Practical Keyword SPIR with Lightweight Clients: Hintless and Efficient

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 without revealing the queried keyword, generalizing traditional index PIR. This work advances the state of the art of Hao et al. (USENIX Security 2025) by addressing several key limitations of their constructions, most notably the efficiency bottleneck of their generic framework and the over-revealing nature of their more efficient designs. Although using oblivious PRFs (OPRFs) can solve the over-revealing issue, doing so introduces additional trade-offs that may hinder practical deployment.**

We present a practical, hintless keyword PIR framework that natively eliminates over-revelation without relying on OPRFs, achieving symmetric PIR security. Building on KsPIR (CCS 2024) and the linearly homomorphic technique of Peikert and Pepin (TCC 2025), our server homomorphically evaluates key-value store decoding over encrypted queries, retrieving exactly the target record. To accelerate evaluation, we introduce a generalized baby-step giant-step (BSGS) algorithm for coefficient-encoded Galois sums, an offline/online decomposition, and a Horner-style evaluation chain that reduces uploaded evaluation keys to three. For databases up to 2^{22} entries, our construction achieves a mean online speedup of $2.65\times$ over their generic framework under the same index-PIR backend. Compared to their specialized design, our scheme attains comparable online time while requiring neither a client-side hint (saving 14.5–143.6 MB) nor an OPRF.

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], [3], [4], [5], [6], [7], [8], [9]. In many practical applications [10], [11], 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* [12], whereas the traditional PIR setting, in which records are identified by their indices, is referred to as *index PIR*.

Building on a line of advances [4], [12], [13], [14], recent works [15], [16] construct keyword PIR generically from an index PIR used as a black box together with the key-value store (KVS) technique, at the cost of three index-PIR invocations. To avoid this overhead, the state-of-the-art work [16] also gives two specialized constructions ($\mathsf{KPIR}^{\mathtt{hash}}$ and $\mathsf{KPIR}^{\mathtt{index}}$) that integrate SimplePIR [7] with customized data structures and are substantially faster.

Despite these advances, the two specialized constructions of [16] still have two limitations. First, inheriting a drawback of SimplePIR [7], they require a large client-side hint for each database instance, ranging from 14.5 to 143.6 MB for the parameters in our experiments. Second, they retrieve an entire row of the encoded database, allowing the client to learn additional key-value pairs beyond those corresponding to its queries. Consequently, they do not achieve the stronger and more desirable notion of symmetric PIR (SPIR) [17], which requires the server to reveal only the values associated with the queried keys and nothing else.

A generic approach to obtaining SPIR from their two specialized constructions is to incorporate an oblivious PRF (OPRF) [18]. However, existing OPRF constructions present an undesirable trade-off. Notably, a long-term OPRF key masks database entries deterministically, causing updates to reveal relationships between old and new values unless the entire database is re-masked. Using fresh OPRF keys for each session avoids this issue but requires re-masking the database for every client. Furthermore, classical schemes such as 2HashDH [19], [20] are not post-quantum secure, whereas post-quantum alternatives are recent and typically rely on less standard assumptions such as power residues [21] or alternating moduli [22]. These drawbacks introduce substantial overheads that significantly hinder practical deployment.

Our Goal. Motivated by the above, we aim to construct a practical keyword PIR framework that simultaneously achieves (1) exact retrieval that achieves SPIR against semi-honest clients without an additional protocol layer (e.g., OPRF), (2) no client-side hint which is suitable for lightweight clients, (3) post-quantum security, and (4) efficiency surpassing the generic black-box construction based on three invocations of the same (hintless) index PIR, and comparable to the hint-based constructions in [16].

1.1. Our Contributions

In this work, we present a post-quantum, hintless keyword SPIR framework that achieves all four goals. It targets lightweight clients, which cannot keep database-dependent state and benefit from small queries and responses; we therefore build on KsPIR [23], whose client-side costs are the smallest among hintless index PIR (Appendix E), even though our framework works with any linearly homomorphic index PIR. Our contributions are as follows.

- **Post-quantum hintless keyword SPIR.** We extend KsPIR to the keyword setting by combining simple hashing with an α-KVS [24], [25], so that one encrypted query decodes every bin at once and one RGSW ciphertext selects the target bin; the cost is roughly that of a single KsPIR invocation rather than three. We prove that the construction is a SPIR against semi-honest clients without an OPRF; re-encoding at the server once a client exhausts its query budget, which also handles updates, removes the bound on the number of queries.
- **Efficient homomorphic evaluation.** Building on the linear-map technique of [26], which only requires the plaintext modulus to be coprime to the ring dimension, we give a generalized BSGS algorithm that reduces the number of automorphisms from $O(m)$ to $O(\sqrt{m})$, a Horner-style chain that reduces the evaluation keys from $O(\sqrt{m})$ to three, and an offline/online split following KsPIR.
- **Implementation and evaluation.** We implement the framework in Java, on the `mpc4j` library [27] used by [16], and in C++ with the engineering optimizations of KsPIR, AVX and multi-threading. For up to 2^{22} entries with 256-byte payloads, our front-end is 2.24–3.06× faster than the generic framework of [16] on the same back-end. Compared with $\mathsf{KPIR}^{\mathtt{index}}$, their construction with the shortest online time, our Java implementation achieves comparable online latency on the same library and our C++ implementation is 1.40–7.47× faster, without any client hint (saving up to 143.6 MB). The remaining gap in Java stems from the back-end: our front-end could equally run on the hint-based SimplePIR, and we use KsPIR to stay hintless. The query is 280–300 KB regardless of the database size.

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, $\mathsf{DB} = \{(k_z, v_z)\}_{z\in[n]}$, the generic framework [15], [16] 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 [16] (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 [6], SimplePIR [7], and KsPIR [23], 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, [16] 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 [7] designs, this approach achieves significantly better communication and response time than the generic construction. However, both inherit the client-side hint of SimplePIR and return an entire bin or row, not only the requested value.

Our New Design Framework. Our starting point is the generic α-KVS-based construction [15], [16]. A natural way to avoid multiple client queries is to compress the α queried locations into a single query using linear PIR [8]. 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 $\ell^\star = h(k)$ and then determines the corresponding α-KVS locations $t_1, \ldots, t_\alpha$. Recovering the desired value therefore reduces to obtaining the matrix entries $(\ell^\star, t_1), \ldots, (\ell^\star, t_\alpha)$.

One possibility is to issue a linear PIR query for the locations $t_1, \ldots, t_\alpha$ 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 a SIMD-style approach that applies the same private query vector $\vec{w}$ simultaneously across all r bins. Since the encoded database is represented as an $r \times d$ 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 $\mathsf{Enc}(\vec{w})$ to the server, which homomorphically computes $\mathsf{Enc}(\mathbf{M} \cdot \vec{w})$. 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 $\mathsf{Enc}(X^{-\ell^\star})$ for the target bin index $\ell^\star$. Using homomorphic multiplication together with the extraction techniques underlying KsPIR [23], the server can homomorphically isolate the $\ell^\star$-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 [23]. 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 [28], Hintless PIR [8], and KsPIR [23]. Existing approaches typically rely on either NTT-friendly plaintext moduli or Chinese Remainder Theorem (CRT)-based encodings. Instead, we adopt another technique from [26] that achieves comparable efficiency while supporting arbitrary plaintext moduli coprime 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 [26] evaluates expressions of the form

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

where σ denotes an automorphism in the Galois group G, $\{c_\sigma\}$ are arbitrary coefficients, and $\mathsf{Eval}_\sigma(\mathsf{Enc}(x))$ denotes the homomorphic evaluation of σ on ciphertext $\mathsf{Enc}(x)$. 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 $|\mathsf{G}|$ 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 $O(\sqrt{|\mathsf{G}|})$, yielding the decomposition

$$\sum_{\sigma \in \mathsf{G}} c_\sigma \cdot \sigma(x) = \sum_{\sigma_\mathsf{g} \in \Sigma_{\mathsf{giant}}} \sigma_\mathsf{g}\Big(\sum_{\sigma_\mathsf{b} \in \Sigma_{\mathsf{baby}}} \tilde{c}_{\sigma_\mathsf{g},\sigma_\mathsf{b}} \cdot \sigma_\mathsf{b}(x)\Big),$$

where $\tilde{c}_{\sigma_\mathsf{g},\sigma_\mathsf{b}} := {\sigma_\mathsf{g}}^{-1}\big(c_{\sigma_\mathsf{g} \circ \sigma_\mathsf{b}}\big)$. This decomposition gives rise to a generalized BSGS algorithm whose homomorphic counterpart follows naturally. As a result, the number of automorphism evaluations can be reduced from $O(|\mathsf{G}|)$ to $O(\sqrt{|\mathsf{G}|})$, yielding a substantial performance improvement.

Reducing Evaluation Keys. A naive implementation of the above BSGS decomposition appears to require $O(\sqrt{|\mathsf{G}|})$ 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 $O(\sqrt{|\mathsf{G}|})$.

To illustrate the main idea, consider the simplified setting where $\mathsf{G} = \langle\sigma\rangle$ 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

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

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.

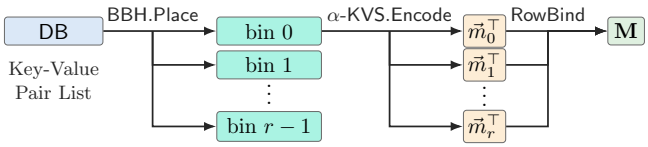

Figure 1. High-level overview of our database encoding.

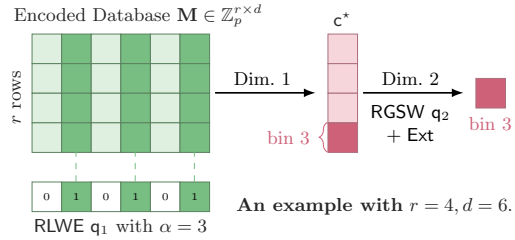

Figure 2. Our two-fold keyword PIR. Dimension 1 combines $\alpha = 3$ columns. Dimension 2 selects bin 3 via one RGSW.

Further Offline/Online Optimization. Our framework admits additional optimizations inspired by the offline/online insight of KsPIR [23]. 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 [23]: 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, much of the server workload moves to the offline phase, which reduces the online response time.

Handling Larger Databases. For larger databases, we adopt the idea of KsPIR [23] and split the database DB into multiple blocks $\mathsf{DB}^{(0)}, \ldots, \mathsf{DB}^{(\mathsf{N_{blk}}-1)}$, 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 $v = (v_0 \| v_1 \| \cdots \| v_{L-1})$ as in KsPIR [23]. 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 for large payloads.

1.3. Related Works

Keyword PIR was introduced by Chor et al. [12]. Early works [4], [12] suffer from large buckets and high computation; homomorphic equality tests [13] and sparse encodings (SparsePIR [14]) led to single-invocation designs, yet keyword PIR remained much slower than index PIR.

Recent works narrow this gap with KVS encodings, at the cost of communication or client state. ChalametPIR [15] combines KVS with FrodoPIR [29] but has communication linear in the database size. Hao et al. [16] combine sparse KVS, hashing-to-bins and approximate key-to-index mappings, but need either multiple index-PIR invocations or large client hints, and [30] uses minimal perfect hashing with non-trivial client state. Our construction avoids both client hints and multiple invocations.

Other works add functionality, such as circuit evaluation over retrieved data [31], an updatable oblivious KVS with access control [32] and incremental keyword PIR [33]. They are complementary to ours, which handles updates by re-encoding blocks at the server.

Symmetric PIR and OPRFs. Symmetric PIR [17] additionally protects the database: the client learns nothing beyond the records it queries, including through the noise of a homomorphic response [34]. Keyword PIR schemes that return more than the requested entry, such as a bin or a row, typically obtain SPIR with an oblivious PRF (OPRF) [18]. An OPRF is a two-party protocol in which a server holding a PRF key K and a client holding an input x compute $F_K(x)$, such that the client learns only $F_K(x)$ and the server learns nothing about x. The server stores each pair (k, v) as $(t_k,\ v \oplus \zeta_k)$, where $(t_k \,\|\, \zeta_k) := F_K(k)$: the tag t_k replaces the keyword and ζ_k masks the value. A client first obtains $F_K(k)$ for its own keyword through the OPRF and then retrieves the entry with that tag; every other entry it receives stays masked under PRF values it does not know, which yields SPIR.

Existing OPRFs fall into three groups. DH-based OPRFs such as 2HashDH [19], standardized in RFC 9497 [20], need one round trip and little computation, but are not post-quantum. Batched OPRFs from oblivious transfer or VOLE [35], [36] are fast and can be post-quantum, but derive the key from correlations set up for each session. Post-quantum OPRFs with a long-term key [21], [22], [37], [38], [39] have become much more efficient, but still need heavy proofs, several rounds or a preprocessing phase, or rely on less standard assumptions such as power residues or alternating moduli. All three groups face a problem when the database is updated. A long-term key masks every entry deterministically, so a client that retrieves the same row before and after an update learns the difference between the old and new values of every updated entry in that row. Avoiding this requires re-masking the whole database under a fresh key, and for a hint-based PIR also rebuilding the hint and sending it to every client again. Per-session keys avoid the leakage, but the server must then re-mask the database, and rebuild any hint, for every session. In addition, the server holds a secret whose exposure unmasks every row released so far. Keyword PIR from homomorphic equality tests [13], [40] returns only the matching value without an OPRF, but at a much higher computational cost.

Hintless Index PIR. HintlessPIR [8], YPIR [9] and KsPIR [23] remove the hint of SimplePIR by evaluating

its hint-dependent part at the server. Our framework works with any $\mathbb{Z}_p$-linearly homomorphic back-end; we use KsPIR for its small client-side costs (Appendix E).

2. Preliminaries

Notations. Let $\lambda \in \mathbb{N}$ denote the security parameter, $\mathsf{negl}(\cdot)$ and $\mathsf{poly}(\cdot)$ denote negligible and polynomial functions. For $n \in \mathbb{N}$, define $[n] := \{0, \ldots, n-1\}$. 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 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. We write $\Delta := \lfloor q/p \rfloor$ for the scaling factor and χ for the error distribution, of standard deviation ξ. 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], [43], [44] of Gentry–Sahai–Waters [45], and standard operations including coefficient extraction [46], gadget decomposition, key switching, and homomorphic Galois automorphisms [47], [48], [49], [50]. As these primitives are used as black boxes in our framework, we summarize their syntax in Table 1 and defer details to Appendix B.1. Ciphertexts are denoted by $\vec{c} \in \mathsf{LWE}_{\vec{s}}(\mu)$ for $\mu \in \mathbb{Z}_p$, $\mathsf{c} \in \mathsf{RLWE}_{\mathsf{s}}(\mu) \subset \mathcal{R}_q^2$, and $\mathsf{C} \in \mathsf{RGSW}_{\mathsf{s}}(\mu)$ for $\mu \in \mathcal{R}_p$, all under the secret key $\mathsf{sk} = (1, \mathsf{s})^\top$. The LWE algorithms are the natural analogues of the RLWE ones over $\mathbb{Z}_p$ and are omitted.

Four 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:

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

where B denotes the gadget base and $\ell_B := \lceil \log_B q \rceil$ is the number of gadget digits. Second, the external product

TABLE 1. SYNTAX OF THE LATTICE PRIMITIVES. ALGORITHMS ARE USED AS BLACK BOXES THROUGHOUT THE PAPER; APPENDIX B.1 GIVES THEIR INSTANTIATIONS AND NOISE.

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

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 Section 3.5 and in our noise analysis, which can be found in the supplementary material in our artifact repository. Third, each automorphism evaluation $\mathsf{Auto.Eval}_t$ 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 extraction Ext uses only public ciphertext coefficients, so either party can run it.

2.2. Key-Value Store

A key-value store (KVS) [24] 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 [16]. This variant will serve as a central component of our keyword PIR framework.

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

Definition 1 (Key-Value Store [24]) *A KVS over $\mathcal{L}$ consists of ppt algorithms* $(\mathsf{Encode}, \mathsf{Decode})$*:*

- $\mathsf{Encode}(1^\lambda, \mathcal{L}) \to S$*: on input λ and a list $\mathcal{L}$ with distinct keys, outputs a data structure S.*
- $\mathsf{Decode}(S, k_i) \to v_i$*: on input S and a key $k_i \in \mathcal{K}$, outputs $v_i \in \mathcal{V}$.*

Below, we present a variant of KVS specific to our framework, where decoding is performed as a weighted linear combination of α entries in the encoded data structure. We explicitly separate out a one-time setup procedure that samples the hash functions and induces the decoding mappings, and we let Encode sample its output uniformly

among all valid data vectors. Uniform sampling costs nothing online, and it makes the decodings of non-encoded keys look uniformly random, which our SPIR guarantee rests on.

Definition 2 (α-KVS [16]) *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* $(\mathsf{Setup}, \mathsf{Encode}, \mathsf{Decode})$*:*

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

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

  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*.*
- $\alpha\text{-KVS.Encode}(\mathsf{pp}_{\alpha\text{-KVS}}, \mathcal{L}) \to \mathbf{D}$ *or $\perp$: on input $\mathsf{pp}_{\alpha\text{-KVS}}$ and a list $\mathcal{L} \subseteq \mathcal{K} \times \mathbb{Z}_p$ containing at most d/η distinct-key pairs, outputs a data vector $\mathbf{D} \in \mathbb{Z}_p^d$ sampled uniformly from those 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 $(k, v) \in \mathcal{L}$, or outputs $\perp$ if no such $\mathbf{D}$ exists for the sampled hash family.
- $\alpha\text{-KVS.Decode}(\mathsf{pp}_{\alpha\text{-KVS}}, \mathbf{D}, k) \to v$*: outputs $v := \langle \vec{w}(k), \mathbf{D} \rangle \bmod p$.*

The ratio $\eta := d/|\mathcal{L}| \geq 1$ 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 the supplementary material in our artifact repository.

Remark 3 *The above definition generalizes the KVS abstraction of [24] by introducing a key-to-decoding-coefficient mapping $\vec{w}(\cdot) : \mathcal{K} \to \mathbb{Z}_p^d$ with Hamming weight α. The sparse KVS of [16] is recovered as the special case $\alpha = 3$ in which every nonzero coefficient equals 1; our instantiation instead uses pseudorandom nonzero coefficients (Appendix C.1). A polynomial-interpolation KVS [51], [52] 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 formalisms, and our instantiation is given in Section 3. With uniform* Encode*, the α-KVS is a linear OKVS [24]. We do not use obliviousness, which concerns uniform values, but the following property for arbitrary values: for keys K not in $\mathcal{L}$ whose decode vectors are linearly independent of each other and of those of $\mathcal{L}$, the decodings $(\langle \vec{w}(k), \mathbf{D} \rangle)_{k\in K}$ are uniform and independent of the values in $\mathcal{L}$. We write $\varepsilon_{\mathsf{dep}}$ for the probability over α-KVS.Setup that one further key breaks this independence; it is small as long as the number of decode vectors stays below the threshold of Remark 10. Any linear OKVS fits our framework, e.g., RB-OKVS [25]; we use that of [16] for comparability.*

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 [53], [54]) *For positive integers n, r, an (n, r)-BBH with key space $\mathcal{K}$ consists of ppt algorithms:*

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

2.4. Keyword Private Information Retrieval

A keyword PIR scheme allows a client to retrieve the value v associated with a queried keyword k from a server-held database DB without revealing k to the server. A *symmetric* keyword PIR (SPIR) additionally guarantees that the client learns nothing about DB beyond the values of its queried keywords [17]. We consider a semi-honest server and semi-honest clients in an authenticated model.

Definition 6 (Keyword PIR [12]) *A keyword PIR over $(\mathcal{K}, \mathcal{V})$ consists of ppt algorithms:*

- $\mathsf{Server.Setup}(1^\lambda, \mathsf{DB}) \to (\mathsf{pp}_{\mathsf{kPIR}}, \mathsf{st}_\mathsf{S})$*: server takes* DB *as input, then outputs public parameters $\mathsf{pp}_{\mathsf{kPIR}}$ and a server state st_S.*
- $\mathsf{Client.Setup}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$*: client generates a secret key sk and a public key pk, and in some PIR schemes $\mathsf{pk} := \perp$.*
- $\mathsf{Client.Query}(\mathsf{pp}_{\mathsf{kPIR}}, \mathsf{sk}, k) \to (\mathsf{st}, \mathsf{qu})$*: client generates a query qu for $k \in \mathcal{K}$ and a state st.*

TABLE 2. PARAMETERS IN CONSTRUCTION 8.

Parameter	Meaning
$n,\ n_\iota$	database size; per-block capacity $n/\mathsf{N_{blk}}$
$n_\iota^{\max}$	admissible per-block capacity
$\mathsf{N_{blk}}$	number of partition blocks
r	bin count of the BBH
$\alpha,\ \eta$	α-KVS sparsity; expansion rate $d/\lvert\mathcal{L}\rvert$
$\Lambda,\ \Lambda^{\max}$	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
$B,\ \ell_B$	gadget base and digit count

- $\mathsf{Server.Answer}(\mathsf{pp_{kPIR}}, \mathsf{st_S}, \mathsf{pk}, \mathsf{qu}) \to \mathsf{ans}$: *server computes an answer* ans.
- $\mathsf{Client.Recover}(\mathsf{st}, \mathsf{pp}, \mathsf{ans}) \to v$ *or* $\perp$: *client decodes the value* $v \in \mathcal{V}$, *or outputs* $\perp$.

Definition 7 (Correctness and Security) *A keyword PIR scheme must satisfy the following properties:*

- ***Correctness.*** *For any database* DB *and keyword* $k \in \mathcal{K}$, *let* $(\mathsf{st}, \mathsf{qu}) \leftarrow \mathsf{Client.Query}(\mathsf{pp_{kPIR}}, \mathsf{sk}, k)$ *and* $\mathsf{ans} \leftarrow \mathsf{Server.Answer}(\mathsf{pp_{kPIR}}, \mathsf{st_S}, \mathsf{pk}, \mathsf{qu})$. *We require, if* $(k, v) \in \mathsf{DB}$ *and if* k *is not in* DB, *respectively,*

$$\Pr[\mathsf{Client.Recover}(\mathsf{st}, \mathsf{pp}, \mathsf{ans}) = v] = 1,$$
$$\Pr[\mathsf{Client.Recover}(\mathsf{st}, \mathsf{pp}, \mathsf{ans}) = \perp] \geq 1 - \mathsf{negl}(\lambda).$$

- ***Security.*** *The scheme is secure if for any ppt adversary* $\mathcal{A}$ *and any* $k, k' \in \mathcal{K}$,

$$\big|\Pr[\mathcal{A}(\mathsf{pp}, \mathsf{pk}, \mathsf{qu}) = 1] - \Pr[\mathcal{A}(\mathsf{pp}, \mathsf{pk}, \mathsf{qu}') = 1]\big| \leq \mathsf{negl}(\lambda),$$

  where qu *is generated by* $\mathsf{Client.Query}(\mathsf{pp_{kPIR}}, \mathsf{sk}, k)$ *and* qu' *by* $\mathsf{Client.Query}(\mathsf{pp_{kPIR}}, \mathsf{sk}, k')$.
- ***SPIR Privacy.*** *Let* $\mathsf{DB}(k)$ *denote the value* v *with* $(k, v) \in \mathsf{DB}$, *or* $\perp$ *if there is none. The scheme is SPIR if there is a ppt simulator* Sim *such that, for any* DB *and any keywords* $k_1, \ldots, k_q$ *queried by a semi-honest client, no ppt adversary distinguishes the client's view (its keys, queries, answers and random coins) from* $\mathsf{Sim}(1^\lambda, \mathsf{pp_{kPIR}}, (k_i, \mathsf{DB}(k_i))_{i\in[q]})$ *with advantage more than* $\mathsf{negl}(\lambda)$.

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. Section 3.2 discusses the choice of the α-KVS, Section 3.3 instantiates the linear map f, and Sections 3.4 and 3.5 detail two optimizations. The detailed parameter selection and noise analysis can be found in the supplementary material in our artifact repository.

3.1. Our Construction

Our core protocol extends and generalizes the idea of KsPIR's [23] 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 $\mathsf{DB} := \{(k_z, v_z)\}_{z\in[n]}$ 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. In particular, each value is first prefixed by $\mathsf{tag}(k) \in \{0,1\}^\gamma$ using a tag function $\mathsf{tag} : \mathcal{K} \to \{0,1\}^\gamma$, 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 $\mathcal{S}_{\mathsf{plane}} := \mathbb{Z}_p^L$, denoted as $\mathsf{pl}(k, v)$, where $L := \left\lceil (\gamma + \ell_v) / \lfloor \log_2 p \rfloor \right\rceil$. The j-th entry $\mathsf{pl}(k, v)[j] \in \mathbb{Z}_p$ holds the j-th $\lfloor \log_2 p \rfloor$-bit part of $\mathsf{tag}(k) \,\|\, v$, called a *plane*. This yields the normalized list $\mathcal{L} := \{(k_z, \mathsf{pl}(k_z, v_z))\}_{z\in[n]} \subseteq \mathcal{K} \times \mathcal{S}_{\mathsf{plane}}$, 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 $\mathsf{N_{blk}} := \lceil n / n_\iota^{\max} \rceil$ equally sized blocks $\mathsf{DB}^{(0)}, \ldots, \mathsf{DB}^{(\mathsf{N_{blk}}-1)}$, where $n_\iota^{\max}$ 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 $n_\iota^{\max}$ rather than equal to it, since $\Lambda^{\max}$ must cover the largest bin while the capacity only fills the average one. Since all blocks operate independently but share the identical $\mathsf{pp_{BBH}}$ and $\mathsf{pp}_{\alpha\text{-}\mathsf{KVS}}$, a single client query can be evaluated on every block.

Within each block $\iota \in [\mathsf{N_{blk}}]$, the encoding proceeds in two steps. First, the server routes each pair of $\mathsf{DB}^{(\iota)}$ 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 $\mathbf{D}_{\ell,j}^{(\iota)} \in \mathbb{Z}_p^d$. Every bin, plane and block is encoded under the same $\mathsf{pp}_{\alpha\text{-}\mathsf{KVS}} = (H, \vec{w}(\cdot))$, which is what lets a single query $\vec{w}(k)$ serve all of them.

Second, the server stacks these rL vectors into a matrix $\mathbf{M}^{(\iota)} \in \mathbb{Z}_p^{rL\times d}$ using a layout as $\mathbf{M}^{(\iota)}[\,\ell \cdot L + j,\ \cdot\,] := \mathbf{D}_{\ell,j}^{(\iota)}$ 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) *See Figure 3.*

$\mathsf{Client.Setup}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$:

1) Sample $\mathsf{sk} := (1, \mathsf{s})^\top \leftarrow \mathsf{RLWE.KeyGen}(1^\lambda)$.
2) RGSW.KeyGen uses the same keys sk.
3) Generate $\mathsf{evk} \leftarrow \mathsf{RLWE.EvalKeyGen}(1^\lambda)$.
4) Output $\mathsf{sk} := (1, \mathsf{s})^\top$ and $\mathsf{pk} := \mathsf{evk}$.

$\mathsf{Client.Query}(\mathsf{pp}_{\mathsf{kPIR}}, \mathsf{sk}, k) \to (\mathsf{st}, \mathsf{qu})$:

1) $\ell_k^\star := \mathsf{BBH.Locate}(\mathsf{pp}_{\mathsf{BBH}}, k) \in [r]$.
2) $W_k(X) := \mathsf{coef}^{-1}(\vec{w}(k)) = \sum_{i=0}^{d-1} w_i(k)\, X^i$.
3) $\mathsf{q}_1 := (\mathsf{b}_1, \mathsf{a}_1) \leftarrow \mathsf{RLWE.Encrypt}(\mathsf{sk}, W_k(X))$.
4) $\mathsf{q}_2 \leftarrow \mathsf{RGSW.Encrypt}(\mathsf{sk}, X^{-\ell_k^\star L})$.
5) Output $(\mathsf{qu}, \mathsf{st})$ where $\mathsf{qu} := (\mathsf{q}_1, \mathsf{q}_2)$ and $\mathsf{st} := (k, \ell_k^\star, \mathsf{sk})$.

$\mathsf{Client.Recover}(\mathsf{st}, \mathsf{pp}_{\mathsf{kPIR}}, \mathsf{ans}) \to v$ or $\perp$:

1) Parse $\mathsf{st} = (k, \ell_k^\star, \mathsf{sk})$ with $\mathsf{sk} = (1, \mathsf{s})^\top$; set $\vec{s} := \mathsf{coef}(\mathsf{s}(X^{-1}))$ and the derived key $\overrightarrow{\mathsf{sk}} := (1, \vec{s})$.
2) For each $\iota \in [\mathsf{N}_{\mathsf{blk}}]$, decrypt $\mu_\iota[j] := \mathsf{LWE.Decrypt}(\overrightarrow{\mathsf{sk}}, \mathsf{L}_{\iota,j})$ for $j \in [L]$, and parse $\mu_\iota \in \mathcal{S}_{\mathsf{plane}}$ as $\tau_\iota \,\|\, \hat{v}_\iota$ with $|\tau_\iota| = \gamma$.
3) If one index $\iota^\star$ satisfies $\tau_{\iota^\star} = \mathsf{tag}(k)$, output $v := \hat{v}_{\iota^\star}$; otherwise output $\perp$.

Server.ReEncode: when a client has issued $\bar{\nu}$ queries to the current encoding, or on an update, rerun EncodeDB with fresh randomness on the affected blocks and redo steps 4–5 of Server.Setup.

$\mathsf{Server.Setup}(1^\lambda, \mathsf{DB}) \to (\mathsf{pp}_{\mathsf{kPIR}}, \mathsf{st}_{\mathsf{S}})$:

1) Sample $\mathsf{pp}_{\mathsf{BBH}} \leftarrow \mathsf{BBH.Setup}(1^\lambda, n, r)$.
2) Sample $\mathsf{pp}_{\alpha\text{-KVS}} \leftarrow \alpha\text{-KVS.Setup}(1^\lambda)$.
3) Run $\mathsf{EncodeDB}(\mathsf{pp}_{\mathsf{BBH}}, \mathsf{pp}_{\alpha\text{-KVS}}, \mathsf{DB})$ to obtain $\{\mathbf{M}^{(\iota)}\}_{\iota \in [\mathsf{N}_{\mathsf{blk}}]}$.
4) For $\iota \in [\mathsf{N}_{\mathsf{blk}}]$, $t \in [d]$, set $u_t^{(\iota)}(X) := \mathsf{coef}^{-1}\big(\mathbf{M}^{(\iota)}[\,\cdot\,, t]\big) \in \mathcal{R}_p$, where $\mathbf{M}^{(\iota)}[\,\cdot\,, t] \in \mathbb{Z}_p^{rL}$ is zero-padded to length m.
5) For $\iota \in [\mathsf{N}_{\mathsf{blk}}]$, let $f_\iota : \mathcal{R}_p \to \mathcal{R}_p$ be the $\mathbb{Z}_p$-linear map with $f_\iota(X^t) = u_t^{(\iota)}(X)$ for $t \in [d]$ and $f_\iota(X^t) = 0$ otherwise, and let $\mathsf{aux}_f := \{\mathsf{aux}_{f_\iota}\}_{\iota \in [\mathsf{N}_{\mathsf{blk}}]}$ be the auxiliary data of Section 3.3.
6) Output $\mathsf{pp}_{\mathsf{kPIR}} := (\mathsf{pp}_{\mathsf{BBH}}, \mathsf{pp}_{\alpha\text{-KVS}}, \mathsf{N}_{\mathsf{blk}})$; retain $\mathsf{st}_{\mathsf{S}} := (\{f_\iota\}_{\iota \in [\mathsf{N}_{\mathsf{blk}}]}, \mathsf{aux}_f)$.

$\mathsf{Server.Answer}(\mathsf{pp}_{\mathsf{kPIR}}, \mathsf{st}_{\mathsf{S}}, \mathsf{evk}, \mathsf{qu}) \to \mathsf{ans}$:

1) Parse $\mathsf{qu} = (\mathsf{q}_1, \mathsf{q}_2)$ and $\mathsf{st}_{\mathsf{S}} = (\{f_\iota\}, \mathsf{aux}_f)$.
2) Dimension 1. For each $\iota \in [\mathsf{N}_{\mathsf{blk}}]$, compute $\mathsf{c}_\iota^\star := \mathsf{RLWE.Eval}(f_\iota, \mathsf{q}_1, \mathsf{evk}) \in \mathcal{R}_q^2$.
3) Dimension 2. For each $\iota \in [\mathsf{N}_{\mathsf{blk}}]$, compute $\mathsf{T}_\iota := \mathsf{RGSW.ExternalProduct}(\mathsf{c}_\iota^\star, \mathsf{q}_2)$.
4) Extraction. For each ι and each $j \in [L]$, set $\mathsf{L}_{\iota,j} := \mathsf{Ext}(\mathsf{T}_\iota, j) \in \mathbb{Z}_q \times \mathbb{Z}_q^m$.
5) Output $\mathsf{ans} := (\mathsf{L}_{\iota,j})_{\iota \in [\mathsf{N}_{\mathsf{blk}}],\, j \in [L]}$.

Figure 3. Full specification of our Keyword PIR construction.

Algorithm 1 $\mathsf{EncodeDB}(\mathsf{pp}_{\mathsf{BBH}}, \mathsf{pp}_{\alpha\text{-KVS}}, \mathsf{DB})$

Parameters: $\mathsf{N}_{\mathsf{blk}}$, r, L, d, n, α.
Input: $\mathsf{pp}_{\mathsf{BBH}}$; $\mathsf{pp}_{\alpha\text{-KVS}} = (H, \vec{w}(\cdot))$; the key-value pair list $\mathsf{DB} := \{(k_z, v_z)\}_{z \in [n]}$.
Output: $\mathbf{M}^{(\iota)} \in \mathbb{Z}_p^{rL \times d}$ for $\iota \in [\mathsf{N}_{\mathsf{blk}}]$.

For each $z \in [n]$, set $\mathsf{pl}(k_z, v_z) \in \mathcal{S}_{\mathsf{plane}}$ to the plane vector of $\mathsf{tag}(k_z) \,\|\, v_z$.
Partition DB into $\mathsf{N}_{\mathsf{blk}}$ blocks $\mathsf{DB}^{(0)}, \ldots, \mathsf{DB}^{(\mathsf{N}_{\mathsf{blk}}-1)}$.
for $\iota \in [\mathsf{N}_{\mathsf{blk}}]$ **do**
    $\phi_\iota \leftarrow \mathsf{BBH.Place}(\mathsf{pp}_{\mathsf{BBH}}, \{k : (k, \cdot) \in \mathsf{DB}^{(\iota)}\})$.
    Set $S_{\ell,j}^{(\iota)} := \{(k, \mathsf{pl}(k, v)[j]) : \phi_\iota(k) = \ell\}$ for $\ell \in [r], j \in [L]$.
    Set $\mathbf{D}_{\ell,j}^{(\iota)}$ as $\alpha\text{-KVS.Encode}\big(\mathsf{pp}_{\alpha\text{-KVS}}, S_{\ell,j}^{(\iota)}\big)$.
    Set $\mathbf{M}^{(\iota)}[\ell L + j,\ \cdot\,] := \mathbf{D}_{\ell,j}^{(\iota)}$ for all $\ell \in [r]$, $j \in [L]$.
If any bin exceeds $\Lambda^{\max}(\kappa)$, resample and update the seed in $\mathsf{pp}_{\mathsf{BBH}}$, then restart.
If any α-KVS.Encode returned $\perp$, resample and update the seed in $\mathsf{pp}_{\alpha\text{-KVS}}$, then restart.
return $\{\mathbf{M}^{(\iota)}\}_{\iota \in [\mathsf{N}_{\mathsf{blk}}]}$.

Algorithm 2 $\mathsf{SearchDB}(\mathsf{pp}_{\mathsf{BBH}}, \mathsf{pp}_{\alpha\text{-KVS}}, \{\mathbf{M}^{(\iota)}\}_{\iota \in [\mathsf{N}_{\mathsf{blk}}]}, k)$

Parameters: $\mathsf{N}_{\mathsf{blk}}$, r, L, γ.
Input: $\mathsf{pp}_{\mathsf{BBH}}$; $\mathsf{pp}_{\alpha\text{-KVS}} = (H, \vec{w}(\cdot))$; $\{\mathbf{M}^{(\iota)}\}_{\iota \in [\mathsf{N}_{\mathsf{blk}}]}$; query key k.
Output: The value v associated with k, or $\perp$.

$\ell_k^\star \leftarrow \mathsf{BBH.Locate}(\mathsf{pp}_{\mathsf{BBH}}, k) \in [r]$.
for $\iota \in [\mathsf{N}_{\mathsf{blk}}]$ **do**
    For $j \in [L]$, decode $\mu_\iota[j] \leftarrow \langle \vec{w}(k),\, \mathbf{M}^{(\iota)}[\ell_k^\star L + j,\ \cdot\,] \rangle \bmod p$.
    Parse $\mu_\iota \in \mathcal{S}_{\mathsf{plane}}$ as $\tau_\iota \,\|\, \hat{v}_\iota$ with $|\tau_\iota| = \gamma$.
    if $\tau_\iota = \mathsf{tag}(k)$ **then**
        return $\hat{v}_\iota$.
return $\perp$.

Correctness. First, fix a block $\iota \in [\mathsf{N}_{\mathsf{blk}}]$. By the $\mathbb{Z}_p$-linearity of f_ι, the Dimension-1 fold inherently evaluates the query polynomial $W_k(X)$ to yield $\mu_\iota^\star := f_\iota(W_k(X)) = \sum_{\ell=0}^{r-1} \sum_{j=0}^{L-1} V_{\ell,j}^{(\iota)} X^{\ell L + j}$, where the coefficient $V_{\ell,j}^{(\iota)} := \langle \vec{w}(k), \mathbf{M}^{(\iota)}[\ell L + j,\ \cdot\,] \rangle$ exactly recovers the α-KVS.Decode result for bin ℓ on plane j. Dimension 2 then applies the external product with the selector $X^{-\ell_k^\star L}$ to compute $\mathsf{T}_\iota \in \mathsf{RLWE}_{\mathsf{s}}(\mu_\iota^\star \cdot X^{-\ell_k^\star L})$. This algebraic shift maps the

target coefficient at $X^{\ell_k^\star L+j}$ down to index j, enabling the server to directly extract the target planes via $\mathsf{Ext}(\mathsf{T}_\iota, j) \in \mathsf{LWE}_{\vec{s}}(V^{(\iota)}_{\ell_k^\star, j})$ for all $j \in [L]$. Because BBH.Locate and BBH.Place agree on every key, the bin $\ell_k^\star$ named by the client is exactly the bin $\phi_{\iota^\star}(k)$ into which the server placed k in its block $\iota^\star$, so that block returns the payload of k. Decryption recovers the $\mathsf{N_{blk}}$ candidate payloads, and since tag is modelled as a random oracle a union bound restricts the probability of a false-positive tag match across the remaining blocks to $(\mathsf{N_{blk}}-1)\cdot 2^{-\gamma}$, which is negligible.

SPIR Privacy. In every block, the client obtains only the decoding of k in bin $\ell_k^\star$. Only the block storing k returns its payload; in every other block, k is not encoded in that bin, so its decoding is uniform by Remark 4. Each such decoding is one linear equation in the data vector $\mathbf{D}$ of that bin, and uniform encoding keeps $\mathbf{D}$ random among all solutions for the stored keys. Hence the decodings reveal nothing about stored values as long as the decode vectors of the stored and the queried keys in each bin remain linearly independent. This holds with high probability as long as each client issues at most $\bar{\nu}$ queries to one encoding (Remark 10), whatever keywords it chooses; the server rate-limits clients accordingly and re-encodes the database with fresh randomness once a client exhausts this budget, which yields SPIR for an unbounded number of queries.

Theorem 9 (SPIR) *Assume that α-KVS.Encode samples uniformly and that the noise of the extracted LWE ciphertexts reveals nothing beyond their plaintexts, e.g., by noise flooding [34]. Then Construction 8 is SPIR (Definition 7) against a semi-honest client: for $q \le \bar{\nu}$ queries of the client to one encoding of DB (Remark 10), the simulation error is at most $q\mathsf{N_{blk}}\,\varepsilon_{\mathsf{dep}}$. Re-encoding with fresh randomness once the client has issued $\bar{\nu}$ queries extends this to any number of queries.*

We prove Theorem 9 in Appendix C.2. The simulator answers with the payload of k in its block and with uniform values elsewhere; this is indistinguishable as long as, in every bin, the decode vectors of the stored and the queried keys are linearly independent.

Remark 10 (Query budget) *Linear independence in a bin with $|\mathcal{L}| \le \Lambda^{\max}$ stored keys and ν distinct queried keywords not stored there means that a random sparse matrix with $|\mathcal{L}|+\nu$ rows, d columns and three nonzero entries per row has full row rank. Asymptotically, this holds with high probability while $(|\mathcal{L}|+\nu)/d < 0.918$ [55], [56], so a bin tolerates $\bar{\nu} = 0.918d - \Lambda^{\max} = \Theta(d)$ queried keywords.*

3.2. Choice of the α-KVS

Construction 8 accepts any α-KVS. We compare two representative instantiations: the dense polynomial KVS [57], whose decode vector $\vec{w}(k) = (1, k, \ldots, k^{d-1})$ is a Vandermonde row of Hamming weight $\alpha = d$, and the sparse 3-KVS based on segmented binary fuse filters [58] (Appendix C.1), which maps a key to one pseudorandom position in each of three consecutive segments, so that $\vec{w}(k)$ has Hamming weight $\alpha = 3$. They trade storage against encoding and update cost:

- **Plaintext side.** The polynomial KVS attains the optimal expansion rate $\eta = 1$, whereas the 3-KVS needs $\eta > 1$. In exchange, the 3-KVS encodes in expected $O(|\mathcal{L}|)$ time and the client computes only three positions and coefficients to build its query, whereas the polynomial KVS costs $O(d \log^2 d)$ to encode and a full degree-$(d-1)$ evaluation per query.
- **Homomorphic side.** Sparsity brings no saving under encryption: to hide the queried positions, the server processes all d coefficients, so the cost of Dimension 1 depends on d and not on α, and both instantiations cost the same. This is the opposite of the generic framework, whose cost grows with α. Two further properties separate them. The 3-KVS only inverts its unit coefficients and thus leaves the choice of p as free as the Galois-based linear map does, whereas the polynomial KVS inverts a Vandermonde system and requires $\mathbb{Z}_p$ to be a field. An update re-encodes its bin, which costs expected $O(|\mathcal{L}|)$ for the 3-KVS but a full re-interpolation for the polynomial KVS (Remark 10).

We therefore use the 3-KVS, which is also the KVS of [16] and keeps our comparison like-for-like.

3.3. Instantiation of f

Now we instantiate $\mathsf{RLWE.Eval}(f_\iota, \cdot)$ used in Construction 8. The $\mathsf{N_{blk}}$ 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_ι, u_t for $u_t^{(\iota)}$, and $\mathbf{M}$ for $\mathbf{M}^{(\iota)}$; the server repeats the entire evaluation once per block. Thus $f : \mathcal{R}_p \to \mathcal{R}_p$ is the $\mathbb{Z}_p$-linear map fixed by $f(X^t) = u_t(X)$ on the d data-carrying basis 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 [26].

Lemma 11 (Existence of $\mathbb{F}$-linear function f [26]) *Let $\mathbb{E}/\mathbb{F}$ be any finite Galois extension with $\mathsf{G} := \mathrm{Gal}(\mathbb{E}/\mathbb{F})$, and let $\vec{\beta}, \vec{\gamma}$ be vectors over $\mathbb{E}$ with same index set, such that $\vec{\beta}$ is $\mathbb{F}$-linearly independent with dual $\vec{\beta}^\vee$. Then $f_{\mathsf{sel}} : \mathbb{E} \to \mathbb{E}$ defined by*

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

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

Instantiation of f. Our $\mathcal{R}_p$ is a quotient ring rather than a field extension, so we need the ring version of Lemma 11, which can be found in [59, 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 $\mathsf{G} \cong \mathbb{Z}_{2m}^{\times}$ 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 $d \le t < m$. With $\hat{u}_i(X) := \sum_{t=0}^{d-1} X^{-it}\, u_t(X) \in \mathcal{R}_p$, the lemma yields $f(X^t) = u_t(X)$ via

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

the sum running only over the d nonzero columns. The server stores $\mathsf{aux}_f := \{\hat{u}_i\}_{i\in\mathbb{Z}_{2m}^{\times}}$ as part of its state $\mathsf{st_S}$; 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 $\mathsf{Auto.Eval}_i$ as Section 2.1. Thus, on input $\mathsf{c} \in \mathsf{RLWE_s}(\mu)$,

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

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):

$$\mathsf{RLWE.Eval}(f, \mathsf{q}_1, \mathsf{evk}) = m^{-1} \sum_{i\in\mathbb{Z}_{2m}^{\times}} \hat{u}_i \cdot \mathsf{Auto.Eval}_i(\mathsf{q}_1, \mathsf{evk}),$$

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 [28]. Our observation is that it needs no slot structure at all: it applies to any expression $\sum_{\sigma\in\mathbb{Z}_{2m}^{\times}} c_\sigma \cdot \mathsf{Auto.Eval}_\sigma(\mathsf{Enc}(x), \mathsf{evk})$ with arbitrary plaintext coefficients $\{c_\sigma\}$, which is exactly the form Section 3.3 gives to our Dimension 1. BSGS reduces 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 $\mathsf{G} \cong \mathbb{Z}_{2m}^{\times}$, 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 $\sigma = \sigma_{\mathsf{g}} \circ \sigma_{\mathsf{b}}$, where $\sigma_{\mathsf{b}} \in \Sigma_{\mathsf{baby}} := \{\sigma_{g^k}\}_{k\in[M_2]}$ and $\sigma_{\mathsf{g}} \in \Sigma_{\mathsf{giant}} := \{\sigma_{ig^{jM_2}}\}_{(i,j)\in\{\pm1\}\times[M_1]}$. Note that $|\Sigma_{\mathsf{baby}}| = M_2$ while $|\Sigma_{\mathsf{giant}}| = 2M_1$, 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

$$\sum_{\sigma\in\mathbb{Z}_{2m}^{\times}} c_\sigma \cdot \sigma(x) = \sum_{\sigma_{\mathsf{g}}\in\Sigma_{\mathsf{giant}}} \sigma_{\mathsf{g}}\Big(\sum_{\sigma_{\mathsf{b}}\in\Sigma_{\mathsf{baby}}} \tilde{c}_{\sigma_{\mathsf{g}},\sigma_{\mathsf{b}}} \cdot \sigma_{\mathsf{b}}(x)\Big),$$

where $\tilde{c}_{\sigma_{\mathsf{g}},\sigma_{\mathsf{b}}} := {\sigma_{\mathsf{g}}}^{-1}\big(c_{\sigma_{\mathsf{g}}\circ\sigma_{\mathsf{b}}}\big)$.

Writing $\sigma_{bg^{jM_2}} = \sigma_b \circ (\sigma_{g^{M_2}})^j$ for $b \in \{\pm1\}$, the outer sum splits along the two cosets of $\langle g\rangle$ and each coset becomes a power series in the *single* automorphism $\sigma_{g^{M_2}}$:

$$\sum_{\sigma_{\mathsf{g}}\in\Sigma_{\mathsf{giant}}} \sigma_{\mathsf{g}}\big(\mathsf{bs}_{\sigma_{\mathsf{g}}}\big) = \sum_{b\in\{\pm1\}} \sigma_b\Big(\sum_{j\in[M_1]} (\sigma_{g^{M_2}})^j \big(\mathsf{bs}_j^{(b)}\big)\Big), \tag{4}$$

where $\mathsf{bs}_j^{(b)} := \sum_{\sigma_{\mathsf{b}}\in\Sigma_{\mathsf{baby}}} \tilde{c}_{\sigma_{bg^{jM_2}},\sigma_{\mathsf{b}}} \cdot \sigma_{\mathsf{b}}(x)$. 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.

We formalize this procedure in Algorithm 3, instantiated with $\mathsf{c} := \mathsf{q}_1$ 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

$$\mathsf{c}^\star \leftarrow \mathsf{BSGSEval}\big(\mathsf{q}_1,\ \{\hat{u}_i\}_{i\in\mathbb{Z}_{2m}^{\times}},\ \mathsf{evk}_g,\ \mathsf{evk}_{g^{M_2}},\ \mathsf{evk}_{-1}\big),$$

replacing the direct evaluation in Equation (3).

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

Input: $\mathsf{c} \in \mathsf{RLWE_s}(\mu)$; $\{c_\sigma\}_{\sigma\in\mathbb{Z}_{2m}^{\times}} \subset \mathcal{R}_p$; evk_g, $\mathsf{evk}_{g^{M_2}}$, evk_{-1}.
Output: Ciphertext in $\mathsf{RLWE_s}\big(\sum_\sigma c_\sigma \cdot \sigma(\mu)\big)$.

$\mathsf{c}'_{\sigma_1} \leftarrow \mathsf{c}$.
for $k = 1, \ldots, M_2 - 1$ **do** ▷ Baby chain.
  $\mathsf{c}'_{\sigma_{g^k}} \leftarrow \mathsf{Auto.Eval}_g\big(\mathsf{c}'_{\sigma_{g^{k-1}}}, \mathsf{evk}_g\big) \in \mathcal{R}_q^2$.
For $b \in \{\pm1\}$, $j \in [M_1]$, $\mathsf{bs}_j^{(b)} := \sum_{\sigma_{\mathsf{b}}\in\Sigma_{\mathsf{baby}}} \tilde{c}_{\sigma_{bg^jM_2},\sigma_{\mathsf{b}}} \cdot \mathsf{c}'_{\sigma_{\mathsf{b}}}$.
for $b \in \{\pm1\}$ **do** ▷ Giant chain.
  $\mathsf{c}_b \leftarrow \mathsf{bs}_{M_1-1}^{(b)}$.
  for $j = M_1 - 2$ **down to** 0 **do**
    $\mathsf{c}_b \leftarrow \mathsf{Auto.Eval}_{g^{M_2}}\big(\mathsf{c}_b, \mathsf{evk}_{g^{M_2}}\big) + \mathsf{bs}_j^{(b)}$.
return $m^{-1}\big(\mathsf{c}_{+1} + \mathsf{Auto.Eval}_{-1}(\mathsf{c}_{-1}, \mathsf{evk}_{-1})\big)$.

Three Evaluation Keys. Algorithm 3 runs on evk_g, $\mathsf{evk}_{g^{M_2}}$ and evk_{-1} alone instead of all keys uploaded by the client, 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 $|\Sigma_{\mathsf{giant}}| - 1 = 2M_1 - 1$ 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,

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

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. The noise analysis in the supplementary material in our artifact repository 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 [23]. Even seemingly query-dependent computation admits a useful decomposition. Specifically, an RLWE query ciphertext q_1 can be written as $(\mathsf{b}_1, \mathsf{a}_1)$, 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 D. A complete formalization is given in Algorithm 4.

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

Precomputing 2: Dimension 1 (The BSGS Fold). We observe that BSGSEval can be split into two branches: a keyword-dependent and a keyword-independent one. 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 $\mathsf{c}_0^\star = \mathsf{c}_{0,\mathsf{off}}^\star + \mathsf{c}_{0,\mathsf{on}}^\star$. 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.

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

Parameters: L.
Input: $\mathbf{M}$ via $\{c_\sigma\}_{\sigma\in\mathbb{Z}_{2m}^\times}$, $c_\sigma := \hat{u}_\sigma$; $(\mathsf{b}_1, \mathsf{a}_1)$; q_2; evk_g, $\mathsf{evk}_{g^{M_2}}$, evk_{-1}.
Output: $\mathsf{ans} := (\mathsf{L}_j)_{j\in[L]}$ for the fixed block.

Static phase (per database):
For every $(\sigma_\mathsf{g}, \sigma_\mathsf{b}) \in \Sigma_{\mathsf{giant}} \times \Sigma_{\mathsf{baby}}$, set $\tilde{c}_{\sigma_\mathsf{g},\sigma_\mathsf{b}} \leftarrow {\sigma_\mathsf{g}}^{-1}(c_{\sigma_\mathsf{g}\circ\sigma_\mathsf{b}})$.
Offline phase (per query):
$(\mathsf{c}_{0,\mathsf{off}}^\star, \mathsf{c}_1^\star) \leftarrow \mathsf{BSGSEval}\big((0, \mathsf{a}_1), \{c_\sigma\}, \mathsf{evk}\big)$.
Online phase :
$\mathsf{c}_{0,\mathsf{on}}^\star \leftarrow m^{-1}\sum_{\sigma_\mathsf{g}\in\Sigma_{\mathsf{giant}}} \sigma_\mathsf{g}\big(\sum_{\sigma_\mathsf{b}\in\Sigma_{\mathsf{baby}}} \tilde{c}_{\sigma_\mathsf{g},\sigma_\mathsf{b}} \cdot \sigma_\mathsf{b}(\mathsf{b}_1)\big)$; set $\mathsf{c}^\star \leftarrow (\mathsf{c}_{0,\mathsf{off}}^\star + \mathsf{c}_{0,\mathsf{on}}^\star, \mathsf{c}_1^\star)$.
Set $\mathsf{T} \leftarrow \mathsf{c}^\star \boxdot \mathsf{q}_2$.
For every $j \in [L]$, set $\mathsf{L}_j \leftarrow \mathsf{Ext}(\mathsf{T}, j)$.
return $\mathsf{ans} := (\mathsf{L}_j)_{j\in[L]}$.

4. Implementation and Evaluation

To evaluate concrete performance, we implement Construction 8 and compare it with the fastest prior keyword PIR constructions of [16]. For a direct algorithmic comparison, we implement our scheme in Java on top of the `mpc4j` library, matching the implementation and environment of [16]. We also develop a C++ implementation, incorporating the engineering optimizations of KsPIR [23] 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

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; the plaintext modulus $p \in \{40961, 65537\}$ and the gadget base $\log B \in \{3, 4, 5\}$ are selected per cell by a dynamic search. The noise analysis, the parameter-selection rule and the detailed per-cell parameters are given in the supplementary material in our artifact repository. Every reported configuration passes a 24-keyword exact-bytes recovery check. Each value carries a 64-bit tag, as

TABLE 3. EVALUATION RESULTS MEASURED OVER THE ENTIRE DATABASE. HINT IS THE PER-CLIENT DOWNLOAD BEFORE THE FIRST QUERY; OPRF STATES WHETHER SPIR RELIES ON AN ADDITIONAL OPRF IN THE STANDARD COMPOSITION. OUR CONSTRUCTION IS SHADED IN GRAY. 3-KSPIRV COMBINES THE FRONT-END OF [16] WITH OUR BACK-END, WHEREAS OURS (*Java*) AND OURS (*C++*) ARE OUR FULL CONSTRUCTION.

Settings		Hint	OPRF$^{\mathcal{S}}$		Latency					Communication‡		
n / ℓ_v	Scheme	Size (MB)	for SPIR	N_{blk}	Online* (ms)	Speed-up¶	Thr.* (MB/s)	Setup† (s)	Offline (s)	Query (KB)	Evk (MB)	Resp. (KB)
2^{18}/32 B	KPIR^{index}	14.5	Yes	–	15.0	–	533	2.7	–	12.7	–	14.4
	3-KsPIRV	**0**	**No**	1	121.6	–	66	2.2	4.87	90+810	3.9	39.2
	Ours (Java)	**0**	**No**	1	39.8	**3.06**×	201	2.2	1.64	30+270	3.9	12.6
	Ours (C++)	**0**	**No**	1	10.7	–	748	0.3	0.45	28+252	9.2	13.1
2^{18}/64 B	KPIR^{index}	20.3	Yes	–	28.0	–	571	4.7	–	17.0	–	20.3
	3-KsPIRV	**0**	**No**	1	121.9	–	131	2.4	4.90	90+810	3.9	39.3
	Ours (Java)	**0**	**No**	1	40.3	**3.02**×	397	2.2	1.62	30+270	3.9	13.1
	Ours (C++)	**0**	**No**	1	8.7	–	1839	0.4	0.47	28+252	9.2	13.1
2^{18}/128 B	KPIR^{index}	29.3	Yes	–	120.0	–	267	8.7	–	23.3	–	29.2
	3-KsPIRV	**0**	**No**	2	242.8	–	132	3.9	7.48	90+810	3.9	79.3
	Ours (Java)	**0**	**No**	2	81.3	**2.99**×	394	3.7	2.49	30+270	3.9	25.4
	Ours (C++)	**0**	**No**	2	16.2	–	1975	1.0	1.00	28+252	9.2	26.4
2^{18}/256 B	KPIR^{index}	44.4	Yes	–	248.0	–	258	17.9	–	32.5	–	44.3
	3-KsPIRV	**0**	**No**	3	363.9	–	176	5.3	9.96	90+810	3.9	120.8
	Ours (Java)	**0**	**No**	3	121.5	**3.00**×	527	5.6	3.35	30+270	3.9	40.3
	Ours (C++)	**0**	**No**	3	33.2	–	1928	1.0	1.54	28+252	9.2	41.8
2^{20}/32 B	KPIR^{index}	27.3	Yes	–	74.0	–	432	10.1	–	25.3	–	27.0
	3-KsPIRV	**0**	**No**	2	242.7	–	132	4.5	7.49	90+810	3.9	75.4
	Ours (Java)	**0**	**No**	2	82.2	**2.95**×	389	4.0	2.48	30+270	3.9	26.1
	Ours (C++)	**0**	**No**	2	22.1	–	1448	1.0	0.93	28+252	9.2	26.1
2^{20}/64 B	KPIR^{index}	37.4	Yes	–	135.0	–	474	16.8	–	33.9	–	37.1
	3-KsPIRV	**0**	**No**	3	365.1	–	175	5.8	10.03	90+810	3.9	118.0
	Ours (Java)	**0**	**No**	4	161.6	**2.26**×	396	7.0	4.21	30+270	3.9	52.5
	Ours (C++)	**0**	**No**	4	32.9	–	1945	1.8	1.95	28+252	9.2	52.5
2^{20}/128 B	KPIR^{index}	52.9	Yes	–	261.0	–	490	31.8	–	46.6	–	52.6
	3-KsPIRV	**0**	**No**	6	731.3	–	175	10.2	17.80	90+810	3.9	237.9
	Ours (Java)	**0**	**No**	7	289.0	**2.53**×	443	11.5	6.81	30+270	3.9	92.5
	Ours (C++)	**0**	**No**	7	57.3	–	2234	4.1	3.48	28+252	9.2	92.5
2^{20}/256 B	KPIR^{index}	77.7	Yes	–	492.0	–	520	63.4	–	65.0	–	77.3
	3-KsPIRV	**0**	**No**	11	1361.9	–	188	18.5	30.65	90+810	3.9	442.8
	Ours (Java)	**0**	**No**	13	538.7	**2.53**×	475	20.6	11.95	30+270	3.9	174.4
	Ours (C++)	**0**	**No**	13	103.0	–	2485	4.9	7.52	28+252	9.2	174.4
2^{22}/32 B	KPIR^{index}	53.6	Yes	–	186.0	–	688	39.7	–	50.6	–	52.3
	3-KsPIRV	**0**	**No**	6	742.9	–	172	12.2	17.80	90+810	3.9	235.1
	Ours (Java)	**0**	**No**	8	331.4	**2.24**×	386	14.6	7.65	30+270	3.9	100.5
	Ours (C++)	**0**	**No**	8	64.2	–	1994	4.8	4.24	28+252	9.2	104.5
2^{22}/64 B	KPIR^{index}	72.4	Yes	–	346.0	–	740	66.8	–	67.9	–	71.2
	3-KsPIRV	**0**	**No**	11	1351.1	–	189	19.7	30.74	90+810	3.9	432.8
	Ours (Java)	**0**	**No**	14	575.9	**2.35**×	445	24.6	12.94	30+270	3.9	176.5
	Ours (C++)	**0**	**No**	14	111.0	–	2306	7.7	7.75	28+252	9.2	183.6
2^{22}/128 B	KPIR^{index}	100.6	Yes	–	810.0	–	632	132.3	–	93.3	–	99.3
	3-KsPIRV	**0**	**No**	21	2557.5	–	200	37.1	56.72	90+810	3.9	832.6
	Ours (Java)	**0**	**No**	26	1076.4	**2.38**×	476	42.4	23.08	30+270	3.9	343.6
	Ours (C++)	**0**	**No**	26	198.9	–	2574	12.4	15.52	28+252	9.2	343.6
2^{22}/256 B	KPIR^{index}	143.6	Yes	–	1107.0	–	925	267.5	–	130.0	–	142.3
	3-KsPIRV	**0**	**No**	41	5034.2	–	203	69.5	107.58	90+810	3.9	1650.5
	Ours (Java)	**0**	**No**	49	2009.0	**2.51**×	510	80.0	43.05	30+270	3.9	657.5
	Ours (C++)	**0**	**No**	49	396.4	–	2583	38.5	43.16	28+252	9.2	657.5

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

† **Setup:** BBH and α-KVS encoding and auxiliary table building (Section 3.3), on 16 threads for every scheme. Per-client key generation ($\approx$ 16 s) is reported separately (supplementary material).

$^{\mathcal{S}}$ **OPRF:** KPIR^{index} returns a whole row and typically needs an additional OPRF for SPIR (Section 4.2); the hintless rows are SPIR without one (Theorem 9).

¶ **Speedup:** online time of 3-KsPIRV divided by that of Ours (*Java*), which shares its back-end and library.

‡ **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.

in both baselines. Client and server run in a single process; latencies are medians of repeated runs after JVM warm-up.

Baselines. We call the *front-end* the layer that turns a keyword into what the server must fetch, and the *back-end* the index-PIR layer that evaluates the encrypted query. 3-KsPIRV combines the three-query front-end of the generic construction KPIR^{kvs} in [16] with our back-end and the same randomized α-KVS, implemented on the same Java library as Ours $(Java)$, so the two differ only in the front-end; it issues $\alpha = 3$ queries per block. KPIR^{index} is the fastest construction of [16], run from the authors' implementation with only its existing parallel setup enabled; it is hint-based and returns a whole row. We do not benchmark KPIR^{hash}, which [16] reports to be less efficient than KPIR^{index} and which also returns a whole row.

4.2. Evaluation Results

Table 3 reports the full grid. Setup is a one-time per-database cost, and the offline phase runs per query but before the keyword is known, so neither contributes to the online latency, following the setting of KsPIR [23].

Front-end. Ours $(Java)$ and 3-KsPIRV share the back-end, so their ratio isolates the front-end: ours is faster in every cell, by 2.24–3.06× with a mean of 2.65×, owing to linear homomorphism, without decaying as n grows. 3-KsPIRV's filter reaches a lower expansion rate (1.125 against our 1.309) and hence fewer blocks (41 against 49 at 2^{22}/256 B), reducing its loss from 3× to 2.65×.

Hintless versus hint-based. Compared with Ours $(Java)$ on the same `mpc4j` framework, KPIR^{index} achieves lower online latency in ten of the twelve settings, by up to 2.65× at 2^{18}/32 B, but every client must first download and store a 14.5–143.6 MB hint. Ours $(C++)$ is faster than KPIR^{index} in all twelve settings, by 1.40–7.47× with a geometric mean of 3.75×. The difference stems from the underlying SimplePIR and KsPIR back-ends; our C++ implementation also benefits from native execution, the optimizations of KsPIR [23] and AVX, and since [16] provides no C++ implementation, this is not a same-language comparison.

Communication and setup. Due to the multi-block design, our query is constant in n: 300 KB in Java (30 KB offline for a_1 and 270 KB online) and 280 KB in C++. 3-KsPIRV sends three times as much, and KPIR^{index}'s query grows from 12.7 to 130.0 KB. Our response grows with n, up to 657.5 KB, as every block returns one answer. Evaluation keys (3.9 MB in Java, 9.2 MB in C++) are uploaded once per client; thanks to the three-key evaluation of Section 3.4, this is 42× less than uploading all 126 keys. Setup runs on 16 threads for every scheme and takes 0.3–80.0 s for ours against 2.7–267.5 s for KPIR^{index}; client key generation (16–17 s, flat in n) is per client and reported separately. The per-query offline phase costs 0.45–43.2 s against 4.87–107.6 s for 3-KsPIRV, which runs it three times.

TABLE 4. COST OF SPIR: KPIR^{INDEX} WITH OPRFS VERSUS OURS.

	KPIR^{index} + OPRF			
	2HashDH [20]	Pool [39]	VOLE-based [35]	Ours
Static database				
Post-quantum	×	✓	✓[a]	✓
Online round trips[b]	2	2	1	1
Server secret	long-term	long-term	per session	none
Hint per client (MB)	14.5–143.6	14.5–143.6	14.5–143.6[c]	0
Updatable database				
Update w/o re-mask[e]	leaks diff.	leaks diff.	–[c]	no leak
Update, server (s)	2.7–267.5	2.7–267.5	2.7–267.5[c]	0.3–2.2[d]
Update, client (MB)	14.5–143.6	14.5–143.6	14.5–143.6[c]	0

Ranges over the twelve settings of Table 3. [a] If the VOLE correlations are generated post-quantumly. [b] Keyword-dependent, including the PIR query; OPRF and PIR query are sequential for KPIR^{index} unless the key is per session. [c] Paid for every session, independently of updates. [d] Re-encoding one block, estimated as Setup/$\mathsf{N}_{\mathsf{blk}}$ (0.3–0.8 s in C++, 1.6–2.2 s in Java). [e] "Update" rows: leak-free update, i.e., re-mask under a fresh key, rebuild and redistribute the hint.

Cost of SPIR. As KPIR^{index} returns a whole row, it typically obtains SPIR through an OPRF (Section 1.3); in this composition the OPRF precedes the PIR query, because the key-to-index map is built over PRF images; this adds a keyword-dependent round trip that cannot be moved offline. The OPRF computation itself is small, e.g., below 1 ms and 11.9 kB per query for Pool [39]; the dominant costs arise from updates (Table 4).

Why updates leak. With a long-term key, the mask $\zeta_{k'}$ of an entry never changes: if the row a client retrieves also holds another user's value under k', the client receives $v' \oplus \zeta_{k'}$ and, after an update, $v'' \oplus \zeta_{k'}$, whose XOR reveals how the value changed although the client never queried k'.

Cost of a leak-free update. With a long-term key, a leak-free update re-masks the whole database under a fresh key and rebuild the key-to-index map and the hint which every client downloads again, unless clients track per-entry state such as version numbers; with per-client or per-session keys, this recurs for every client or session.

Our construction keeps no server secret, and re-encoding with fresh randomness both resets the query budget and handles updates (Theorem 9). For our parameters, simulation finds no dependence up to 400 keywords per bin and dependence at 600, so we set $\bar{\nu} = 300$ queries per client and encoding (supplementary material). If one client queries continuously, re-encoding every 300 queries costs 1.0–128.3 ms per query in C++ and 7.3–266.7 ms in Java (9–32% and 13–18% of online time), and it can be prepared in the background. An update re-encodes only the affected block, 9–340× (C++) and 1.2–164× (Java) cheaper than rebuilding the hint in KPIR^{index} [16]. 3-KsPIRV can use the same technique with a budget about a third of ours.

Lightweight clients. Although KPIR^{index} transfers less per query (Table 3), once its hint is counted a client needs 20 to 228 queries between two updates before KPIR^{index} transfers less in total than ours, whose evaluation keys (3.9–9.2 MB) are uploaded only once.

References

[1] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *Proc. IEEE FOCS*, 1995, pp. 41–50.

[2] 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.

[3] H. Corrigan-Gibbs and D. Kogan, "Private information retrieval with sublinear online time," in *Proc. EUROCRYPT*, 2020, pp. 44–75.

[4] 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.

[5] M. H. Mughees, H. Chen, and L. Ren, "OnionPIR: Response efficient single-server PIR," in *Proc. ACM CCS*, 2021, pp. 2292–2306.

[6] 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.

[7] 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.

[8] B. Li, D. Micciancio, M. Raykova, and M. Schultz, "Hintless single-server private information retrieval," in *Proc. CRYPTO*, 2024, pp. 183–217.

[9] S. J. Menon and D. J. Wu, "YPIR: High-throughput single-server PIR with silent preprocessing," in *Proc. USENIX Security*, 2024.

[10] 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.

[11] D. Kogan and H. Corrigan-Gibbs, "Private blocklist lookups with checklist," in *Proc. USENIX Security*, 2021, pp. 875–892.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin, "Protecting data privacy in private information retrieval schemes," in *Proc. ACM STOC*, 1998, pp. 151–160.

[18] S. Casacuberta, J. Hesse, and A. Lehmann, "SoK: Oblivious pseudorandom functions," in *Proc. IEEE EuroS&P*, 2022, pp. 625–646.

[19] S. Jarecki, A. Kiayias, and H. Krawczyk, "Round-optimal password-protected secret sharing and T-PAKE in the password-only model," in *Proc. ASIACRYPT*, 2014, pp. 233–253.

[20] A. Davidson, A. Faz-Hernandez, N. Sullivan, and C. A. Wood, "Oblivious pseudorandom functions (OPRFs) using prime-order groups," RFC 9497, 2023.

[21] Y. Yang, F. Benhamouda, S. Halevi, H. Krawczyk, and T. Rabin, "Gold OPRF: Post-quantum oblivious power-residue PRF," in *Proc. IEEE S&P*, 2025, pp. 259–278.

[22] D. F. Aranha, A. van Baarsen, A. B. Hansen, K. Nielsen, and P. Scholl, "A maliciously-secure post-quantum OPRF from crypto dark matter," in *Proc. IEEE S&P*, 2026.

[23] M. Luo, F.-H. Liu, and H. Wang, "Faster FHE-based single-server private information retrieval," in *Proc. ACM CCS*, 2024, pp. 1405–1419.

[24] 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.

[25] 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.

[26] C. Peikert and Z. Pepin, "Vive galois! Part 1: Optimal SIMD packing and packed bootstrapping for FHE," in *Proc. TCC*, 2025, pp. 185–219.

[27] W. Liu *et al.*, "mpc4j: An efficient multi-party computation library," https://github.com/alibaba-edu/mpc4j, 2024.

[28] S. Halevi and V. Shoup, "Faster homomorphic linear transformations in HElib," in *Proc. CRYPTO*, 2018, pp. 93–120.

[29] A. Davidson, G. Pestana, and S. Celi, "FrodoPIR: Simple, scalable, single-server private information retrieval," *popets*, vol. 2023, no. 1, pp. 365–383, 2023.

[30] 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.

[31] 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.

[32] 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.

[33] B. Ninh, "Incremental keyword private information retrieval from d-ary segmented cuckoo filters," Cryptology ePrint Archive, Paper 2026/2082, 2026.

[34] C. Lin, Z. Liu, and T. Malkin, "XSPIR: Efficient symmetrically private information retrieval from ring-LWE," in *Proc. ESORICS*, 2022, pp. 217–236.

[35] S. Raghuraman and P. Rindal, "Blazing fast PSI from improved OKVS and subfield VOLE," in *Proc. ACM CCS*, 2022, pp. 2505–2517.

[36] K. Han, S. Kim, B. Lee, and Y. Son, "Revisiting OKVS-based OPRF and PSI: Cryptanalysis and better construction," in *Proc. ASIACRYPT*, 2024, pp. 266–296.

[37] M. R. Albrecht, A. Davidson, A. Deo, and N. P. Smart, "Round-optimal verifiable oblivious pseudorandom functions from ideal lattices," in *Proc. PKC*, 2021, pp. 261–289.

[38] L. Heimberger, D. Kales, R. Lolato, O. Mir, S. Ramacher, and C. Rechberger, "Leap: A fast, lattice-based OPRF with application to private set intersection," in *Proc. EUROCRYPT*, 2025, pp. 254–283.

[39] A. Davidson, A. Deo, and L. T. Thibault, "Pool: A practical OT-based OPRF from learning with rounding," in *Proc. ACM CCS*, 2025, pp. 1038–1052.

[40] I. Ahmad, D. Agrawal, A. E. Abbadi, and T. Gupta, "Pantheon: Private retrieval from public key-value store," *Proc. VLDB Endow.*, vol. 16, no. 4, pp. 643–656, 2022.

[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] S. Patel, G. Persiano, and K. Yeo, "Private stateful information retrieval," in *Proc. ACM CCS*, 2018, pp. 1002–1019.

[52] M. J. Freedman, K. Nissim, and B. Pinkas, "Efficient private matching and set intersection," in *Proc. EUROCRYPT*, 2004, pp. 1–19.

[53] 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.

[54] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations (extended abstract)," in *Proc. ACM STOC*, 1994, pp. 593–602.

[55] P. Ayre, A. Coja-Oghlan, P. Gao, and N. Müller, "The satisfiability threshold for random linear equations," *Combinatorica*, 2020, arXiv:1710.07497.

[56] M. Dietzfelbinger and S. Walzer, "Dense peelable random uniform hypergraphs," in *27th Annual European Symposium on Algorithms (ESA)*, ser. LIPIcs, vol. 144, 2019, p. 38.

[57] 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.

[58] 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.

[59] Z. Pepin, "Algebraic structure in lattice cryptography," Ph.D. dissertation, University of Michigan, 2025.

[60] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proc. EUROCRYPT*, 2010, pp. 1–23.

[61] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *Proc. ACM STOC*, 2008, pp. 197–206.

Appendix A. Ethics Considerations

This work involves no human subjects, no personal or otherwise sensitive data, and no interaction with third-party systems: all experiments run on our own machine over synthetic databases generated from a fixed seed, and the work discloses no vulnerability.

Appendix B. Deferred Preliminaries

B.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 $\mathcal{R}_t := \mathcal{R}/t\mathcal{R}$ for $t \in \{p, q\}$. 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 several plausible LWE-based homomorphic encryption schemes [41], [50], [60], [61]; 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 $\vec{c} \in \mathsf{LWE}_{\vec{s}}(\mu)$ for the resulting ciphertext, where the LWE Encryption consists of the following *ppt* algorithms parameterized by dimension N:

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

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., $\mathsf{c} := (\mathsf{b}, \mathsf{a}) = (-\mathsf{as} + e + \Delta\mu,\ \mathsf{a}) \in \mathcal{R}_q^2$ with $\mu \in \mathcal{R}_p$, $\mathsf{a} \xleftarrow{\$} \mathcal{R}_q$ and $\mathsf{s} \xleftarrow{\$} \mathcal{R}_q$. We write $\mathsf{c} \in \mathsf{RLWE_s}(\mu)$ for the resulting ciphertext.

Extraction. Given a ring element, we denote the process that isolates its coefficient vector as coef, i.e., given $\mathsf{a} := \sum_i a_i X^i \in \mathcal{R}_q$ with $a_i \in \mathbb{Z}_q$, let $\mathsf{coef}(\mathsf{a}) := (a_0, \ldots, a_{m-1})$. Conversely, $\mathsf{coef}^{-1}(a_0, \ldots, a_{m-1}) := \mathsf{a}$. For $\mathsf{a} \in \mathcal{R}_q$, define $\vec{a} := \mathsf{coef}(\mathsf{a}(X^{-1})) \in \mathbb{Z}_q^m$. Consequently, given an RLWE ciphertext $\mathsf{c} = (\mathsf{b}, \mathsf{a}) \in \mathsf{RLWE_s}(\mu)$, there is a simple way to extract an 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 $\mathsf{Ext}(\mathsf{c}, j) := \big(\mathsf{coef}(\mathsf{b})[j], \mathsf{coef}(\mathsf{a})^{(j)}\big) \in \mathsf{LWE}_{\vec{s}}\big(\mathsf{coef}(\mu)[j]\big)$, where $\mathsf{coef}(\mathsf{a})^{(j)}$ is the negacyclically reindexed coefficient vector for index j (with $\mathsf{coef}(\mathsf{a})^{(0)} = \mathsf{coef}(\mathsf{a})$).

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

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

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

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

RGSW **Encryption.** RGSW [42], [43], [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 $\mathsf{C} \in \mathsf{RGSW}_\mathsf{s}(\mu)$ denote an encryption of μ under the key $\mathsf{sk} := (1, \mathsf{s})^\top$. The scheme consists of the following *ppt* algorithms:

- $\mathsf{RGSW.KeyGen}(1^\lambda) \to \mathsf{sk}$: outputs $\mathsf{sk} := (1, \mathsf{s})^\top \leftarrow \mathsf{RLWE.KeyGen}(1^\lambda)$.
- $\mathsf{RGSW.Encrypt}(\mathsf{sk}, \mu \in \mathcal{R}_p) \to \mathsf{C}$: 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

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

  satisfying $\mathsf{C} \cdot \mathsf{sk} = \vec{e} + \mu \cdot \mathsf{PowersOfB}(\mathsf{sk})$ for $\vec{e} \leftarrow \chi^{2\ell_B}$. For $i \in [\ell_B], j \in [2]$, let $\mathsf{C}[i, j]$ denote the row at index $j\ell_B + i$, yielding $\langle \mathsf{C}[i, j], \mathsf{sk} \rangle = \vec{e}[j\ell_B + i] + B^i \mu\, \mathsf{sk}[j]$.
- $\mathsf{RGSW.Decrypt}(\mathsf{sk}, \mathsf{C}) \to \mu$: inputs $(\mathsf{sk}, \mathsf{C})$, outputs $\big\lfloor \langle \mathsf{C}[\ell_B - 1, 0], \mathsf{sk} \rangle / B^{\ell_B - 1} \big\rceil$ if $\|\vec{e}\| < B^{\ell_B - 1}/2$.
- $\mathsf{RGSW.ExternalProduct}(\mathsf{c}, \mathsf{C}) \to \mathsf{c}'$ (denoted as $\mathsf{c} \boxdot \mathsf{C}$): given $\mathsf{c} = (\mathsf{b}, \mathsf{a}) \in \mathsf{RLWE}_\mathsf{s}(\mu')$ and $\mathsf{C} \in \mathsf{RGSW}_\mathsf{s}(\mu)$, outputs $\mathsf{c} \boxdot \mathsf{C} := \mathsf{G}^{-1}(\mathsf{c}) \cdot \mathsf{C} \in \mathsf{RLWE}_\mathsf{s}(\mu\mu')$ which is

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

  with noise $e_\boxdot = \mu\, e_\mathsf{c} + \big\langle \mathsf{G}^{-1}(\mathsf{c}), \vec{e} \big\rangle$ with

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

  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 ($\le B/2$), 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 $(\mathsf{KS.KeyGen}, \mathsf{KS})$:

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

Automorphisms. Galois automorphisms provide a natural mechanism for permuting plaintext elements in homomorphic encryption, which was proposed in [47], [48], [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 an RLWE ciphertext via the following *ppt* algorithms:

- $\mathsf{Auto.KeyGen}(\mathsf{s}, t) \to \mathsf{ksk}$: on inputs (s, t), outputs $\mathsf{ksk} \leftarrow \mathsf{KS.KeyGen}(\mathsf{s}, \sigma_t(\mathsf{s}))$.
- $\mathsf{Auto.Eval}_t(\mathsf{c}, \mathsf{ksk}) \to \mathsf{c}'$: given $\mathsf{c} = (\mathsf{b}, \mathsf{a})$ and ksk, outputs $\mathsf{c}' := \mathsf{KS}\big((\sigma_t(\mathsf{b}), \sigma_t(\mathsf{a})), \mathsf{ksk}\big) \in \mathsf{RLWE}_\mathsf{s}(\sigma_t(\mu))$.

Appendix C. Deferred Material for Our Construction

C.1. Instantiations of the α-KVS

In this section, we present the 3-KVS [16] (sparse, based on a segmented binary fuse filter), which instantiates the α-KVS of Definition 2 in EncodeDB (Algorithm 1).

Instantiation: the 3-KVS. Fix $\alpha = 3$ and partition the domain $[d]$ into $n_{\mathsf{seg}} := d/\varsigma$ disjoint segments $\mathcal{S}_i := \{i\varsigma, \ldots, (i+1)\varsigma - 1\}$, $i \in [n_{\mathsf{seg}}]$, of a common segment length ς. 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 $\mathsf{sd} \xleftarrow{\$} \{0,1\}^\lambda$ for a pseudorandom function PRF which takes (j, k) as inputs. Define the start segment and, for $j \in [\alpha]$, the positions and the decode coefficients

  $$\mathsf{seg}(k) := \mathsf{PRF}_\mathsf{sd}(0, k) \bmod (n_{\mathsf{seg}} - \alpha + 1),$$
  $$h_j(k) := \big(\mathsf{seg}(k) + j\big)\varsigma + \big(\mathsf{PRF}_\mathsf{sd}(1 + j, k) \bmod \varsigma\big),$$
  $$\rho_j(k) \in \mathbb{Z}_p^\times \text{ from } \mathsf{PRF}_\mathsf{sd}(1 + \alpha + j, k).$$

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

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

  The α segments $\mathcal{S}_{\mathsf{seg}(k)}, \ldots, \mathcal{S}_{\mathsf{seg}(k)+\alpha-1}$ are distinct, so the α positions are mutually distinct and the Hamming weight of $\vec{w}(k)$ is exactly $\alpha = 3$.

The coefficients are public and computed from k by client and server alike; random nonzero decode coefficients are also used by band-based OKVS [25]. They matter for SPIR: with unit coefficients, a queried key $k^* \notin \mathcal{L}$ whose three positions coincide with those of a stored key k would decode exactly to the value of k, which happens with probability about $|\mathcal{L}|/\big((n_{\mathsf{seg}}-\alpha+1)\varsigma^3\big)$ per query. With pseudorandom coefficients, the two decode vectors are dependent only if, in addition, $\vec{\rho}(k^*)$ is proportional to $\vec{\rho}(k)$, which multiplies this probability by about $(p-1)^{-2}$ (Remark 4).

- α-KVS.Encode$(\mathsf{pp}_{\alpha\text{-KVS}}, \mathcal{L})$**:** Solve the sparse linear system $\{\langle \vec{w}(k), \mathbf{D}\rangle = v\}_{(k,v)\in\mathcal{L}}$ 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 with high probability, and $\mathbf{D}$ computable in expected time $O(|\mathcal{L}|)$, whenever the expansion rate satisfies $\eta \geq \eta_3(|\mathcal{L}|)$ for the sizing law given in the supplementary material in our artifact repository; otherwise α-KVS.Encode fails and the seed sd is resampled. The solver follows the standard approach for linear OKVS [24]: positions at which no key is peeled are free and sampled uniformly from $\mathbb{Z}_p$, and the remaining positions are determined by back-substitution. It 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 $h_{j^\star}(k)$ for the position at which k was peeled, assigns

$$\mathbf{D}[h_{j^\star}(k)] := \rho_{j^\star}(k)^{-1} \cdot \Big(v - \textstyle\sum_{j\neq j^\star} \rho_j(k)\, \mathbf{D}[h_j(k)]\Big) \bmod p.$$

  Peeling is purely combinatorial and depends only on the positions, so its success probability is the same as with unit coefficients. Back-substitution inverts only the coefficients $\rho_j(k)$, which are units by construction; hence α-KVS.Encode never requires $\mathbb{Z}_p$ to be a field and imposes no constraint on the plaintext modulus beyond the one of the homomorphic evaluation. The positions not assigned by back-substitution keep their uniform values, so $\mathbf{D}$ is uniform over the solution set. Output $\mathbf{D}$.
- α-KVS.Decode$(\mathsf{pp}_{\alpha\text{-KVS}}, \mathbf{D}, k)$**:** Output $\sum_{j\in[3]} \rho_j(k)\, \mathbf{D}[h_j(k)] \bmod p$.

The Segment Length is a Free Knob. The segment length ς decides only which α positions a key occupies; d, α and the shape of the decode vectors are unchanged, so neither the homomorphic evaluation nor its noise depends on ς. Longer segments, however, spread a key's positions over a wider window and sharply lower the peel-failure probability θ. Since the expansion rate approaches 1.125 only asymptotically, an encoder needs headroom that depends on the list length, obtained by enlarging ς or d. We therefore treat ς as a free parameter rather than fixing it by the standard sizing law [58] (supplementary material; Section 4 reports our value).

C.2. Proof of Theorem 9

Proof Fix DB and a semi-honest client querying $k_1, \ldots, k_q$. For a keyword k, let $\ell_k^\star$ be its bin, which is the same in every block since all blocks share $\mathsf{pp}_{\mathsf{BBH}}$, and if $\mathsf{DB}(k) \neq \perp$ let $\iota(k)$ be the block storing k. We assume, as in our implementation, that blocks are assigned by a public function of the key, so $\iota(k)$ is computable from k. By the correctness argument of Construction 8, the answer to a query on k consists, for every block ι and plane j, of an extracted LWE ciphertext whose plaintext is $V_{\ell_k^\star,j}^{(\iota)} = \langle \vec{w}(k), \mathbf{M}^{(\iota)}[\ell_k^\star L + j, \cdot]\rangle$, and of nothing else. By the noise assumption, the client's view is determined, up to fresh encryption noise, by $\mathsf{pp}_{\mathsf{kPIR}}$, its own keys and queries, and these plaintexts.

Simulator. Sim generates the client keys and queries honestly. For each k_i and each block ι, it sets the plaintexts to the planes of $\mathsf{pl}(k_i, \mathsf{DB}(k_i))$ if $\mathsf{DB}(k_i) \neq \perp$ and $\iota = \iota(k_i)$; otherwise it reuses the values chosen for an earlier query on the same keyword, or samples fresh uniform values in $\mathbb{Z}_p^L$. It then encrypts them under the client's key.

Indistinguishability. In the block storing k_i, the real plaintexts equal the planes of $\mathsf{pl}(k_i, \mathsf{DB}(k_i))$ by correctness. Now fix a block ι, a bin ℓ and a plane j, and let $\mathcal{L}$ be the keys stored in that bin and K the distinct queried keywords with $\ell_k^\star = \ell$ that are not in $\mathcal{L}$. The vector $\mathbf{D}_{\ell,j}^{(\iota)}$ is sampled uniformly from the solutions for $\mathcal{L}$, independently across bins, planes and blocks. If $\{\vec{w}(k)\}_{k\in\mathcal{L}\cup K}$ are linearly independent, Remark 4 implies that $(\langle \vec{w}(k), \mathbf{D}_{\ell,j}^{(\iota)}\rangle)_{k\in K}$ is uniform and independent of the values in $\mathcal{L}$, which is exactly the simulated distribution. Repeated queries on the same keyword return the same value in both worlds. Hence, as long as independence holds, the real and the simulated views are indistinguishable under the noise assumption. Each query adds at most one new keyword per block; as the client issues $q \leq \bar{\nu}$ queries, every bin holds at most $\bar{\nu}$ of its keywords, whichever keywords it chooses, and adding one breaks independence with probability at most $\varepsilon_{\mathsf{dep}}$; a union bound over the q queries and $\mathsf{N}_{\mathsf{blk}}$ blocks shows that independence fails with probability at most $q\mathsf{N}_{\mathsf{blk}}\,\varepsilon_{\mathsf{dep}}$. A re-encoding samples every $\mathbf{D}_{\ell,j}^{(\iota)}$ afresh, independently of earlier encodings, so the argument applies to each encoding separately, and the simulation errors of the encodings add up.

Appendix D. Formal Derivations of the Offline Precomputing Optimizations

This appendix derives the offline/online precomputations of Section 3.5.

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 $u_t = \mathsf{coef}^{-1}(\mathbf{M}[\cdot, t])$ alone. Since $\{c_\sigma\}$ is a function of $\mathbf{M}$ and nothing else, the entire rotated family $\{\tilde{c}_{\sigma_g,\sigma_b}\}_{(\sigma_g,\sigma_b)\in\Sigma_{\mathsf{giant}}\times\Sigma_{\mathsf{baby}}}$, 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 $\mathsf{q}_1 = (\mathsf{b}_1, \mathsf{a}_1) = (\mathsf{b}_1, 0) + (0, \mathsf{a}_1)$ and applying linearity,

$$\mathsf{c}^\star = \underbrace{\mathsf{BSGSEval}\big((0,\mathsf{a}_1), \{c_\sigma\}, \mathsf{evk}_g, \mathsf{evk}_{g^{M_2}}, \mathsf{evk}_{-1}\big)}_{\text{keyword-independent}} + \underbrace{\mathsf{BSGSEval}\big((\mathsf{b}_1,0), \{c_\sigma\}, \mathsf{evk}_g, \mathsf{evk}_{g^{M_2}}, \mathsf{evk}_{-1}\big)}_{\text{keyword-dependent}}.$$

The keyword-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 cache $(\mathsf{c}^\star_{0,\mathsf{off}}, \mathsf{c}^\star_1) \in \mathcal{R}_q^2$. The key-dependent branch then reduces to pure ring arithmetic:

$$\mathsf{BSGSEval}\big((\mathsf{b}_1,0), \{c_\sigma\}, \mathsf{evk}_g, \mathsf{evk}_{g^{M_2}}, \mathsf{evk}_{-1}\big) = (\mathsf{c}^\star_{0,\mathsf{on}}, 0),$$

$$\mathsf{c}^\star_{0,\mathsf{on}} := m^{-1} \sum_{\sigma_g\in\Sigma_{\mathsf{giant}}} \sigma_g\Big(\sum_{\sigma_b\in\Sigma_{\mathsf{baby}}} \tilde{c}_{\sigma_g,\sigma_b}\cdot\sigma_b(\mathsf{b}_1)\Big),$$

and the server recombines $\mathsf{c}^\star = (\mathsf{c}^\star_{0,\mathsf{off}} + \mathsf{c}^\star_{0,\mathsf{on}}, \mathsf{c}^\star_1)$. 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. These are 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 $\mathsf{c}^\star_0 = \mathsf{c}^\star_{0,\mathsf{off}} + \mathsf{c}^\star_{0,\mathsf{on}}$. 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

$$\vec{d}^{\,\mathsf{off}} := \mathsf{G}^{-1}(\mathsf{c}^\star_{0,\mathsf{off}}), \qquad \vec{d}^{\,\mathsf{on}} := \mathsf{G}^{-1}(\mathsf{c}^\star_{0,\mathsf{on}}),$$

the sum $\vec{d}^{\,\mathsf{off}} + \vec{d}^{\,\mathsf{on}}$ forms a valid non-canonical decomposition of $\mathsf{c}^\star_0$ with digit norm bounded by B. Substituting this into Equation (5) yields $\mathsf{T} = \mathsf{T}^{\mathsf{pre}} + \mathsf{T}^{\mathsf{on}}$, grouping the terms by whether their digits can be precomputed offline:

$$\mathsf{T}^{\mathsf{pre}} := \sum_{i\in[\ell_B]} d_i^{\mathsf{off}} \cdot \mathsf{q}_2[i,0] + \sum_{i\in[\ell_B]} \mathsf{G}^{-1}(\mathsf{c}^\star_1)_i \cdot \mathsf{q}_2[i,1], \tag{8}$$

$$\mathsf{T}^{\mathsf{on}} := \sum_{i\in[\ell_B]} d_i^{\mathsf{on}} \cdot \mathsf{q}_2[i,0]. \tag{9}$$

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 are more expensive than multiplications in this context, this trade yields a saving. This shows that the split also applies to Dimension 2. 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., $\Theta(m^2)$.

The noise analysis of the two-dimensional homomorphic evaluation, the parameter selection together with its derivations, and the per-cell parameters and additional evaluation plots are provided in the supplementary material in our artifact repository.

TABLE 5. HINTLESS INDEX PIR ON A 1 GB DATABASE, AS REPORTED IN TABLE 4 OF [23]; RECORD SIZES IN PARENTHESES.

	HintlessPIR (1 B, 32 KB)	YPIR (1 B)	YPIR+SP (32 KB)	KsPIR (8 KB)
Query	443 KB	846 KB	686 KB	140 KB
Response	3008 KB	12 KB	120 KB	26 KB
Client comp.	3958 ms	2050 ms	1223 ms	7.6 ms
Server resp.	866 ms	178 ms	426 ms	801 ms

Appendix E. Client-Side Costs of Hintless Index PIR

Table 5 reproduces the client-relevant metrics reported by [23] for a 1 GB database. Timings come from different implementations and record sizes and are therefore indicative, but the client-side quantities are largely implementation-independent: KsPIR has the smallest query, a small response and milliseconds of client computation, which is why we build on it.