

More Efficient (Hintless) Keyword Private Information Retrieval

Minzhang Li Feng-Hao Liu Guangbei Yi

School of Electrical Engineering and Computer Science,
Washington State University, Pullman

`{minzhang.li, feng-hao.liu, guangbei.yi}@wsu.edu`

Abstract

Keyword private information retrieval (Keyword PIR) enables a client to retrieve the value associated with a keyword from a database while keeping the queried keyword private, thereby generalizing traditional private information retrieval, known as index PIR. The state-of-the-art by Hao et al. (USENIX 2025) has several limitations. First, their generic construction requires three invocations of an underlying index-PIR protocol. Second, to reduce this overhead, they propose a specialized design combining SimplePIR with hash tables. However, this approach inherits SimplePIR’s large client-side hint, resulting in substantial per-database storage costs on the client side. Moreover, it allows clients to retrieve information beyond the value associated with the queried keyword.

In this work, we present a novel and practical Keyword PIR framework that addresses these limitations. Our construction extends the hintless KsPIR scheme of Luo et al. (CCS 2024) to the keyword setting, ensuring that a semi-honest client retrieves only the value corresponding to the queried keyword. The construction leverages the linear homomorphic technique of Peikert and Pepin (TCC 2025). To accelerate homomorphic evaluation, we design a baby-step giant-step (BSGS) implementation and an offline/online decomposition based on a Galois-theoretic formulation. Experimental results show a mean online speedup of $2.65\times$ over the generic framework when instantiated with the same index-PIR scheme, for databases containing up to 2^{22} entries.

1 Introduction

Private information retrieval (PIR) [1] enables a client to retrieve a record from a server-hosted database without revealing which record is being accessed. In recent years, the field of PIR has seen rapid development with numerous works [2–19]. In many practical applications [20–23], however, users issue queries in a more natural key-value form: given a keyword (or key) k , the client wishes to retrieve the corresponding value v . This setting is known as *keyword PIR* [24], whereas the

traditional PIR setting, in which records are identified by their indices, is referred to as *index PIR*.

Motivated by a line of advances [5, 24–27], recent works [25, 28] proposed an elegant and generic construction for keyword PIR by leveraging index PIR in a black-box manner, together with the key-value store (KVS) technique. This framework requires three invocations of the underlying index PIR protocol, resulting in a generic and broadly applicable solution, but one that incurs the overhead of multiple PIR queries. To reduce this cost, the state-of-the-art work [28] further introduced two specialized constructions that tightly integrate SimplePIR [8] with customized hashing mechanisms. These schemes significantly outperform the generic approach and are substantially faster than three independent invocations of SimplePIR.

Despite these advances, the proposed schemes still suffer from two limitations. First, inheriting a drawback of SimplePIR [8], they require a large client-side hint for each database instance, ranging from 14.5 to 143.6 MB for parameters in our experiments. Second, the schemes retrieve an entire row of the encoded database, giving more information than the specific value requested by the client (e.g., the client may learn other key-value pairs). While this behavior does not weaken the protocol’s privacy guarantees, since the queried keyword remains hidden from the server, it leads to over-disclosure from the server’s perspective. Particularly, this may be undesirable in applications where pricing or access control is tied to the amount of data revealed to the client.

Our Goal. Motivated by the above, we aim to construct a practical keyword PIR framework that simultaneously achieves (1) precise element-level retrieval in semi-honest client model, (2) no or substantially reduced client-side hint, and (3) efficiency surpassing the generic black-box construction based on three invocations of an underlying (hintless) index PIR protocol.

1.1 Our Contributions

In this work, we present a new framework for keyword PIR that achieves all three of the aforementioned goals: precise element-level retrieval, no client-side hint, and better efficiency than the generic black-box construction. Our framework extends KsPIR [29], a highly efficient index PIR protocol, to the keyword setting by combining homomorphic computation with simple hashing and key-value store (KVS) techniques [30–33]. The resulting protocol incurs a cost roughly comparable to a single invocation of KsPIR and is therefore substantially more efficient than the generic construction, which requires three index-PIR invocations.

Particularly, our contributions are as follows:

- We leverage a recent homomorphic method for evaluating linear functions [34], which expresses arbitrary linear functions as weighted sums of automorphisms. Compared with prior approaches that require NTT-friendly plaintext moduli or CRT-style decompositions such as [8, 16], this method imposes significantly weaker algebraic requirements on the plaintext space, i.e., only that the plaintext modulus be relatively prime to the ring dimension. At the same time, it achieves essentially the same asymptotic complexity. For our framework, this flexibility significantly broadens the range of practical parameter choices.
- We develop new techniques to substantially improve the time and space efficiency to implement the above homomorphic method. First, we introduce a generalized baby-step giant-step (BSGS) algorithm for weighted sums of automorphisms, reducing the number of homomorphic automorphism evaluations from $O(m)$ to $O(\sqrt{m})$, where m denotes the order of the Galois group (equivalently, the ring dimension in our setting). This optimization significantly accelerates a key building block of our framework. Second, we integrate a Horner-style evaluation strategy into the BSGS procedure, reducing the number of required evaluation keys from $O(\sqrt{m})$ to just three.
- We show how to incorporate the offline-online architecture of KsPIR [29] into our framework, yielding a practical protocol with low online response latency.

We implement our framework in both C++ and Java. To enable a direct comparison with prior work, our Java implementation is built using the same `mpc4j` library [35] employed by [28]. We evaluate the framework on databases containing up to 2^{22} key-value pairs with payloads of 256 bytes.

Compared with the first scheme of [28] instantiated using KsPIR [29], our solution achieves a $2.24\text{--}2.99\times$ speedup across database configurations. Compared with the hint-based constructions of [28], under the same Java implementation environment, our framework achieves comparable online latency while completely eliminating client-side hints, reducing client storage by up to 143.6 MB. In addition, we develop a high-performance C++ implementation that incorporates the engineering optimizations introduced in the KsPIR implemen-

tation [29], together with hardware acceleration techniques such as AVX and multi-threading. This implementation substantially reduces online response time and demonstrates the practicality of our framework at large scales. Moreover, the query size remains independent of the database size, requiring only 280 KB in C++ and 300 KB in Java regardless of the number of key-value pairs stored in the database.

1.2 Technique Overview

To highlight our new insights, we first recall the main approaches for constructing keyword PIR from index PIR, and then describe our new framework.

The Generic Construction and Prior Designs. Given a database of key-value pairs, $\text{DB} = \{(k_z, v_z)\}_{z \in [n]}$, the generic framework [25, 28] first encodes the key-value pairs using α -KVS. Informally, an α -KVS can be viewed as an array of size d such that the value v associated with a key k can be recovered by reading α locations in the array and computing an appropriate weighted linear combination of their contents. Consequently, retrieving the value associated with a key reduces to retrieving these α array locations, which can be accomplished privately using an index PIR protocol.

The first construction of [28] (state-of-the-art) directly applies this idea and further arranges the encoded array into a two-dimensional layout. This representation aligns naturally with efficient index PIR protocols that benefit from two-fold queries, such as Spiral [10], SimplePIR [8], and KsPIR [29], resulting in improved response time. In practice, choosing $\alpha = 3$ provides a favorable balance between efficiency and decoding failure probability.

To eliminate the overhead of repeating α PIR retrievals, [28] further proposed two alternative constructions based on hashing rather than α -KVS. Specifically, the key-value pairs are mapped into r bins, and retrieving a key k reduces to identifying the corresponding bin and retrieving all items stored in that bin. Integrated with SimplePIR [8] designs, this approach achieves significantly better communication and response time than the generic construction. However as discussed, the two schemes still suffer from two drawbacks: the large client-side hint inherited from SimplePIR and the over-disclosure resulting from returning an entire bin (or encoded row) rather than only the requested value.

Our New Design Framework. Our starting point is the generic α -KVS-based construction [25, 28]. A natural way to avoid multiple client queries is to compress the α queried locations into a single query using linear PIR [16]. In a linear PIR scheme, the client privately queries a coefficient vector $\vec{w}$, and the server responds with the linear combination $\sum \vec{d}[i] \cdot \vec{w}[i]$, where $\vec{d}$ denotes the database viewed as an array. However, applying linear PIR directly to a large database (encoded in

α -KVS) would require a large ring dimension, significantly limiting practical efficiency.

To overcome this challenge, we combine the hashing idea with the generic α -KVS construction. We first partition the n key-value pairs into r bins using a hash function h , i.e., the ℓ -th bin contains all pairs satisfying $h(k) = \ell$. The contents of each bin are then encoded individually via α -KVS (using the same hashing across all bins). Each bin yields an array of size d , which is substantially smaller than an α -KVS encoding of the entire database.

Applying this encoding to every bin transforms the database into a matrix $\mathbf{M} \in \mathbb{Z}^{r \times d}$, where the ℓ -th row $\mathbf{M}[\ell, \cdot]$ represents the α -KVS encoding of the ℓ -th bin. To retrieve the value associated with a key k , the client first computes its bin index $\ell^* = h(k)$ and then determines the corresponding α -KVS locations $t_1, \dots, t_\alpha$. Recovering the desired value therefore reduces to obtaining the matrix entries $(\ell^*, t_1), \dots, (\ell^*, t_\alpha)$.

One possibility is to issue a linear PIR query for the locations $t_1, \dots, t_\alpha$ to every bin. However, this would require r linear-PIR queries. In general, reducing the number of queried locations from α to one at the cost of increasing the number of PIR queries from 1 to r yields an unfavorable tradeoff.

Instead, we adopt an SIMD-style approach that applies the same private query vector $\vec{w}$ simultaneously across all r bins. Since the encoded database is represented as a $r \times d$ matrix $\mathbf{M}$, evaluating the desired linear combination for all bins at once amounts to a matrix-vector multiplication. The client encrypts $\vec{w}$ and sends $\text{Enc}(\vec{w})$ to the server, which homomorphically computes $\text{Enc}(\mathbf{M} \cdot \vec{w})$. This computation produces the desired linear combinations for all bins simultaneously in a single packed evaluation.

Returning this ciphertext directly, however, would reveal information about every bin rather than only the queried one. To prevent such over-disclosure, the client additionally sends an RGSW ciphertext $\text{Enc}(X^{-\ell^*})$ for the target bin index ℓ^* . Using homomorphic multiplication together with the extraction techniques underlying KsPIR [29], the server can homomorphically isolate the ℓ^* -th entry of $\mathbf{M} \cdot \vec{w}$ and return only the information corresponding to the queried bin.

We illustrate the resulting two-fold query structure in Figure 2. At a high level, the query format is reminiscent of KsPIR [29]. The similarity, however, is purely structural. In our construction, the first fold encodes the coefficients needed for α -KVS reconstruction, while the second fold encodes the target bin. By contrast, KsPIR uses its two query ciphertexts to encode the row and column indices of the database matrix. Thus, although both protocols employ a two-fold encrypted query structure, the semantic roles of the two components are fundamentally different.

Our New Accelerations. The homomorphic matrix-vector computation described above has been extensively studied in prior work, including HELib [36], Hintless PIR [16], and

KsPIR [29]. Existing approaches typically rely on either NTT-friendly plaintext moduli or Chinese Remainder Theorem (CRT)-based encodings. Instead, we adopt another technique from [34] that achieves comparable efficiency while supporting arbitrary plaintext moduli co-prime to the ring dimension.

The key idea is to express an arbitrary linear transformation as a weighted sum of automorphisms. This approach extends naturally to general number field extensions and imposes substantially weaker constraints on the plaintext space, thereby providing greater flexibility in parameter selection. Such flexibility is particularly desirable in applications where the plaintext modulus is dictated by the data representation rather than by algebraic requirements. For the commonly used power-of-two cyclotomic rings in FHE, the plaintext space can be defined over $\mathbb{Z}_p$ for any odd modulus p , without requiring p to be prime or NTT-friendly. For example, when encoding ℓ -bit messages, one may simply choose $p = 2^\ell + 1$, yielding a nearly tight plaintext representation.

Briefly, the method of [34] evaluates expressions of the form

$$\sum_{\sigma \in G} c_\sigma \cdot \text{Eval}_\sigma(\text{Enc}(x)),$$

where σ denotes an automorphism in the Galois group G , $\{c_\sigma\}$ are arbitrary coefficients, and $\text{Eval}_\sigma(\text{Enc}(x))$ denotes the homomorphic evaluation of σ on ciphertext $\text{Enc}(x)$. The dominant computational cost arises from the automorphism evaluations Eval_σ , motivating the goal of reducing their number. Moreover, a straightforward realization requires as many as $|G|$ evaluation keys, leading to substantial upload and storage overhead. Reducing both the number of automorphism evaluations and the number of evaluation keys is therefore critical to practical deployment.

To address these challenges, we introduce two new insights. First, we develop a decomposition of the Galois group that enables a baby-step giant-step (BSGS)-style acceleration, significantly reducing the number of required automorphism evaluations. Second, we observe that this decomposition admits a Horner-style evaluation strategy for both the baby-step and giant-step computations. As a result, the number of required evaluation keys can be reduced to three. We outline these ideas below.

BSGS Ideas. For the power-of-two cyclotomic rings commonly used in current FHE libraries, the Galois group G admits a structure that is close to, although not exactly, cyclic. Exploiting this structure, we decompose G into two sets, Σ_{giant} and Σ_{baby} , each of size $O(\sqrt{|G|})$, yielding the decomposition

$$\sum_{\sigma \in G} c_\sigma \cdot \sigma(x) = \sum_{\sigma_g \in \Sigma_{\text{giant}}} \sigma_g \left(\sum_{\sigma_b \in \Sigma_{\text{baby}}} \tilde{c}_{\sigma_g \circ \sigma_b} \cdot \sigma_b(x) \right),$$

where $\tilde{c}_{\sigma_g \circ \sigma_b} := \sigma_g^{-1}(c_{\sigma_g \circ \sigma_b})$. This decomposition gives rise to a generalized BSGS algorithm whose homomorphic counterpart follows naturally. As a result, the number of automor-

phism evaluations can be reduced from $O(|G|)$ to $O(\sqrt{|G|})$, yielding a substantial performance improvement.

Reducing Evaluation Keys. A naive implementation of the above BSGS decomposition appears to require $O(\sqrt{|G|})$ evaluation keys, one for each automorphism in Σ_{baby} and Σ_{giant} . We observe, however, that our particular decomposition structures of Σ_{baby} and Σ_{giant} are close to cyclic groups. This allows us to exploit a Horner-style evaluation strategy and dramatically reduce the number of required evaluation keys. As a result, the entire computation can be carried out using only three evaluation keys, instead of $O(\sqrt{|G|})$.

To illustrate the main idea, consider the simplified setting where $G = \langle \sigma \rangle$ is a cyclic group and the goal is to compute $\sum_i \sigma^i(x)$. Rather than evaluating each automorphism separately, Horner’s rule rewrites the computation as

$$\sigma(\cdots \sigma(\sigma(x) + x) + x).$$

In the homomorphic setting, this computation requires only the evaluation key corresponding to σ , since all automorphism evaluations are performed using repeated applications of the same generator. Extending this idea to our setting is more involved because the Galois group is not exactly cyclic and the terms are weighted by coefficients. Nevertheless, by carefully incorporating the coefficients into the Horner-style evaluation and exploiting the near-cyclic structure of the Galois group, we reduce the number of evaluation keys to just three.

Further Offline/Online Optimization. Our framework admits additional optimizations inspired by the offline/online insight of KsPIR [29]. First, all coefficients appearing in the above summations, including the transformed coefficients introduced by the BSGS and Horner-style procedures, depend only on the database DB and can therefore be precomputed offline. Second, our framework shares a key structural feature with KsPIR [29]: each query ciphertext naturally decomposes into a keyword-independent component and a keyword-dependent component. This property enables our homomorphic computation to be similarly partitioned. In particular, a substantial portion of the computation depends only on the keyword-independent component and can therefore be performed offline, before the client knows the key to query. Thus, a large fraction of the server workload can be shifted to the offline phase, substantially reducing the online response time.

Handling Larger Databases. For larger databases, we adopt the idea of KsPIR [29] and split the database DB into multiple blocks $\text{DB}^{(0)}, \dots, \text{DB}^{(\text{N}_{\text{blk}}-1)}$, which avoids increasing the ring dimension to load more entries. Furthermore, for databases storing large entries, increasing the plaintext modulus p is not always the most efficient solution, as doing so may also necessitate a larger ring dimension. Instead, we split each entry into multiple planes $v = (v_0 \| v_1 \| \cdots \| v_{L-1})$

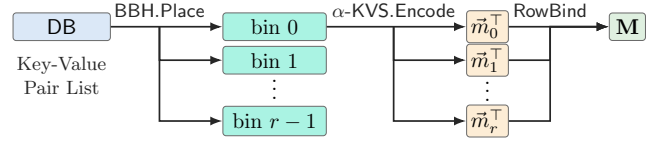


Figure 1: High-level overview of our database encoding.

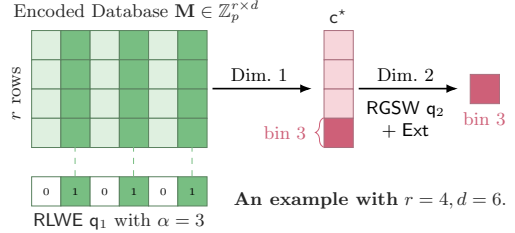


Figure 2: Our two-fold keyword PIR. Dimension 1 combines $\alpha = 3$ columns. Dimension 2 selects bin 3 via one RGSW.

like that in KsPIR [29]. A query retrieves the corresponding planes and reconstructs the value locally. This additional design dimension allows us to optimize the tradeoff among the plaintext modulus p , the ring dimension, and the number of planes L , leading to improved efficiency for large payloads.

1.3 Related Works

Keyword PIR was first introduced by Chor et al. [24] to enable private retrieval from key-value databases. Early works like [5, 24] suffer from large bucket sizes and high computational overhead. Subsequent works sought to close this efficiency gap through single-invocation designs. For example, the work [26] employed homomorphic equality tests, while SparsePIR [27] reduced costs through sparse linear-combination encodings. Despite these advances, keyword PIR remained significantly slower than state-of-the-art index PIR.

Recent works have explored KVS encodings to narrow this gap, albeit at the cost of increased communication or client state. ChalametPIR [25] combines KVS with a variant of FrodoPIR [37], shifting much of the computation to a pre-processing phase, but still incurs communication linear in the database size. Hao et al. [28] significantly improve both communication and runtime using sparse KVS, hashing-to-bins, and approximate key-to-index mappings. However, their constructions still require either multiple index-PIR invocations or substantial client-side hints. Similarly, the work [38] instantiates KVS using minimal perfect hash functions and a *Rewind & Skip* technique, but requires non-trivial client state to store hash descriptions and fingerprints. In contrast, our construction eliminates both database-dependent client-side hints and multi-query overhead.

Beyond retrieval efficiency, several recent works have ex-

plored additional functionalities. The work of [39] focuses on non-interactive evaluation of arbitrary circuits over retrieved data using TFHE bootstrapping, while [40] introduces an updatable oblivious KVS supporting incremental updates and fine-grained access control. These goals are orthogonal to ours. Whereas such works prioritize richer functionality and dynamic storage, our framework targets stateless, high-throughput keyword retrieval, making it a natural backend component for more expressive systems.

2 Preliminaries

Notations. Let $\lambda \in \mathbb{N}$ denote the security parameter, $\text{negl}(\cdot)$ and $\text{poly}(\cdot)$ denote negligible and polynomial functions. For $n \in \mathbb{N}$, define $[n] := \{0, \dots, n-1\}$. Let $a \xleftarrow{\$} S$ denote sampling a uniformly at random from a finite set S , and $y \leftarrow A(x)$ for the output of a randomized algorithm A . Two distributions X, Y are computationally (resp., statistically) indistinguishable, denoted by $X \approx_c Y$ (resp., $X \approx_s Y$). Vectors and matrices are denoted by arrows (e.g., $\vec{x}$) and bold uppercase (e.g., $\mathbf{A}$) letters, respectively. We write $\vec{x}[i]$ for the i -th entry of $\vec{x}$, and $\mathbf{A}[i, j]$, $\mathbf{A}[i, \cdot]$, $\mathbf{A}[\cdot, j]$ for the (i, j) -th entry, i -th row, and j -th column of $\mathbf{A}$. Let $\vec{e}_i$ denote the i -th standard basis vector, and let $\langle \vec{x}, \vec{y} \rangle$ be the inner product. Let $\|\cdot\|_\infty$ denote the infinity norm; all unmarked norms $\|\cdot\|$ throughout this paper are assumed to be the infinity norm. Let $\mathbb{Z}_{2m}^\times = (\mathbb{Z}/2m\mathbb{Z})^\times$.

2.1 Lattice Background

Assumptions. Let $p \ll q$ be the positive integer moduli. For a power-of-two m , we define the cyclotomic ring $\mathcal{R} := \mathbb{Z}[X]/(X^m + 1)$ and its quotients $\mathcal{R}_q := \mathcal{R}/q\mathcal{R}$ for q , $\mathcal{R}_p := \mathcal{R}/p\mathcal{R}$ for p . And we write $\Delta := \lfloor q/p \rfloor$ for the scaling factor and χ for the error distribution, of standard deviation σ . The (N, M, q, χ) -LWE assumption was introduced by Regev [41], which states that $(\mathbf{A}, \vec{s} \cdot \mathbf{A} + \vec{e}) \approx_c (\mathbf{A}, \vec{u})$ for $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{N \times M}$, $\vec{s} \xleftarrow{\$} \mathbb{Z}_q^N$, $\vec{u} \xleftarrow{\$} \mathbb{Z}_q^M$, and $\vec{e} \leftarrow \chi^M$. The $(\mathcal{R}, q, \chi)$ -RLWE assumption is defined analogously over $\mathcal{R}_q$ (of degree m), stating $(a, a \cdot s + e) \approx_c (a, u)$ for $a, s, u \xleftarrow{\$} \mathcal{R}_q$ and $e \leftarrow \chi$.

Primitives. We use the standard Regev-style secret-key LWE scheme, its ring analogue RLWE, the ring variant RGSW [42–44] of Gentry–Sahai–Waters [45], and standard operations including coefficient extraction [46], gadget decomposition, key switching, and homomorphic Galois automorphisms [47–50]. As these primitives are used as black boxes in our framework, we summarize their syntax in Table 1 and defer details to Appendix A.1. Ciphertexts are denoted by $\vec{c} \in \text{LWE}_{\vec{s}}(\mu)$ for $\mu \in \mathbb{Z}_p$, $c \in \text{RLWE}_s(\mu) \subset \mathcal{R}_q^2$, and $C \in \text{RGSW}_s(\mu)$ for $\mu \in \mathcal{R}_p$, all under the secret key $\text{sk} = (1, s)^\top$. The LWE algorithms are the natural analogues of the RLWE ones over $\mathbb{Z}_p$ and are omitted.

Table 1: Syntax of the lattice primitives. Algorithms are used as black boxes throughout the paper; Appendix A.1 gives their instantiations and noise.

Algorithm	Output
<i>Encryption</i>	
RLWE.KeyGen(1^λ)	$\text{sk} = (1, s)^\top$; RGSW.KeyGen reuses it
RLWE.EvalKeyGen(1^λ)	evaluation key evk
RLWE.Encrypt(sk, μ)	$c \in \text{RLWE}_s(\mu)$
RLWE.Decrypt(sk, c)	$\mu \in \mathcal{R}_p$
RLWE.Eval(f, c, evk)	$c' \in \text{RLWE}_s(f(\mu))$, $f \in \mathbb{Z}_p$ -linear
RGSW.Encrypt(sk, μ)	$C \in \text{RGSW}_s(\mu)$
RGSW.Decrypt(sk, C)	$\mu \in \mathcal{R}_p$
$c \boxtimes C$	$c' \in \text{RLWE}_s(\mu\mu')$, external product
<i>Extraction and gadget decomposition</i>	
$\text{coef}(\mathbf{a}), \text{coef}^{-1}(\vec{a})$	coefficient vector of $\mathbf{a}$, and its inverse
$\text{Ext}(c, j)$	$\vec{c} \in \text{LWE}_{\vec{s}}(\text{coef}(\mu)[j])$, $\vec{s} := \text{coef}(s(X^{-1}))$
$G^{-1}(\vec{x}), \text{PowersOfB}(\vec{y})$	digits with $\ G^{-1}(\vec{x})\ \leq B/2$, and its dual
<i>Key switching and automorphisms</i>	
KS.KeyGen(s, s')	switching key ksk from s' to s
KS(c, ksk)	$c' \in \text{RLWE}_s(\mu)$, message unchanged
Auto.KeyGen(s, t)	ksk for $\sigma_t : X \mapsto X^t$, $t \in \mathbb{Z}_{2m}^\times$
Auto.Eval(c, ksk)	$c' \in \text{RLWE}_s(\sigma_t(\mu))$, one KS each

Three properties from Table 1 are used repeatedly throughout the paper, and we state them here for convenience. First, the gadget pair exactly reconstructs an inner product:

$$G^{-1}(\vec{x}) \cdot \text{PowersOfB}(\vec{y}) = \langle \vec{x}, \vec{y} \rangle \pmod{q}, \quad (1)$$

where B denotes the gadget base and $\ell_B := \lceil \log_B q \rceil$ is the number of gadget digits. Second, the external product is *asymmetric*: the noise growth is amplified only by the bounded gadget digits and is therefore independent of p . This property plays a key role in the offline optimization of Sections 3.5 and C.1. Third, each automorphism evaluation Auto.Eval_t requires exactly one key-switching operation. Consequently, the number of automorphism evaluations serves as an accurate proxy for the computational cost, motivating the optimization target in Section 3.4. Finally, the external product operation Ext depends only on public ciphertext coefficients and therefore can be executed by either party.

2.2 Key-Value Store

A key-value store (KVS) [31] encodes a collection of key-value pairs into a data structure from which associated values can be efficiently recovered. We briefly recall the KVS definition below and introduce a closely related variant, denoted by α -KVS, obtained by modifying the α -sparse KVS notion of [28]. This variant will serve as a central component of our keyword PIR framework.

Let $L := \{(k_z, v_z)\}_{z \in [n]} \subseteq \mathcal{K} \times \mathcal{V}$ be a set of key-value pairs.

Definition 1 (Key-Value Store [31]) A KVS over L consists of ppt algorithms (Encode, Decode):

- $\text{Encode}(1^\lambda, L) \rightarrow S$: on input λ and a list L with distinct keys, outputs a data structure S .

- $\text{Decode}(S, k_i) \rightarrow \mu$: on input S and a key $k_i \in \mathcal{K}$, outputs $v_i \in \mathcal{V}$.

Below, we present a variant of KVS in which decoding is performed as a weighted linear combination of α entries in the encoded data structure. In addition, we explicitly separate out a one-time setup procedure that samples the hash functions and induces the decoding mappings. This abstraction captures both sparse and dense KVS constructions that can be used with our keyword PIR framework.

Definition 2 (α -KVS [28]) For an integer $\alpha \geq 1$ and a plaintext modulus p , an α -KVS with output length d over $\mathcal{K} \times \mathbb{Z}_p$ (i.e., $\mathcal{V} = \mathbb{Z}_p$) consists of PPT algorithms (Setup, Encode, Decode):

- $\alpha\text{-KVS.Setup}(1^\lambda) \rightarrow \text{pp}_{\alpha\text{-KVS}}$: samples a public hash family $H = (h_0, \dots, h_{\alpha-1})$, $h_j: \mathcal{K} \rightarrow [d]$, and outputs public parameters $\text{pp}_{\alpha\text{-KVS}} := (H, \vec{w}(\cdot))$, where

$$\vec{w}(\cdot): \mathcal{K} \rightarrow \mathbb{Z}_p^d, \quad \vec{w}(k) = (w_0(k), \dots, w_{d-1}(k)),$$

is the induced decoding-coefficient mapping. For every key k , the vector $\vec{w}(k)$ has Hamming weight α , and its nonzero entries are located at positions $\{h_j(k)\}_{j \in [\alpha]}$. The public parameters are generated once and reused across all subsequent invocations of Encode.

- $\alpha\text{-KVS.Encode}(\text{pp}_{\alpha\text{-KVS}}, L) \rightarrow \mathbf{D}$ or $\perp$: on input $\text{pp}_{\alpha\text{-KVS}}$ and a list $L \subseteq \mathcal{K} \times \mathbb{Z}_p$ containing at most d/η distinct-key pairs, outputs a data vector $\mathbf{D} \in \mathbb{Z}_p^d$ satisfying

$$\langle \vec{w}(k), \mathbf{D} \rangle = \sum_{i=0}^{d-1} w_i(k) \cdot \mathbf{D}[i] \equiv v \pmod{p}$$

for every $(k, v) \in L$, or outputs $\perp$ if no such $\mathbf{D}$ exists for the sampled hash family.

- $\alpha\text{-KVS.Decode}(\text{pp}_{\alpha\text{-KVS}}, \mathbf{D}, k) \rightarrow v$: $\langle \vec{w}(k), \mathbf{D} \rangle \bmod p$.

The ratio $\eta := d/|L| \geq 1$ between the length of the data vector and the number of encoded pairs is called the *expansion rate* of the α -KVS. It quantifies the storage overhead required for successful encoding. The optimal value is $\eta = 1$, which is achievable only by dense constructions. Sparse constructions require $\eta > 1$, and minimizing this expansion rate is a central objective of the parameter selection discussed in Appendix C.2.

Remark 3 The above definition generalizes the KVS abstraction of [31] by introducing a key-to-decoding-coefficient mapping $\vec{w}(\cdot): \mathcal{K} \rightarrow \mathbb{Z}_p^d$ with Hamming weight α . The sparse KVS of [28] is recovered as the special case $\alpha = 3$ in which every nonzero coefficient equals 1. A polynomial-interpolation KVS [14, 51] corresponds to the dense setting $\alpha = d$ with $w_i(k) := k^i, i \in [d]$, where $\mathbf{D}$ stores interpolation coefficients.

Remark 4 An α -KVS must satisfy the standard correctness properties inclusion and non-inclusion. We omit the standard formalisms here for brevity. The concrete instantiation of our α -KVS will be detailed later in Section 3.

2.3 Balls-into-Bins Hash

A balls-into-bins hash BBH schedules n keys into r bins by assigning every key to its corresponding bin determined by its hash. We give the definition below, which is tailored to our keyword PIR construction with the bin lookup BBH. Locate deterministic in the keyword alone.

Definition 5 (Balls-into-Bins Hash [52, 53]) For positive integers n, r , a (n, r) -BBH with key space $\mathcal{K}$ consists of ppt algorithms:

- $\text{BBH.Setup}(1^\lambda, n, r) \rightarrow \text{pp}_{\text{BBH}}$: generates and outputs public parameters.
- $\text{BBH.Locate}(\text{pp}_{\text{BBH}}, k) \rightarrow \ell_k^*$: inputs $k \in \mathcal{K}$, outputs the bin index $\ell_k^* \in [r]$ that k is assigned to.
- $\text{BBH.Place}(\text{pp}_{\text{BBH}}, K) \rightarrow \phi$: inputs a finite set $K \subset \mathcal{K}$ of distinct keys, outputs $\phi: K \rightarrow [r]$ defined by $\phi(k) := \text{BBH.Locate}(\text{pp}_{\text{BBH}}, k)$, i.e., the bin lookup for K .

We use $\Lambda := n/r$ to denote the *mean bin load* of a (n, r) -BBH, i.e. the number of keys an average bin receives, and $\Lambda^{\max}$ to denote its *max bin load* at statistical parameter κ : the smallest per-bin count that no bin of the deployment exceeds except with probability at most $2^{-\kappa}$. We keep κ separate from λ because a bin overflow is detected by the server inside EncodeDB and repaired by resampling pp_{BBH} .

2.4 Keyword Private Information Retrieval

A keyword PIR scheme allows a client retrieve v associated with a queried keyword k from a server-held database DB, without revealing k to the server.

Definition 6 (Keyword Private Information Retrieval [24])

A keyword PIR over $(\mathcal{K}, \mathcal{V})$ consists of ppt algorithms:

- $\text{Server.Setup}(1^\lambda, \text{DB}) \rightarrow \text{pp}_{\text{kPIR}}$: server takes DB as input, then outputs public parameters pp_{kPIR} .
- $\text{Client.Setup}(1^\lambda) \rightarrow (\text{sk}, \text{pk})$: client generates a secret key sk and a public key pk , and in some PIR schemes $\text{pk} := \perp$.
- $\text{Client.Query}(\text{pp}_{\text{kPIR}}, \text{sk}, k) \rightarrow (\text{st}, \text{qu})$: client generates a query qu for $k \in \mathcal{K}$ and a state st .
- $\text{Server.Answer}(\text{pp}_{\text{kPIR}}, \text{pk}, \text{qu}) \rightarrow \text{ans}$: server homomorphically computes an answer ans .
- $\text{Client.Recover}(\text{st}, \text{pp}, \text{ans}) \rightarrow v$ or $\perp$: client decodes the value $v \in \mathcal{V}$, or outputs $\perp$.

Definition 7 (Correctness and Security) The kPIR scheme defined above must satisfy:

- **Correctness.** For any database DB and keyword $k \in \mathcal{K}$, let $(\text{st}, \text{qu}) \leftarrow \text{Client.Query}(\text{pp}_{\text{kPIR}}, \text{sk}, k)$ and $\text{ans} \leftarrow \text{Server.Answer}(\text{pp}_{\text{kPIR}}, \text{pk}, \text{qu})$. We require for $(k, v) \in \text{DB}$:

$$\begin{aligned} \Pr[\text{Client.Recover}(\text{st}, \text{pp}_{\text{kPIR}}, \text{ans}) = v] &= 1, \quad \text{o.w.}, \\ \Pr[\text{Client.Recover}(\text{st}, \text{pp}_{\text{kPIR}}, \text{ans}) = \perp] &\geq 1 - \text{negl}(\lambda). \end{aligned}$$

Table 2: Parameters in Construction 8.

Parameter	Meaning
n, n_1	database size; per-block capacity n/N_{blk}
$n_1^{\max}$	admissible per-block capacity, Eq. (13)
N_{blk}	number of partition blocks
r	bin count of the BBH
α, η	α -KVS sparsity; expansion rate $d/ L $
$\Lambda, \Lambda^{\max}$	BBH mean and max bin load
L	number of payload planes
γ	tag length
d	α -KVS length = number of column polynomials
m	cyclotomic ring degree, a power of two
p, q	plaintext and ciphertext moduli
B, ℓ_B	gadget base and digit count

- **Security.** The scheme is (T, ϵ) -secure if for any T -time adversary $\mathcal{A}$ and any $k, k' \in \mathcal{K}$,

$$|\Pr[\mathcal{A}(\text{pp}, \text{pk}, \text{qu}) = 1] - \Pr[\mathcal{A}(\text{pp}, \text{pk}, \text{qu}') = 1]| \leq \epsilon,$$

where qu is generated by $\text{Client.Query}(\text{pp}_{\text{KPIR}}, \text{sk}, k)$ and qu' by $\text{Client.Query}(\text{pp}_{\text{KPIR}}, \text{sk}, k')$.

3 Our Keyword PIR

In this section, we present our keyword PIR construction. First we present the core framework of the protocol in Section 3.1, which composes BBH, α -KVS, and an FHE-based design. In Sections 3.2 and C.2, we provide concrete instantiations and parameter selection strategies for these building blocks. We detail two optimizations and analysis in Sections 3.3 to C.1.

3.1 Our Construction

Our core protocol extends and generalizes the idea of ksPIR 's [29] to the keyword setting, using the technique of two-dimensional folding together with α -KVS and BBH. First we describe how the server encodes a key-value pair list into a matrix to facilitate later responses, and then present the protocol itself.

Encoding Preparation. The server holds a raw key-value pair list $\text{DB} := \{(k_z, v_z)\}_{z \in [n]}$ with keys $k_z \in \mathcal{K} \subseteq \{0, 1\}^*$ and values $v_z \in \{0, 1\}^{\ell_v}$. The server's goal is to prepare an encoding of DB that facilitates the responses (i.e., homomorphic computation) to later clients' queries. Particularly, each value is first prefixed by $\text{tag}(k) \in \{0, 1\}^\gamma$ using a tag function $\text{tag} : \mathcal{K} \rightarrow \{0, 1\}^\gamma$, which we model as a random oracle. Since our plaintext space is $\mathbb{Z}_p$, the tag-prefixed value is then viewed as a vector over $\mathcal{S}_{\text{plane}} := \mathbb{Z}_p^L$, denoted as $\text{pl}(k, v)$, where $L := \lceil (\gamma + \ell_v) / \lfloor \log_2 p \rfloor \rceil$. The j -th entry $\text{pl}(k, v)[j] \in \mathbb{Z}_p$ holds the j -th $\lfloor \log_2 p \rfloor$ -bit part of $\text{tag}(k) \parallel v$, which is called *plane*. This yields the normalized

list $L := \{(k_z, \text{pl}(k_z, v_z))\}_{z \in [n]} \subseteq \mathcal{K} \times \mathcal{S}_{\text{plane}}$, which is the input to the encoding phase.

Database Encoding. Let m be the degree of the cyclotomic ring used by the RLWE, and d the α -KVS length. To process the entire database DB of n pairs, we first partition it into $N_{\text{blk}} := \lceil n/n_1^{\max} \rceil$ equally sized blocks $\text{DB}^{(0)}, \dots, \text{DB}^{(N_{\text{blk}}-1)}$, where $n_1^{\max}$ is the admissible capacity of one block. Each block operates as an independent ring block over r bins, and reserves $\Lambda^{\max}(\kappa)$ positions in every bin, $\Lambda^{\max}$ being the BBH maximum bin load. That reservation is governed by two constraints: (1) $rL \leq m$ ensures that every bin-plane pair occupies a distinct polynomial coefficient; (2) $\eta\Lambda^{\max} \leq d \leq m$ ensures that each bin's encoded α -KVS fits within a single query ciphertext. The reserved space $r\Lambda^{\max}$ is an upper bound on $n_1^{\max}$ rather than equal to it, since $\Lambda^{\max}$ must cover the largest bin while the capacity only fills the average one. Due to all blocks operate independently but share the identical pp_{BBH} and $\text{pp}_{\alpha\text{-KVS}}$, a single client query can be used to evaluate for every block.

Within each block $\mathfrak{t} \in [N_{\text{blk}}]$, the encoding proceeds in two steps. First, the server routes each pair of $\text{DB}^{(\mathfrak{t})}$ into one of r bins via BBH.Place , and for each bin $\ell \in [r]$ and plane $j \in [L]$ encodes the corresponding pairs into a data vector $\mathbf{D}_{\ell,j}^{(\mathfrak{t})} \in \mathbb{Z}_p^d$. Every bin, plane and block is encoded under the same $\text{pp}_{\alpha\text{-KVS}} = (H, \vec{w}(\cdot))$, which is what lets a single query $\vec{w}(k)$ serve all of them.

Second, the server stacks these rL vectors into a matrix $\mathbf{M}^{(\mathfrak{t})} \in \mathbb{Z}_p^{rL \times d}$ using a layout as $\mathbf{M}^{(\mathfrak{t})}[\ell \cdot L + j, \cdot] := \mathbf{D}_{\ell,j}^{(\mathfrak{t})}$ with $\ell \in [r]$, $j \in [L]$, so that the L planes of a bin occupy L consecutive rows. We present the encoding in EncodeDB (Algorithm 1) and its counterpart in SearchDB (Algorithm 2).

Construction 8 (Our Keyword PIR) Our construction is presented in Figure 3.

Correctness. First, fix a block $\mathfrak{t} \in [N_{\text{blk}}]$. By the $\mathbb{Z}_p$ -linearity of f , the Dimension-1 fold inherently evaluates the query polynomial $W_k(X)$ to yield $\mu_k^* := f_1(W_k(X)) = \sum_{\ell=0}^{r-1} \sum_{j=0}^{L-1} V_{\ell,j}^{(\mathfrak{t})} X^{\ell L + j}$, where the coefficient $V_{\ell,j}^{(\mathfrak{t})} := \langle \vec{w}(k), \mathbf{M}^{(\mathfrak{t})}[\ell L + j, \cdot] \rangle$ exactly recovers the α -KVS.Decode result for bin ℓ on plane j . Dimension 2 then applies the external product with the selector $X^{-\ell_k^* L}$ to compute $T_1 \in \text{RLWE}_s(\mu_k^* \cdot X^{-\ell_k^* L})$. This algebraic shift maps the target coefficient at $X^{\ell_k^* L + j}$ down to index j , enabling the server to directly extract the target planes via $\text{Ext}(T_1, j) \in \text{LWE}_{\vec{s}}(V_{\ell_k^*, j}^{(\mathfrak{t})})$ for all $j \in [L]$. Because BBH.Place and BBH.Locate agree on every key, the bin ℓ_k^* named by the client is exactly the bin $\phi_{\mathfrak{t}^*}(k)$ into which the server placed k in its block $\mathfrak{t}^*$, so that block returns the payload of k . Decryption recovers the N_{blk} candidate payloads, and since tag is modelled as a random oracle a union bound restricts the probability of a false-positive tag

Algorithm 1 EncodeDB(pp_{BBH}, pp_{α-KVS}, DB)

Parameters: N_{blk}, r, L, d, n, α.

Input: pp_{BBH}; pp_{α-KVS} = (H, $\vec{w}(\cdot)$); the key-value pair list DB := {(k_z, v_z)}_{z ∈ [n]}.

Output: $\mathbf{M}^{(t)} \in \mathbb{Z}_p^{rL \times d}$ for $t \in [N_{\text{blk}}]$.

- 1: For each $z \in [n]$, set $\text{pl}(k_z, v_z) \in \mathcal{S}_{\text{plane}}$ to the plane vector of $\text{tag}(k_z) \parallel v_z$.
 - 2: Partition DB into N_{blk} blocks DB⁽⁰⁾, ..., DB^(N_{blk}-1).
 - 3: **for** $t \in [N_{\text{blk}}]$ **do**
 - 4: $\phi_t \leftarrow \text{BBH.Place}(\text{pp}_{\text{BBH}}, \{k : (k, \cdot) \in \text{DB}^{(t)}\})$.
 - 5: Set $S_{\ell,j}^{(t)} := \{(k, \text{pl}(k, v)[j]) : \phi_t(k) = \ell\}$ for $\ell \in [r], j \in [L]$.
 - 6: Set $\mathbf{D}_{\ell,j}^{(t)}$ as α -KVS.Encode(pp_{α-KVS}, $S_{\ell,j}^{(t)}$).
 - 7: Set $\mathbf{M}^{(t)}[\ell L + j, \cdot] := \mathbf{D}_{\ell,j}^{(t)}$ for all $\ell \in [r], j \in [L]$.
 - 8: If any bin exceeds $\Lambda^{\max}(\kappa)$, resample and update the seed in pp_{BBH}, then restart.
 - 9: If any α -KVS.Encode returned $\perp$, resample and update the seed in pp_{α-KVS}, then restart.
 - 10: **return** $\{\mathbf{M}^{(t)}\}_{t \in [N_{\text{blk}}]}$.
-

Algorithm 2 SearchDB(pp_{BBH}, pp_{α-KVS}, $\{\mathbf{M}^{(t)}\}_{t \in [N_{\text{blk}}]}$, k)

Parameters: N_{blk}, r, L, γ.

Input: pp_{BBH}; pp_{α-KVS} = (H, $\vec{w}(\cdot)$); $\{\mathbf{M}^{(t)}\}_{t \in [N_{\text{blk}}]}$; query key k.

Output: The value v associated with k, or $\perp$.

- 1: $\ell_k^* \leftarrow \text{BBH.Locate}(\text{pp}_{\text{BBH}}, k) \in [r]$.
 - 2: **for** $t \in [N_{\text{blk}}]$ **do**
 - 3: For $j \in [L]$, decode $\mu_t[j] \leftarrow \langle \vec{w}(k), \mathbf{M}^{(t)}[\ell_k^* L + j, \cdot] \rangle \bmod p$.
 - 4: Parse $\mu_t \in \mathcal{S}_{\text{plane}}$ as $\tau_t \parallel \hat{v}_t$ with $|\tau_t| = \gamma$.
 - 5: **if** $\tau_t = \text{tag}(k)$ **then**
 - 6: **return** $\hat{v}_t$.
 - 7: **return** $\perp$.
-

match across the remaining blocks to $(N_{\text{blk}} - 1) \cdot 2^{-\gamma}$, which is overwhelming unlikely.

3.2 Comparison of the α-KVS

In this section, we compare two concrete instantiations of the α-KVS: the dense polynomial KVS based on polynomial interpolation [54], and the sparse 3-KVS based on segmented binary fuse filters [55]. We defer their formal algorithmic descriptions to Appendix B.1. Fundamentally, both schemes encode a database L of key-value pairs into a vector $\mathbf{D} \in \mathbb{Z}_p^d$ such that the value associated with a key k is recovered via an inner product $\langle \vec{w}(k), \mathbf{D} \rangle \bmod p$. They differ structurally in the decode vector $\vec{w}(k)$: the polynomial KVS defines $\vec{w}(k) := (1, k, k^2, \dots, k^{d-1})$ as a dense Vandermonde

row with Hamming weight $\alpha = d$, whereas the 3-KVS partitions the bin into many segments and then maps the key to a pseudorandom position in each of three consecutive ones, yielding a highly sparse $\vec{w}(k)$ with a fixed Hamming weight of $\alpha = 3$.

These structural differences present a trade-off between spatial capacity, offline encoding complexity, and query performance:

- **Plaintext Context.** For the same length key-value pair list, because the polynomial KVS achieves an ideal expansion rate of $\eta = 1$ (i.e., $d = |L|$), it introduces no inflation to the encoded database size. The 3-KVS, conversely, requires $\eta > 1$ to guarantee successful encoding, expanding the required storage capacity. However, this capacity penalty buys overwhelming efficiency in plaintext retrieval: encoding the 3-KVS takes only expected $O(|L|)$ time, and the client's plaintext query generation requires computing only three nonzero positions. In contrast, the polynomial KVS requires $O(d \cdot \log^2 d)$ operations to encode and requires evaluating a full degree- $(d - 1)$ polynomial to generate a query.
- **Homomorphic Context.** While the 3-KVS sparsity accelerates plaintext retrieval, this advantage vanishes in the PIR protocol under the homomorphic encryption context. Because oblivious homomorphic evaluation must securely process all d coefficients of the ciphertext to prevent information leakage, the server's PIR workload is proportional to the allocated query length d , independent of the Hamming weight α . Thus, for a fixed query capacity d , both instantiations incur identical homomorphic evaluation complexity. A second, algebraic difference does separate them: decoding a 3-KVS is a plain sum of α entries and its encoding only peels and subtracts, so it never inverts an element of $\mathbb{Z}_p$ and constrains the plaintext modulus no further than the homomorphic evaluation already does, whereas the polynomial KVS inverts a Vandermonde system and therefore additionally requires $\mathbb{Z}_p$ to be a field. The 3-KVS thus preserves the freedom in choosing p that the Galois-based linear map gives us. Furthermore, modifying a single key-value pair in the polynomial KVS invalidates the entire interpolating polynomial, forcing a global re-encoding. The 3-KVS, by contrast, confines insertions to localized bins, enabling highly efficient database maintenance. We ultimately select the 3-KVS as our concrete instantiation primarily due to these advantages.

3.3 Instantiation of f

Now we instantiate RLWE.Eval($f_i, \cdot$) used in Construction 8. The N_{blk} maps are independent and identically structured, so throughout this subsection and the two optimizations that follow we fix a single block and drop the index, writing f for f_i , u_t for $u_t^{(i)}$, and $\mathbf{M}$ for $\mathbf{M}^{(t)}$; the server repeats the entire evaluation once per block. Thus $f : \mathcal{R}_p \rightarrow \mathcal{R}_p$ is the $\mathbb{Z}_p$ -linear map fixed by $f(X^t) = u_t(X)$ on the d data-carrying basis

Client.Setup(1^λ) $\rightarrow$ (sk, pk): 1. Sample $\text{sk} := (1, s)^\top \leftarrow \text{RLWE.KeyGen}(1^\lambda)$. 2. RGSW.KeyGen uses the same sk keys. 3. Generate $\text{evk} \leftarrow \text{RLWE.EvalKeyGen}(1^\lambda)$. 4. Output $\text{sk} := (1, s)^\top$ and $\text{pk} := \text{evk}$. Client.Query($\text{pp}_{\text{kPIR}}, \text{sk}, k$) $\rightarrow$ (st, qu): 1. $\ell_k^* := \text{BBH.Locate}(\text{pp}_{\text{BBH}}, k) \in [r]$. 2. $W_k(X) := \text{coef}^{-1}(\vec{w}(k)) = \sum_{i=0}^{d-1} w_i(k) X^i$. 3. $q_1 := (b_1, a_1) \leftarrow \text{RLWE.Encrypt}(\text{sk}, W_k(X))$. 4. $q_2 \leftarrow \text{RGSW.Encrypt}(\text{sk}, X^{-\ell_k^* L})$. 5. Output (qu, st) where $\text{qu} := (q_1, q_2)$ and $\text{st} := (k, \ell_k^*, \text{sk})$. Client.Recover(st, pp_{kPIR}, ans) $\rightarrow v$ or $\perp$: 1. Parse $\text{st} = (k, \ell_k^*, \text{sk})$ with $\text{sk} = (1, s)^\top$; set $\vec{s} := \text{coef}(s(X^{-1}))$ and the derived key $\overline{\text{sk}} := (1, \vec{s})$. 2. For each $\mathbf{u} \in [\text{N}_{\text{blk}}]$, decrypt $\mu_{\mathbf{u}}[j] := \text{LWE.Decrypt}(\overline{\text{sk}}, L_{\mathbf{u}, j})$ for $j \in [L]$, and parse $\mu_{\mathbf{u}} \in \mathcal{S}_{\text{plane}}$ as $\tau_{\mathbf{u}} \parallel \hat{v}_{\mathbf{u}}$ with $\tau_{\mathbf{u}} = \gamma$. 3. If one index $\mathbf{u}^*$ satisfies $\tau_{\mathbf{u}^*} = \text{tag}(k)$, output $v := \hat{v}_{\mathbf{u}^*}$; otherwise output $\perp$.	Server.Setup($1^\lambda, \text{DB}$) $\rightarrow$ ($\text{pp}_{\text{kPIR}}, \text{st}_S$): 1. Sample $\text{pp}_{\text{BBH}} \leftarrow \text{BBH.Setup}(1^\lambda, n, r)$. 2. Sample $\text{pp}_{\alpha\text{-KVS}} \leftarrow \alpha\text{-KVS.Setup}(1^\lambda)$. 3. $\{\mathbf{M}^{(\mathbf{u})}\}_{\mathbf{u} \in [\text{N}_{\text{blk}}]} \leftarrow \text{EncodeDB}(\text{pp}_{\text{BBH}}, \text{pp}_{\alpha\text{-KVS}}, \text{DB})$. 4. For $\mathbf{u} \in [\text{N}_{\text{blk}}], t \in [d]$, set $u_t^{(\mathbf{u})}(X) := \text{coef}^{-1}(\mathbf{M}^{(\mathbf{u})}[\cdot, t]) \in \mathcal{R}_p$, where $\mathbf{M}^{(\mathbf{u})}[\cdot, t] \in \mathbb{Z}_p^{rL}$ is zero-padded to length m. 5. For $\mathbf{u} \in [\text{N}_{\text{blk}}]$, let $f_{\mathbf{u}} : \mathcal{R}_p \rightarrow \mathcal{R}_p$ be the $\mathbb{Z}_p$-linear map with $f_{\mathbf{u}}(X^t) = u_t^{(\mathbf{u})}(X)$ for $t \in [d]$ and $f_{\mathbf{u}}(X^t) = 0$ otherwise, and let $\text{aux}_f := \{\text{aux}_{f_{\mathbf{u}}}\}_{\mathbf{u} \in [\text{N}_{\text{blk}}]}$ be the auxiliary data instantiated in Section 3.3. 6. Output $\text{pp}_{\text{kPIR}} := (\text{pp}_{\text{BBH}}, \text{pp}_{\alpha\text{-KVS}}, \text{N}_{\text{blk}})$ and retain $\text{st}_S := (\{f_{\mathbf{u}}\}_{\mathbf{u} \in [\text{N}_{\text{blk}}]}, \text{aux}_f)$ as server-side state. Server.Answer($\text{pp}_{\text{kPIR}}, \text{st}_S, \text{evk}, \text{qu}$) $\rightarrow \text{ans}$: 1. Parse $\text{qu} = (q_1, q_2)$ and $\text{st}_S = (\{f_{\mathbf{u}}\}, \text{aux}_f)$. 2. Dimension 1. For each $\mathbf{u} \in [\text{N}_{\text{blk}}]$, compute $c_{\mathbf{u}}^* := \text{RLWE.Eval}(f_{\mathbf{u}}, q_1, \text{evk}) \in \mathcal{R}_q^2$. 3. Dimension 2. For each $\mathbf{u} \in [\text{N}_{\text{blk}}]$, compute $T_{\mathbf{u}} := \text{RGSW.ExternalProduct}(c_{\mathbf{u}}^*, q_2)$. 4. Extraction. For each $\mathbf{u}$ and each $j \in [L]$, set $L_{\mathbf{u}, j} := \text{Ext}(T_{\mathbf{u}}, j) \in \mathbb{Z}_q \times \mathbb{Z}_q^m$. 5. Output $\text{ans} := (L_{\mathbf{u}, j})_{\mathbf{u} \in [\text{N}_{\text{blk}}], j \in [L]}$.
---	--

Figure 3: Full specification of our Keyword PIR construction.

elements $\{X^t\}_{t \in [d]}$ (and $f(X^t) = 0$ on the remaining $m - d$). The construction of f is inspired by Lemma 4.1 in [34].

Lemma 9 (Existence of $\mathbb{F}$ -linear function f [34]) *Let $\mathbb{E}/\mathbb{F}$ be any finite Galois extension with $G := \text{Gal}(\mathbb{E}/\mathbb{F})$, and let $\vec{\beta}, \vec{\gamma}$ be over $\mathbb{E}$ with same index set, such that $\vec{\beta}$ is $\mathbb{F}$ -linearly independent with dual $\vec{\beta}^\vee$. Then $f_{\text{sel}} : \mathbb{E} \rightarrow \mathbb{E}$ defined by*

$$f_{\text{sel}}(x) := \sum_{\sigma \in G} \langle \vec{\gamma}, \sigma(\vec{\beta}^\vee) \rangle \cdot \sigma(x)$$

is $\mathbb{F}$ -linear and satisfies $f_{\text{sel}}(\vec{\beta}) = \vec{\gamma}$.

Instantiation of f . Our $\mathcal{R}_p$ is a quotient ring rather than a field extension, so we need the ring version of Lemma 9, which can be found in [56, Lemma 4.2]. In our setting, we can just set $\mathbb{E}$ to be the $2m$ -th cyclotomic field (for two's powers), $\mathbb{F} = \mathbb{Q}$ and $p > 1$ be any odd positive integer (coprime to m). Then $\mathcal{R}_p$ is the ring of integers modulo p .

Consequently $G \cong \mathbb{Z}_{2m}^\times$ has order m with automorphisms $\sigma_i : X \mapsto X^i, i \in \mathbb{Z}_{2m}^\times$. Take the full coefficient basis $\vec{\beta} = (X^t)_{t \in [m]}$ (dual $\vec{\beta}^\vee = m^{-1}(X^{-t})_{t \in [m]}$) and the target vector $\vec{\gamma} = (u_t(X))_{t \in [m]} \in \mathcal{R}_p^m$, where $u_t(X)$ is the t -th column polynomial for $t \in [d]$ and $u_t(X) := 0$ for $d \leq t < m$. Let $\hat{u}_i(X) := \sum_{t=0}^{d-1} X^{-it} u_t(X) \in \mathcal{R}_p$, the lemma yields $f(X^t) = u_t(X)$ via

$$f(x) = m^{-1} \sum_{i \in \mathbb{Z}_{2m}^\times} \hat{u}_i(X) \cdot \sigma_i(x), \quad (2)$$

the sum running only over the d nonzero columns. The server stores $\text{aux}_f := \{\hat{u}_i\}_{i \in \mathbb{Z}_{2m}^\times}$ as part of its state st_S ; it is a deterministic function of $\mathbf{M}$, is used only inside Server.Answer, and is never transmitted. In our paper, each σ_i is represented by Auto.Eval_i as Section 2.1. Thus, on input $c \in \text{RLWE}_s(\mu)$,

$$\text{RLWE.Eval}(f, c, \text{evk}) := m^{-1} \sum_{i \in \mathbb{Z}_{2m}^\times} \hat{u}_i \cdot \text{Auto.Eval}_i(c, \text{evk}). \quad (3)$$

3.4 Optimization 1: BSGS Acceleration

We present a BSGS realization that accelerates the server's response time. Recall that the Dimension 1 component of the target answer `Server.Answer` requires computing the Galois sum in Equation (3):

$$\text{RLWE.Eval}(f, q_1, \text{evk}) = m^{-1} \sum_{i \in \mathbb{Z}_{2m}^\times} \hat{u}_i \cdot \text{Auto.Eval}_i(q_1, \text{evk}),$$

which consists of m homomorphic automorphisms applied to a single ciphertext.¹

BSGS is well studied in FHE for matrix–vector multiplication over *NTT-slot-encoded* RLWE plaintexts [36]. Our observation is that it needs no slot structure at all: it applies to any expression $\sum_{\sigma \in \mathbb{Z}_{2m}^\times} c_\sigma \cdot \text{Auto.Eval}_\sigma(\text{Enc}(x), \text{evk})$ with arbitrary plaintext coefficients $\{c_\sigma\}$, which is exactly the form Section 3.3 gives to our Dimension 1. BSGS improves the $m - 1$ nontrivial automorphisms to $O(\sqrt{m})$ while leaving the m plaintext–ciphertext multiplications untouched, at the price of $O(\sqrt{m})$ additional additions and one further layer of automorphism noise. To present the idea, we describe the group decomposition on the plaintext computation, i.e., $\sum_\sigma c_\sigma \cdot \sigma(x)$. Its homomorphic counterpart follows immediately.

We begin by factoring the Galois group into baby and giant steps. Let G denote the Galois group of the $2m$ -th cyclotomic field. It is well known that $G \cong \mathbb{Z}_{2m}^\times$, where the automorphism corresponding to $t \in \mathbb{Z}_{2m}^\times$ is $\sigma_t : X \mapsto X^t$. Furthermore, $\mathbb{Z}_{2m}^\times \cong \langle -1 \rangle \times \langle g \rangle$, where $g = 5$ generates the index-2 subgroup of order $m/2$. Choose block sizes M_1, M_2 satisfying $M_1 M_2 = m/2$. Then every automorphism admits a unique factorization $\sigma = \sigma_g \circ \sigma_b$, where $\sigma_b \in \Sigma_{\text{baby}} := \{\sigma_{g^k} \mid k \in [M_2]\}$ and $\sigma_g \in \Sigma_{\text{giant}} := \{\sigma_{ig^{jM_2}} \mid (i, j) \in \{\pm 1\} \times [M_1]\}$. Note that $|\Sigma_{\text{baby}}| = M_2$ while $|\Sigma_{\text{giant}}| = 2M_1$, so the two sets are balanced at $M_2 = 2M_1 = O(\sqrt{m})$, the $\mathbb{Z}_p$ -linear Galois sum can then be rewritten in BSGS form as

$$\sum_{\sigma \in \mathbb{Z}_{2m}^\times} c_\sigma \cdot \sigma(x) = \sum_{\sigma_g \in \Sigma_{\text{giant}}} \sigma_g \left(\sum_{\sigma_b \in \Sigma_{\text{baby}}} \tilde{c}_{\sigma_g \cdot \sigma_b} \cdot \sigma_b(x) \right),$$

where $\tilde{c}_{\sigma_g \cdot \sigma_b} := \sigma_g^{-1}(c_{\sigma_g \circ \sigma_b})$.

Writing $\sigma_{bg^{jM_2}} = \sigma_b \circ (\sigma_{g^{M_2}})^j$ for $b \in \{\pm 1\}$, the outer sum splits along the two cosets of $\langle g \rangle$ and each coset becomes a power series in the *single* automorphism $\sigma_{g^{M_2}}$:

$$\sum_{\sigma_g \in \Sigma_{\text{giant}}} \sigma_g(\text{bs}_{\sigma_g}) = \sum_{b \in \{\pm 1\}} \sigma_b \left(\sum_{j \in [M_1]} (\sigma_{g^{M_2}})^j (\text{bs}_j^{(b)}) \right), \quad (4)$$

where $\text{bs}_j^{(b)} := \sum_{\sigma_b \in \Sigma_{\text{baby}}} \tilde{c}_{\sigma_{bg^{jM_2}} \cdot \sigma_b} \cdot \sigma_b(x)$. Each inner sum is a polynomial in $\sigma_{g^{M_2}}$ and is therefore evaluated by Horner's rule, exactly as the baby step evaluates its own chain under σ_g .

¹We write evk without a subscript to denote the entire evaluation key. In practice, each automorphism Auto.Eval_i uses only its corresponding component of the evaluation key, e.g., evk_{σ_i} .

Algorithm 3 BSGSEval($c, \{c_\sigma\}, \text{evk}_g, \text{evk}_{g^{M_2}}, \text{evk}_{-1}$)

Input: $c \in \text{RLWE}_s(\mu)$; $\{c_\sigma\}_{\sigma \in \mathbb{Z}_{2m}^\times} \subset \mathcal{R}_p$; $\text{evk}_g, \text{evk}_{g^{M_2}}, \text{evk}_{-1}$.

Output: Ciphertext in $\text{RLWE}_s(\sum_\sigma c_\sigma \cdot \sigma(\mu))$.

```

1:  $c'_{\sigma_1} \leftarrow c$ .
2: for  $k = 1, \dots, M_2 - 1$  do ▷ Baby chain.
3:    $c'_{\sigma_{g^k}} \leftarrow \text{Auto.Eval}_g(c'_{\sigma_{g^{k-1}}}, \text{evk}_g) \in \mathcal{R}_q^2$ .
4: For  $b \in \{\pm 1\}$ ,  $j \in [M_1]$ ,  $\text{bs}_j^{(b)} := \sum_{\sigma_b \in \Sigma_{\text{baby}}} \tilde{c}_{\sigma_{bg^{jM_2}} \cdot \sigma_b} \cdot c'_{\sigma_b}$ .
5: for  $b \in \{\pm 1\}$  do ▷ Giant chain.
6:    $c_b \leftarrow \text{bs}_{M_1-1}^{(b)}$ .
7:   for  $j = M_1 - 2$  down to 0 do
8:      $c_b \leftarrow \text{Auto.Eval}_{g^{M_2}}(c_b, \text{evk}_{g^{M_2}}) + \text{bs}_j^{(b)}$ .
9: return  $m^{-1}(c_{+1} + \text{Auto.Eval}_{-1}(c_{-1}, \text{evk}_{-1}))$ .
```

We formalize this procedure in Algorithm 3, instantiated with $c := q_1$ and $\{c_\sigma\} := \{\hat{u}_i\}$, so as to evaluate RLWE.Eval in Construction 8. With this optimization, the Dimension 1 component of `Server.Answer` is computed as

$$c^* \leftarrow \text{BSGSEval}(q_1, \{\hat{u}_i\}_{i \in \mathbb{Z}_{2m}^\times}, \text{evk}_g, \text{evk}_{g^{M_2}}, \text{evk}_{-1}),$$

replacing the direct evaluation in Equation (3).

Three Evaluation Keys. Algorithm 3 runs on $\text{evk}_g, \text{evk}_{g^{M_2}}$ and evk_{-1} alone instead of client uploading all the keys, because both of its loops advance along a single generator and therefore reuse a single key: the baby chain by σ_g , the giant chain by $\sigma_{g^{M_2}}$, with evk_{-1} spent once to join the two cosets. No key is derived from another, so the server stores exactly what the client sent.

The number of automorphisms is unchanged by this choice: a direct evaluation of Equation (4) applies $|\Sigma_{\text{giant}}| - 1 = 2M_1 - 1$ giant automorphisms, and the two Horner chains plus the coset join total the same. What changes is which key each one uses, and hence how noise accumulates, and here the two steps are genuinely asymmetric. Baby-step automorphisms precede the multiplication by the $\tilde{c}$ entries, so their key-switching error is p -amplified and sets the noise floor of the whole evaluation; chaining them under the fresh evk_g keeps that error purely additive along the chain. Giant-step automorphisms follow that multiplication and escape the p -amplification entirely, so their contribution stays far below the floor however the chain is arranged, which is why the giant step is free to be optimized for storage rather than for noise. Appendix C.1 quantifies both.

3.5 Optimization 2: Offline Precomputing

For PIR deployments, many schemes exploit offline precomputation to further reduce the server's online response time.

In our framework, we identify two sources of such precomputable work.

The first is straightforward: any computation that depends only on the database is fixed once during the server setup phase `Server.Setup` and can subsequently be reused across all clients and queries. The second stems from an observation introduced in `ksPIR` [29]. Even seemingly query-dependent computation admits a useful decomposition. Specifically, an RLWE query ciphertext q_1 can be written as (b_1, a_1) , where the component a_1 is independent of the queried keyword and may therefore be transmitted or processed in advance. Since the homomorphic evaluation performed by the server is linear in the ciphertext, it naturally decomposes along the same partition. As a result, the entire computation associated with the a_1 branch, including all subsequent key-switching operations, can be performed offline before the client specifies its query.

Combining these two observations yields a three-stage offline/online workflow, which we outline below. Formal derivations and complexity analyses are deferred to Appendix C. A complete formalization is given in Algorithm 4.

Precomputing 1: Static BSGS Components. Section 3.4 introduced the rotated coefficients $\tilde{c}_{\sigma_g, \sigma_b} := \sigma_g^{-1}(c_{\sigma_g \circ \sigma_b})$ consumed by the giant step of Algorithm 3. For our Dimension-1 map f , the entire rotated family is fixed at `Server.Setup` which can be precomputed.

Precomputing 2: Dimension 1 (The BSGS Fold). We observe that the `BSGSEval` can be split into two branches: key-independent and key independent. The latter depends only on a_1 , $\{c_\sigma\}$ and the evaluation keys, and crucially it absorbs every key-switching operation. The server can evaluate that during the per-query offline phase.

Precomputing 3: Dimension 2 (The External Product). Applying the offline/online split to Dimension 2 requires splitting the gadget decomposition of the RLWE operand $c_0^* = c_{0, \text{off}}^* + c_{0, \text{on}}^*$. Although G^{-1} is non-linear, the external product in Equation (5) requires only a bounded representation satisfying the reconstruction, not canonical digits. while Precomputing 3 optimizes the external product, this layer remains a $\Theta(\ell_B m)$ lower-order term, which results in a minor speedup in the overall term as the major complexity is dominated by the Dimension-1 fold, i.e., $\Theta(m^2)$. Thus, we only formalize Precomputing 1 and 2 in Algorithm 4.

4 Implementation and Evaluation

To evaluate concrete performance, we implement Construction 8 and compare it with the fastest prior keyword PIR constructions of [28]. For a direct algorithmic comparison, we implement our scheme in Java on top of the `mpc4j` library,

Algorithm 4 `Server.Answer` (Offline/Online + BSGS).

Parameters: L .

Input: M via $\{c_\sigma\}_{\sigma \in \mathbb{Z}_{2^m}^\times}$, $c_\sigma := \hat{u}_\sigma$; (b_1, a_1) ; q_2 ; $\text{evk}_g, \text{evk}_{g^{M_2}}, \text{evk}_{-1}$.

Output: $\text{ans} := (L_j)_{j \in [L]}$ for the fixed block.

- 1: **Static phase** (per database):
 - 2: For every $(\sigma_g, \sigma_b) \in \Sigma_{\text{giant}} \times \Sigma_{\text{baby}}$, set $\tilde{c}_{\sigma_g, \sigma_b} \leftarrow \sigma_g^{-1}(c_{\sigma_g \circ \sigma_b})$.
 - 3: **Offline phase** (per query):
 - 4: $(c_{0, \text{off}}^*, c_1^*) \leftarrow \text{BSGSEval}((0, a_1), \{c_\sigma\}, \text{evk}_g, \text{evk}_{g^{M_2}}, \text{evk}_{-1})$.
 - 5: **Online phase** :
 - 6: $c_{0, \text{on}}^* \leftarrow m^{-1} \sum_{\sigma_g \in \Sigma_{\text{giant}}} \sigma_g(\sum_{\sigma_b \in \Sigma_{\text{baby}}} \tilde{c}_{\sigma_g, \sigma_b} \cdot \sigma_b(b_1))$; set $c^* \leftarrow (c_{0, \text{off}}^* + c_{0, \text{on}}^*, c_1^*)$.
 - 7: Set $T \leftarrow c^* \boxtimes q_2$.
 - 8: For every $j \in [L]$, set $L_j \leftarrow \text{Ext}(T, j)$.
 - 9: **return** $\text{ans} := (L_j)_{j \in [L]}$.
-

matching the implementation and environment of [28]. We also develop a C++ implementation, incorporating the engineering optimizations of `ksPIR` [29] together with AVX and multi-threading support to further reduce online latency.

Both implementations use the same dynamic parameter-selection procedure and are reported as two variants, `Ours(Java)` and `Ours(C++)`. We evaluate them on databases of size $n \in \{2^{18}, 2^{20}, 2^{22}\}$ and payload lengths $\ell_v \in \{32, 64, 128, 256\}$ bytes. Experiments are conducted on a Mini PC with an AMD Ryzen 9 PRO 8945HS processor (8 cores, 16 threads, 3.99 GHz) and 93.8 GB memory, running Windows 11. Java uses JDK 26, while C++ is compiled with GCC 15.2. Unless otherwise stated, all reported online latencies are measured in single-threaded mode.

4.1 Implementation Details

Parameter Selection. We defer the detailed asymptotic noise analysis of our final construction to Appendix C.1, and present the rule for parameter selection in Appendix C.2.

Our protocol. We set the security parameter to 128 bits. The back-end uses ring degree $m := 4096$ over a single 60-bit working prime, with a further 49-bit special prime used only inside key switching, plaintext modulus $p \in \{40961, 65537\}$ and gadget base $\log B \in \{3, 4, 5\}$, the last three selected per cell by the dynamic search following the analysis in Appendix C.2. All reported configuration is admitted only after passing a 24-keyword exact-bytes recovery gate on the instance it reports; all 36 cells pass 24/24 functional check at the load they report, with no step-down. Each value is prefixed by a 64-bit tag, and the two baselines use a tag of the same width. Client and server run in a single process as a benchmark instead of a real-world deployment. Latencies are medians of repeated runs taken after JVM warm-up rather

than single cold calls. Appendix D gives the detailed parameters and other settings.

Baselines. Throughout this section we call the *front-end* the layer that turns a keyword into what the server must fetch such as our BBH placement and α -KVS decode vector, and the key-value lookup of [28], while the *back-end* is the underlying index-PIR layer that evaluates the encrypted query over the encoded database and forms the response. The two are independent, which is what makes the ablation below meaningful. We compare ours against two schemes. 3-KsPIRV replaces our front-end with the three-query front-end of KPIR^{kvs} [28] while retaining *our* back-end unchanged; it is implemented in Java on the same library as Ours (*Java*), so the difference between the two isolates the front-end and nothing else. Because it cannot aggregate, it issues $\alpha = 3$ independent queries per block and ships $3N_{\text{blk}}$ answers to our N_{blk} . KVS^{index} is the third construction of [28], run from the authors’ own implementation, and is a hint-based scheme: its online speed is accompanied with a per-client hint of 14.5–143.6 MB that the client must download and store before its first query. We run their implementation as is, only including existing optimizations within their library and documentation. We note that their code does not use the multi-block technique in Section 3.1, so its query grows with n where ours does not.

4.2 Evaluation Result

Table 3 reports the full grid. Setup is a one-time per-database cost and sums two stages, the BBH and α -KVS encoding and the auxiliary table build of Section 3.3; client key generation is per client rather than per database and is charged to the evaluation keys instead. The offline phase runs once per query but before the queried keyword is known, so neither setup nor offline contributes to the online latency a client observes. This offline/online structure follows the setting of ksPIR [29].

Comparison of the front-end. Ours (*Java*) and 3-KsPIRV share an identical back-end and differ only in the front-end, so their ratio is the cleanest measurement in the table. Our front-end is faster in every cell, by 2.24–2.99 $\times$ with a mean of 2.65 $\times$, attributed to our framework utilizing linear homomorphism. The gain is stable across both axes of the grid and does not decay as n grows. It is also not free of a side effect: 3-KsPIRV’s filter reaches an expansion rate of 1.125 against our 1.309, so it packs the same database into fewer blocks, 41 against our 49 at 2^{22} with 256-byte payloads, and wins back part of its loss per block from the theoretical $3\times$ to 2.64 $\times$.

Hintless versus hint-based. KVS^{index} is the fastest construction reported in [28]. When compared with Ours (*Java*), which is implemented on the same mpc4j framework, KVS^{index} achieves lower online latency in ten of the twelve

parameter settings, with up to a 2.75 $\times$ advantage at $n = 2^{18}$ and a 32-byte payload. However, KVS^{index} requires a per-client hint of 14.5–143.6 MB, which must be generated and stored by every client, whereas our scheme is hintless. Compared with Ours (C++), our scheme is faster in all twelve settings, achieving a 1.42–10.33 $\times$ speedup with a geometric mean of 3.80 $\times$.

At the algorithmic level, this performance difference stems from the characteristics of the underlying SimplePIR and KsPIR designs. The keyword PIR constructions built upon them naturally inherit these tradeoffs. At the implementation level, our Java implementation is already competitive with the fastest construction of [28]. The C++ implementation further benefits from a native execution environment, together with engineering optimizations adapted from the KsPIR implementation [29] to our keyword PIR framework, as well as AVX acceleration and multi-threading. Since [28] does not provide a C++ implementation, we do not include a direct comparison in that setting.

Communication. As to the Java version, our query is constant across the entire grid at 300 KB due to the multi-block design: 30 KB offline for a_1 and 270 KB online for the query-dependent part; while the response grows with n from 13.1 to 657.5 KB, because the server returns one answer per block. 3-KsPIRV pays 900 KB for its three independent queries, exactly three times ours, and KVS^{index}’s query grows with the database from 12.7 to 130.0 KB. Both implementations send the same query object, one ring element offline for a_1 and nine online, namely b_1 together with the eight components of the RGSW selector; the C++ build’s 280 KB against Java’s 300 KB is therefore only the coefficient width each serializer packs at, 56 bits against 60. Evaluation keys cost 3.9 MB once per client in Java and 9.2 MB in C++, the difference being the finer gadget the C++ search selects. Both figures owe to the three-key upload of Section 3.4: at $m = 4096$ the full $\Theta(\sqrt{m})$ family is 126 keys, so uploading it directly would cost $42\times$ more, 165 MB in Java and 386 MB in C++.

Setup and offline latency. The setup of KVS^{index} is benchmarked in a single-threaded configuration to keep its implementation unmodified. Our one-time setup runs 0.7–107.0 s and splits into two parts: the BBH and α -KVS encoding and the per-block table build of Section 3.3; only the latter is multi-threaded, on 16 cores, and it is the sole exception to the single-threaded rule stated above. Client key generation is 16–17 s, flat in n ; we report it separately because it is per-client rather than per-database. The per-query offline phase absorbs every key switch in the evaluation and costs 0.45–43.2 s against 4.92–108.3 s for 3-KsPIRV, which must repeat it for each of its three queries.

Table 3: Full evaluation results measured over the entire database. KVS^{index} [28] relies on a per-client hint (last column), differentiating it from the three hintless schemes below. 3-KsPIRV combines the front-end of [28] with our back-end, whereas Ours (Java) and Ours (C++) are our full construction. The best results in the six comparable columns are shaded **green**.

Settings		N_{blk}	Latency				Communication [‡]			Hint
n / ℓ_v	Scheme		Online* (ms)	Thr.* (MB/s)	Setup [†] (s)	Offline (s)	Query (KB)	Evk (MB)	Resp. (KB)	Size (MB)
$2^{18}/32\text{ B}$	KVS^{index}	–	15.0	533	6.9	–	12.7	–	14.4	14.5
	3-KsPIRV	1	120.5	66	2.2	4.92	90+810	3.9	39.2	–
	Ours (Java)	1	41.3	194	2.3	1.66	30+270	3.9	13.1	–
	Ours (C++)	1	10.6	755	0.7	0.45	28+252	9.2	12.6	–
$2^{18}/64\text{ B}$	KVS^{index}	–	28.0	571	10.9	–	17.0	–	20.3	20.3
	3-KsPIRV	1	120.9	132	2.3	4.93	90+810	3.9	39.3	–
	Ours (Java)	1	40.5	395	2.4	1.66	30+270	3.9	13.1	–
	Ours (C++)	1	9.6	1667	0.7	0.47	28+252	9.2	12.6	–
$2^{18}/128\text{ B}$	KVS^{index}	–	120.0	267	23.9	–	23.3	–	29.2	29.3
	3-KsPIRV	2	243.0	132	4.1	7.54	90+810	3.9	79.3	–
	Ours (Java)	2	81.7	392	4.6	2.54	30+270	3.9	26.4	–
	Ours (C++)	2	19.2	1667	1.5	1.00	28+252	9.2	26.4	–
$2^{18}/256\text{ B}$	KVS^{index}	–	248.0	258	52.8	–	32.5	–	44.3	44.4
	3-KsPIRV	3	364.4	176	6.2	10.25	90+810	3.9	120.8	–
	Ours (Java)	3	121.8	525	6.3	3.41	30+270	3.9	40.3	–
	Ours (C++)	3	24.0	2667	3.2	1.54	28+252	9.2	40.3	–
$2^{20}/32\text{ B}$	KVS^{index}	–	74.0	432	26.3	–	25.3	–	27.0	27.3
	3-KsPIRV	2	242.0	132	4.8	7.53	90+810	3.9	78.4	–
	Ours (Java)	2	81.5	393	5.2	2.51	30+270	3.9	25.1	–
	Ours (C++)	2	17.7	1808	4.6	0.93	28+252	9.2	25.1	–
$2^{20}/64\text{ B}$	KVS^{index}	–	135.0	474	47.4	–	33.9	–	37.1	37.4
	3-KsPIRV	3	373.8	171	6.9	10.12	90+810	3.9	118.0	–
	Ours (Java)	4	162.8	393	9.2	4.25	30+270	3.9	50.4	–
	Ours (C++)	4	31.8	2013	3.3	1.95	28+252	9.2	54.5	–
$2^{20}/128\text{ B}$	KVS^{index}	–	261.0	490	78.6	–	46.6	–	52.6	52.9
	3-KsPIRV	6	729.9	175	12.6	17.86	90+810	3.9	237.9	–
	Ours (Java)	7	284.8	449	14.8	6.88	30+270	3.9	92.5	–
	Ours (C++)	7	66.0	1939	5.3	3.48	28+252	9.2	92.5	–
$2^{20}/256\text{ B}$	KVS^{index}	–	492.0	520	163.6	–	65.0	–	77.3	77.7
	3-KsPIRV	11	1348.0	190	21.5	31.04	90+810	3.9	442.8	–
	Ours (Java)	13	528.9	484	25.1	12.10	30+270	3.9	174.4	–
	Ours (C++)	13	104.5	2450	19.1	7.52	28+252	9.2	174.4	–
$2^{22}/32\text{ B}$	KVS^{index}	–	186.0	688	91.1	–	50.6	–	52.3	53.6
	3-KsPIRV	6	731.5	175	16.0	18.03	90+810	3.9	235.1	–
	Ours (Java)	8	326.1	393	21.4	7.71	30+270	3.9	104.5	–
	Ours (C++)	8	68.0	1882	26.5	4.24	28+252	9.2	100.5	–
$2^{22}/64\text{ B}$	KVS^{index}	–	346.0	740	169.5	–	67.9	–	71.2	72.4
	3-KsPIRV	11	1348.0	190	25.6	30.77	90+810	3.9	432.8	–
	Ours (Java)	14	571.1	448	50.5	12.94	30+270	3.9	176.5	–
	Ours (C++)	14	115.7	2213	17.7	7.75	28+252	9.2	183.6	–
$2^{22}/128\text{ B}$	KVS^{index}	–	810.0	632	317.8	–	93.3	–	99.3	100.6
	3-KsPIRV	21	2585.6	198	45.2	56.68	90+810	3.9	832.6	–
	Ours (Java)	26	1063.2	482	67.9	23.33	30+270	3.9	343.6	–
	Ours (C++)	26	194.8	2628	34.9	15.52	28+252	9.2	356.8	–
$2^{22}/256\text{ B}$	KVS^{index}	–	1107.0	925	644.8	–	130.0	–	142.3	143.6
	3-KsPIRV	41	5108.6	200	96.5	108.31	90+810	3.9	1650.5	–
	Ours (Java)	49	2013.8	508	107.0	43.20	30+270	3.9	657.5	–
	Ours (C++)	49	360.2	2843	87.8	43.16	28+252	9.2	657.5	–

* **Online Time/Throughput:** Single-threaded. Throughput is database size over pure server-side online time. KVS^{index} is measured unmodified via its native log appender.

† **Setup:** Includes single-threaded encoding (BBH and α -KVS) and 16-threaded auxiliary table building (Section 3.3). Per-client key generation (≈ 16 s) is reported separately under Evk (Appendix D). However, KVS^{index} is benchmarked in a single-threaded configuration to keep its implementation unmodified.

‡ **Communication:** Serialized buffer lengths packed at the implementation’s coefficient width (56 bits in C++, 60 in Java). Responses are rescaled before measurement. Queries are constant in n .

References

- [1] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, “Private information retrieval,” in *Proc. IEEE FOCS*, 1995, pp. 41–50.
- [2] H. Corrigan-Gibbs and D. Kogan, “Private information retrieval with sublinear online time,” in *Proc. EUROCRYPT*, 2020, pp. 44–75.
- [3] Z. Dvir and S. Gopi, “2-server pir with subpolynomial communication,” *Journal of the ACM (JACM)*, vol. 63, no. 4, pp. 1–15, 2016.
- [4] I. Ahmad, Y. Yang, D. Agrawal, A. El Abbadi, and T. Gupta, “Addra: Metadata-private voice communication over fully untrusted infrastructure,” in *15th {USENIX} symposium on operating systems design and implementation ({OSDI} 21)*, 2021.
- [5] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo, “Communication-computation trade-offs in PIR,” in *Proc. USENIX Security*, 2021, pp. 1811–1828.
- [6] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, “PIR with compressed queries and amortized query processing,” in *Proc. IEEE S&P*, 2018, pp. 962–979.
- [7] H. Corrigan-Gibbs, A. Henzinger, and D. Kogan, “Single-server private information retrieval with sublinear amortized time,” in *Proc. EUROCRYPT*, 2022, pp. 3–33.
- [8] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. Vaikuntanathan, “One server for the price of two: Simple and fast single-server private information retrieval,” in *Proc. USENIX Security*, 2023, pp. 3889–3905.
- [9] E. Kushilevitz and R. Ostrovsky, “Replication is NOT needed: SINGLE database, computationally-private information retrieval,” in *Proc. IEEE FOCS*, 1997, pp. 364–373.
- [10] S. J. Menon and D. J. Wu, “SPIRAL: Fast, high-rate single-server PIR via FHE composition,” in *Proc. IEEE S&P*, 2022, pp. 930–947.
- [11] M. H. Mughees, H. Chen, and L. Ren, “OnionPIR: Response efficient single-server PIR,” in *Proc. ACM CCS*, 2021, pp. 2292–2306.
- [12] M. Zhou, A. Park, W. Zheng, and E. Shi, “Piano: Extremely simple, single-server PIR with sublinear server computation,” in *Proc. IEEE S&P*, 2024, pp. 4296–4314.
- [13] A. Lazzeretti and C. Papamanthou, “TreePIR: Sublinear-time and polylog-bandwidth private information retrieval from DDH,” in *Proc. CRYPTO*, 2023, pp. 284–314.
- [14] S. Patel, G. Persiano, and K. Yeo, “Private stateful information retrieval,” in *Proc. ACM CCS*, 2018, pp. 1002–1019.
- [15] L. de Castro, K. Lewi, and E. Suh, “Whispir: Stateless private information retrieval with low communication,” *Cryptology ePrint Archive*, 2024.
- [16] B. Li, D. Micciancio, M. Raykova, and M. Schultz, “Hintless single-server private information retrieval,” in *Proc. CRYPTO*, 2024, pp. 183–217.
- [17] S. J. Menon and D. J. Wu, “YPIR: High-throughput single-server PIR with silent preprocessing,” in *Proc. USENIX Security*, 2024.
- [18] A. Ghoshal, M. Zhou, and E. Shi, “Efficient preprocessing PIR without public-key cryptography,” in *Proc. EUROCRYPT*, 2024, pp. 210–240.
- [19] A. Ghoshal, M. Zhou, E. Shi, and B. Peng, “Pseudo-random functions with weak programming privacy and applications to private information retrieval,” in *Proc. EUROCRYPT*, 2025, pp. 284–313.
- [20] A. Kwon, D. Lazar, S. Devadas, and B. Ford, “Riffle: An efficient communication system with strong anonymity,” *popets*, vol. 2016, no. 2, pp. 115–134, 2016.
- [21] P. Mittal, F. G. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, “PIR-tor: Scalable anonymous communication using private information retrieval,” in *Proc. USENIX Security*, 2011.
- [22] E. Fung, G. Kellaris, and D. Papadimas, “Combining differential privacy and pir for efficient strong location privacy,” in *International Symposium on Spatial and Temporal Databases*. Springer, 2015, pp. 295–312.
- [23] D. Kogan and H. Corrigan-Gibbs, “Private blocklist lookups with checklist,” in *Proc. USENIX Security*, 2021, pp. 875–892.
- [24] B. Chor, N. Gilboa, and M. Naor, “Private information retrieval by keywords,” *Cryptology ePrint Archive*, Paper 1998/003, 1998, <https://eprint.iacr.org/1998/003>.
- [25] S. Celi and A. Davidson, “Call me by my name: Simple, practical private information retrieval for keyword queries,” in *Proc. ACM CCS*, 2024, pp. 4107–4121.
- [26] R. A. Mahdavi and F. Kerschbaum, “Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators,” in *Proc. USENIX Security*, 2022, pp. 1723–1740.

- [27] S. Patel, J. Y. Seo, and K. Yeo, “Don’t be dense: Efficient keyword PIR for sparse databases,” in *Proc. USENIX Security*, 2023, pp. 3853–3870.
- [28] M. Hao, W. Liu, L. Peng, C. Zhang, P. Wu, L. Zhang, H. Li, and R. H. Deng, “Practical keyword private information retrieval from key-to-index mappings,” in *Proc. USENIX Security*, 2025, pp. 3397–3416.
- [29] M. Luo, F.-H. Liu, and H. Wang, “Faster FHE-based single-server private information retrieval,” in *Proc. ACM CCS*, 2024, pp. 1405–1419.
- [30] A. Bienstock, S. Patel, J. Y. Seo, and K. Yeo, “Near-optimal oblivious key-value stores for efficient PSI, PSU and volume-hiding multi-maps,” in *Proc. USENIX Security*, 2023, pp. 301–318.
- [31] G. Garimella, B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, “Oblivious key-value stores and amplification for private set intersection,” in *Proc. CRYPTO*, 2021, pp. 395–425.
- [32] S. Raghuraman and P. Rindal, “Blazing fast PSI from improved OKVS and subfield VOLE,” in *Proc. ACM CCS*, 2022, pp. 2505–2517.
- [33] C. Zhang, Y. Chen, W. Liu, M. Zhang, and D. Lin, “Linear private set union from multi-query reverse private membership test,” in *Proc. USENIX Security*, 2023, pp. 337–354.
- [34] C. Peikert and Z. Pepin, “Vive galois! Part 1: Optimal SIMD packing and packed bootstrapping for FHE,” in *Proc. TCC*, 2025, pp. 185–219.
- [35] W. Liu *et al.*, “mpc4j: An efficient multi-party computation library,” <https://github.com/alibaba-edu/mpc4j>, 2024.
- [36] S. Halevi and V. Shoup, “Faster homomorphic linear transformations in HELib,” in *Proc. CRYPTO*, 2018, pp. 93–120.
- [37] A. Davidson, G. Pestana, and S. Celi, “FrodoPIR: Simple, scalable, single-server private information retrieval,” *popets*, vol. 2023, no. 1, pp. 365–383, 2023.
- [38] H. Liang, H. Yu, C. Xu, D. Zhan, J. Yang, and H. Zhang, “Bridging keyword PIR and index PIR via minimal perfect hash function and batch PIR,” Cryptology ePrint Archive, Paper 2025/2252, 2025, <https://eprint.iacr.org/2025/2252>.
- [39] A. Arastehfard, W. Liu, Q. Zhou, Z. Shen, L. Peng, L. Qu, S. Feng, and Y. Hong, “KPIR-C: Keyword PIR with arbitrary server-side computation,” Cryptology ePrint Archive, Paper 2025/1952, 2025, <https://eprint.iacr.org/2025/1952>.
- [40] B. Fuller, A. Hamlin, A. Paul, M. Rezapour, R. Sahu, A. Shukla, and M. Stuart, “Updatable oblivious key value stores with access control and application to multi key searchable encryption,” Cryptology ePrint Archive, Paper 2026/1590, 2026, <https://eprint.iacr.org/2026/1590>.
- [41] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” in *Proc. ACM STOC*, 2005, pp. 84–93.
- [42] L. Ducas and D. Micciancio, “FHEW: Bootstrapping homomorphic encryption in less than a second,” in *Proc. EUROCRYPT*, 2015, pp. 617–640.
- [43] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds,” in *Proc. ASIACRYPT*, 2016, pp. 3–33.
- [44] J. Alperin-Sheriff and C. Peikert, “Faster bootstrapping with polynomial error,” in *Proc. CRYPTO*, 2014, pp. 297–314.
- [45] C. Gentry, A. Sahai, and B. Waters, “Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based,” in *Proc. CRYPTO*, 2013, pp. 75–92.
- [46] H. Chen, W. Dai, M. Kim, and Y. Song, “Efficient homomorphic conversion between (ring) LWE ciphertexts,” in *Proc. ACNS*, 2021, pp. 460–479.
- [47] D. Micciancio and C. Peikert, “Trapdoors for lattices: Simpler, tighter, faster, smaller,” in *Proc. EUROCRYPT*, 2012, pp. 700–718.
- [48] N. P. Smart and F. Vercauteren, “Fully homomorphic SIMD operations,” *dcc*, vol. 71, no. 1, pp. 57–81, 2014.
- [49] C. Gentry, S. Halevi, and N. P. Smart, “Homomorphic evaluation of the AES circuit,” in *Proc. CRYPTO*, 2012, pp. 850–867.
- [50] Z. Brakerski and V. Vaikuntanathan, “Fully homomorphic encryption from ring-LWE and security for key dependent messages,” in *Proc. CRYPTO*, 2011, pp. 505–524.
- [51] M. J. Freedman, K. Nissim, and B. Pinkas, “Efficient private matching and set intersection,” in *Proc. EUROCRYPT*, 2004, pp. 1–19.
- [52] M. Raab and A. Steger, ““balls into bins” — a simple and tight analysis,” in *Randomization and Approximation Techniques in Computer Science (RANDOM)*, 1998, pp. 159–170.

- [53] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, “Balanced allocations (extended abstract),” in *Proc. ACM STOC*, 1994, pp. 593–602.
- [54] R. Moenck and A. Borodin, “Fast modular transforms via division,” in *13th Annual Symposium on Switching and Automata Theory (swat 1972)*. IEEE, 1972, pp. 90–96.
- [55] T. M. Graf and D. Lemire, “Binary fuse filters: Fast and smaller than xor filters,” *ACM Journal of Experimental Algorithmics*, vol. 27, pp. 1–15, 2022, <https://arxiv.org/abs/2201.01174>.
- [56] Z. Pepin, “Algebraic structure in lattice cryptography,” Ph.D. dissertation, University of Michigan, 2025.
- [57] V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings,” in *Proc. EUROCRYPT*, 2010, pp. 1–23.
- [58] C. Gentry, C. Peikert, and V. Vaikuntanathan, “Trapdoors for hard lattices and new cryptographic constructions,” in *Proc. ACM STOC*, 2008, pp. 197–206.

A Deferred Preliminaries

A.1 Lattice Primitives

This subsection is the full version of Section 2.1, which states only the syntax. Nothing here is new; we record the concrete algorithms, the two identities the construction relies on, and the noise of the external product.

Assumptions. Let $p \ll q$ be the positive integer moduli. For a power-of-two m , we define the cyclotomic ring $\mathcal{R} := \mathbb{Z}[X]/(X^m + 1)$ and its quotients $\mathcal{R}_q := \mathcal{R}/q\mathcal{R}$ for $q \in \{p, q\}$. The (N, M, q, χ) -LWE assumption was introduced by Regev [41], which states that $(\mathbf{A}, \vec{s} \cdot \mathbf{A} + \vec{e}) \approx_c (\mathbf{A}, \vec{u})$ for $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{N \times M}$, $\vec{s} \xleftarrow{\$} \mathbb{Z}_q^N$, $\vec{u} \xleftarrow{\$} \mathbb{Z}_q^M$, and $\vec{e} \leftarrow \chi^M$. The $(\mathcal{R}, q, \chi)$ -RLWE assumption is defined analogously over $\mathcal{R}_q$ (of degree m), stating $(a, a \cdot s + e) \approx_c (a, u)$ for $a, s, u \xleftarrow{\$} \mathcal{R}_q$ and $e \leftarrow \chi$.

RLWE Encryption. There are a number of plausible LWE-based homomorphic encryption schemes [41, 50, 57, 58]; we adopt the standard Regev-style [41] secret-key variant throughout this paper, while our paper is compatible with other variants with proper modification. We write $\vec{c} \in \text{LWE}_{\vec{s}}(\mu)$ for the resulting ciphertext, where the LWE Encryption consists of the following *ppt* algorithms parameterized by dimensions N :

- $\text{LWE.KeyGen}(1^\lambda) \rightarrow \text{sk}_{\text{LWE}}$: samples a secret key $\vec{s} \xleftarrow{\$} \mathbb{Z}_q^N$ and outputs $\text{sk}_{\text{LWE}} := (1, \vec{s})^\top$.
- $\text{LWE.EvalKeyGen}(1^\lambda) \rightarrow \text{evk}$: sample the randomness and generate an evaluation key evk .
- $\text{LWE.Encrypt}(\text{sk}_{\text{LWE}}, \mu \in \mathbb{Z}_p) \rightarrow c$: samples $\vec{a} \xleftarrow{\$} \mathbb{Z}_q^N$, $\vec{e} \leftarrow \chi$, and outputs $\vec{c} := (b, \vec{a}) = (-\vec{s} \cdot \vec{a} + \vec{e} + \Delta \cdot \mu, \vec{a}) \in \mathbb{Z}_q^{N+1}$, where $\Delta := \lfloor q/p \rfloor$.
- $\text{LWE.Decrypt}(\text{sk}_{\text{LWE}}, \vec{c}) \rightarrow \mu$: on inputs $(\text{sk}_{\text{LWE}}, \vec{c})$, outputs $\lfloor (b + \vec{s} \cdot \vec{a})/\Delta \rfloor \bmod p$.
- $\text{LWE.Eval}(f, \vec{c}, \text{evk})$: given function f , $c \in \text{LWE}_{\vec{s}}(\mu)$ and an evaluation key evk , outputs a ciphertext in $\text{LWE}_{\vec{s}}(f(\mu))$.

Similarly, the RLWE encryption scheme consists of *ppt* algorithms RLWE.KeyGen , RLWE.Encrypt , RLWE.Decrypt , RLWE.Eval obtained by replacing $\mathbb{Z}_p$ and $\mathbb{Z}_q^N$ with $\mathcal{R}_p$ and $\mathcal{R}_q$, respectively, i.e., $c := (b, a) = (-as + e + \Delta\mu, a) \in \mathcal{R}_q^2$ with $\mu \in \mathcal{R}_p$, $a \xleftarrow{\$} \mathcal{R}_q$ and $s \xleftarrow{\$} \mathcal{R}_q$. We write $c \in \text{RLWE}_s(\mu)$ for the resulting ciphertext.

Extraction. Given a ring element, we denote the process that isolates its coefficient vector as coef , i.e., given $a := \sum_i a_i X^i \in \mathcal{R}_q$ with $a_i \in \mathbb{Z}_q$, let $\text{coef}(a) := (a_0, \dots, a_{m-1})$. On the contrary, $\text{coef}^{-1}(a_0, \dots, a_{m-1}) := a$. For $a \in \mathcal{R}_q$, define $\vec{a} := \text{coef}(a(X^{-1})) \in \mathbb{Z}_q^m$. Consequently, given an RLWE ciphertext $c = (b, a) \in \text{RLWE}_s(\mu)$, there is a simple way to extract a LWE ciphertext encrypting the j -th coefficient of μ under the derived secret $\vec{s} \in \mathbb{Z}_q^m$ [46]. We denote

this coefficient extraction, at any target index $j \in [m]$, by $\text{Ext}(c, j) := (\text{coef}(b)[j], \text{coef}(a)^{(j)}) \in \text{LWE}_{\vec{s}}(\text{coef}(\mu)[j])$, where $\text{coef}(a)^{(j)}$ is the negacyclically reindexed coefficient vector for index j (with $\text{coef}(a)^{(0)} = \text{coef}(a)$).

Gadget Decomposition. Gadget decomposition is a frequently used technique in homomorphic encryption to reduce the noise growth. Fix a gadget base $B \in \mathbb{N}$, then $\ell_B := \lceil \log_B q \rceil$, and for $k \in \mathbb{N}$, define the gadget matrix

$$\mathbf{G}_k := \mathbf{I}_k \otimes \vec{g} \in \mathbb{Z}_q^{k\ell_B \times k}, \vec{g} := (B^0, \dots, B^{\ell_B-1})^\top \in \mathbb{Z}_q^{\ell_B}.$$

For $\vec{x} \in \mathcal{R}_q^{1 \times k}$, its gadget decomposition $\mathbf{G}^{-1}(\vec{x}) \in \mathcal{R}^{1 \times k\ell_B}$ is applied coefficient-wise, satisfying $\|\mathbf{G}^{-1}(\vec{x})\| \leq B/2$ and $\mathbf{G}^{-1}(\vec{x}) \cdot \mathbf{G}_k = \vec{x} \pmod{q}$. Dually, for $\vec{y} \in \mathcal{R}_q^k$, defining $\text{PowersOfB}(\vec{y}) := \mathbf{G}_k \cdot \vec{y} \in \mathcal{R}_q^{k\ell_B}$, it is well known that

$$\mathbf{G}^{-1}(\vec{x}) \cdot \text{PowersOfB}(\vec{y}) = \langle \vec{x}, \vec{y} \rangle \pmod{q}.$$

RGSW Encryption. RGSW [42–44] is the ring analogue of the Gentry-Sahai-Waters scheme [45], providing an asymmetric external product between an RLWE and an RGSW ciphertext. Let $C \in \text{RGSW}_s(\mu)$ denote an encryption of μ under the key $\text{sk} := (1, s)^\top$. The scheme consists of the following *ppt* algorithms:

- $\text{RGSW.KeyGen}(1^\lambda) \rightarrow \text{sk}$: outputs $\text{sk} := (1, s)^\top \leftarrow \text{RLWE.KeyGen}(1^\lambda)$.
- $\text{RGSW.Encrypt}(\text{sk}, \mu \in \mathcal{R}_p) \rightarrow C$: samples $\mathbf{Z} \in \mathcal{R}_q^{2\ell_B \times 2}$ whose $2\ell_B$ rows are independent RLWE encryptions of 0 under sk , and outputs

$$C := \mathbf{Z} + \mu \cdot \mathbf{G} \in \mathcal{R}_q^{2\ell_B \times 2}$$

, satisfying $C \cdot \text{sk} = \vec{e} + \mu \cdot \text{PowersOfB}(\text{sk})$ for $\vec{e} \leftarrow \chi^{2\ell_B}$. For $i \in [\ell_B]$, $j \in [2]$, let $C[i, j]$ denote the row at index $j\ell_B + i$, yielding $\langle C[i, j], \text{sk} \rangle = \vec{e}[j\ell_B + i] + B^i \mu \text{sk}[j]$.

- $\text{RGSW.Decrypt}(\text{sk}, C) \rightarrow \mu$: inputs (sk, C) , outputs $\lfloor \langle C[\ell_B - 1, 0], \text{sk} \rangle / B^{\ell_B-1} \rfloor$ if $\|\vec{e}\| < B^{\ell_B-1}/2$.
- $\text{RGSW.ExternalProduct}(c, C) \rightarrow c'$ (denoted as $c \boxtimes C$): given $c = (b, a) \in \text{RLWE}_s(\mu')$ and $C \in \text{RGSW}_s(\mu)$, outputs $c \boxtimes C := \mathbf{G}^{-1}(c) \cdot C \in \text{RLWE}_s(\mu\mu')$ which is

$$\sum_{i \in [\ell_B]} \mathbf{G}^{-1}(b)_i \cdot C[i, 0] + \sum_{i \in [\ell_B]} \mathbf{G}^{-1}(a)_i \cdot C[i, 1], \quad (5)$$

with noise $e_{\boxtimes} = \mu e_c + \langle \mathbf{G}^{-1}(c), \vec{e} \rangle$ with

$$\|\langle \mathbf{G}^{-1}(c), \vec{e} \rangle\| = O(\ell_B B \sqrt{m} \sigma), \quad (6)$$

where e_c is the noise of c .

Equation (6) highlights the scheme's asymmetry: the fresh noise $\vec{e}$ is amplified only by bounded gadget digits ($\leq B/2$), making this additive error independent of p .

Key Switching. Here we recall the key-switching operation proposed by [50], which is useful for our design. It consists of *ppt* algorithms (KS.KeyGen, KS):

- KS.KeyGen(s, s') $\rightarrow$ ksk: given $s' \in \mathcal{R}_q$ and target $s \in \mathcal{R}_q$, outputs $\text{ksk} \leftarrow \text{RLWE.Encrypt}(s, \text{PowersOfB}(s')) \in \mathcal{R}_q^{\ell_B \times 2}$, i.e. the row-wise encryption under s of the powers-of- B expansion of s' .
- KS(c, ksk) $\rightarrow$ c' : given $c = (b, a) \in \text{RLWE}_s(\mu)$ and ksk, outputs $c' := (b, 0) + G^{-1}(a) \cdot \text{ksk} \in \text{RLWE}_s(\mu)$.

Automorphisms. Galois automorphisms provide a natural mechanism for permuting plaintext elements in homomorphic encryption, which was proposed in [47–49]. For $\mathcal{R}_q$ of power-of-two degree m , the Galois automorphism $\sigma_t: X \mapsto X^t, t \in \mathbb{Z}_{2m}^\times$ can be applied homomorphically to a RLWE ciphertext via the following *ppt* algorithms:

- Auto.KeyGen(s, t) $\rightarrow$ ksk: on inputs (s, t) , outputs $\text{ksk} \leftarrow \text{KS.KeyGen}(s, \sigma_t(s))$.
- Auto.Eval_t(c, ksk) $\rightarrow$ c' : given $c = (b, a)$ and ksk, outputs $c' := \text{KS}((\sigma_t(b), \sigma_t(a)), \text{ksk}) \in \text{RLWE}_s(\sigma_t(\mu))$.

B Deferred Material for Our Construction

B.1 Instantiations of the α -KVS

In this section, we present two concrete instantiations of α -KVS (ref. Definition 2): the 3-KVS [28] (sparse, based on a segmented binary fuse filter) and the classical polynomial KVS [54] (dense, based on polynomial interpolation). Both fit into EncodeDB (Algorithm 1) and our keyword PIR framework.

Instantiation I: the 3-KVS. Fix $\alpha = 3$ and partition the domain $[d]$ into $n_{\text{seg}} := d/\zeta$ disjoint segments $S_i := \{i\zeta, \dots, (i+1)\zeta - 1\}$, $i \in [n_{\text{seg}}]$, of a common segment length ζ . The segment count n_{seg} is a free parameter and is *not* tied to α : a key occupies one position in each of α consecutive segments, starting from a segment index derived from the key itself.

- α -KVS.Setup(1^λ): Sample a seed $\text{sd} \xleftarrow{\$} \{0, 1\}^\lambda$ for a pseudorandom function PRF which takes (j, k) as inputs. Define the start segment and, for $j \in [\alpha]$, the positions

$$\begin{aligned} \text{seg}(k) &:= \text{PRF}_{\text{sd}}(0, k) \bmod (n_{\text{seg}} - \alpha + 1), \\ h_j(k) &:= (\text{seg}(k) + j)\zeta + (\text{PRF}_{\text{sd}}(1 + j, k) \bmod \zeta). \end{aligned}$$

Output $\text{pp}_{\alpha\text{-KVS}} := (H, \vec{w}(\cdot))$ with $H = (h_0, h_1, h_2)$ and

$$\vec{w}(k) := \sum_{j \in [3]} \vec{e}_{h_j(k)} \in \mathbb{Z}_p^d. \quad (7)$$

The α segments $S_{\text{seg}(k)}, \dots, S_{\text{seg}(k)+\alpha-1}$ are distinct, so the α positions are mutually distinct and the Hamming weight of $\vec{w}(k)$ is exactly $\alpha = 3$. Every nonzero entry of $\vec{w}(k)$ equals 1, so this instantiation is the special case of Definition 2

noted in Remark 3; the generality of arbitrary decode coefficients is carried by the definition itself and exercised by Instantiation II.

- α -KVS.Encode($\text{pp}_{\alpha\text{-KVS}}, L$): Solve the sparse linear system $\{\langle \vec{w}(k), \mathbf{D} \rangle = v\}_{(k,v) \in L}$ for $\mathbf{D} \in \mathbb{Z}_p^d$, or output $\perp$ if it is unsolvable for the sampled H . Because $\vec{w}(k)$ has exactly one nonzero entry in each of the three segments, the system is solvable, and $\mathbf{D}$ computable in expected time $O(|L|)$, whenever the expansion rate satisfies $\eta \geq \eta_3(|L|)$ for the law of Appendix C.2; otherwise α -KVS.Encode fails and the seed sd is resampled. The solver makes two passes. A *peeling* pass repeatedly removes a key that owns a position no other remaining key touches, which yields an elimination order; a *back-substitution* pass then walks that order in reverse and, writing $h_{j^*}(k)$ for the position at which k was peeled, assigns

$$\mathbf{D}[h_{j^*}(k)] := v - \sum_{j \neq j^*} \mathbf{D}[h_j(k)] \bmod p.$$

Peeling is purely combinatorial and back-substitution only subtracts, because every decode coefficient is 1; neither pass inverts an element of $\mathbb{Z}_p$, so α -KVS.Encode imposes no invertibility requirement of its own on the plaintext modulus. Output $\mathbf{D}$.

- α -KVS.Decode($\text{pp}_{\alpha\text{-KVS}}, \mathbf{D}, k$): Output
 $\sum_{j \in [3]} \mathbf{D}[h_j(k)] \bmod p.$

The Segment Length is a Free Knob. Two properties of ζ matter for parameter selection and are easy to overlook, because in the degenerate case $n_{\text{seg}} = \alpha$ the segment length is pinned to d/α and stops being a parameter at all. First, ζ decides only *which* α positions a key occupies. It leaves d , the Hamming weight α , and therefore the decode vector's shape untouched, so the encoded table is the same size and the same density whatever ζ is: neither the homomorphic evaluation of Section 3 nor its noise sees ζ at all. Second, the one quantity ζ *does* move is the per-bin peel-failure probability θ of α -KVS.Encode, which falls sharply as segments grow, because long segments spread each key's three positions over a wider window and make the underlying hypergraph easier to peel. Together these make ζ the right instrument for the encoding budget (C4) of Appendix C.2: it buys a lower θ , and hence a smaller expected number of seed draws, at no cost in online time, response size, or noise. Prior work using this filter fixes ζ by the standard fuse-filter sizing law [55]; we treat it instead as a parameter to be chosen against (C4), and Section 4 reports the value we deploy.

A related caution concerns η . The load factor of the sizing law is not a constant but a slowly decreasing function of the list length, approaching 1.125 only asymptotically, which is exactly what the law $\eta_3(\cdot)$ of Appendix C.2 expresses. A filter sized at precisely its nominal capacity therefore sits on the peel threshold rather than safely below it, and larger filters sit closer to it than small ones do. An encoder must be given

headroom by enlarging ς , or by allowing slack in d ; and the amount is a property of the list length, not a single constant that can be calibrated once and reused.

Instantiation II: the polynomial KVS. Fix $\alpha = d$ and identify $\mathcal{K}$ with a subset of $\mathbb{Z}_p$.

- α -KVS.Setup(1^λ): Output $\text{pp}_{\alpha\text{-KVS}} := (\perp, \vec{w}(\cdot))$ with empty hash family $H = \perp$ and the decode-coefficient mapping defined as the Vandermonde row:

$$\vec{w}(k) := (1, k, k^2, \dots, k^{d-1}) \in \mathbb{Z}_p^d. \quad (8)$$

- α -KVS.Encode($\text{pp}_{\alpha\text{-KVS}}, L$): Solve the Vandermonde system $\{\langle \vec{w}(k), \mathbf{D} \rangle = v\}_{(k,v) \in L}$ for $\mathbf{D} \in \mathbb{Z}_p^d$. This determines the unique interpolating polynomial $P(X) = \sum_{i \in [d]} \mathbf{D}[i] X^i$ of degree strictly less than $|L|$ such that $P(k) = v$ for all $(k, v) \in L$. Pad the remaining $d - |L|$ entries of $\mathbf{D}$ with zeros, and output $\mathbf{D}$. (This always succeeds).
- α -KVS.Decode($\text{pp}_{\alpha\text{-KVS}}, \mathbf{D}, k$): $\sum_{i \in [d]} k^i \mathbf{D}[i] \bmod p$.

C Formal Derivations of the Offline Precomputing Optimizations

In Section 3.5, we introduced a series of offline/online precomputing techniques that drastically reduce the online latency of our keyword PIR framework. In this appendix, we provide the formal algebraic derivations and detailed step-by-step formulations for these optimizations.

Precomputing 1: Static BSGS Components. Section 3.4 introduced the rotated coefficients $\tilde{c}_{\sigma_g, \sigma_b} := \sigma_g^{-1}(c_{\sigma_g \circ \sigma_b})$ consumed by the giant step of Algorithm 3. For our Dimension-1 map f , the underlying family is instantiated by Equation (3) as $c_\sigma := \hat{u}_\sigma$, where $\hat{u}_i = \sum_{t \in [d]} X^{-it} u_t$ is determined by the column polynomials $u_t = \text{coef}^{-1}(\mathbf{M}[\cdot, t])$ alone. Since $\{c_\sigma\}$ is a function of $\mathbf{M}$ and nothing else, the entire rotated family $\{\tilde{c}_{\sigma_g, \sigma_b}\}_{(\sigma_g, \sigma_b) \in \Sigma_{\text{giant}} \times \Sigma_{\text{baby}}}$, is fixed at Server.Setup. This is Line 2 of Algorithm 4.

Precomputing 2: Dimension 1. BSGSEval consists of plaintext-ciphertext multiplications and automorphisms, both of which are $\mathbb{Z}_p$ -linear in the input ciphertext. Writing $\mathbf{q}_1 = (b_1, a_1) = (b_1, 0) + (0, a_1)$ and applying linearity,

$$\begin{aligned} c^* &= \underbrace{\text{BSGSEval}((0, a_1), \{c_\sigma\}, \text{evk}_g, \text{evk}_{g^{M_2}}, \text{evk}_{-1})}_{\text{key-independent}} \\ &+ \underbrace{\text{BSGSEval}((b_1, 0), \{c_\sigma\}, \text{evk}_g, \text{evk}_{g^{M_2}}, \text{evk}_{-1})}_{\text{key-dependent}}. \end{aligned}$$

The key-independent branch depends only on a_1 , $\{c_\sigma\}$ and the evaluation keys, and crucially it absorbs every key-switching operation. The server can evaluate that during the

per-query offline interval and caches $(c_{0,\text{off}}^*, c_1^*) \in \mathcal{R}_q^2$. The key-dependent branch then reduces to pure ring arithmetic:

$$\begin{aligned} \text{BSGSEval}((b_1, 0), \{c_\sigma\}, \text{evk}_g, \text{evk}_{g^{M_2}}, \text{evk}_{-1}) &= (c_{0,\text{on}}^*, 0), \\ c_{0,\text{on}}^* &:= m^{-1} \sum_{\sigma_g \in \Sigma_{\text{giant}}} \sigma_g \left(\sum_{\sigma_b \in \Sigma_{\text{baby}}} \tilde{c}_{\sigma_g, \sigma_b} \cdot \sigma_b(b_1) \right), \end{aligned}$$

and the server recombinces $c^* = (c_{0,\text{off}}^* + c_{0,\text{on}}^*, c_1^*)$. Since the Dimension-1 fold dominates the online phase at $\Theta(m^2)$ ring operations and contributes all $O(\sqrt{m})$ key-switching operations, removing the latter is where the practical value of this optimization lies. This is Lines 4 and 6 of Algorithm 4.

Precomputing 3: Dimension 2. Applying the offline/online split to Dimension 2 requires splitting the gadget decomposition of the RLWE operand $c_0^* = c_{0,\text{off}}^* + c_{0,\text{on}}^*$. Although G^{-1} is non-linear, the external product in Equation (5) requires only a bounded representation satisfying the reconstruction, not canonical digits. By defining

$$\vec{d}^{\text{off}} := G^{-1}(c_{0,\text{off}}^*), \quad \vec{d}^{\text{on}} := G^{-1}(c_{0,\text{on}}^*),$$

the sum $\vec{d}^{\text{off}} + \vec{d}^{\text{on}}$ forms a valid non-canonical decomposition of c_0^* with digit norm bounded by B . Substituting this into Equation (5) yields $T = T^{\text{pre}} + T^{\text{on}}$, grouping the terms by whether their digits can be precomputed offline:

$$T^{\text{pre}} := \sum_{i \in [\ell_B]} d_i^{\text{off}} \cdot q_2[i, 0] + \sum_{i \in [\ell_B]} G^{-1}(c_1^*)_i \cdot q_2[i, 1], \quad (9)$$

$$T^{\text{on}} := \sum_{i \in [\ell_B]} d_i^{\text{on}} \cdot q_2[i, 0]. \quad (10)$$

The correctness cost of the split is a doubled digit bound from $B/2$ to B , which inflates the Dimension-2 gadget noise by at most a factor of two and is a negligible constant subsumed by the Dimension-1 noise floor. By migrating the gadget decompositions offline, it halves the online decomposition count (from $2\ell_B$ to ℓ_B) at the cost of increasing online products (from $2\ell_B$ to $3\ell_B$). Since decompositions is more expensive than multiplications in this context, this trade yields a saving. This structurally confirms that our offline/online splitting technique generalizes to every layer of the evaluation. In summary, while Precomputing 3 optimizes the external product, this layer remains a $\Theta(\ell_B m)$ lower-order term, which results in a minor speedup in the overall term as the major complexity is dominated by the Dimension-1 fold, i.e., $\Theta(m^2)$.

C.1 Noise Analysis

We analyze the noise accumulated during the homomorphic evaluation (Server.Answer) to establish the correctness condition for the ciphertext modulus q . Following standard heuristic analysis, errors are modeled as independent subgaussians whose variances add linearly. Throughout, σ denotes the standard deviation of the error distribution χ .

Theorem 10 (Homomorphic Evaluation Noise) *Let p be the plaintext modulus, m the ring dimension, and (B, ℓ_B) the gadget decomposition parameters. Assuming a balanced BSGS split $M_2 = 2M_1 = \Theta(\sqrt{m})$, the total noise accumulated after the two-dimensional homomorphic evaluation is asymptotically bounded by*

$$E_{\text{out}} = \Theta(pm^{5/4} E_0),$$

where $E_0 = \Theta(B\sqrt{\ell_B m} \sigma)$ is the base key-switching noise. Consequently, to hold the decryption failure probability below $2^{-\lambda}$, the ciphertext modulus q must satisfy:

$$q = \Omega(p^2 m^{7/4} B \sqrt{\ell_B} \sigma \sqrt{\lambda}). \quad (11)$$

It is worth noting that the bound is governed entirely by the baby chain because the giant automorphisms of Algorithm 3 are applied after the multiplication by the $\tilde{c}$ entries, they contribute only $\Theta(\sqrt{M_1} E_0)$. The proof is given below.

Proof The proof analyzes the noise propagation sequentially through the base key-switching operations, the Dimension-1 fold, and the Dimension-2 external product.

Base Key-Switching Noise. A key-switching operation computes the inner product between ℓ_B gadget digits (bounded by $B/2$) and the evaluation key errors across m coefficients. This amplifies the base error by an expansion factor $\rho := B\sqrt{\ell_B m}/12$. Thus, a single key switch with a fresh key contributes noise $E_0 := \rho \sigma$. Two fundamental properties govern the subsequent analysis: ρ represents the asymptotic cost of evaluating an error embedded within a switching key, and multiplying a ciphertext by a plaintext scalar in $\mathbb{Z}_p$ amplifies the ciphertext error by a factor of $\Theta(p\sqrt{m})$.

Dimension 1 (BSGS Fold). The noise in the BSGS evaluation (Algorithm 3) comprises two components, distinguished by their interaction with the plaintext scalar multiplication:

- *Baby steps:* The Horner chain (Line 3) reuses the *single* uploaded evk_g at every step, accumulating $\Theta(\sqrt{k} E_0)$ noise in the k -th baby ciphertext.

$$\begin{aligned} E_{\text{baby}} &= \Theta(p\sqrt{m} \cdot E_0 \cdot \sqrt{|\Sigma_{\text{giant}}|} \cdot (\sum_{k \in [M_2]} k)^{1/2}) \\ &= \Theta(p\sqrt{m} \sqrt{M_1} M_2 E_0). \end{aligned}$$

Note that chaining automorphisms reduces the evaluation key upload bandwidth by $\Theta(\sqrt{m})$, at the acceptable cost of inflating the noise by a factor of $\Theta(\sqrt{M_2}) = \Theta(m^{1/4})$: uploading a fresh key per baby automorphism would replace $(\sum_{k \in [M_2]} k)^{1/2}$ by $\sqrt{M_2}$, giving $\Theta(pm E_0)$ instead of $\Theta(pm^{5/4} E_0)$.

- *Giant steps:* The giant automorphisms occur after the plaintext multiplication, thus avoiding the p -amplification. Every one of them switches under the same uploaded $\text{evk}_{g_{M_2}}$, so it contributes a $\Theta(E_0)$ key-switching term and merely permutes whatever noise already sits in its operand. The chain

of Line 8 therefore accumulates additively: summing the $\Theta(M_1)$ independent contributions in variance gives

$$E_{\text{giant}} = \Theta(\sqrt{M_1} E_0),$$

and the single coset-joining Auto.Eval_{-1} adds one further $\Theta(E_0)$. Note that this is the term the three-key design of Section 3.4 improves: had the giant family been derived instead, a key obtained by j compositions would carry $\Theta(\sqrt{j} E_0)$ noise and applying it would amplify that by ρ , giving $\Theta(\rho E_0 M_1)$ in place of $\Theta(\sqrt{M_1} E_0)$.

Balancing $|\Sigma_{\text{baby}}| = M_2$ against $|\Sigma_{\text{giant}}| = 2M_1$, i.e. $M_2 = 2M_1 = \Theta(\sqrt{m})$, the total output noise of Dimension 1 is $E^* = \Theta(E_{\text{baby}} + E_{\text{giant}}) = \Theta(E_0(pm^{5/4} + m^{1/4})) = \Theta(pm^{5/4} E_0)$, the giant term being dominated for every $p \geq 2$. The evaluation's noise ceiling is therefore set by the p -amplified baby chain alone, and no assumption on the relative size of p and ρ is needed.

Both chains reuse a single uploaded evaluation key rather than a fresh key per step, so the per-step key-switching errors are distinct gadget-digit combinations of one fixed key error rather than independent samples. Following the standard convention in noise analysis we treat them as independent and add them in variance; the bounds above are to be read under that heuristic, and without it each $\sqrt{\cdot}$ degrades to a linear factor.

Dimension 2 (External Product). The external product introduces noise $e_{\square} = \mu e^* + \langle G^{-1}(c^*), \vec{e} \rangle$, where $\vec{e}$ is the RGSW encryption noise. Because the selector $\mu = X^{-\ell_k^* L}$ is a monomial ($\|\mu\| = 1$), the propagated noise remains $\Theta(E^*)$. The additive gadget noise is $\Theta(E_0)$: the sum $\langle G^{-1}(c^*), \vec{e} \rangle$ ranges over $\Theta(\ell_B)$ digits of variance $\Theta(B^2)$ across m coefficients, whose standard deviation is exactly $B\sqrt{\ell_B m}/12 \sigma = \rho \sigma = E_0$. Consequently, the final output noise is $E_{\text{out}} = \Theta(E^* + E_0) = \Theta(E^*)$, since $E^*/E_0 = \Theta(pm^{5/4}) \gg 1$. Therefore, Dimension 2 adds negligible noise asymptotically; it requires no modulus rescaling and consumes no multiplicative depth. (Even if Precomputing 3 doubles the digit bound, E_{out} remains strictly dominated by Dimension 1).

Correctness Condition on q . Decryption succeeds if the accumulated noise is bounded by $\Delta/2 \approx q/(2p)$. To ensure a decryption failure probability of $2^{-\lambda}$, we bound the subgaussian noise by $\Theta(\sqrt{\lambda})$ standard deviations, which requires $q = \Omega(p \cdot \sqrt{\lambda} \cdot E_{\text{out}})$. Substituting E_{out} and E_0 directly yields the stated condition on q . Furthermore, because E_{out} is completely independent of the database size n and the block count N_{blk} , partitioning the database scales the system with strictly zero noise penalty.

Architectural Trade-off. Taking logarithms of Theorem 10, we have $\log q$ equals:

$$\Omega\left(2 \log p + \frac{7}{4} \log m + \log B + \frac{1}{2} \log \ell_B + \log \sigma + \frac{1}{2} \log \lambda\right).$$

This formulation exposes a fundamental architectural tension: increasing the plaintext modulus p reduces the required plane

count L (thereby improving online throughput), but exacts a quadratic cost on q . Since RLWE hardness at a fixed security level requires $m = \Omega(\log q)$, this quadratic growth in q will inevitably force a larger ring dimension m .

C.2 Parameter Selection

Setting. Recall the quantities fixed by Section 3.1. The database of n pairs is cut into N_{blk} blocks of $n_t := n/N_{\text{blk}}$ pairs each, a block spreads its pairs over r bins, and every bin is encoded into one α -KVS of length d . Let $\Lambda := n_t/r$ denote the mean bin load and $\Lambda^{\max}(\kappa)$ the max bin load of Section 2.3 at statistical parameter κ , let $\eta := d/\Lambda^{\max} \geq 1$ denote the expansion rate the α -KVS requires, and let θ denote the probability that α -KVS.Encode fails to peel a single bin. The homomorphic back-end fixes (m, p, q, B) and, through them, the plane count L . What is left for us to choose is the bin count r , the α -KVS sparsity α , and its segment length ς .

Constraints. Four conditions bound the load a block may carry. Two are combinatorial, one is a decryption condition, and one is a budget on the encoder:

- **(C1)** $rL \leq m$: every bin–plane pair must own a distinct ring coefficient, so that the Dimension-2 shift isolates a bin without wrap-around.
- **(C2)** $d = \eta\Lambda^{\max}(\kappa) \leq m$: one bin’s encoding must fit inside a single query ciphertext. In the deployment d and η are fixed by the ring, so this reads in the reverse direction: it pins $\Lambda^{\max}(\kappa) = \lfloor d/\eta \rfloor$ and thereby caps the load a block may carry at the budget κ .
- **(C3)** $\Lambda \leq \Lambda_{\text{noise}}(L, B, p)$: the Dimension-1 fold over a block that full must still decrypt. A fuller bin leaves more nonzero positions for the fold to accumulate, so the noise of Theorem 10 grows with the load; this is not implied by (C1)–(C2), and Λ_{noise} has to be measured.
- **(C4)** $\mathbb{E}[\#\text{seeds}] = \exp(N_{\text{blk}}r \cdot \theta)$ must stay within a small budget: α -KVS.Encode redraws the shared seed until every bin of the deployment peels.

Consequences. Three conclusions follow; the derivations follow in Appendix C.3.

(i) *The deployable load is a minimum of two ceilings.* Writing Λ_{comb} for the largest mean load admitted by (C1)–(C2),

$$\Lambda_{\text{dep}} = \min\{\Lambda_{\text{comb}}, \Lambda_{\text{noise}}(L, B, p)\}. \quad (12)$$

Which of the two binds is not fixed in advance: at small L the combinatorial ceiling is the tighter and the noise ceiling is slack, while as L grows the two converge. A parameter set must therefore be verified, not merely computed.

(ii) *A block never reaches its reserved capacity.* Provisioning $\Lambda^{\max}$ positions in each of r bins reserves room for $\bar{n}_t := rd/\eta = r\Lambda^{\max}$ pairs, but the reservation must cover the

fullest bin while the keys only fill the average one. The admissible capacity is smaller by exactly that balls-into-bins margin,

$$n_t^{\max} = \bar{n}_t \left(1 - \Theta\left(\sqrt{\frac{\eta(\kappa + \log N_{\text{blk}}r)}{d}}\right)\right), \quad (13)$$

and the deployment uses $N_{\text{blk}} = \lceil n/n_t^{\max} \rceil$ blocks. The discount is second order in d , and since N_{blk} is a ceiling it costs nothing unless n lands within it of a block boundary.

(iii) (C3) and (C4) pull in opposite directions. The deployment holds $R = N_{\text{blk}}r$ bins, which is n/Λ when every block is filled to capacity and strictly more otherwise, since $N_{\text{blk}} = \lceil n/n_t^{\max} \rceil$ is a ceiling; either way the count falls as the mean load grows, so (C4) is eased by making Λ large while (C3) demands it be small. At large n the two cannot both be satisfied by moving Λ , and the budget must be bought elsewhere.

Our Strategy. We take r as small as (C1) permits, since fewer and fuller bins shrink the relative margin in Equation (13); we fix $\alpha := 3$, for which η approaches 1.125 from above; and we use ς to satisfy (C4). The last is the point of consequence (iii): the segment length decides only which α positions of $[d]$ a key occupies, so it leaves d , L , r and the noise untouched and moves θ alone. Enlarging it is thus the correct response to a violated (C4), and lowering Λ is not. Section 4 reports the deployed values.

C.3 Parameter Selection: Derivations

This subsection supplies the arguments behind the four constraints and three consequences of Appendix C.2, and uses its notation throughout. Let $R := N_{\text{blk}}r$ denote the total number of bins in the deployment. Since $R = N_{\text{blk}}r = n_t N_{\text{blk}}/\Lambda = n/\Lambda$, the bin count of a deployment is determined by the mean load alone and not by how that load is divided into blocks; this identity is what drives consequence (iii).

The Combinatorial Ceiling. (C1) and (C2) bound the space one block may reserve by

$$C := r\Lambda^{\max} \leq m^2/(\eta L),$$

obtained by substituting $r \leq m/L$ from (C1) into $\Lambda^{\max} \leq m/\eta$ from (C2). Both are statements about the α -KVS geometry: they say how many keys the encoding can be made to hold, and nothing about whether a fold over a table that full still decrypts, nor about what the encoder pays to build it. We write Λ_{comb} for the largest mean load these two admit.

Sizing d for a Target Failure Probability. The provisioned length $d = \eta\Lambda^{\max}$ must absorb the max load, not the mean, and the probability that *some* bin of the deployment overflows it must stay below $2^{-\kappa}$. Our regime is heavily loaded, $\Lambda \gg 1$.

Since every key lands in the bin named by its own hash, the bin loads follow a multinomial distribution, and a Chernoff bound together with a union bound over the R bins places the max load at $\Lambda^{\max}(\kappa) = \Lambda + \Theta(\sqrt{\Lambda(\kappa + \log R)})$. Hence the minimum provisioning is

$$d = \eta \Lambda \left(1 + \Theta \left(\sqrt{\frac{\kappa + \log R}{\Lambda}} \right) \right). \quad (14)$$

The margin the tail demands above the mean is a $\Theta(\Lambda^{-1/2})$ fraction of d , so it is the statistical tail, and not the mean load itself, that first collides with the ceiling $d \leq m$ of (C2). Because that fraction shrinks as Λ grows, bins should be as few and as full as (C1) allows: at fixed n_i , halving r doubles Λ and cuts the relative margin by $\sqrt{2}$. This is the rule quoted in Appendix C.2.

The Expansion Rate. We minimize η in order to maximize C while keeping the *peeling* failure probability θ of (C4) small; this is a different event from the bin overflow of $\Lambda^{\max}(\kappa)$ and is repaired by a different resample. For the $\alpha = 3$ sparse α -KVS of Appendix B.1, the sizing law requires $\eta_3(\Lambda^{\max}) = \max\{1.125, 0.875 + 0.25 \frac{\ln(10^6)}{\ln \Lambda^{\max}}\}$ [55], so the optimal 1.125 floor is reached only at max loads beyond 10^6 . Constraint (C2) forbids that at our ring dimension: with $m = 4096$ the largest sustainable $\Lambda^{\max}$ is three orders of magnitude short of the floor, and the best attainable rate we measure is $\eta \approx 1.31$. We therefore drive $\Lambda^{\max}$ as close to the (C2) limit as the other constraints permit, which is the most efficient point a single ring admits. Conceding the asymptotic 1.125 in exchange for a strictly single-ciphertext query is what preserves the framework’s advantage in total communication.

Deployable Block Capacity. Inverting Equation (14) at fixed (m, d) gives the admissible capacity. Reserving $\Lambda^{\max}$ positions in each of r bins sets aside room for $\bar{n}_i := rd/\eta = r\Lambda^{\max}$ pairs, the capacity a perfectly balanced placement would attain. A real placement cannot reach it, because the reservation must cover the fullest bin, $\Lambda^{\max}$, while the keys only fill the average one, Λ . Discounting $\bar{n}_i$ by that margin, which is exactly what buys the $2^{-\kappa}$ overflow probability, yields Equation (13), so a block realizes a $1 - \Theta(d^{-1/2})$ fraction of its reserved capacity and the deployment needs $N_{\text{blk}} = \lceil n/n_i^{\max} \rceil$ blocks.

The Deployed Budget κ . Instantiating (C2) at $d = m = 4096$ and the measured $\eta = 1.31$ fixes $\Lambda^{\max}(\kappa) = 3129$. What κ this buys depends on how full a block actually is: the realized mean load is $n/(N_{\text{blk}}r)$, which the ceiling $N_{\text{blk}} = \lceil n/n_i^{\max} \rceil$ leaves between 45% and 99% of Λ across our grid. A multiplicative Chernoff bound with a union over the $R = N_{\text{blk}}r$ bins gives $\kappa \geq 16.8$ in the tightest configuration (2^{18} with 256-byte payloads, where 99.4% occupancy meets only 93 bins), $\kappa \geq 22.6$ at 2^{22} with 256-byte payloads, and

$\kappa \geq 37$ in the remaining ten cells. The expected number of re-encodes is therefore $1 + 8.7 \cdot 10^{-6}$, and an overflow costs one pp_{BBH} resample at setup, before any client has interacted with the server.

We report κ rather than fold it into λ because the two guard different things. A client never observes a bin overflow: EncodeDB detects it, resamples and restarts, so κ budgets server work and not privacy, and it does not have to match the 128-bit computational parameter. A deployment that prefers $\kappa = \lambda$ can have it by raising N_{blk} : doing so across the whole grid costs 15.4% more blocks and hence 15.4% more online time, and four of our twelve cells already satisfy it unchanged.

(C3): the Noise Ceiling. A heavier bin leaves more nonzero filter positions for the Dimension-1 fold to accumulate, so the output noise of Theorem 10 grows with the load and the response modulus after modulus switching must be chosen to clear it. Two consequences are worth recording. First, that modulus must be *measured per configuration* rather than fixed: a constant calibrated at one bin load rejects loads the scheme handles perfectly well, and it conceals the fact that the response size grows with the load. Second, the gadget base is the effective lever on Λ_{noise} , since a finer gadget lowers E_0 in Theorem 10 at a cost in evaluation-key size and offline time but none in online time.

(C4): the Encoding Budget and the Role of ς . A single shared seed is drawn for the whole deployment and α -KVS.Encode succeeds only if every one of the R bins peels, so the expected number of draws is

$$\mathbb{E}[\#\text{seeds}] = \exp(R\theta), \quad R = N_{\text{blk}}r. \quad (15)$$

Bounding this by a budget t requires $\Lambda \geq n\theta/\ln t$, a lower bound on the load, whereas (C3) imposes an upper one; for large n the two windows can fail to intersect, and no choice of Λ satisfies both. The escape is that θ is itself free. By Appendix B.1 the segment length ς decides only which α of the d positions a key occupies: it leaves d, L, r, m and B untouched, so the table the fold walks is identical and neither the online time nor the noise sees it, while θ falls sharply as ς grows. Enlarging ς is therefore the correct response to a violated (C4).

The Deployed Load. Combining the above gives Equation (12), with ς then chosen as small as (C4) allows at that load. The reading to take from it is that the ring dimension alone does not determine the deployable load: at small L the combinatorial ceiling binds and the noise ceiling is slack, while as L grows the two converge and the margin becomes thin, so a parameter set must be verified on the instance it will run on rather than computed once.

D Deferred Material for the Evaluation

D.1 Per-Cell Parameters and Verification

The Dimension-2 Selector. The RGSW selector is instantiated in its approximate form. Its gadget is $\{PP, PPB_{\text{sel}}, \dots, PPB_{\text{sel}}^{\ell_{\text{sel}}-1}\}$ and must span the ciphertext modulus, so the design rule at $\ell_{\text{sel}} = 2$ is $PP \cdot B_{\text{sel}}^{\ell_{\text{sel}}} \approx q$; the two implementations therefore instantiate it differently because their moduli differ, at $(PP, B_{\text{sel}}) = (2^{26}, 2^{17})$ for the 60-bit Java prime and $(2^{20}, 2^{18})$ for the ≈ 56 -bit C++ one. What both share is $\ell_{\text{sel}} = 2$, and it is the digit count alone that fixes the online query: a selector of ℓ_{sel} digits occupies $4\ell_{\text{sel}}$ ring elements, so the eight elements reported in Section 4 follow from $\ell_{\text{sel}} = 2$, where an exact selector would need five digits and a correspondingly larger query.

Timing the Hint-Based Baseline. KVS^{index} is run unmodified. Its online column in Table 3 is the library’s own *Server responses query* stopwatch, read out through a log appender without patching any of its source, so that it measures the same quantity as the other three rows: server-side computation, with client query generation and recovery excluded. Timing it instead by the client-side wall clock around the library’s query call would charge it that library’s client overhead, which inflates it by 0.7–20%, mean 4.9%, concentrated at the small-payload cells where it is most competitive against us; at 2^{18} with 32-byte payloads the wall clock reads 18 ms against 15 ms of server compute. We report the smaller, like-for-like figure.

The Response Modulus is Measured, not Derived. The response modulus of Table 4 is not fixed in advance: for each cell we take the smallest width that still recovers and add three bits of margin, which lands on 25, 26 or 27 bits. Because the underlying implementation draws its secret from an unseeded generator, re-running a cell can move both the recovery threshold and, with it, the reported response size by one bit. The published α -KVS seed therefore fixes the filter geometry and the block structure, but not the response bytes; those are a property of the run.

Table 4 lists the parameters of our construction and the outcome of the recovery gate, one row per database configuration; the two baselines do not share our front-end parameters and are omitted, which is why the table has 12 rows where the gate of Section 4 covers 36 cells. The two implementations of ours agree on every scheme parameter in the table, but their libraries fix different ciphertext moduli: a 60-bit working prime under SEAL against the ≈ 56 -bit modulus of $kSPiR$. And each parameter sized to that modulus differs accordingly: the key-switch gadget is base 2^3 with 20 digits in Java against base 2 with 56 in C++, which is the whole of the evaluation-key gap in Table 3, and the selector gadget follows the per-modulus

Table 4: Per-cell parameters of *our* construction, one row per database configuration. Both implementations select the same *scheme* parameters $(L, r, \Lambda, N_{\text{blk}}, \zeta)$ but not the same *cryptographic* ones, and each measures its own response width as the smallest that still recovers plus three bits of margin; a one-bit disagreement between the two therefore reflects a different key draw, not a different configuration.

n / ℓ_v	L	r	Λ	N_{blk}	$\mathbb{E}[\#s]$	resp. bits	gate
$2^{18}/32\text{ B}$	20	204	2835	1	1.6	25/26	24/24
$2^{18}/64\text{ B}$	36	113	2835	1	1.3	25/26	24/24
$2^{18}/128\text{ B}$	68	60	2835	2	1.3	26/26	24/24
$2^{18}/256\text{ B}$	132	31	2835	3	1.2	26/26	24/24
$2^{20}/32\text{ B}$	20	204	2811	2	2.5	25/25	24/24
$2^{20}/64\text{ B}$	36	113	2811	4	2.7	27/25	24/24
$2^{20}/128\text{ B}$	68	60	2811	7	2.5	26/26	24/24
$2^{20}/256\text{ B}$	132	31	2811	13	2.4	26/26	24/24
$2^{22}/32\text{ B}$	20	204	2790	8	36.2	25/26	24/24
$2^{22}/64\text{ B}$	36	113	2790	14	32.5	26/25	24/24
$2^{22}/128\text{ B}$	68	60	2790	26	30.9	27/26	24/24
$2^{22}/256\text{ B}$	132	31	2790	49	28.3	26/26	24/24

rule stated above. Both nonetheless upload the same three keys and send the same $\ell_{\text{sel}} = 2$ selector digits. Λ is the mean bin load a block carries *at its admissible capacity* n_t^{max} of Equation (13), and $\mathbb{E}[\#seeds]$ is Equation (15) evaluated at the deployed bin count $R = N_{\text{blk}}r$. Since N_{blk} is a ceiling, the last block is in general only partly filled, so the realized mean load of the deployment is at most Λ and is materially below it when n sits just past a block boundary.

Explanation of Columns in Table 4. L , payload planes per value; r , bins per block; Λ , mean bin load at the admissible block capacity n_t^{max} of Equation (13); N_{blk} , blocks; $\mathbb{E}[\#s]$, expected number of α -KVS seeds before every bin peels, from Equation (15); *resp. bits*, the width the server rescales the response to, given as C++/Java; *gate*, passes out of attempts in a 24-keyword exact-bytes recovery test on the reported instance, which is a functional check on that instance and not a bound on the failure probability.

D.2 Comparison in Multiple Dimensions

Figure 4 replots Table 3 on the four axes that matter to a deployment. The orderings are stable across the whole grid. On online time and throughput, Ours (C++) is first in all twelve cells, Ours (Java) is ahead of 3-KsPIRV in all twelve, and KVS^{index} falls between Ours (C++) and 3-KsPIRV, ahead of Ours (Java) in ten cells of the twelve. On setup, KVS^{index} cannot be compared because it is benchmarked in a single-threaded configuration to keep its implementation unmodified. Among the other three schemes, Ours (C++) is the cheapest in eleven of twelve. On per-query communication the ordering reverses: KVS^{index} is the smallest in every cell at 27–272 KiB,

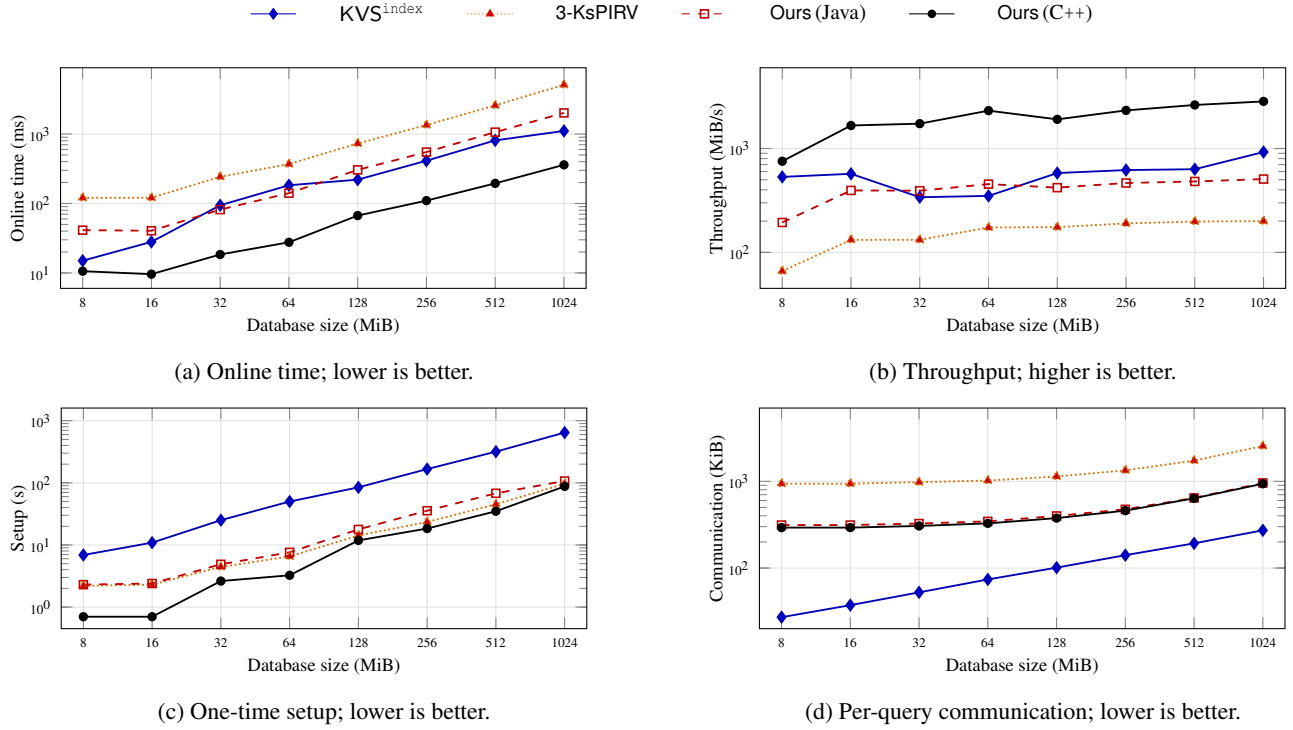


Figure 4: The twelve cells of Table 3 on four axes, against total database size $n \cdot \ell_v$; both axes are logarithmic. Four of the eight sizes carry two *different* (n, ℓ_v) configurations rather than two runs of one: at those sizes the marker is the geometric mean of the two cells. It is negligible for the three homomorphic schemes and widest for KVS_{index} , whose cost depends on the matrix shape — 135 against 248 ms at 64 MiB. Panel d is the per-query communication, query plus response; the one-time per-client transfer is excluded and is discussed below.

our two implementations sit together at 293–958 KiB, and 3-KsPIRV is $2.7\text{--}3.0\times$ ours. Panel d excludes the one-time per-client transfer, which is where that ordering is paid for: KVS^{index} ships a hint of 14.5–143.6 MB that grows with the database, against our evaluation keys of 3.9 MB in Java and 9.2 MB in C++, flat in n . The reasons behind each of these orderings are as analysed in Section 4.2 and are not repeated here.