. The final normalization step ensures that the outputs collectively preserve the original message.

Algorithm 13: $\mathsf{Expand}(ct, r)$

Input: Ciphertext ct and $0 \leq r \leq \log_2 d$
Output: Ciphertexts $\{ct_j\}_{j\in[2^r]}$

$ct_0 \leftarrow ct$
for $i = 0, ..., r-1$ **do**
  $\ell \leftarrow d/2^i + 1$
  for $j = 0, ..., 2^i - 1$ **do**
    $ct' \leftarrow \mathsf{Auto}(ct_j, \ell)$
    $ct_{j+2^i} \leftarrow (ct_j - ct') \cdot X^{-2^i}$
    $ct_j \leftarrow ct_j + ct'$
  end
end
$ct_j \leftarrow ct_j \cdot 2^{-r}$
return $\{ct_j\}_{j\in[2^r]}$

F.5 Ring Switch

Let $\iota : R_{d_2,q} \to R_{d_1,q}$ denote the embedding from the smaller ring to the larger ring, and let ι^{-1} denote the induced coefficient map on embedded ciphertext components. We model ring switching [24] as projection to the smaller ring followed by a key-switching step.

Algorithm 14: $\mathsf{RingSwitch}(ct, \mathsf{ksk})$

Input: An RLWE ciphertext $ct = (c_0, c_1) \in R^2_{d_1,q}$ and a key-switching key ksk from $\iota^{-1}(s_1)$ to s_2
Output: An RLWE ciphertext $ct' = (c'_0, c'_1) \in R^2_{d_2,q}$

$(\hat{c}_0, \hat{c}_1) \leftarrow (\iota^{-1}(c_0), \iota^{-1}(c_1))$
$ct' \leftarrow \mathsf{KeySwitch}((\hat{c}_0, \hat{c}_1), \mathsf{ksk})$
return ct'

F.6 RLWE2RGSW

The RLWE2RGSW [44] transformation converts a collection of RLWE ciphertexts into an RGSW ciphertext under the same secret key. Given an RLWE encryption of a message m represented as a set of ciphertexts $\{ct_j\}$, the algorithm produces an RGSW encryption by decomposing the ciphertext components and combining them with the relevant switching material. Concretely, each RLWE component is decomposed into smaller digits, which are then linearly combined with the corresponding key material to encode the multiplicative structure required by the RGSW representation.

Algorithm 15: $\mathsf{RLWE2RGSW}(\{ct_j\}_{j\in[t]}, \mathsf{ksk})$

Input: A set of ciphertexts $\{ct_j\}_{j\in[t]}$ and the relevant switching material ksk
Output: A RGSW ciphertext $\mathsf{RGSW}(m, s)$

$\mathsf{Rlev}(m, s) \leftarrow (ct_0, ..., ct_{t-1})$
for $i \leftarrow 0, ..., t-1$ **do**
  $(c_0, c_1) \leftarrow ct_i$
  $\{a_j\} \leftarrow \mathsf{Decompose}(c_0)$
  $\mathsf{Rlev}(s \cdot m, s)_i = (c_1, 0) + \langle\{a_j\}, \mathsf{ksk}\rangle$
end
return $[\mathsf{Rlev}(s \cdot m, s), \mathsf{Rlev}(m, s)]$

F.7 Polynomial Embedding

Given ρ polynomials $f_1, \ldots, f_\rho$, each of degree n_1, the polynomial embedding algorithm outputs a single polynomial f of degree $n_2 = \rho n_1$. We construct f by interleaving the coefficients of $f_1, \ldots, f_\rho$. Specifically, for each coefficient index i, we place the i-th coefficients of $f_1, \ldots, f_\rho$ consecutively in f. As a result, the coefficient of f at position $i \cdot \rho + (j-1)$ is set to the i-th coefficient of f_j.

Algorithm 16: $\mathsf{Interleave}(\{f_j\}_{j\in[\rho]})$

Input: Polynomials $\{f_j\}_{j\in[\rho]}$, each of degree n_1
Output: A polynomial f of degree $n_2 = \rho n_1$

Initialize f with n_2 zero coefficients
for $i \in \{0, \ldots, n_1 - 1\}$ **do**
  for $j \in \{1, \ldots, \rho\}$ **do**
    set $f[i \cdot \rho + (j-1)] \leftarrow f_j[i]$
  end
end
return f

Table 6: Efficiency comparison of keyword PIR protocols in the setup phase across different record sizes. This phase is run only once and can be reused for any polynomial number of queries. We highlight the best performance in yellow and the second-best in blue. SparsePIR is omitted because it has no public implementation and its paper [47] does not report setup efficiency.

Database	**Metric**	**ChalametPIR**	**KPIR**	**LetoPIR-v1**	**LetoPIR-v2**
$2^{18} \times 64$ B	Setup Comm.	0.35 MB	15.15 MB	14.76 MB	14.76 MB
	Setup Comp.	108.1 s	9.8 s	4.2 s	54.1 s
	Client Storage	0.35 MB	15.15 MB	0.16 MB	0.16 MB
$2^{18} \times 1024$ B	Setup Comm.	5.55 MB	114.77 MB	14.67 MB	14.67 MB
	Setup Comp.	2611.8 s	378.1 s	12.7 s	59.2 s
	Client Storage	5.55 MB	114.77 MB	0.07 MB	0.07 MB
$2^{18} \times 4096$ B	Setup Comm.	22.18 MB	333.66 MB	14.65 MB	14.65 MB
	Setup Comp.	13874.4 s	2406.9 s	49.1 s	95.8 s
	Client Storage	22.18 MB	333.66 MB	0.06 MB	0.06 MB

Table 7: Comparison of communication and computation between LetoPIR-v2 and constant-weight-code-based keyword PIR

Scheme	Threads	Setup Comm.(KB)	Setup Comp.(s)	Online Comm.(KB)	Online Comp.(s)
TCWKPIR	64	0.01	8.84	138.04	1901.01
TCAKPIR+	64	0.01	296.81	9225.46	1.72
LetoPIR-v2	1	14417.92	205.83	2.78	0.47

F.8 Coefficient Compression and Expansion

This part corresponds to the query-generation step of our protocol, where the client first compresses the positions selected by the keyword into one RLWE ciphertext and then expands it into a collection of RGSW ciphertexts for the server-side evaluation. Let

$$U = \{u_0, \ldots, u_{k-1}\}$$

be the set of selected positions. Each u_i is represented by an ℓ-bit binary vector

$$u_i = (u_{i,0}, \ldots, u_{i,\ell-1}) \in \{0,1\}^{\ell},$$

where $\ell = \lceil \log n \rceil$. Here k is the number of non-zero positions in the query, $\mathbf{g}$ denotes the public gadget vector, and t is the gadget dimension used in the subsequent RLWE-to-RGSW transformation. Alg. 17 packs the bit decomposition of all elements in U into one RLWE ciphertext c. Alg. 18 then takes c as input, applies coefficient expansion, and converts the resulting RLWE components into $k \log n$ RGSW ciphertexts. These ciphertexts encode the same selection pattern

Table 8: Bucket-merging evaluation over successful trials.

#buckets	2^8	2^9	2^{10}	2^{11}	2^{12}
trivial mean	301.77	582.02	1128.53	2198.50	4321.94
trivial variance	54.49	75.79	99.56	159.23	339.66
merging mean	261.51	518.32	1031.27	2056.78	4105.17
merging variance	10.44	11.01	15.05	13.92	26.63
reduction in $B_{\max}$	13.34%	10.94%	8.62%	6.45%	5.02%

as U, but in a form suitable for the homomorphic operations used in the online phase.

Algorithm 17: $\mathsf{VecGen}(U, s)$

Input: The index set U, the secret key s
Output: A RLWE ciphertext c

$(u_0, ..., u_{k-1}) \leftarrow U$
$p \leftarrow []$
for $i \in [k]$ **do**
  $(u_{i,0}, ..., u_{i,\ell-1}) \leftarrow u_i$
  for $j \in [\ell]$ **do**
    $p \leftarrow p \cup u_{i,j} \cdot \mathbf{g}$
  end
end
return $\mathsf{RLWE}(p, s)$

Algorithm 18: $\mathsf{VecExpand}(c, \mathsf{ksk})$

Input: The query c and the switching material ksk
Output: The RGSW ciphertexts $\{C_j\}_{j \in [k \log n]}$

$\{c_i\}_{i \in [kt \log n]} \leftarrow \mathsf{Expand}(c)$
for $j \in [k \log n]$ **do**
  $C_j \leftarrow \mathsf{RLWE2RGSW}(\{c_{jt+\ell}\}_{\ell \in [t]}, \mathsf{ksk})$
end

G Additional Experimental Results

This section presents additional experimental results. We evaluate performance under a fixed number of records while varying the record size, and report the setup-phase communication cost, computation cost, and client storage overhead.

Evaluation of bucket merging. Table 8 reports the bucket-merging results over successful trials for 2^{2b} balls and 2^b buckets. The *trivial* case uses direct hash-to-bucket assignment, while *bucket merging* applies the technique from Section 4.1. We report the mean and variance of the maximum bucket size across independent trials. Bucket merging reduces the mean of maximum bucket size by $5.02\% \sim 13.34\%$. Let X_i denote the load of bucket i. We define the maximum bucket load as

$$B_{\max} = \max_{i \in [T]} X_i.$$

We define the reduction in the maximum bucket load as

$$\mathsf{Reduction} = \frac{\mathbb{E}[B_{\max}^{\mathsf{trivial}}] - \mathbb{E}[B_{\max}^{\mathsf{merge}}]}{\mathbb{E}[B_{\max}^{\mathsf{trivial}}]} \times 100\%.$$

Performance under varying record sizes in the setup phase Table 6 shows two main trends. First, for both LetoPIR variants, the setup communication and client storage remain nearly unchanged as the record size grows. This is consistent with our design, where the setup information depends mainly on the encoded keyword structure rather than on the record payload. In contrast, the setup cost of ChalametPIR and KPIR increases substantially with the record size. Second, LetoPIR-v1 consistently gives the lowest setup computation among the schemes with sublinear setup communication, while LetoPIR-v2 incurs higher setup time due to the heavier preprocessing used to further reduce the online communication. Even so, both variants keep client storage very small across all tested settings.

Comparison with keyword PIR schemes based on constant-weight codes. We also compare LetoPIR-v2 with the recent keyword PIR schemes TCWKPIR and TCAKPIR+ [6]; the latter additionally supports arbitrary server-side computation. Since the authors do not provide an open-source implementation, we use the numbers reported in their paper instead of rerunning the schemes on our machine. For a database with 2^{20} entries and record size 256 bytes, their paper reports that, under 64 threads, TCWKPIR uses 138.04 KB of communication and 1901.01 s of server online computation, while TCAKPIR+ uses 9225.46 KB and 1.72 s. On our machine, using a single thread, LetoPIR requires 2.78 KB of communication and 0.47 s of online computation. The comparison is shown in Table 7. The communication numbers are directly comparable, but the runtime numbers should be interpreted with care because the thread counts, implementations, and experimental environments are different. Even so, under these respective settings, LetoPIR has much lower communication cost and lower reported online computation time than both TCWKPIR and TCAKPIR+. This suggests that, despite the different design paradigm, our KVS-based logarithmic-communication approach reaches a more favorable concrete operating point for single-record retrieval.