KPIR-C: Keyword PIR with Arbitrary Server-Side Computation

Ali Arastehfard*, Weiran Liu†, Qixian Zhou‡, Zinan Shen§, Liqiang Peng†, Lin Qu†, Shuya Feng¶, Yuan Hong*
*University of Connecticut, †Alibaba Group, ‡Ant International, §Peking University, ¶University of Alabama at Birmingham
{ali.arastehfard,yuan.hong}@uconn.edu; {weiran.lwr,plq270998}@alibaba-inc.com; xide.ql@taobao.com;
qixian.zqx@ant-intl.com; sznnzs@pku.edu.cn; fengs@uab.edu

***Abstract*—Keyword Private Information Retrieval (KPIR) enables clients to retrieve database entries by keyword without revealing their query, with many efficient schemes relying on Fully Homomorphic Encryption (FHE). However, existing FHE-based KPIR schemes fix parameters to a bounded computation depth, which becomes a bottleneck in privacy-sensitive settings such as distributed ledger systems and private web search, where databases grow dynamically and clients are lightweight: exceeding this bound incurs significant overhead, forces larger parameters, and leaves any post-retrieval computation to the client or to communication-heavy interactive protocols.**

We introduce Keyword PIR with Computation (KPIR-C), a new paradigm that enables arbitrary, non-interactive server-side computation on query responses while preserving query privacy. Unlike prior designs, KPIR-C fixes parameters independently of database size or computation depth by leveraging bootstrapping via a co-design of Torus Fully Homomorphic Encryption (TFHE) and KPIR that keeps bootstrapping overhead practical. We further introduce Keyword PIR with Default (KPIR-D) to handle mismatched queries non-interactively. We instantiate KPIR-C with two constructions: one based on equality circuits and constant-weight codes, and one based on Learning With Errors (LWE)-based KPIR, offering complementary communication-runtime trade-offs. We demonstrate experimentally that both achieve competitive performance compared to state-of-the-art KPIR schemes that do not support arbitrary computation.[1]

I. Introduction

Private Information Retrieval (PIR) [1] enables clients to retrieve data entries from a server-hosted database without revealing any information about the requested entry. PIR offers a variety of applications, such as anonymous messaging [2], private contact discovery [3], private blocklist lookups [4], and private Wikipedia access [5], and has been recently deployed in real-world systems including password breach checkups [6], private web search [7], and live caller ID lookups [8].

In its original form, PIR requires the server to organize its database as an array, enabling the client to retrieve entries by index. This formulation is therefore commonly referred to as *index PIR*. However, this abstraction differs from real-world databases, where entries are typically indexed by arbitrary strings such as keywords. This limitation motivates a variant of PIR known as *keyword PIR* (KPIR) [9]. Beyond its primary application in private data retrieval, KPIR has increasingly been used as a building block in settings requiring privacy-preserving post-retrieval computation [10], [11], [12]. In these settings, the client and server first run a KPIR protocol to retrieve the matching encrypted records, then evaluate functions over the retrieved data via either general-purpose two-party computation (2PC) [13] or protocols tailored to specific tasks [14], [15]. 2PC-based computation, however, incurs high communication costs and requires continued client participation (i.e., it is interactive), which is undesirable for lightweight clients with limited bandwidth, such as those in distributed ledgers [16], [17]. Private web search is a similar case: the client evaluates a top-k function over the retrieved results, but current systems [7] rely on communication-heavy multi-round protocols, imposing significant overhead on bandwidth-constrained clients.

Since Fully Homomorphic Encryption (FHE) in theory supports arbitrary functions over encrypted data [18], a natural alternative is to outsource all post-retrieval computation on the encrypted response to the server, replacing the interactive 2PC protocols entirely. Most recent efficient KPIR solutions are notably based on FHE [19], [20], [21]. Yet existing FHE-based KPIR schemes support only limited computation depth: bootstrapping—necessary to control noise growth in FHE ciphertexts—remains prohibitively expensive beyond shallow circuits, so prior works [19], [22], [23] instead fix the computation depth in advance (e.g., by upper-bounding the number of database entries) and tune their FHE parameters accordingly. This limits the *expressiveness* of these schemes: they cannot support arbitrary post-retrieval computation, and even for retrieval itself, they cannot accommodate arbitrary database sizes without significant overhead. A concrete example is the Bitcoin transaction ledger (exceeding 2^{30} entries as of 2026[2]), which grows over time and can easily exceed the pre-defined depth of existing schemes [19], [22]; once exceeded, these schemes fail to return correct results, and recovery requires re-tuning parameters, re-processing the whole database, and re-running the entire protocol. Moreover, provisioning

[1]Code is publicly available at https://github.com/datasec-lab/kpirc.

[2]https://www.blockchain.com/explorer/charts/n-transactions-total

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.230124
www.ndss-symposium.org

for deeper computation forces larger parameters, substantially increasing per-operation runtime and communication overhead.

Motivated by these challenges, we introduce *Keyword PIR with Computation* (KPIR-C), a new paradigm that enables the server to perform *non-interactive, expressive computation* over KPIR responses via a database-independent setup phase, while preserving query privacy. We illustrate this functionality through two representative examples.

1) *Retrieving Aggregated Results from Multiple Bitcoin Accounts*: A lightweight Bitcoin client with limited bandwidth and computational resources uses a KPIR-based Bitcoin account lookup service [17] to privately fetch the total balance across several accounts.[3] Instead of receiving and decrypting a separate response per account, KPIR-C enables the server to privately retrieve the matching entries, aggregate their encrypted values, and return only a single encrypted result. The aggregation can further be replaced by more complex server-side computations, such as computing the average balance, applying a threshold function, retrieving the top-k accounts by balance, or performing anti-money-laundering (AML) checks.
2) *Predicate KPIR*: A client wants to know whether a retrieved entry satisfies a predicate. With standard KPIR, the client retrieves the entire entry and evaluates the predicate locally, wasting bandwidth when the entry is large but the result is small (often a single bit); alternatively, the server can precompute the predicate for all entries and expose the results through KPIR, but this is unsuitable when the predicate is specified at query time, since the server must evaluate it over the entire database per query, adding linear overhead. KPIR-C enables the server to privately retrieve the matching entry, evaluate the query-time predicate only on the encrypted result, and return an encrypted bit, with post-retrieval cost independent of database size.

A. Our Contributions

- **KPIR-C: Keyword PIR with Computation.** We formalize KPIR-C's correctness and security requirements, extending KPIR with an evaluation phase for server-side post-retrieval computation, and introduce Keyword PIR with Default (KPIR-D) as a subroutine ensuring well-defined responses under query mismatches.
- **Database-independent setup.** We decouple the setup parameters of KPIR-C from database cardinality, allowing the same setup to support *dynamically growing databases* without re-parametrization. Consequently, KPIR-C's per-entry overhead remains unaffected by database cardinality.
- **KPIR-specific bootstrapping strategy.** We instantiate KPIR-C using Torus Fully Homomorphic Encryption (TFHE); a direct adaptation, however, incurs unacceptable overhead, rendering the KPIR component of KPIR-C impractical (Table III). We therefore introduce a *KPIR-specific bootstrapping strategy* (Section V-E) that exploits KPIR's structure to significantly reduce the number of bootstrapping operations, keeping our KPIR phase competitive with prior schemes optimized for shallow circuits.
- **Two complementary constructions.** We design two KPIR-C constructions targeting distinct communication–computation trade-offs (Section II-B). Our first construction reduces communication by up to $10\times$, favoring bandwidth-constrained clients, with only $1.2\times$ runtime overhead. Our second construction achieves a $60\times$ runtime speedup (over $14\times$ faster online latency in its offline–online variant) on databases with 2^{16} entries of 8 B each, while maintaining a comparable download cost to the first construction.

[3] See https://github.com/blyssprivacy/clients as an example.

II. Design Overview and Background

The expressiveness and non-interactivity requirements of KPIR-C give rise to three distinct design challenges, which we discuss next.

A. Challenges in Constructing KPIR-C

Challenge 1 (Bootstrapping Efficiency). Supporting non-interactive post-retrieval computation requires a fully homomorphic encryption (FHE)-based design. However, existing KPIR schemes are designed to support shallow circuit depths and are thus either built on leveled homomorphic encryption schemes such as Brakerski/Fan–Vercauteren (BFV) [27], [28] or Brakerski–Gentry–Vaikuntanathan (BGV) [29], or directly constructed on Learning With Errors (LWE) or Ring-LWE (RLWE) assumptions (see Table I). In contrast, KPIR-C does not bound the depth of the post-retrieval computation, which requires *bootstrapping* [18]—a costly operation that refreshes ciphertexts and controls noise growth so that computation can proceed indefinitely. While BGV/BFV schemes can in principle be bootstrapped, doing so is prohibitively expensive in practice: a single operation takes approximately 35 seconds [30].

Third-generation FHE schemes such as Gentry–Sahai–Waters (GSW) [31], FHEW [32], and TFHE [33], [34] offer significantly faster bootstrapping. In particular, TFHE bootstraps a single ciphertext in about 10 milliseconds, making it a promising candidate for KPIR-C. To the best of our knowledge, no prior (K)PIR scheme has been built on TFHE, and the efficiency of TFHE-based (K)PIR remains largely unexplored.

Challenge 2 (Bootstrapping-Driven KPIR Design). The expressiveness requirement of KPIR-C means that the supported computation cannot be tied to a predetermined circuit depth. This requirement also extends to the KPIR phase itself, whose depth can grow with the database size. In particular, constructing the KPIR response requires aggregating ciphertext contributions across database entries; therefore, fixing a maximum database size would effectively impose a predetermined bound on the KPIR circuit. As the database grows, the number of ciphertexts aggregated within the KPIR circuit increases, which increases the accumulated noise and can eventually exhaust the available noise budget. Bootstrapping must therefore be performed within the KPIR circuit itself to preserve correctness.

TFHE makes such bootstrapping efficient, but its fast-bootstrapping parameter regime makes this requirement more

TABLE I
COMPARISON OF EXISTING KPIR SCHEMES, RELATED RETRIEVAL APPROACHES, AND OUR KPIR-C CONSTRUCTIONS.

Related Schemes	KPIR/Retrieval Strategy	FHE Scheme/ Assumption	Return on Mismatch	Arbitrary Computation
LabelledPSI [24]	Polynomial Interpolation	BFV/FV	A random value	✗
MulPIR [21], SealPIR [20]	Key-to-Index (Cuckoo hashing)	BFV/FV	A random value	✗
ChalametPIR [22], Hao et al. [25]	Key-to-Index (BFFs/PLA), with Hints	LWE/RLWE	A random value	✗
CWKPIR [19], Pantheon [26]	Hom. Eq. (CW encoding/Fermat's Little Theorem)	BFV/FV	Always zero	✗
Hao et al. [12]†	OKVR (sparse OKVS + batch PIR)	BFV/OPRF	A random value	✓ (Interactive)
TCWKPIR (ours)	Hom. Eq. (CW encoding)	TFHE	**Server-specified default**	✓ **(Non-interactive)**
TCAKPIR (ours)	Key-to-Index (BFFs), Hintless	TFHE	**Server-specified default**	✓ **(Non-interactive)**

†We include Hao et al. [12] for completeness because their oblivious key-value retrieval (OKVR) primitive is described as a batched variant of symmetric KPIR. However, their work targets unbalanced circuit private set intersection (PSI) rather than KPIR, and arbitrary computation requires subsequent interactive 2PC, whereas our KPIR-C constructions support arbitrary non-interactive server-side computation on the retrieved response.

pronounced. TFHE employs relatively small ciphertext moduli and LWE dimensions to achieve fast bootstrapping [35], which leaves a relatively limited noise margin and therefore requires frequent bootstrapping as noise accumulates during aggregation. Its bootstrapping efficiency also degrades with plaintext precision, since supporting higher-precision messages requires larger bootstrapping parameters [36]. Practical parameterizations therefore provide limited plaintext space, which leaves little room for intermediate carry values; once the available carry space is exhausted, bootstrapping is required to clear the accumulated carry before aggregation can continue [37]. Thus, although each bootstrapping operation is relatively inexpensive, a black-box use of TFHE requires frequent bootstrapping for both noise and carry management, and their cumulative cost renders the protocol impractical (see Table III).

As a solution, we propose a **paradigm shift** from circuit-driven to bootstrapping-driven KPIR design. Rather than provisioning parameters for a predetermined circuit depth, we fix a database-independent parameter set and optimize when bootstrapping is performed. Concretely, we introduce a *KPIR-specific bootstrapping strategy* that exploits the structural properties of KPIR to minimize this cost (see Section V-E). Prior designs, in contrast, must provision larger parameters as the supported circuit depth increases, which substantially increases per-entry runtime and upload costs (see Section VII-A).

Challenge 3 (Mismatch Handling). In standard KPIR, the response on a mismatch—where the client's queried keyword is absent from the database—is scheme-dependent (see Table I), but the client can always detect it via a scheme-specific predicate and treat it as $\perp$. For instance, schemes based on polynomial interpolation or key-to-index mappings [24], [21], [20], [22] return an arbitrary value on a mismatch, while equality-based schemes [19], [26] return 0; in both cases, the client performs *mismatch detection* and treats the response as $\perp$ accordingly. In KPIR-C, however, the challenge goes beyond *mismatch detection*: since the server evaluates post-retrieval computation on encrypted responses without client interaction, a mismatch must produce a well-defined default value δ determined by the post-retrieval functionality rather than by the underlying KPIR scheme; otherwise, the post-retrieval computation receives an arbitrary input (what we refer to as *ill-defined*), silently corrupting its output.

In Section IV, we formalize KPIR-C along with its correctness and security properties, and introduce KPIR-D as a subroutine for resolving the third challenge by enforcing well-defined inputs to the post-retrieval computation on mismatches.

B. Two Instantiations for KPIR-C

Having identified these challenges, we next describe how they shape our two instantiations of KPIR-C. We adopt TFHE [38] as the underlying FHE scheme and employ our KPIR-specific bootstrapping strategy to efficiently support computation within the KPIR phase. For mismatch handling, the server obtains an encrypted indicator bit that signals whether a match exists and uses it to select between the encrypted KPIR response and a default value via a multiplexer operation. While existing solutions adopt this mechanism in an interactive setting [14], our constructions realize it non-interactively.

A Communication-efficient Design: Torus Constant-Weight KPIR (TCWKPIR). One approach is to build on equality-based KPIR constructions that achieve optimal communication complexity and inherently return 0 on mismatches (see Table I), which can serve as an indicator bit for KPIR-D (Section IV-A). Among existing candidates [19], [26], CWKPIR [19] is the only viable option,[4] but it lacks native support for well-defined, scheme-independent default values and is built on BFV. We therefore adapt the constant-weight encoding to construct TCWKPIR over the torus (see Section V-A).

This construction achieves a query size of $O(\rho)$, where $\rho \ll m$ denotes the codeword length and m the database cardinality, at the cost of one homomorphic equality test per database entry, which requires multiplying $h-1$ encrypted bits, where h denotes the Hamming weight. Since TFHE does not natively support ciphertext-ciphertext multiplication, a standard realization of each such multiplication requires two programmable bootstrapping (PBS) operations, a variant of bootstrapping that evaluates a function while refreshing the ciphertext (see Section III-E). However, because each operand encrypts a single bit, the two inputs can be packed into a single ciphertext and their multiplication evaluated as a bivariate function using a single PBS operation [39]. This optimization keeps the TFHE realization computationally practical. Overall,

[4]Pantheon [26] is incompatible with TFHE's power-of-two torus modulus

this construction is best suited for settings where the database is not excessively sparse or where the client has limited bandwidth but the server has ample computational resources, as the server-side workload can be parallelized (see Appendix A, Table VII).

Trading Computation for Communication: Torus Client-Aided KPIR (TCAKPIR). An alternative approach is to build on Key-to-Index KPIR constructions (see Table I), where the client constructs the selection vector in plaintext via a key-to-index mapping, encrypts it, and sends it to the server. For KPIR-C, however, the server must additionally obtain an encrypted indicator bit for KPIR-D (Section IV-A). Adapting Cuckoo hashing-based KPIR schemes [21], [20] would require κ homomorphic equality checks and $\kappa - 1$ homomorphic AND operations to obtain this indicator, where κ denotes the number of hash functions. Similarly, adapting the Piecewise Linear Approximation (PLA)-based KPIR scheme [25] (built on the SimplePIR paradigm [40]) would require $O(\sqrt{m})$ homomorphic equality checks and AND operations, where m denotes the database cardinality. Both approaches are prohibitively expensive.

We therefore adopt a construction based on Binary Fuse Filter (BFF) [22], which requires a single PIR invocation and equality check. Although this increases the query size to $O(m)$, the concrete communication remains acceptable with LWE-based encryption [41]. Since Torus LWE (TLWE) ciphertexts (Section III-E) are a variant of LWE ciphertexts, this enables a TFHE-based KPIR following the LWE-based paradigm [42], [22], yielding our second construction TCAKPIR (Section V-B). However, unlike ChalametPIR [22], we cannot reuse its client-side hint precomputation to reduce online cost: the KPIR-C computation may only be known online, while our evaluation key is fixed per client (Section V-B). We therefore introduce TCAKPIR$^+$, an optimized variant with online efficiency comparable to ChalametPIR, without extra communication.

III. Preliminaries

A. Notations

For convenience, a consolidated reference for the common notations and terminologies appears in Table VI (Appendix A).

B. Keyword Private Information Retrieval

We recall the definition of KPIR [9] in the standard PIR setting. The server first runs a query-independent, database-dependent setup algorithm that generates a set of public parameters pp shared with all clients. As in most PIR schemes [20], [21], [19], each client also runs a query-independent, database-independent setup algorithm, which outputs a query key qk known only to the client. The same pp and qk can be reused across multiple queries, thereby amortizing the setup cost across many PIR queries.

Let $\mathsf{DB} \in (\mathcal{K} \times \mathcal{V})^m$ be a key-value database of m pairs. A KPIR scheme consists of the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB})$: On input the security parameter λ and a key-value database DB, the setup algorithm outputs a set of public parameters pp.
- $\mathsf{qk} \leftarrow \mathsf{ClientSetup}(\mathsf{pp})$: On input the public parameters pp, the client setup algorithm outputs a query key qk.
- $\overline{\mathsf{DB}} \leftarrow \mathsf{Encode}(\mathsf{DB})$: On input a key-value database DB, outputs an encoded database $\overline{\mathsf{DB}}$.
- $q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw})$: On input the public parameters pp, the client's query key qk, and a keyword kw, the query algorithm outputs a query q.
- $r \leftarrow \mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, q)$: On input the public parameters pp, the encoded database $\overline{\mathsf{DB}}$, and a query q, the response algorithm outputs a response r.
- $\{v, \perp\} \leftarrow \mathsf{Recover}(\mathsf{qk}, r)$: On input the query key qk and a response r, the recover algorithm outputs a value v or a special symbol $\perp$ in case of a mismatch.

Correctness. A KPIR scheme is correct if honest execution returns the matching value or $\perp$ on a mismatch. Formally, a KPIR scheme has negligible correctness error $\mathsf{negl}(\lambda)$ if, for every database $\mathsf{DB} := \{(k_i, v_i)\}_{i\in[m]} \in (\mathcal{K}\times\mathcal{V})^m$ and keyword $\mathsf{kw} \in \mathcal{K}$, the following probability is at least $1 - \mathsf{negl}(\lambda)$:

$$\Pr\left[v = \begin{cases} v_i & \mathsf{kw} = k_i \\ \perp & \text{otherwise} \end{cases} \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB}) \\ \mathsf{qk} \leftarrow \mathsf{ClientSetup}(\mathsf{pp}) \\ \overline{\mathsf{DB}} \leftarrow \mathsf{Encode}(\mathsf{DB}) \\ q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}) \\ r \leftarrow \mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, q) \\ v \leftarrow \mathsf{Recover}(\mathsf{qk}, r) \end{array} \right]$$

Security. The client's query should not reveal any information about the keyword being retrieved, which is captured by the *query indistinguishability* notion. Formally, a KPIR scheme satisfies *query indistinguishability* if, for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$ and for all $\mathsf{kw}_i, \mathsf{kw}_j \in \mathcal{K}$,

$$\Big| \Pr[\mathcal{A}(1^\lambda, q) = 1 : q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_i)] - \Pr[\mathcal{A}(1^\lambda, q) = 1 : q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_j)] \Big| \leq \mathsf{negl}(\lambda).$$

Efficiency. For a KPIR scheme to be non-trivial, it should be more communication-efficient than the naive scheme of downloading the entire key-value database.

The efficiency bottleneck in a non-trivial KPIR scheme lies in enabling the server to acquire a selection vector sv. In its basic form, sv is an encrypted binary vector that allows the server to obliviously select database entries matching the client's query kw, such that $\sum_{v_i \in \mathsf{DB}} \mathsf{sv}_i v_i = v$ whenever there exists a matching entry $(k, v) \in \mathsf{DB}$ (i.e., $\mathsf{kw} = k$). This selection vector can be constructed either by the client, at the expense of communication (Section V-B), or outsourced to the server, at the expense of computation (Section V-A).

C. Constant-Weight Code

A constant-weight code is an error-detecting code where all codewords have the same Hamming weight. Mahdavi and Kerschbaum [19] were the first to apply constant-weight codes in KPIR. They designed an encoding algorithm (hereafter denoted as $\mathsf{CWEncode}$; see Appendix A, Algorithm 3) that encodes any keyword with a maximum bit length of ℓ_k into a constant-weight code c with a codeword length of ρ and a Hamming weight of h, such that $\binom{\rho}{h} \geq 2^{\ell_k}$. Crucially, equality

between two constant-weight codes can be checked efficiently, reducing multiplicative depth from $O(\log \ell_k)$ to $O(h)$. We leverage this encoding in Section V-A to let the server construct the selection vector efficiently.

D. Key-Value Stores

A key-value store (KVS) [43], [44] encodes a set $\mathcal{M}$ of m key-value pairs $(k, v) \in \mathcal{K} \times \mathcal{V}$ into an output of length M using κ hash functions. It consists of two algorithms:

- $(F, H) \leftarrow \mathsf{write}(\mathcal{M})$: On input $\mathcal{M} := \{(k_i, v_i)\}_{i\in[m]}$, outputs a vector F and hash functions $H := \{h_i\}_{i\in[\kappa]}$.
- $v \leftarrow \mathsf{recon}(H, k, F[H(k)])$: On input a key k and entries $F[H(k)] := \{F[h_i(k)]\}_{i\in[\kappa]}$, outputs $v \in \mathcal{V} \cup \{\perp\}$.

Thus, recon accesses κ entries of F to reconstruct a value. We rely on the standard KVS correctness properties of inclusion and non-inclusion [22]: reconstruction returns the associated value for a stored key and $\perp$ with overwhelming probability for a non-member key.

Instantiation from Binary Fuse Filters. A Binary Fuse Filter (BFF) [45] instantiates a KVS with $\kappa := 3$ hash functions and output length $M \leq 1.156m$, yielding a compact encoding vector. We denote this instantiation by `BFF.write` and use it to construct the selection vector client-side in Section V-B.

E. Torus Fully Homomorphic Encryption (TFHE)

TFHE is a third-generation FHE scheme building on GSW [31] and FHEW [32]. Its key feature is efficient programmable bootstrapping, which evaluates functions while refreshing ciphertexts. TFHE operates over the torus $(\mathbb{T}, +)$, an Abelian group whose elements can be represented by real numbers in $[0, 1)$ modulo 1, i.e., $\mathbb{T} := \mathbb{R}/\mathbb{Z}$. In practice, torus elements are represented with finite precision using $\Omega \in \{32, 64\}$ bits. In binary form, $t := \sum_{i=1}^{\Omega} t_i 2^{-i} \bmod 1$, where $t_i \in \{0, 1\}$. This yields the discretized torus $\mathbb{T}_q := q^{-1}\mathbb{Z}/\mathbb{Z} \subseteq \mathbb{T}$, where $q := 2^{\Omega}$, whose elements take the form $\frac{i}{q} \bmod 1$ for $i \in \mathbb{Z}$.

TFHE supports several ciphertext types. We focus on TLWE ciphertexts and defer the remaining types and naming conventions to Appendix C.

TLWE Ciphertexts. Let $\mathsf{pp} := (n, \sigma, p, q)$ be the public parameters under security parameter λ, where n is the LWE dimension, σ is the noise standard deviation, and $p \mid q$ are the plaintext and ciphertext moduli. The TFHE scheme over TLWE ciphertexts supports the following core algorithms:

- $(\mathbf{s}, \mathsf{evk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$: On input public parameters pp, returns a secret key $\mathbf{s} := (s_1, \ldots, s_n) \leftarrow\$\ \mathbb{B}^n$ and an evaluation key evk.
- $\mu \leftarrow \mathsf{Encode}(v, \mathsf{pp})$: On input $v \in \mathbb{Z}$ and pp, returns $\mu := \frac{v \bmod p}{p} \in \mathbb{T}_p \subseteq \mathbb{T}_q$.
- $\mathsf{ct} \leftarrow \mathsf{Encrypt}_{\mathbf{s}}(\mu, \mathsf{pp})$: On input $\mu \in \mathbb{T}_p$ and pp, defines a Gaussian error distribution $\chi := \mathcal{N}(0, \sigma^2)$ over $q^{-1}\mathbb{Z}$. Samples $\mathbf{a} := (a_1, \ldots, a_n) \leftarrow\$\ \mathbb{T}_q^n$ and $e \leftarrow \chi$, and returns $\mathsf{ct} := (\mathbf{a}, b) \in \mathbb{T}_q^{n+1}$, where $b := \sum_{i=1}^{n} s_i \cdot a_i + \mu + e$. The ciphertext is denoted $\mathrm{TLWE}_{\mathbf{s}}(\mu)$.
- $\mu \leftarrow \mathsf{Decrypt}_{\mathbf{s}}(\mathsf{ct}, \mathsf{pp})$: On input $\mathsf{ct} := \mathrm{TLWE}_{\mathbf{s}}(\mu)$ and pp, computes $\mu^* := b - \sum_{i=1}^{n} s_i \cdot a_i \in \mathbb{T}_q$ and returns $\mu := (\lfloor p \cdot \mu^* \rceil \bmod p)/p \in \mathbb{T}_p$.
- $v \leftarrow \mathsf{Decode}(\mu, \mathsf{pp})$: On input $\mu \in \mathbb{T}_p$ and pp, returns $v := \lfloor \mu \cdot p \rceil \bmod p \in \mathbb{Z}$.
- $\mathrm{TLWE}_{\mathbf{s}}(f(\mu)) \leftarrow \mathsf{PBS}_{\mathsf{evk}}(f, \mathrm{TLWE}_{\mathbf{s}}(\mu))$: On input a function $f : \mathbb{T}_p \to \mathbb{T}_p$ and ciphertext $\mathsf{ct} := \mathrm{TLWE}_{\mathbf{s}}(\mu)$ with noise e, returns $\mathsf{ct}' := \mathrm{TLWE}_{\mathbf{s}}(f(\mu))$ with reduced noise $e' < e$.
- $\mathrm{TLWE}_{\mathbf{s}}(\mu) \leftarrow \mathsf{Bootstrap}_{\mathsf{evk}}(\mathrm{TLWE}_{\mathbf{s}}(\mu))$: On input $\mathsf{ct} := \mathrm{TLWE}_{\mathbf{s}}(\mu)$, returns $\mathsf{PBS}_{\mathsf{evk}}(f, \mathsf{ct})$ with f as the identity.
- $\mathrm{TLWE}(\mu_1 + \mu_2) \leftarrow \mathsf{HomAdd}(\mathrm{TLWE}(\mu_1), \mathrm{TLWE}(\mu_2))$: On input TLWEs of $\mu_1, \mu_2 \in \mathbb{T}_p$, returns $\mathrm{TLWE}(\mu_1 + \mu_2)$.
- $\mathrm{TLWE}(z \cdot \mu) \leftarrow \mathsf{ScalarMul}(z, \mathrm{TLWE}(\mu))$: On input $z \in \mathbb{Z}$ and a TLWE ciphertext of $\mu \in \mathbb{T}_p$, returns $\mathrm{TLWE}(z \cdot \mu)$.

Remark 1. *While* $\mathsf{PBS}_{\mathsf{evk}}(f, \mathsf{ct})$ *is defined for univariate functions, it can be used to evaluate multivariate functions through functional circuits. By Kolmogorov's superposition theorem [46], any multivariate function can be expressed as a linear combination of univariate functions. Consequently, using* HomAdd *and PBS, we can implement arbitrary multivariate functions. For a multivariate function* $\mathcal{F}$ *and a vector of ciphertexts* $\mathbf{c}$*, we write* $\mathcal{F}(\mathbf{c})$ *to denote its homomorphic evaluation using these operations.*

PBS thus enables the evaluation of circuits such as HomAND (homomorphic bitwise AND), HomMul (homomorphic multiplication), and HomEq (homomorphic equality check) [33]. See Appendix C for details on TFHE's bootstrapping procedure.

IV. Formalization of KPIR-C

In this section, we formally define KPIR with Computation (KPIR-C), which extends the standard KPIR definition by allowing the server to perform expressive computation over the KPIR response non-interactively. However, to address the mismatch challenge (Section II-A, Challenge 3), we first introduce a subroutine called KPIR with Default (KPIR-D).

A. KPIR with Default

To support subsequent computation over the response r, we introduce *KPIR with Default* (KPIR-D), which ensures a well-defined output for both matched and mismatched queries in the KPIR phase.[5]

To allow KPIR-D functionality, we assume that the server chooses a default value $\delta \in \mathcal{V}$. We then modify the Recover algorithm by replacing $\perp$ with δ. In addition, the server must have the capability to modify the response r for a query q from a designated client in the mismatched case without the involvement of the client. To achieve this, the client shares an evaluation key evk corresponding to its query key qk with the server in the ClientSetup phase. Finally, the Setup algorithm is made **database independent**, which allows for dynamic database updates without requiring a new setup phase.

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^{\lambda})$: On input the security parameter λ, the setup algorithm outputs public parameters pp independent of the database DB.

[5]While Lepoint et al. [14] propose this concept in an *interactive* setting, KPIR-D achieves this *non-interactively*.

- $(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp})$: On input the public parameters pp, the client setup algorithm outputs an evaluation key evk and a query key qk.
- $r \leftarrow \mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, \mathsf{evk}, \delta, q)$: On input the public parameters pp, an encoded database $\overline{\mathsf{DB}}$, a query q from a client, and its evaluation key evk, the response algorithm outputs a response r (that encrypts δ in the mismatched case).

Correctness. The correctness should ensure that the KPIR output is the value corresponding to the queried keyword in the matched case, and the default value δ in the mismatched case with overwhelming probability. Formally, the following probability is at least $1 - \mathsf{negl}(\lambda)$:

$$\Pr\left[v = \begin{cases} v_i & \mathsf{kw} = k_i \\ \delta & \text{otherwise} \end{cases} \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp}) \\ \overline{\mathsf{DB}} \leftarrow \mathsf{Encode}(\mathsf{pp}, \mathsf{DB}) \\ q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}) \\ r \leftarrow \mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, \mathsf{evk}, \delta, q) \\ v \leftarrow \mathsf{Recover}(\mathsf{qk}, r) \end{array}\right]$$

Security. In addition to the query-indistinguishability requirement of KPIR (Section III-B), KPIR-D requires that the server cannot distinguish a mismatch from a match. Formally, for all PPT adversaries $\mathcal{A}$, for any $\mathsf{kw}_i \in \mathsf{DB.keys}$, $\mathsf{kw}_j \notin \mathsf{DB.keys}$, and any $\delta \in \mathcal{V}$,

$$\Big|\Pr\big[\mathcal{A}(1^\lambda, \mathsf{evk}, q, r) = 1 \mid \mathsf{Exp}_i\big] - \Pr\big[\mathcal{A}(1^\lambda, \mathsf{evk}, q, r) = 1 \mid \mathsf{Exp}_j\big]\Big| \le \mathsf{negl}(\lambda),$$

where Exp_b is

$$q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_b),\ r \leftarrow \mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, \mathsf{evk}, \delta, q).$$

We note that while our constructions can also signal mismatches to the client, the primary role of this subroutine in KPIR-C is to ensure well-defined inputs to the post-retrieval computation. The server replaces the response r with δ in the mismatched case, where δ is chosen according to the circuit $\mathcal{F}$. For instance, for aggregation, the server may choose $\delta \coloneqq 0$ so that mismatched responses do not contribute to the result. For predicate checking, it may instead choose δ to produce the desired default predicate outcome.

B. KPIR with Computation

KPIR-C introduces a new algorithm named $\mathsf{Evaluate}$ run by the server to obliviously evaluate a circuit $\mathcal{F}$ on the response r, so that the client can retrieve $v' \coloneqq \mathcal{F}(v)$.

- $r' \leftarrow \mathsf{Evaluate}(\mathsf{pp}, \mathsf{evk}, \mathcal{F}, r)$: On input the public parameters pp, a response r, the evaluation key evk, and a circuit $\mathcal{F}$, the evaluate algorithm outputs a new response $r' \coloneqq \mathcal{F}(r)$.

Single-query correctness. The correctness guarantee can naturally be extended by requiring that the following probability is at least $1 - \mathsf{negl}(\lambda)$:

$$\Pr\left[v' = \begin{cases} \mathcal{F}(v_i) & \mathsf{kw} = k_i \\ \mathcal{F}(\delta) & \text{otherwise} \end{cases} \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp}) \\ \overline{\mathsf{DB}} \leftarrow \mathsf{Encode}(\mathsf{pp}, \mathsf{DB}) \\ q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}) \\ r \leftarrow \mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, \mathsf{evk}, \delta, q) \\ r' \leftarrow \mathsf{Evaluate}(\mathsf{pp}, \mathsf{evk}, \mathcal{F}, r) \\ v' \leftarrow \mathsf{Recover}(\mathsf{qk}, r') \end{array}\right]$$

Multi-Query Correctness. For a set of t queries, let $\mathcal{F}$ be a circuit that takes t inputs (Remark 1). The correctness guarantee requires that the following probability is at least $1 - \mathsf{negl}(\lambda)$:

$$\Pr\left[v' = \mathcal{F}(\mathbf{v}) \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \overline{\mathsf{DB}} \leftarrow \mathsf{Encode}(\mathsf{pp}, \mathsf{DB}) \\ (\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp}) \\ \mathbf{q} \leftarrow \big(\mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_i)\big)_{i \in [t]} \\ \mathbf{r} \leftarrow \big(\mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, \mathsf{evk}, \delta_i, q_i)\big)_{i \in [t]} \\ r' \leftarrow \mathsf{Evaluate}(\mathsf{pp}, \mathsf{evk}, \mathcal{F}, \mathbf{r}) \\ v' \leftarrow \mathsf{Recover}(\mathsf{qk}, r') \end{array}\right]$$

where $\mathbf{v} \coloneqq (v_1, \ldots, v_t)$ and, for each $i \in [t]$,

$$v_i \coloneqq \begin{cases} v_j & \text{if } \mathsf{kw}_i = k_j \text{ for some } j \in [m], \\ \delta_i & \text{otherwise.} \end{cases}$$

Depending on $\mathcal{F}$, the default values δ_i may be chosen identically across all queries or set independently.

Security. A KPIR-C scheme is secure if, for all PPT adversaries $\mathcal{A}$, all circuits $\mathcal{F}$, any $\delta \in \mathcal{V}$, and all $\mathsf{kw}_i, \mathsf{kw}_j \in \mathcal{K}$,

$$\Big|\Pr\big[\mathcal{A}(1^\lambda, \mathsf{evk}, q, r, r') = 1 \mid \mathsf{Exp}_i\big] - \Pr\big[\mathcal{A}(1^\lambda, \mathsf{evk}, q, r, r') = 1 \mid \mathsf{Exp}_j\big]\Big| \le \mathsf{negl}(\lambda),$$

where Exp_b denotes the experiment

$$q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_b),\quad r \leftarrow \mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, \mathsf{evk}, \delta, q),\quad r' \leftarrow \mathsf{Evaluate}(\mathsf{pp}, \mathsf{evk}, \mathcal{F}, r).$$

We refer the reader to Appendix D-A, where we show that this security definition captures the security requirements of KPIR-D and that standard query indistinguishability, commonly used to capture KPIR security (Section III-B), is sufficient to guarantee KPIR-C security.

V. Detailed KPIR-C Constructions

In this section, we detail two instantiations for KPIR-C. When necessary to distinguish between the two constructions, we prefix an algorithm with the corresponding construction name; within each protocol definition, we omit this prefix when the construction is clear from context. For simplicity, we assume values are within $[0 : p - 1]$; for larger entries, the formal descriptions in Figures 2 and 4 include a partitioning step to handle values of arbitrary bit length.

A. First Construction: Constant-Weight Keyword PIR over the Torus (TCWKPIR)

1) Construction Details: Here, we detail our TCWKPIR construction which requires only one round of communication.

Overview. TCWKPIR consists of five steps (see Figure 1): ① The client encodes its keyword kw as a constant-weight codeword (see Section III-C), encrypts it, and sends it to the server. ② The server, which has encoded all database keywords into constant-weight codewords during preprocessing, performs

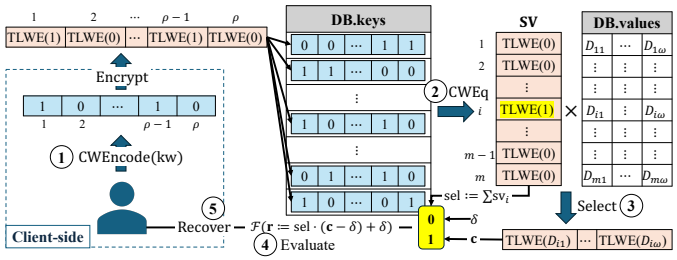


Fig. 1. TCWKPIR construction overview. $\omega := \lceil \ell_d / \log p \rceil$ is the count of entry partitions, with ℓ_d as the max data bit length.

an equality check (Algorithm 1) between the query and each encoded keyword to construct the selection vector sv. ③ The server computes the inner product (Algorithm 2) between sv and the database values to obliviously "select" the row at the position of the encrypted 1 in sv. ④ The server then applies a multiplexer that obliviously returns a default value δ on a mismatch. It evaluates $\mathcal{F}$ on the multiplexer's output. ⑤ The client decrypts the response and recovers the result. Figure 2 describes a formal instantiation of TCWKPIR.

Generating the Selection Vector. Section III-B defines the selection vector sv as a binary vector of length m that identifies the database entry matching the client's query. In TCWKPIR, the server obtains sv in encrypted form by evaluating CWEq (Algorithm 1) on the encrypted query keyword and each stored keyword, both under the constant-weight encoding. Whenever a matching entry $(k, v) \in \mathsf{DB}$ exists, the vector contains $\mathrm{TLWE}(1)$ at the matching position and $\mathrm{TLWE}(0)$ at all others, so the inner product $\sum_{i=1}^{m} \mathsf{sv}_i \cdot v_i$ returns $\mathrm{TLWE}(v)$.

Handling Mismatches on the Server. Recall from Section IV that the server evaluates an arbitrary function $\mathcal{F}$ on the retrieved value, and that it needs to ensure well-defined inputs for $\mathcal{F}$ even when no keywords match the client's query.

Letting the default value $\delta := 0$, Algorithm 1 aligns, as is, with the definition of KPIR-D (Section IV-A): the constant-weight equality check (CWEq, Algorithm 1) outputs an encryption of 0 for every non-matching query–keyword pair, so when no keyword in the database matches the query, the selection vector sv consists entirely of 0s. The server then computes $\mathsf{sel} := \sum_{i=1}^{m} \mathsf{sv}_i$ and returns the ciphertext $\mathsf{sel} \cdot r$ in the Response phase (Figure 2), which is $\mathrm{TLWE}(0)$ on mismatch and r otherwise.

However, zero may not be the appropriate default value for a given post-retrieval computation. We therefore allow δ to be chosen according to $\mathcal{F}$ by the server (Section IV-A). Accordingly, the server applies a homomorphic multiplexer and responds with $\mathsf{sel} \cdot (r - \delta) + \delta$ in the Response stage (Figure 2). After the Evaluate phase, the server then responds with $\mathsf{TLWE}\big(\mathcal{F}(v)\big)$ or $\mathsf{TLWE}\big(\mathcal{F}(\delta)\big)$.

B. Second Construction: Client-Aided Keyword PIR over the Torus (TCAKPIR)

1) Construction Details: In this section, we present our second construction TCAKPIR. Unlike the first construction, which builds the selection vector sv on the server, in TCAKPIR

Protocol TCWKPIR

Description: A KPIR-C scheme (Setup, ClientSetup, Encode, Query, Response, Recover, Evaluate) is defined between a server holding a database $\mathsf{DB} \in (\mathcal{K} \times \mathcal{V})^m$ and a client who, upon requesting a keyword $\mathsf{kw} \in \mathcal{K}$, aims to retrieve $\mathcal{F}(\mathsf{DB}[\mathsf{kw}])$ if $\mathsf{kw} \in \mathsf{DB.keys}$, or $\mathcal{F}(\delta)$ otherwise, where $\mathcal{F}$ denotes the post-processing circuit evaluated by the server.
Parameters: Security parameter λ, Hamming weight h, database cardinality m, data bit length ℓ_d, codeword length ρ.
Protocol execution:

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$:
- Parties agree on public parameters $\mathsf{pp} := (n, \sigma, p, q)$ determined by the security parameter λ.

$(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp})$:
- The client generates a private query key qk and a public evaluation key evk by invoking $(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$.

$\overline{\mathsf{DB}} \leftarrow \mathsf{Encode}(\mathsf{pp}, \mathsf{DB})$:
- For each key-value pair $(k, v) \in \mathsf{DB}$, the key k is encoded as a constant-weight codeword using $\mathsf{CWEncode}(k, h, \rho)$, and the value v is split into $\omega := \lceil \ell_d / \log p \rceil$ partitions, each of $\log(p)$ bits. Let $\overline{\mathsf{DB}}$ denote the preprocessed database.

$\mathbf{q} := (q_1, \ldots, q_\rho) \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw})$:
- The client encodes its query kw into a constant-weight codeword: $\mathsf{cw} \leftarrow \mathsf{CWEncode}(\mathsf{kw}, h, \rho)$.
- The client encrypts the codeword to generate the query $\mathbf{q} := (q_1, \ldots, q_\rho)$ by invoking $q_i \leftarrow \mathsf{Encrypt}_{\mathsf{qk}}(\mathsf{Encode}(\mathsf{cw}_i)), i \in [\rho]$.

$\mathbf{r} \leftarrow \mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, \mathsf{evk}, \delta, \mathbf{q})$:
- The server generates the selection vector sv by running $\forall i \in [m],\ \mathsf{sv}_i \leftarrow \mathsf{CWEq}(\mathbf{q}, K[i])$, where $K := \overline{\mathsf{DB}}.\mathsf{keys}$.
- Next, the server obliviously "selects" a row by invoking $\mathsf{r} := (\mathsf{r}_1, \ldots, \mathsf{r}_\omega) \leftarrow \mathsf{InnerProduct}(\mathsf{sv}, \overline{\mathsf{DB}}.\mathsf{values}, \mathsf{evk})$.
- The server then computes $r_j := \mathsf{sel} \cdot (\mathsf{r}_j - \delta_j) + \delta_j$, where $\mathsf{sel} := \sum_{i=1}^{m} \mathsf{sv}_i$, $j \in [\omega]$, and the values δ_j satisfy $\delta = \sum_{j=1}^{\omega} \delta_j \cdot p^{j-1}$.

$\mathbf{r}' \leftarrow \mathsf{Evaluate}(\mathsf{pp}, \mathsf{evk}, \mathcal{F}, \mathbf{r})$:
- The server homomorphically evaluates the circuit $\mathcal{F}$ on the response $\mathbf{r}$ to obtain $\mathbf{r}' \leftarrow \mathcal{F}(\mathbf{r})$ (Remark 1).

$v' \leftarrow \mathsf{Recover}(\mathsf{qk}, \mathbf{r}')$:
- The client decrypts the response $\mathbf{r}'$ by invoking $v'_i \leftarrow \mathsf{Decode}(\mathsf{Decrypt}_{\mathsf{qk}}(r'_i)), i \in [|\mathbf{r}'|]$.
- Next, the client reconstructs the value $v' := \sum_{i=1}^{|\mathbf{r}'|} v'_i \cdot p^{i-1}$.

Fig. 2. Our first construction: TCWKPIR.

the client constructs sv and sends it to the server as part of the query, which allows for a much more efficient runtime (see tradeoff details in Section II-B).

Overview. Figure 3 outlines our TCAKPIR construction: ① the server encodes each key-value pair into a Binary Fuse Filter (BFF); ② the client generates a query consisting of an encrypted selection vector (see Section III-B) and an encrypted digest of the queried keyword; ③ the server obliviously reconstructs the corresponding BFF entry using the selection vector, ④ performs an equality test between the embedded and encrypted digests, ⑤ and returns the evaluation of $\mathcal{F}$ on the reconstructed value or a default value based on the

Algorithm 1 CWEq: Bitwise Constant-Weight Eq. Check

Input: ($\mathbf{c}$: encoded query (ciphertext), $\mathbf{k}$: encoded keyword (plaintext))

Initialize e
for $i = 1$ **to** ρ **do**
  if $\mathbf{k}[i] = 1$ **then**
    if e is the initial value **then**
      $e \leftarrow \mathbf{c}[i]$
    else
      $e \leftarrow \mathsf{HomAND}(e, \mathbf{c}[i])$
Return: $e \in \mathbb{T}_q^{n+1}$

Algorithm 2 InnerProduct

Input: Selection vector sv, entries $D \in \mathbb{Z}_p^{m \times \omega}$, and evaluation key evk

$\mathbf{r} := (r_1, \ldots, r_\omega)$, where $r_i := \mathrm{TLWE}(0) \in \mathbb{T}_q^{n+1}$ for $i \in [\omega]$
for $i = 1$ **to** ω **do**
  for $j = 1$ **to** m **do**
    $t' \leftarrow \mathsf{ScalarMul}(D_{ji}, \mathsf{sv}_j)$
    $r_i \leftarrow \mathsf{HomAdd}(r_i, t')$
    $r_i \leftarrow \mathsf{Bootstrap}_{\mathsf{evk}}(r_i)$ ▷ Every τ_{bf} times (Section V-E)
Return: Response $\mathbf{r}$

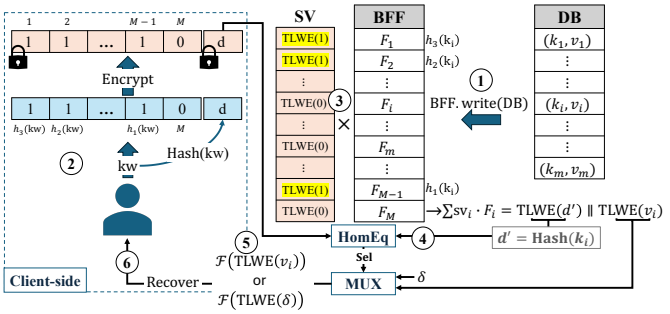


Fig. 3. TCAKPIR construction overview. $M \approx 1.156m$.

test outcome; ⑥ finally, the client decrypts the response to obtain the result. Figure 4 describes a formal instantiation of TCAKPIR as a KPIR-C scheme.

Generating the Selection Vector. In TCAKPIR, the server encodes the database into a BFF F (of cardinality $M \approx 1.156m$) using hash functions H shared with the client during the Encode phase, inserting the encoding $\mathsf{Hash}(k_i)||v_i$ for each pair (k_i, v_i). By correctness of inclusion (Section III-D), summing the κ rows indexed by $H(k_i)$ recovers this encoding. Accordingly, the client sets $\overline{\mathsf{sv}}_i := \mathbb{I}[i \in H(\mathsf{kw})]$ for $i \in [M]$, which selects exactly these κ rows, and encrypts $\overline{\mathsf{sv}}$ to obtain sv. Whenever a matching entry $(k, v) \in \mathsf{DB}$ exists (i.e., $\mathsf{kw} = k$), the inner product of sv and F yields an encryption of $\mathsf{Hash}(k)||v$: the server obtains $\mathbf{d}'$, the encryption of the embedded digest $\mathsf{Hash}(k)$, and r, the encryption of v.

Handling Mismatches on the Server. To handle mismatches, the client sends the encrypted digest of its query along with the encrypted selection vector. The server extracts and stores this digest, denoted $\mathbf{d}$, and performs a homomorphic equality test HomEq (Section III-E) between $\mathbf{d}$ and the digest $\mathbf{d}'$ embedded in the inner product result, yielding a selector bit sel. From this point, mismatch handling proceeds identically to our first construction (Section V-A).

Protocol TCAKPIR

Description: A KPIR-C scheme (Setup, $\mathsf{ClientSetup}$, Encode, Query, $\mathsf{Response}$, $\mathsf{Recover}$, $\mathsf{Evaluate}$) is defined between a server holding a database $\mathsf{DB} \in (\mathcal{K} \times \mathcal{V})^m$ and a client who, upon requesting a keyword $\mathsf{kw} \in \mathcal{K}$, aims to retrieve $\mathcal{F}(\mathsf{DB}[\mathsf{kw}])$ if $\mathsf{kw} \in \mathsf{DB.keys}$, or $\mathcal{F}(\delta)$ otherwise, where $\mathcal{F}$ denotes the post-processing circuit evaluated by the server.
Parameters: Security parameter λ, data bit length ℓ_d, database cardinality m, and a hash function $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^\mu$.
Protocol execution:
$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$: Similar to Figure 2.
$(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp})$:
- The client generates a private query key qk and a public evaluation key evk by invoking $(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$.

$F \leftarrow \mathsf{Encode}(\mathsf{pp}, \mathsf{DB})$:
- The server initializes an empty mapping $\mathcal{M} := \emptyset$.
- For each key-value pair $(k_i, v_i) \in \mathsf{DB}$, the server inserts $(k_i, f(k_i, v_i))$ into $\mathcal{M}$, where $f : \mathcal{K} \times \mathcal{V} \to \mathbb{Z}_p^\omega$ is an encoding function with $f(k_i, v_i) := \mathsf{Hash}(k_i) \parallel v_i$ and $\omega := \lceil (\mu + \ell_d)/\log p \rceil$.
- The server then constructs a BFF by running $(F, H) \leftarrow \mathtt{BFF.write}(\mathcal{M})$, and obtains the filter storage $F \in \mathbb{Z}_p^{M \times \omega}$ and κ hash functions $H(x) := (h_1(x), \ldots, h_\kappa(x))$.

$\mathbf{q} := (q_1, \ldots, q_{M+\omega'}) \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw})$:
- For a given query kw, the client computes κ positions $H(\mathsf{kw}) := (h_1(\mathsf{kw}), \ldots, h_\kappa(\mathsf{kw}))$ and generates the plaintext selection vector $\overline{\mathsf{sv}}$, where $\overline{\mathsf{sv}}_i := \mathbb{I}[i \in H(\mathsf{kw})]$ for all $i \in [M]$.
- Next, the client hashes the keyword kw into a digest $\overline{\mathbf{d}} := (\overline{\mathbf{d}}_1, \ldots, \overline{\mathbf{d}}_{\omega'}) \leftarrow \mathsf{Hash}(\mathsf{kw}) \in \mathbb{Z}_p^{\omega'}$ where $\omega' := \lceil \mu/\log p \rceil$.
- Then, for each $i \in [M]$, the client sets $\mathbf{q}[1{:}M] := (q_1, \ldots, q_M)$, where $q_i \leftarrow \mathsf{Encrypt}_{\mathsf{qk}}(\mathsf{Encode}(\overline{\mathsf{sv}}_i))$, and for each $i \in [\omega']$, the client sets $\mathbf{q}[M+1{:}M+\omega'] := (q_{M+1}, \ldots, q_{M+\omega'})$, where $q_{M+i} \leftarrow \mathsf{Encrypt}_{\mathsf{qk}}(\mathsf{Encode}(\overline{d}_i))$.

$\mathbf{r} \leftarrow \mathsf{Response}(\mathsf{pp}, F, \mathsf{evk}, \delta, \mathbf{q})$:
- The server retrieves $\mathsf{sv} := \mathbf{q}[1{:}M]$ and $\mathbf{d} := \mathbf{q}[M+1{:}M+\omega']$.
- Next, the server obliviously "selects" a row by invoking $\mathsf{z} \leftarrow \mathsf{InnerProduct}(\mathsf{sv}, F, \mathsf{evk})$, and sets $\mathbf{d}' := \mathsf{z}[1{:}\omega']$ and $\mathsf{r} := \mathsf{z}[\omega'+1{:}\omega]$.
- The server then invokes $\mathsf{sel} \leftarrow \mathsf{HomEq}(\mathbf{d}, \mathbf{d}')$ and computes $\mathbf{r}[j] := \mathsf{sel} \cdot (\mathsf{r}[j] - \delta_j) + \delta_j$, where $j \in [\bar{\omega}]$. The values δ_j satisfy $\delta = \sum_{j=1}^{\bar{\omega}} \delta_j \cdot p^{j-1}$, and $\bar{\omega} := \lceil \ell_d/\log p \rceil$.

$\mathbf{r}' \leftarrow \mathsf{Evaluate}(\mathsf{pp}, \mathsf{evk}, \mathcal{F}, \mathbf{r})$: Similar to Figure 2.
$v' \leftarrow \mathsf{Recover}(\mathsf{qk}, \mathbf{r}')$: Similar to Figure 2.

Fig. 4. Our second construction: TCAKPIR.

C. Common Extensions to Both Constructions

Supporting Multiple Queries. Our constructions extend naturally to t concurrent queries. The query key qk must remain fixed across queries,[6] while the client generates a fresh random mask for each query, since reusing the same mask across

[6]Cross-query computation under different keys requires server-side key switching, which is costly and increases communication overhead due to the need to communicate a different evaluation key per query key.

queries can leak information about the client's queries [47], [40], [46]. The client runs Query once per keyword to obtain $q^{(1)}, \ldots, q^{(t)}$ and sends them in a single round. The server runs Response independently for each query to obtain $r^{(i)}$, which encrypts the corresponding matched value or the prescribed default δ_i on a mismatch. During Evaluate, it evaluates $\mathcal{F}$ on $(r^{(1)}, \ldots, r^{(t)})$ (Remark 1) and returns a single r' encrypting $\mathcal{F}(r^{(1)}, \ldots, r^{(t)})$. The final response size therefore depends on the output size of $\mathcal{F}$ rather than directly on t.

Signaling Mismatches to the Client. It might be the case that the client needs to know when a mismatch occurs. In this case, the server may additionally return the encrypted selector $\mathsf{sel}^{(i)}$ for each query to indicate its match status. For an aggregate over t queries, the server may instead return a single indicator $\mathsf{sel}_{\text{all}} := \bigwedge_{i=1}^{t} \mathsf{sel}^{(i)}$, indicating whether all constituent queries matched. This signaling is optional and is not required for correctness, since an honest execution of KPIR-C already replaces each mismatched response with its prescribed default value δ_i before evaluating $\mathcal{F}$.

D. An Offline/Online Construction for TCAKPIR

Motivated by the offline preprocessing technique of ChalametPIR [22], we adopt a similar strategy to reduce the online cost of TCAKPIR: precomputing the query-independent KPIR computation offline and generating only the query-dependent ciphertext bodies on demand. In ChalametPIR, this offline precomputation takes the form of a "hint"—obtained by evaluating the fixed retrieval computation on random masks—which is generated once and reused across queries. These hints are sent to the client during the offline phase, allowing it to independently reconstruct the response without further interaction with the server.

However, unlike ChalametPIR, we cannot precompute such hints and share them with the client in advance, since in KPIR-C the evaluated circuit $\mathcal{F}$ is arbitrary and may not be known during the offline phase. Nor can we fix the masks across queries to amortize their generation cost, because doing so leaks information about the client's queries, as discussed in Section V-C. We can, however, precompute the KPIR computation on fresh random masks offline and retain the resulting states on the server. Although a fresh mask is required for every query, this does not require a separate mask exchange for each query: we show that a single shared pseudorandom generator (PRG) seed is sufficient to derive all such masks. These offline states are then combined with query-dependent ciphertext bodies generated on demand to complete the online computation. We call the resulting construction TCAKPIR$^+$ and describe its changes to TCAKPIR below.

The client and server share a single uniformly random seed for the PRG. From this seed, both parties independently derive the same sequence of pseudorandom masks, so no mask needs to be communicated separately. For practical reasons, we assume the client and server agree in advance on a fixed upper bound C on the total number of queries supported per offline phase. Once this limit is reached, they rerun the offline step without exchanging a new seed. Specifically, the TCAKPIR$^+$.Offline phase (which follows the setup phases in Fig. 4 and subsumes TCAKPIR.Encode) introduces two additional steps:

- The server generates $\theta \leftarrow\$ \{0,1\}^\lambda$. Then, it samples $t := M \cdot C + 1$ pseudorandom seeds $\tilde{\theta} := (\tilde{\theta}_1, \ldots, \tilde{\theta}_t) \leftarrow \mathsf{PRG}(\theta)$. The server shares θ with the client, who performs the same computation to obtain and store $\tilde{\theta}$.
- Next, the server precomputes the masked portion of the inner product. For each $i \in [C]$, the server samples a matrix $A_i \in \mathbb{T}_q^{M \times n}$, where each row is derived from a distinct seed among the first $M \cdot C$ elements of $\tilde{\theta}$ and represents the mask part of a row in the selection vector. It multiplies the matrix $A_i^\intercal \in \mathbb{T}_q^{n \times M}$ with the filter matrix $F \in \mathbb{Z}_p^{M \times \omega}$, where $\omega := \lceil (\mu + \ell_d) / \log p \rceil$ denotes the number of partitions. The resulting matrix $A_i' := (A_i^\intercal \times F)^\intercal \in \mathbb{T}_q^{\omega \times n}$ is stored.

Next, in TCAKPIR$^+$.Query, after constructing the cleartext selection vector $\overline{\mathsf{sv}}^{(i)}$, the client encrypts each entry by sampling A_i ($i \in [C]$) from the pseudorandom seeds under the same secret key qk, and sends only the body parts $B \in \mathbb{T}_q^{C \times M}$ to the server, where each entry is computed as $B[i][j] := \sum_{k=1}^{n} A_i[j][k] \cdot \mathsf{qk}[k] + \overline{\mathsf{sv}}^{(i)}[j] + e$, for $i \in [C]$ and $j \in [M]$.

In TCAKPIR$^+$.Response, the server computes the matrix product $B' := B \times F \in \mathbb{T}_q^{C \times \omega}$, and prepends the precomputed A_i' to construct the responses $\mathscr{R}_i[w] := (A_i'[w], B'[i][w]) \in \mathbb{T}_q^{n+1}$ for each $w \in [\omega]$ and $i \in [C]$. Once all $M \cdot C$ seeds have been used, the final seed $\tilde{\theta}_{M \cdot C + 1}$ is used to repeat the preprocessing step without any further communication.

In Table II, we provide a complexity analysis of TCAKPIR, TCAKPIR$^+$, and ChalametPIR. As discussed, the first difference between ChalametPIR and TCAKPIR$^+$ is that, to satisfy KPIR-C, no hint is sent during the offline phase; instead, the mask is returned to the client as part of the final response. Another key difference is that to satisfy KPIR-D, the client must send its encrypted digest to the server during the online phase to ensure correctness. In contrast, ChalametPIR ensures correctness by having the server send the response digest. Finally, as per our goal, the online runtime of TCAKPIR$^+$ remains the same as ChalametPIR. For $C := 1$, TCAKPIR$^+$ and TCAKPIR have the same communication complexity as ChalametPIR; for $C > 1$, an additional $n(C-1)\log q$ bits must be communicated per data partition, a cost imposed by KPIR-C that cannot be avoided. However, Appendix B-B shows how response packing amortizes the download cost across data partitions, yielding better communication than ChalametPIR even for $C := 1$ (Table V).

E. KPIR-Specific Bootstrapping Strategy

In the protocols of Figures 2 and 4, data entries are partitioned into ω blocks and encrypted as TLWE ciphertext blocks. Performing arithmetic over these blocks requires detecting modular carries and propagating them across blocks. In the generic use of TFHE [38], this is handled via frequent bootstrapping. Applied directly to our constructions, such aggressive bootstrapping would dominate the total cost (see Section VI-C, Table III). However, unlike arbitrary computation,

TABLE II
COMPARISON OF TCAKPIR, TCAKPIR$^+$, AND CHALAMETPIR; SECURITY PARAMETER λ, LWE DIMENSION n, NUMBER OF QUERIES C, CIPHERTEXT MODULUS q, STORAGE LENGTH M, AND $\omega \coloneqq (\bar{\omega} + \omega')$; $\bar{\omega}$ AND ω' DENOTE THE NUMBER OF ENTRY AND DIGEST PARTITIONS.

Scheme	Client Storage (bits)	Offline Comm. (bits)		Online Comm. (bits)		Total Comm. (bits)	Online Runtime (complex.)
		Upload	Download	Upload	Download		
ChalametPIR [22]	$\omega n \log q + (C+1)\lambda$	-	$\bar{\omega} n \log q + \lambda$	$CM \log q$	$\omega C \log q$	$\lambda + (\bar{\omega} n + C(M+\omega)) \log q$	$O(\omega CM)$
TCAKPIR	$(C+1)\lambda$	$C\lambda$	-	$C(M+\omega') \log q$	$\bar{\omega} C(n+1) \log q$	$C\big(\lambda + (M+\omega+n\bar{\omega}) \log q\big)$	$O(\omega CnM)$
TCAKPIR$^+$	$(CM+1)\lambda$	-	λ	$C(M+\omega') \log q$	$\bar{\omega} C(n+1) \log q$	$\lambda + C(M+\omega+n\bar{\omega}) \log q$	$O(\omega CM)$

the KPIR component in KPIR-C has a highly structured form: its main arithmetic consists of an inner product between an encrypted binary selection vector and plaintext database entries. Although all terms contribute to ciphertext noise, carry accumulation arises only from database entries multiplied by a selection bit of 1; entries multiplied by 0 contribute nothing.

Let κ denote the number of active entries encrypting 1 in the selection vector *of a single query*, with $\kappa = 1$ for TCWKPIR and $\kappa \coloneqq 3$ for TCAKPIR. During the inner product, summing these κ active entries may cause a wraparound if the accumulated value exceeds $p-1$. To prevent this, we partition the plaintext space using a message modulus p_m chosen such that $\kappa \cdot (p_m - 1) < p$. The remaining factor $p_c \coloneqq p/p_m$ is referred to as the *carry modulus* (see Section VI-A).

This leads to the following design principle: *When evaluating KPIR over TFHE, bootstrapping should be invoked only to bound noise growth.* Therefore, we define a parameter τ_{bf} (bootstrapping-free depth) that controls the number of bootstrappings during the inner product. In other words, τ_{bf} bounds the number of bootstrapping-safe entries that can be aggregated when performing the inner product. We formalize the requirements for this KPIR-specific bootstrapping (KBS) strategy as follows:

Property 1 (Worst-case validity). A KBS-based KPIR construction must remain correct under worst-case data values, including the largest possible partition entries (e.g., $p_m - 1$), and when all bounded noise contributions accumulate in the same direction within the prescribed failure probability. This ensures that the strategy remains valid under conservative worst-case noise accumulation.

Property 2 (Correctness for subsequent computation). The ciphertext output of the KBS-based KPIR circuit must remain decryptable within the prescribed failure probability after bootstrapping. Equivalently, the ciphertext produced by the KPIR circuit should be *bootstrapping-safe*. This guarantees that it can serve reliably as input to the next phase.

In Section VI, we instantiate this strategy.

F. Security and Correctness Analysis

Threat Model. Similar to prior KPIR works [19], [22], [25], we consider a semi-honest server that follows the protocol specification but attempts to infer information about the client's queried keyword. This adversarial view is captured by the KPIR-C security definition in Section IV-B, where the server knows the database and its view includes the client's evaluation key and query ciphertexts, together with the response and evaluated response ciphertexts it computes.

Our security model focuses on protocol-level query privacy and excludes network metadata, such as client IP addresses and packet timing. Such leakage can be mitigated separately via anonymous communication systems [48]. Active network attacks, including message modification, injection, and replay, are outside our model and may be mitigated with authenticated transport protocols such as Transport Layer Security (TLS).

Security. To analyze the security of our constructions, we consider query indistinguishability, which we show in Appendix D-A is sufficient to guarantee KPIR-C security. We then prove query indistinguishability of both TCWKPIR and TCAKPIR by reduction to the IND-CPA security of TFHE in Appendix D-B. Moreover, TCAKPIR$^+$ is an offline/online variant of TCAKPIR in which the server precomputes the query-independent KPIR computation over the public TLWE masks before the query is known. Since these masks are already public in TCAKPIR, this preprocessing reveals no additional query-dependent information. Thus, TCAKPIR$^+$ preserves the query indistinguishability of TCAKPIR.

Correctness. For TCWKPIR, correctness is determined by the TFHE decryption-failure probability and the failure probability of the KBS strategy. For TCAKPIR, correctness additionally depends on the collision probability of the query digest. Although BFFs may also produce false positives, these do not affect correctness, since the resulting mismatches are detected through the digest comparison. We analyze these sources of error in more detail and bound their probabilities in Appendix D-C. Finally, TCAKPIR$^+$ follows the same correctness argument as TCAKPIR. Its offline/online split only preprocesses the public random masks and therefore introduces no additional correctness error.

VI. INSTANTIATION OF OUR CONSTRUCTIONS

A. Parameters for TFHE

We adopt TFHE-rs (v0.10.0) [38] to instantiate the TFHE scheme, and select the default parameter set `PARAM_MESSAGE_2_CARRY_2_KS_PBS` to initialize our protocols. Here, we align these parameters with TFHE's conventions for consistency. In TFHE-rs, the plaintext modulus p is defined as $p \coloneqq p_m \times p_c$, where p_m is the message modulus (i.e., messages are restricted to $\mathbb{Z}_{p_m}$ and then encoded into $\mathbb{T}_{p_m}$), and p_c is the carry modulus, which accommodates overflow during computation. For the chosen parameters, we

set $p_m := 4$ and $p_c := 4$. The ciphertext modulus is fixed at 2^{64}, resulting in a scaling factor of $\Delta := \frac{1}{2p} = 0.03125$. Encryption noise is sampled from a Gaussian distribution with standard deviation $\sigma := 2.845 \cdot 10^{-15}$. The selected TFHE-rs parameter set targets a failure probability of approximately 2^{-64} per PBS operation. We refer readers to Appendix B-A for additional parameters.

B. Parameters for Our Constructions

In TCWKPIR, we use a Hamming weight of $h := 2$ for our constant-weight encoding, which keeps the query size sublinear in the database cardinality, i.e., $O(\sqrt{m})$. Table VIII (Appendix A) illustrates this tradeoff more clearly for a fixed keyword range of m. Although a Hamming weight $h > 2$ would further reduce the query size, the additional AND evaluations needed to construct the selection vector make this impractical.

In TCAKPIR, we initialize the BFF with three hash functions ($\kappa := 3$), resulting in an encoding length of no more than $1.156\times$ the original database cardinality m. Additionally, we set the digest length to $\mu := 64$ bits.

C. KBS for Our Constructions

TFHE-rs [38] provides multiple interfaces for ciphertexts. Higher-level interfaces such as `FheUint` handle bootstrapping automatically, making them suitable for supporting arbitrary computation. However, as discussed in Section V-E, the structured nature of KPIR allows us to reduce the number of bootstrappings required for the KPIR component of KPIR-C (to once per τ_{bf} entries). In this section, we instantiate the τ_{bf} parameter for our KBS-based constructions, ensuring that both KBS properties (Section V-E) are satisfied.

In Appendix D-C, Theorems 4 and 5 show that, under worst-case settings (Property 1), τ_{bf} can be set to 214 and 2^{38} for TCWKPIR and TCAKPIR, respectively, while keeping the per-block failure probability at most 2^{-64} (Property 2). The execution-level bound follows by a union bound over all blocks and PBS operations. Finally, Table III reports the corresponding bootstrapping counts at the KPIR level when instantiated with `FheUint` and our KBS strategy. As shown, KBS requires significantly fewer bootstrappings.

TABLE III
NUMBER OF BOOTSTRAPPINGS DURING THE INNER PRODUCT IN OUR CONSTRUCTIONS INSTANTIATED WITH FHEUINT AND WITH KBS (2^{16} ENTRIES).

	Data Bitlength				
Strategy	**8**	**16**	**32**	**64**	**128**
FheUint (both)	2,555,904	9,109,504	33,030,144	122,748,928	470,024,192
KBS (TCWKPIR)	1,212	2,424	4,848	9,696	19,392
KBS (TCAKPIR)	0	0	0	0	0

VII. EVALUATION

We evaluate our constructions on a server with an AMD Threadripper PRO processor, 512GB of RAM, running Ubuntu 22.04.1 LTS. Unless otherwise specified, evaluations use a single thread, with runtimes averaged over 10 runs, and the implementation is primarily written in Rust. Since our protocols are single-round, they are robust under varying network latencies, and performance under any bandwidth constraint can be directly inferred from communication overhead; we thus keep experiments agnostic of network conditions.

A. Keyword PIR Benchmarks

In this section, we compare our constructions against state-of-the-art KPIR baselines, CWKPIR [19] and ChalametPIR [22]. Our evaluation considers a single query; at the KPIR level, the online runtime, upload cost, and download cost of all schemes grow linearly with the number of queries. In KPIR-C, however, the download cost depends on the output size of the post-processing function rather than directly on the number of queries (Section V-C). Since our baselines support only KPIR, this metric cannot be meaningfully compared across multiple queries. We therefore report single-query benchmarks to show that our constructions remain competitive at the KPIR level while supporting the additional functionality of KPIR-C.

Following prior benchmarks [19], [22], the runtime reported in Tables IV and V denotes server-side computation. Client-side work requires no encrypted-domain computation and consists of query generation (encoding, encryption, and, for TCAKPIR$^+$, mask derivation from the shared seed), response recovery (decryption and decoding), and one-time key generation. Across all settings, query generation takes at most 0.081 s and response recovery at most 0.314 s, for total client-side latency below 0.4 s. Table IX in Appendix A reports the detailed query and recovery times. One-time key generation costs 3.116 s.

Unless stated otherwise, communication costs include seed-compressed uploads and modulus-switched, packed responses, as detailed in Appendix B-B.

vs. CWKPIR. Table IV compares our constructions with CWKPIR [19], which uses BFV with a 218-bit ciphertext modulus, a 21-bit plaintext modulus, and polynomial degree $N \in \{8192, 16384\}$. We denote these configurations CWKPIR (2^{13}) and CWKPIR (2^{14}), respectively. Increasing N provides greater capacity to maintain correctness as database cardinality grows, but imposes additional communication and runtime costs. In contrast, our database-independent setup requires no such parameter adjustment. For a fixed entry size, our per-entry communication and runtime remain constant with database cardinality, whereas CWKPIR's costs increase when its configured capacity is exceeded.

Depending on the value of N, CWKPIR incurs a fixed upload cost of 216 KB or 913 KB and a fixed download cost of 103 KB or 224 KB, for a total of 319 KB or 1137 KB. These costs are independent of the entry size, as long as the plaintext slot capacity is not exceeded. TCWKPIR avoids such fixed overheads in both directions because correctness in our bootstrapping-driven design does not depend on the database size, which allows us to use much smaller ciphertexts with a ciphertext modulus of only 64 bits, far below the 218 bits required by CWKPIR.

For upload, we encrypt the codeword using ρ ciphertexts and apply seed compression, which reduces each ciphertext to a

TABLE IV
COMPARISON WITH CWKPIR [19], $|\text{ENTRY}| \in \{8, 128\}$ B. SV DENOTES SELECTION VECTOR; IP DENOTES INNER PRODUCT.

KPIR Scheme	DB Size (B)	Runtime (s)			Comm. (KB)		
		SV	IP	Total	Up	Down	Total
CWKPIR (2^{13})	8×2^{12}	51.06	4.53	55.99	216	103	319
CWKPIR (2^{14})		242.16	18.49	263.04	913	224	1137
TCWKPIR		61.21	4.02	65.84	8.27	1.68	9.96
TCAKPIR		0.03	0.51	2.32	44	1.73	45.73
CWKPIR (2^{13})	128×2^{12}	49.02	4.49	53.91	216	103	319
CWKPIR (2^{14})		252.22	19.56	274.20	913	224	1137
TCWKPIR		60.65	63.74	133.67	8.27	3.96	12.23
TCAKPIR		0.03	4.27	14.83	44	5.57	49.57
CWKPIR (2^{13})	8×2^{14}	201.06	19.03	220.92	216	103	319
CWKPIR (2^{14})		989.06	84.74	1078.51	913	224	1137
TCWKPIR		245.02	16.07	261.71	16.36	1.68	18.04
TCAKPIR		0.12	1.89	5.36	160	1.73	161.73
CWKPIR (2^{13})	128×2^{14}	202.73	18.84	222.39	216	103	319
CWKPIR (2^{14})		987.66	76.47	1068.99	913	224	1137
TCWKPIR		243.03	254.09	506.39	16.36	3.96	20.32
TCAKPIR		0.13	15.81	27.95	160	5.57	165.57
CWKPIR (2^{13})	8×2^{16}	795.25	73.34	870.28	216	103	319
CWKPIR (2^{14})		3978.66	298.44	4286.92	913	224	1137
TCWKPIR		963.50	63.10	1027.25	32.62	1.68	34.30
TCAKPIR		0.62	7.24	17.40	608	1.73	609.73
CWKPIR (2^{13})	128×2^{16}	784.77	72.78	859.21	216	103	319
CWKPIR (2^{14})		3812.64	292.32	4114.25	913	224	1137
TCWKPIR		972.08	1016.38	1997.87	32.62	3.96	36.58
TCAKPIR		0.61	59.28	77.86	608	5.57	613.57

single 64-bit scalar and yields a $64\times$ overhead over the plaintext codeword. The upload therefore scales with the codeword length ρ, which is sublinear in the database cardinality, whereas CWKPIR incurs the cost of one large ciphertext. This gives an upload of only 32.62 KB for databases with up to 2^{16} entries, independent of the entry size.

For download, we pack multiple TLWE ciphertexts into a single TRLWE ciphertext, which provides a fixed cost for entry sizes up to 2^6 B (see Figure 6). Larger entries require multiple TRLWE ciphertexts, but each uses a substantially smaller polynomial degree than CWKPIR (2048 vs. 8192 or 16384). Thus, the download grows gradually with the entry size rather than incurring CWKPIR's large per-ciphertext cost; for 128 B entries, our download is only 3.96 KB.

Runtime scales differently with entry size in the two schemes. Selection-vector generation is independent of entry size in both schemes. In this phase, CWKPIR ($N = 2^{13}$) performs marginally better than TCWKPIR, while CWKPIR ($N = 2^{14}$) is markedly slower: its ciphertext–ciphertext multiplication costs 12.15 ms at $N = 2^{13}$ but rises to 58.40 ms at $N = 2^{14}$, compared with TCWKPIR's fixed 15.24 ms PBS cost per AND gate. In the inner-product phase, CWKPIR's runtime remains nearly constant with entry size due to coefficient packing, whereas TCWKPIR's cost increases with the number of entry partitions. Consequently, TCWKPIR is faster for small entries, while CWKPIR ($N = 2^{13}$) eventually overtakes it for larger entries. CWKPIR ($N = 2^{14}$) remains slower overall due to its larger parameters. Moreover, supporting larger databases requires increasing CWKPIR's parameters, whereas TCWKPIR uses database-independent parameters and is therefore better suited to dynamic databases.

On the other hand, TCAKPIR has higher communication than TCWKPIR because the client uploads a selection vector proportional to database cardinality. Its upload exceeds CWKPIR (2^{13}) at approximately $m \gtrsim 2^{15}$ but remains well below CWKPIR (2^{14}). Nevertheless, TCAKPIR is significantly faster than both schemes because the client constructs the selection vector directly, eliminating server-side homomorphic AND operations. Fresh ciphertexts also eliminate inner-product bootstrapping through $m = 2^{38}$ (Section VI-C).

Finally, CWKPIR stores one large selection-vector ciphertext per entry, so its memory consumption grows with both database cardinality and ciphertext size. Our constructions are substantially more memory efficient, as shown in Figure 5.

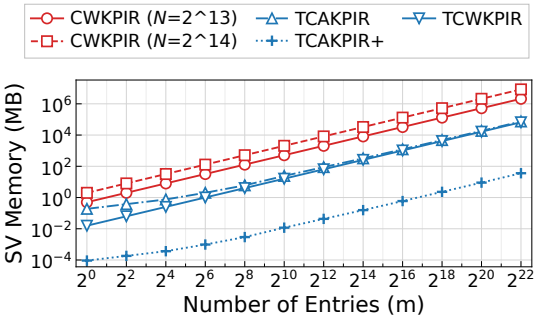


Fig. 5. Server-side Selection-vector memory consumption.

vs. ChalametPIR. In Table V, we benchmark TCAKPIR and TCAKPIR$^+$ against ChalametPIR [22] at its default parameters ($n = 1774, q = 2^{32}, p = 2^{10}$), denoted ChalametPIR ($2^{10}$), which shares TCAKPIR$^+$'s asymptotic online complexity (Table II) but is optimized for static databases and is faster in practice under these parameters. This advantage is parameter-driven: as database cardinality grows, ChalametPIR must adopt parameters closer to ours ($n = 2048, q = 2^{64}, p = 2^2$) to meet our correctness guarantee; we therefore also evaluate it at this reduced plaintext modulus, denoted ChalametPIR (2^2), which already raises offline runtime and download communication substantially. Since ChalametPIR's implementation does not support 64-bit ciphertext moduli, we cannot measure the effect of raising q as well, but expect both costs to converge further toward TCAKPIR$^+$'s.

TCAKPIR$^+$ is significantly faster than TCAKPIR due to its offline/online split, with online runtime competitive with ChalametPIR at small entry sizes (8 B). The $2\times$ upload gap between ChalametPIR and our schemes is entirely parameter-driven: their 32-bit ciphertext modulus versus our 64-bit modulus doubles the ciphertext size, with a marginal additional factor from the smaller LWE dimension ($n = 1774$ vs. $n = 2048$); under matching parameters, upload costs would be equivalent (Figure 6). Both of our schemes, however, achieve substantially lower download costs throughout, owing to modulus switching and response packing.

All Together. Figure 6 summarizes the communication trends across all four schemes. We instantiate ChalametPIR using both its default parameters and a custom configuration ($p =$

TABLE V
COMPARISON WITH CHALAMETPIR [22].

Benchmark	DB Size (B)	Runtime (s)			Comm. (KB)		
		Offline	Online	Total	Up	Down	Total
ChalametPIR (2^{10})	8×2^{12}	0.13	0.04	0.17	22.01	48.71	70.72
ChalametPIR (2^{2})		0.19	0.05	0.24	22.01	222.25	244.25
TCAKPIR		–	2.32	2.32	44	1.73	45.73
TCAKPIR^{+}		1.01	1.17	2.18	44	1.73	45.73
ChalametPIR (2^{10})	128×2^{12}	0.38	0.05	0.43	22.01	715.09	737.09
ChalametPIR (2^{2})		1.45	0.05	1.51	22.01	3554.12	3576.13
TCAKPIR		–	14.83	14.83	44	5.57	49.57
TCAKPIR^{+}		4.06	10.02	14.08	44	5.57	49.57
ChalametPIR (2^{10})	8×2^{16}	1.83	0.57	2.40	304.01	48.71	352.72
ChalametPIR (2^{2})		2.74	0.57	3.31	304.01	222.25	526.25
TCAKPIR		–	17.40	17.40	608	1.73	609.73
TCAKPIR^{+}		13.69	1.25	14.93	608	1.73	609.73
ChalametPIR (2^{10})	128×2^{16}	5.72	0.59	6.31	304.01	715.09	1019.09
ChalametPIR (2^{2})		20.36	0.65	21.01	304.01	3554.12	3858.13
TCAKPIR		–	77.86	77.86	608	5.57	613.57
TCAKPIR^{+}		58.48	10.08	68.56	608	5.57	613.57

$2^2, q = 2^{64}, n = 2048$) to match the failure probability of our own constructions. Because TCAKPIR and TCAKPIR^{+} incur identical communication costs, we plot only TCAKPIR for clarity. Our benchmarks scale up to 2^{20} entries, matching the maximum cardinality reported for ChalametPIR.

Regarding upload costs, as database cardinality grows, TCWKPIR approaches the communication overhead of CWKPIR (2^{13}). However, CWKPIR's upload remains constant only while the encoded query fits within a single ciphertext; beyond this capacity, additional ciphertexts or larger parameters may be required, which increase communication and computation costs, as already observed for CWKPIR (2^{14}). Since CWKPIR reports results only up to 2^{18} entries, we do not extrapolate its correctness beyond this range. Notably, CWKPIR incurs substantial memory overhead, reaching 512 GB at 2^{20} entries and 2 TB at 2^{22} entries, which quickly makes it impractical at larger database sizes (Figure 5).

Download costs follow a similar pattern: TCWKPIR matches CWKPIR's (2^{13}) communication at an entry size of 2^{12} B (4 KB), which already lies well above representative payloads in real-world PIR applications. Apple Live Caller ID Lookup uses KPIR with a 1 B blocking payload [8], Checklist's private Safe Browsing instantiation [4] retrieves 32 B full URL hashes, and Bitcoin PIR [16] uses 758–876 B transaction rows.

Even so, KPIR-C's communication advantage grows sharply once post-retrieval computation is involved: standard KPIR must return raw entries for local evaluation, whereas a threshold predicate in our schemes returns one encrypted bit, a 1.65 KB response regardless of entry size, and an aggregation over t queries returns a response the size of one query's response rather than t entries. Table X in Appendix A reports the corresponding multi-query evaluation, default-handling, and download costs. Evaluation scales with t and ℓ_d, but not with m or the construction; only TCWKPIR's default-handling cost depends on m, while download remains independent of t.

In Table VII (Appendix A), we compare all schemes under larger database sizes, including against another recent KPIR

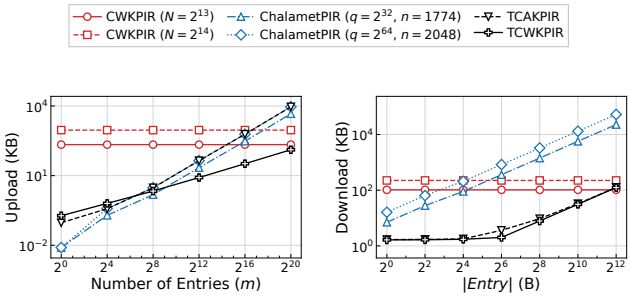


Fig. 6. Communication trend across all four schemes.

scheme, Hao et al. [25], which, as explained in Section II-B, cannot be adapted to KPIR-C. The evaluation keys for CWKPIR (2^{13}), CWKPIR (2^{14}) [19] and TCWKPIR are 15 MB, 121 MB and 26 MB, respectively; however, we do not include them in the upload costs for KPIR since they are data-independent. By contrast, TCAKPIR and TCAKPIR^{+} require no evaluation keys at all, as bootstrapping is entirely eliminated for the KPIR circuit (Section VI-C). Nevertheless, evaluation keys remain necessary when performing KPIR-C.

Overall, our KPIR-C schemes remain highly competitive with their KPIR counterparts: they consistently minimize download costs (e.g., only **3–9 KB** vs. **hundreds to tens of thousands KB** in prior work) and scale to databases as large as 2^{20} entries, where competing schemes either run out of memory or incur much higher communication. Besides communication efficiency, our schemes retain practical online latency, with TCAKPIR^{+} performing comparably to Hao et al. [25] and typically achieving $< \mathbf{1}$ s latency for up to 2^{20} entries.

VIII. RELATED WORK

The idea of Keyword PIR dates back to the work of Chor et al. [9]. More recent works achieve single-round KPIR using FHE. In particular, Chen et al. [10], [49] and Cong et al. [24] showed that symmetric KPIR (with server privacy) can be efficiently realized by homomorphically evaluating two polynomials: a key-interpolated polynomial and a value-interpolated polynomial.

Later, Ali et al. [21] use Cuckoo hashing to map sparse keywords into an array, followed by multiple invocations of index PIR to realize KPIR. A drawback of this approach is that both query and response sizes grow linearly with the number of hash functions used. Subsequent work [50] improves efficiency by encoding sparse key–value pairs as linear combinations of multiple database entries, enabling the use of standard PIR over the resulting encoding.

Alternatively, Pantheon [26] and CWKPIR [19] perform KPIR via equality circuits, achieving optimal communication complexity. In particular, CWKPIR eliminates dependence on entry length by employing constant-weight encoding, making the query size proportional to a fixed Hamming weight.

More recently, ChalametPIR [22] introduces the first hint-based KPIR scheme, allowing the server to provide a pre-computed hint that enables an extremely efficient query phase. ChalametPIR is instantiated using FrodoPIR [42] and Binary

Fuse Filters [45]. Building on this idea, Hao et al. [25] extend ChalametPIR to SimplePIR [40], achieving sublinear query communication. They propose a KPIR scheme based on Piecewise Linear Approximation (PLA) mappings [51], which yields sublinear query and response sizes at the cost of higher preprocessing and storage overhead.

A related line of work considers oblivious key-value retrieval (OKVR), introduced by Hao et al. [12] and described as a batched variant of symmetric KPIR. Their construction uses sparse oblivious key-value stores (OKVSs), a primitive studied in related work [44], [52], [53], together with batch PIR [20] for unbalanced circuit private set intersection (PSI).

IX. Conclusion

We introduced KPIR-C, a paradigm for arbitrary, non-interactive post-retrieval computation on KPIR responses that preserves query privacy. Unlike prior KPIR schemes, KPIR-C supports dynamically growing databases without parameter growth, keeping per-operation costs constant. Our two constructions offer complementary communication–runtime trade-offs and remain competitive with state-of-the-art KPIR baselines despite supporting richer functionality. KPIR-C thereby enables applications such as single-round private web search with server-side ranking, private ledger aggregation for lightweight clients, and related privacy-sensitive services.

Acknowledgment

We sincerely thank the anonymous reviewers for their constructive comments. This work is partially supported by the National Science Foundation (NSF) under Grant Nos. CNS-2302689, CNS-2308730, CNS-2432533, and a Synchrony Fellowship.

References

[1] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*. IEEE Computer Society, 1995, pp. 41–50.

[2] P. Mittal, F. G. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, "Pir-tor: Scalable anonymous communication using private information retrieval," in *USENIX Security 2011*, 2011.

[3] N. Borisov, G. Danezis, and I. Goldberg, "DP5: A private presence service," *PETS 2015*, 2015.

[4] D. Kogan and H. Corrigan-Gibbs, "Private blocklist lookups with checklist," in *USENIX Security 2021*, 2021.

[5] S. J. Menon and D. J. Wu, "SPIRAL: fast, high-rate single-server PIR via FHE composition," in *SP 2022*, 2022.

[6] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh, and E. Bursztein, "Protecting accounts from credential stuffing with password breach alerting," in *USENIX Security 2019*, 2019.

[7] A. Henzinger, E. Dauterman, H. Corrigan-Gibbs, and N. Zeldovich, "Private web search with tiptoe," in *SOSP 2023*, 2023.

[8] Apple, "PIR services: Example service for live caller ID lookup," Apple, Tech. Rep., 2024.

[9] B. Chor, N. Gilboa, and M. Naor, "Private information retrieval by keywords," Cryptology ePrint Archive, Paper 1998/003, 1998.

[10] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *CCS 2017*, 2017.

[11] R. A. Mahdavi, N. Lukas, F. Ebrahimianghazani, T. Humphries, B. Kacsmar, J. A. Premkumar, X. Li, S. Oya, E. Amjadian, and F. Kerschbaum, "PEPSI: practically efficient private set intersection in the unbalanced setting," in *USENIX Security 2024*, 2024.

[12] M. Hao, W. Liu, L. Peng, H. Li, C. Zhang, H. Chen, and T. Zhang, "Unbalanced circuit-psi from oblivious key-value retrieval," in *USENIX Security 2024*, 2024.

[13] P. Buddhavarapu, A. Knox, P. Mohassel, S. Sengupta, E. Taubeneck, and V. Vlaskin, "Private matching for compute," Cryptology ePrint Archive, Paper 2020/599, 2020.

[14] T. Lepoint, S. Patel, M. Raykova, K. Seth, and N. Trieu, "Private join and compute from PIR with default," in *ASIACRYPT 2021*, 2021.

[15] K. Chida, K. Hamada, A. Ichikawa, M. Kii, and J. Tomida, "Communication-efficient inner product private join and compute with cardinality," in *ASIACCS 2023*, 2023.

[16] K. Qin, H. Hadass, A. Gervais, and J. Reardon, "Applying private information retrieval to lightweight bitcoin clients," in *CVCBT 2019*, 2019.

[17] Y. Xie, Q. Wang, R. Li, C. Zhang, and L. Wei, "Private transaction retrieval for lightweight bitcoin clients," *IEEE Trans. Serv. Comput.*, vol. 16, no. 5, pp. 3590–3603, 2023.

[18] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC 2009*, 2009.

[19] R. A. Mahdavi and F. Kerschbaum, "Constant-weight PIR: single-round keyword PIR via constant-weight equality operators," in *USENIX Security 2022*, 2022.

[20] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, "PIR with compressed queries and amortized query processing," in *SP 2018*, 2018.

[21] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo, "Communication-computation trade-offs in PIR," in *USENIX Security 2021*, 2021.

[22] S. Celi and A. Davidson, "Call me by my name: Simple, practical private information retrieval for keyword queries," in *CCS 2024*, B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, Eds., 2024.

[23] J. Song, Z. Xu, Y. Zhang, P. Zhan, M. Li, S. Ma, and R. Xie, "Bkpir: Keyword pir for private boolean retrieval," in *Network and Distributed System Security Symposium, NDSS*, 2026.

[24] K. Cong, R. C. Moreno, M. B. da Gama, W. Dai, I. Iliashenko, K. Laine, and M. Rosenberg, "Labeled PSI from homomorphic encryption with reduced computation and communication," in *CCS 2021*, 2021.

[25] 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 *USENIX Security 2025*, 2025.

[26] I. Ahmad, D. Agrawal, A. E. Abbadi, and T. Gupta, "Pantheon: Private retrieval from public key-value store," *Proceedings of the VLDB Endowment*, vol. 16, pp. 643–656, 2022.

[27] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *CRYPTO 2012*, 2012.

[28] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Paper 2012/144, 2012.

[29] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *ITCS*. ACM, 2012, pp. 309–325.

[30] J. Kim, J. Seo, and Y. Song, "Simpler and faster BFV bootstrapping for arbitrary plaintext modulus from CKKS," in *CCS 2024*, 2024.

[31] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *CRYPTO 2013*, 2013.

[32] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *EUROCRYPT 2015*, 2015.

[33] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *ASIACRYPT 2016*, 2016.

[34] ——, "TFHE: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[35] M. Dahl, D. Demmler, S. E. Kazdadi, A. Meyre, J. Orfila, D. Rotaru, N. P. Smart, S. Tap, and M. Walter, "Noah's ark: Efficient threshold-fhe using noise flooding," in *WAHC@CCS*. ACM, 2023, pp. 35–46.

[36] I. Chillotti, D. Ligier, J. Orfila, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur. (ASIACRYPT), Part III*, 2021, pp. 670–699.

[37] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J. Orfila, and S. Tap, "Parameter optimization and larger precision for (T)FHE," *J. Cryptol.*, vol. 36, no. 3, p. 28, 2023.

[38] Zama, "TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data," 2022, https://github.com/zama-ai/tfhe-rs.

[39] T. Balenbois, J. Orfila, and N. P. Smart, "Trivial transciphering with trivium and TFHE," in *WAHC@CCS*. ACM, 2023, pp. 69–78.
[40] 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 *USENIX Security 2023*, 2023.
[41] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *STOC 2005*, 2005.
[42] A. Davidson, G. Pestana, and S. Celi, "Frodopir: Simple, scalable, single-server private information retrieval," *PETs 2023*, 2023.
[43] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "PSI from paxos: Fast, malicious private set intersection," in *EUROCRYPT 2020*, 2020.
[44] G. Garimella, B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "Oblivious key-value stores and amplification for private set intersection," in *CRYPTO 2021*, 2021.
[45] T. M. Graf and D. Lemire, "Binary fuse filters: Fast and smaller than xor filters," *ACM Journal of Experimental Algorithmics*, vol. 27, pp. 1.5:1–1.5:15, 2022.
[46] M. Joye, "Sok: Fully homomorphic encryption over the [discretized] torus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 4, pp. 661–692, 2022.
[47] S. J. Menon and D. J. Wu, "YPIR: high-throughput single-server PIR with silent preprocessing," in *USENIX Security 2024*, 2024.
[48] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The second-generation onion router," in *USENIX Security Symposium*. USENIX, 2004, pp. 303–320.
[49] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled PSI from fully homomorphic encryption with malicious security," in *CCS 2018*, 2018.
[50] S. Patel, J. Y. Seo, and K. Yeo, "Don't be dense: Efficient keyword PIR for sparse databases," in *USENIX Security 2023*, 2023.
[51] P. Ferragina and G. Vinciguerra, "The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1162–1175, 2020.
[52] 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," *IACR Cryptol. ePrint Arch.*, vol. 2026, p. 1590, 2026.
[53] J. Liu, P. Miao, M. Rosulek, X. Shi, and J. Wang, "Updatable private set intersection from symmetric-key techniques," in *EUROCRYPT (2)*, ser. Lecture Notes in Computer Science, vol. 16542. Springer, 2026, pp. 376–407.
[54] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM J. Comput.*, vol. 43, no. 2, pp. 831–871, 2014.
[55] N. Genise, D. Micciancio, and Y. Polyakov, "Building an efficient lattice gadget toolkit: Subgaussian sampling and more," in *EUROCRYPT 2019*, 2019.
[56] O. Goldreich, *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.
[57] O. Bernard, M. Joye, N. P. Smart, and M. Walter, "Drifting towards better error probabilities in fully homomorphic encryption schemes," in *EUROCRYPT 2025*, 2025.

TABLE VI
NOTATION AND TERMINOLOGY USED IN THE PAPER.

Symbol / term	Meaning
Basic notation	
λ, $\mathsf{negl}(\lambda)$	Security parameter and negligible function.
$[n]$, $[a{:}b]$	Index sets $\{1,\dots,n\}$ and $\{a,\dots,b\}$.
$x \leftarrow\$\ A$, $x \leftarrow A$	Uniform and general sampling from set or distribution A.
$a \gtrsim b$	a is approximately greater than or equal to b.
Database & KPIR(-C) terminology	
Database cardinality (m)	Number of key–value pairs stored by the server.
$\mathsf{DB} \in (\mathcal{K} \times \mathcal{V})^m$	Key–value database with key space $\mathcal{K}$ and value space $\mathcal{V}$.
$\mathsf{kw} \in \mathcal{K}$	Queried keyword.
δ	Default value returned on mismatch (KPIR-D / KPIR-C).
$\mathcal{F}$	Server-side post-processing circuit/function.
Protocol variables	
qk / evk	Query key / evaluation key.
q / r / r'	Query / response / post-processed response.
sv	(Encrypted) selection vector used to obliviously select a row.
sel	Selector bit (e.g., used to mux in δ).
Parameters	
ℓ_k / ℓ_d	Max keyword bit length / max data bit length.
ρ / h	Constant-weight codeword length / Hamming weight.
μ	Digest length (bits) in TCAKPIR.
n / N / k	LWE dimension / polynomial degree / GLWE dimension.

Algorithm 3 CWEncode

Input: Keyword kw, Codeword length ρ, Hamming weight h, s.t. $\binom{\rho}{h} \geq 2^{\ell_k}$
$\mathbf{c} \coloneqq [0]^\rho$ ▷ Initialize codeword as a zero vector of length ρ
$r \coloneqq \mathsf{kw}$
$h_r \coloneqq h$
for $l = \rho - 1$ **to** 0 **do**
  $t \coloneqq \binom{l}{h_r}$
  if $r \geq t$ **then**
    $\mathbf{c}[l] \coloneqq 1$
    $r \coloneqq r - t$
    $h_r \coloneqq h_r - 1$
    if $h_r = 0$ **then**
      break
Return: Codeword c of length ρ with Hamming weight h.

APPENDIX A
ADDITIONAL TABLES AND ALGORITHMS

APPENDIX B
ADDITIONAL IMPLEMENTATION DETAILS

A. Instantiation from TFHE-rs

In Section VI-A, we briefly discuss the parameter set used to instantiate TFHE from TFHE-rs [38], which suffices to understand the main constructions. However, to follow the proofs in Appendix D and to clarify the connection between TFHE and its implementation in TFHE-rs, we also need to discuss additional parameters.

In the parameter set we use (Section VI-A), the encryption key is parameterized as a *Big* encryption key, where the LWE dimension is defined as $n_{\text{Big}} \coloneqq N \cdot k$. Here, $N \coloneqq 2048$ and $k \coloneqq 1$ denote the polynomial degree and the GLWE dimension, respectively. Since the implementation represents all ciphertexts in LWE form, we simply denote the Big LWE dimension as $n \coloneqq n_{\text{Big}} = 2048$ throughout the paper. The parameter set also includes a smaller LWE dimension, n_{Small}, which is used during bootstrapping after key switching. Notably, the implementation adopts a different *atomic* order from Algorithm 4. To bootstrap a ciphertext, the implementation first key switches the ciphertext from n_{Big} to n_{Small}, then performs modulus switching and blind rotation. This order can be seen as a different interpretation of Algorithm 4: rather than switching from small LWE to large and back to small, it switches from large to small and back to large, which keeps ciphertexts in the large LWE dimension without compromising bootstrapping performance. To reproduce our correctness results, we specifically use the parameter set `PARAM_MESSAGE_2_CARRY_2_KS_PBS` in TFHE-rs v0.10.0 [38].

B. Reducing Communication

1) Seed Compression: An immediate implication of having public random masks is that, instead of representing the ciphertext as $\mathsf{ct} \coloneqq (a_1, a_2, \dots, a_n, b)$, one can use the compact form $\mathsf{ct} \coloneqq (\theta, b)$, where θ is a random λ-bit seed [46]. This

TABLE VII

COMPARISON WITH KPIR BENCHMARKS. RESULTS ARE DRAWN WITH 64 THREADS ON 512 GB RAM. CHALAMETPIR DOES NOT IMPLEMENT MULTI-THREADING. OOM STANDS FOR OUT OF MEMORY. **BEST COSTS ARE HIGHLIGHTED IN YELLOW, SECOND BEST IN BLUE, AND WORST DOWNLOAD COSTS IN RED.**

#Entries	Entry (B)	KPIR Benchmark	Upload (KB)	Download (KB)	Online (s)
2^{16}	64 B	CWKPIR (2^{13}) [19]	216	103	50.82
		CWKPIR (2^{14}) [19]	913	224	231.91
		Hao et al. [25]	8.29	12127.74	0.004
		TCWKPIR (ours)	32.62	2.01	57.13
		TCAKPIR (ours)	608	3.63	11.81
		TCAKPIR$^+$ (ours)	608	3.63	0.22
	128 B	CWKPIR (2^{13}) [19]	216	103	50.64
		CWKPIR (2^{14}) [19]	913	224	233.63
		Hao et al. [25]	11.39	18000.52	0.007
		TCWKPIR (ours)	32.62	3.96	79.34
		TCAKPIR (ours)	608	5.57	13.22
		TCAKPIR$^+$ (ours)	608	5.57	0.41
	256 B	CWKPIR (2^{13}) [19]	216	103	51.14
		CWKPIR (2^{14}) [19]	913	224	247.84
		Hao et al. [25]	16.25	28556.87	0.012
		TCWKPIR (ours)	32.62	7.84	124.33
		TCAKPIR (ours)	608	9.46	17.62
		TCAKPIR$^+$ (ours)	608	9.46	0.81
2^{18}	64 B	CWKPIR (2^{13}) [19]	216	103	200.87
		CWKPIR (2^{14}) [19]	913	224	OOM
		Hao et al. [25]	16.97	20842.98	0.006
		TCWKPIR (ours)	65.14	2.01	225.89
		TCAKPIR (ours)	2368	3.63	45.1
		TCAKPIR$^+$ (ours)	2368	3.63	0.26
	128 B	CWKPIR (2^{13}) [19]	216	103	202.05
		CWKPIR (2^{14}) [19]	913	224	OOM
		Hao et al. [25]	23.32	30030.60	0.012
		TCWKPIR (ours)	65.14	3.96	331.30
		TCAKPIR (ours)	2368	5.57	53.71
		TCAKPIR$^+$ (ours)	2368	5.57	0.48
	256 B	CWKPIR (2^{13}) [19]	216	103	204.95
		CWKPIR (2^{14}) [19]	913	224	OOM
		Hao et al. [25]	32.5	45528.30	0.021
		TCWKPIR (ours)	65.14	7.84	484.94
		TCAKPIR (ours)	2368	9.46	70.56
		TCAKPIR$^+$ (ours)	2368	9.46	1.02
2^{20}	64 B	CWKPIR (2^{13}) [19]	216	103	OOM
		CWKPIR (2^{14}) [19]	913	224	OOM
		Hao et al. [25]	33.93	38371.75	0.018
		TCWKPIR (ours)	130.19	2.01	945.19
		TCAKPIR (ours)	9216	3.63	168.58
		TCAKPIR$^+$ (ours)	9216	3.63	0.45
	128 B	CWKPIR (2^{13}) [19]	216	103	OOM
		CWKPIR (2^{14}) [19]	913	224	OOM
		Hao et al. [25]	46.67	54242.50	0.027
		TCWKPIR (ours)	130.19	3.96	1296.35
		TCAKPIR (ours)	9216	5.57	207.58
		TCAKPIR$^+$ (ours)	9216	5.57	0.82
	256 B	CWKPIR (2^{13}) [19]	216	103	OOM
		CWKPIR (2^{14}) [19]	913	224	OOM
		Hao et al. [25]	64.98	79579.29	0.302
		TCWKPIR (ours)	130.19	7.85	1901.01
		TCAKPIR (ours)	9216	9.46	282.80
		TCAKPIR$^+$ (ours)	9216	9.46	1.72

TABLE VIII

QUERY SIZE VS. AND-GATE COUNT ACROSS VARYING HAMMING WEIGHTS h IN A 2^{16}-ENTRY DATABASE. QUERY SIZES ARE UNCOMPRESSED.

Hamming Weight	1	2	3	4	5
Query Size (KB)	12288	68	14	6	4
ANDs Count	0	65536	131072	196608	262144

TABLE IX

CLIENT-SIDE QUERY/RECOVER LATENCY. RECOVER TIMES ARE AVERAGED OVER EQUAL ENTRY LENGTHS.

Scheme	DB Size (B)	Time (s)		
		Query	Recover	Total
TCWKPIR	8×2^{12}	0.010	0.031	0.041
	128×2^{12}	0.010	0.292	0.302
	8×2^{14}	0.020	0.031	0.051
	128×2^{14}	0.020	0.292	0.312
	8×2^{16}	0.040	0.031	0.071
	128×2^{16}	0.040	0.292	0.332
TCAKPIR/TCAKPIR$^+$	8×2^{12}	0.006	0.045	0.051
	128×2^{12}	0.006	0.314	0.320
	8×2^{14}	0.022	0.045	0.067
	128×2^{14}	0.022	0.314	0.336
	8×2^{16}	0.081	0.045	0.126
	128×2^{16}	0.081	0.314	0.395

TABLE X

MULTI-QUERY KPIR-C PERFORMANCE USING 64 THREADS. EVALUATION TIMES ARE AVERAGED ACROSS SCHEMES AND $m \in \{2^{12}, 2^{16}\}$, WHILE DEFAULT TIMES ARE AVERAGED ACROSS FUNCTIONS AND, FOR TCAKPIR, ACROSS m. TCAKPIR ALSO REPRESENTS TCAKPIR$^+$.

t	Default (s)			Sum		Threshold-of-Sum	
	TCWKPIR		TCAKPIR	Eval. (s)	Download (KB)	Eval. (s)	Download (KB)
	$m = 2^{12}$	$m = 2^{16}$					
Entry size: 8 B							
1	0.086	0.133	0.148	0.038	1.684	0.138	1.645
2	0.167	0.260	0.295	0.165	1.684	0.268	1.645
4	0.331	0.516	0.591	0.412	1.684	0.522	1.645
8	0.650	1.001	1.222	0.914	1.684	1.015	1.645
16	1.303	1.997	2.418	1.933	1.684	2.019	1.645
Entry size: 128 B							
1	0.617	0.788	0.761	0.437	3.957	0.493	1.645
2	1.330	1.308	1.498	1.823	3.957	2.008	1.645
4	2.856	2.508	3.138	4.884	3.957	4.907	1.645
8	4.865	5.344	6.050	10.390	3.957	10.354	1.645
16	10.407	12.465	12.968	23.387	3.957	24.036	1.645

technique, known as seed compression, uses a cryptographically secure pseudo-random number generator to deterministically regenerate the masks from θ, allowing both parties to recover the original ciphertext.

2) Modulus Switching: Modulus switching is a standard technique used to reduce the size of lattice-based encodings after homomorphic operations. As shown in [54], an LWE ciphertext with parameters (n, q) can be converted to one with (n, q') for $q' < q$. In our constructions, after the evaluation phase (Evaluate), we apply modulus switching by setting $q' := 2n$ to compress the response. Although this introduces significant noise, it is feasible because modulus switching is applied after PBS (Appendix C-E), which simultaneously evaluates the circuit and refreshes the ciphertext.

3) Response Packing: Using the TLWE-to-TRLWE public functional key switching procedure [34], N scalar TLWE ciphertexts over $\mu_0, \ldots, \mu_{N-1} \in \mathbb{T}_q$ can be packed into a single Torus Ring-LWE (TRLWE) ciphertext (see Appendix C-A) via the coefficient embedding $\mu(X) = \sum_{i=0}^{N-1} \mu_i X^i \in \mathbb{T}_{N,q}[X]$, where N is the polynomial degree. This packing reduces our

download cost by a factor of N (see Section VII-A, Table V).

APPENDIX C
MORE ON TFHE

A. TGLWE Ciphertexts

TGLWE ciphertexts (General TLWE) extend TLWE to torus polynomials over $\mathbb{T}_{N,q}[X]$,[7] where $\mathsf{pp} = (N, k, \sigma, p, q)$ are public parameters: N is the (power-of-2) polynomial degree, k the generalized LWE (GLWE) dimension, σ the noise standard deviation, and $p \mid q$ are the plaintext and ciphertext moduli. When $k = 1$, TGLWE specializes to Torus Ring-LWE (TRLWE). Algorithms for TGLWE extend naturally from the corresponding TLWE algorithms discussed in Section III-E, with the difference that messages are restricted to $\mathbb{Z}_{N,p}[X]$ and encoded into $\mathbb{T}_{N,q}[X]$. We define two additional operations:

- $\text{TGLWE}(\mu_1+\mu_2) \leftarrow \mathsf{HomAdd}(\text{TGLWE}(\mu_1), \text{TGLWE}(\mu_2))$: On input TGLWE ciphertexts of torus polynomials $\mu_1, \mu_2 \in \mathbb{T}_{N,p}[X]$, returns $\text{TGLWE}(\mu_1 + \mu_2)$.
- $\text{TGLWE}(z \cdot \mu) \leftarrow \mathsf{PolyMul}(z, \text{TGLWE}(\mu))$: On input a "small" $z \in \mathbb{Z}_N[X]$ and a TGLWE ciphertext of $\mu \in \mathbb{T}_{N,p}[X]$, returns $\text{TGLWE}(z \cdot \mu)$.

We refer readers to [46] for more details.

B. Gadget Decomposition over the Torus

Definition 1. *For any finite additive group A, an A-gadget of length w and quality β is a vector $\mathbf{g} \in A^w$ such that any group element $v \in A$ can be written as an integer combination $v = \sum_{i=1}^{w} g_i \cdot u_i$ where $\mathbf{u} \coloneqq (u_1, \ldots, u_w) \in \mathbb{Z}^w$ has norm at most $||u|| \leq \beta$ [55].*

Definition 2. *For any finite additive group A, let $\mathbf{g} \in A^w$ be an A-gadget of quality β, and let $\otimes$ denote the Kronecker product. An A^n-gadget is defined as the matrix $G \coloneqq I_n \otimes \mathbf{g}^\intercal \in A^{n \times nw}$ such that any vector $\mathbf{v} \in A^n$ can be represented as $\mathbf{v} = \mathbf{u} \times G^\intercal$ where $\mathbf{u} \in \mathbb{Z}^{nw}$ has norm at most $||u|| \leq \sqrt{n}\beta$.*

Following Definition 1, for the additive group $\mathbb{T}_q$ and quality β, the $\mathbb{T}_q$-gadget is the vector $\mathbf{g} \coloneqq (1/\beta, \ldots, 1/\beta^w) \in \mathbb{T}_q^w$, such that for every $v \in \mathbb{T}_q$, there exists $\mathbf{u} \in \mathbb{Z}^w$ satisfying $v = \sum_{i=1}^{w} g_i \cdot u_i$. To ensure this holds for all $v \in \mathbb{T}_q$, we require $\beta^w = q$ [46]. For any $\ell \leq w$ with $\beta^\ell \mid q$, v can be approximated as $v \approx \sum_{i=1}^{\ell} g_i \cdot u_i$, where ℓ is called the *precision*. We also define the inverse transformation $\mathbf{g}^{-1} : \mathbb{T}_q \to \mathbb{Z}^\ell$ such that $\mathbf{g}^{-1}(v) \times \mathbf{g}^\intercal \approx v$ and $\mathbf{g}^{-1}(v)$ is "small."

Similarly, following Definition 2, define the gadget matrix $G \in \mathbb{T}_q^{n \times n\ell}$ such that every $\mathbf{v} \in \mathbb{T}_q^n$ admits a gadget decomposition $\mathbf{u} \in \mathbb{Z}^{n\ell}$ satisfying $\mathbf{v} \approx \mathbf{u} \times G^\intercal$, where $\ell \leq w$ denotes the precision and $\beta^\ell \mid q$. As in [46], the inverse $G^{-1} : \mathbb{T}_q^n \to \mathbb{Z}^{n\ell}$ maps $\mathbf{v}$ to a "small" decomposition such that $G^{-1}(\mathbf{v}) \times G^\intercal \approx \mathbf{v}$. These definitions extend naturally to torus polynomials over $\mathbb{T}_{N,q}[X]^k$.

[7] $\mathbb{T}_{N,q}[X]$ denotes polynomials modulo $X^N + 1$ with torus coefficients.

C. T(G)GSW Ciphertexts

TGSW ciphertexts are the torus-based variant of Gentry–Sahai–Waters (GSW) ciphertexts [31]. Let G be the gadget matrix over $\mathbb{T}$, β the quality, and ℓ the precision, with $p \coloneqq \beta^\ell$. The TGSW encryption of a message $m \in \mathbb{Z}_p$ under secret key $\mathbf{s} \in \mathbb{B}^n$ is defined as $\text{TGSW}_{\mathbf{s}}(m) \coloneqq Z^\intercal + m \cdot G^\intercal \in \mathbb{T}_q^{(n+1)\ell \times (n+1)}$, where $Z \coloneqq \left[\text{TLWE}_{\mathbf{s}}(0) \mid \cdots \mid \text{TLWE}_{\mathbf{s}}(0)\right] \in \mathbb{T}_q^{(n+1) \times (n+1)\ell}$.

Let $m_1 \in \mathbb{Z}_p$ and $\mu \in \mathbb{T}_p$. Define $C_1 \coloneqq \text{TGSW}(m_1)$ and $c \coloneqq \text{TLWE}(\mu)$. Then:

$$\text{(External Product) } \boxdot : \text{TGSW} \times \text{TLWE} \to \text{TLWE},$$
$$(C_1, c) \mapsto C_1 \boxdot c \coloneqq G^{-1}(c) \times C_1.$$

Similarly, we can define TGGSW ciphertexts over torus polynomials. Let $m \in \mathbb{Z}_{N,p}[X]$ and $\mu \in \mathbb{T}_{N,p}[X]$, with ciphertexts $C \coloneqq \text{TGGSW}(m)$ and $c \coloneqq \text{TGLWE}(\mu)$. Then:

$$\text{(External Product) } \boxdot : \text{TGGSW} \times \text{TGLWE} \to \text{TGLWE},$$
$$(C, c) \mapsto C \boxdot c \coloneqq G^{-1}(c) \times C.$$

The gadget matrix G controls noise growth in both external products, since $G^{-1}(c)$ is small by definition (Appendix C-B).

D. Controlled Multiplexer (CMUX)

The Controlled Multiplexer (CMUX) is an application of the external product in TFHE [46]. Let $c_0 \coloneqq \text{TGLWE}(\mu_0)$, $c_1 \coloneqq \text{TGLWE}(\mu_1)$, and $B \coloneqq \text{TGGSW}(b)$, where $b \in \mathbb{B}$ is the selection bit. The CMUX operator selects between c_0 and c_1 based on b, via the external product of B and $\text{TGLWE}(\mu_1 - \mu_0)$. Specifically, we have: $\mathsf{CMUX}(B, c_0, c_1) \coloneqq B \boxdot (c_1 - c_0) + c_0$.

E. Bootstrapping and Programmable Bootstrapping (PBS)

In TFHE, *bootstrapping* refreshes a ciphertext by homomorphically evaluating its decryption circuit. Let $\mathbf{c} \coloneqq \text{TLWE}_{\mathbf{s}}(\mu)$ be a TLWE ciphertext of $\mu \in \mathbb{T}_p$ under secret key $\mathbf{s} \in \mathbb{B}^n$. Recall that $\mathbf{c}$ takes the form $(a_1, \ldots, a_n, b) \in \mathbb{T}_q^{n+1}$, where $b \coloneqq \sum_{i=1}^{n} s_i \cdot a_i + \mu + e$ for small noise e. The bootstrapping procedure on $\mathbf{c}$ is outlined in Algorithm 4.

In addition, given a function $f : \mathbb{T}_p \to \mathbb{T}_p$, we can re-define the test polynomial in Algorithm 4 as: $v \coloneqq \sum_{j=0}^{N-1} f\left(\frac{\lfloor pj/(2N) \rceil \bmod p}{p}\right) \cdot X^j$. After the *blind rotation*, the encrypted polynomial $X^{-\tilde{\mu}} \cdot v$ will contain the encryption of $f\left(\frac{\lfloor p \cdot \tilde{\mu}/(2N) \rceil \bmod p}{p}\right) = f(\mu)$ in the constant term.

APPENDIX D
SECURITY AND CORRECTNESS PROOFS

A. Extended Security Definition for KPIR-C

In this section, we establish two relationships between the security notions considered in this work. First, the query-indistinguishability notion commonly used for KPIR schemes (Section III-B) is sufficient to guarantee KPIR-C security.

[8] Flattening maps the polynomials $\mathbf{s}' \in \mathbb{B}_N[X]^k$ and $\mathbf{a}' \in \mathbb{T}_q[X]^k$ into length-kN vectors by re-arranging and sign-flipping coefficients according to $X^N \equiv -1$. This ensures that the flat inner product $\langle \bar{\mathbf{a}}, \bar{\mathbf{s}} \rangle$ reproduces the constant term of $\langle \mathbf{a}', \mathbf{s}' \rangle \bmod (X^N + 1)$.

Algorithm 4 Bootstrap [46]

Params: LWE dimension n, polynomial degree N, ciphertext modulus q, plaintext modulus p, quality β, precision ℓ

Input: TLWE ciphertext $\mathbf{c} \coloneqq (a_1, \ldots, a_n, b) \in \mathbb{T}_q^{n+1}$ of μ under secret key $\mathbf{s} \coloneqq (s_1, \ldots, s_n) \in \mathbb{B}^n$; bootstrapping keys (i.e., evaluation keys) $\mathsf{bsk}[j] \coloneqq \mathrm{TGGSW}_{\mathbf{s}'}(s_j)$ for $j \in [n]$, where $\mathbf{s}' \coloneqq (\mathbf{s}'_1, \ldots, \mathbf{s}'_k) \in \mathbb{B}_N[X]^k$ is the TGGSW secret key

1: **[Modulus Switching]** Scale $\mathbf{c}$ by the rounding factor $2N$ to obtain $\tilde{\mathbf{c}} \coloneqq (\tilde{a}_1, \ldots, \tilde{a}_n, \tilde{b}) \in \mathbb{T}_{2N}^{n+1}$, which encrypts $\tilde{\mu} \in \mathbb{T}_p$

2: **[GLWE Encoding]** Construct test polynomial $v \coloneqq \sum_{j=0}^{N-1} \left(\left\lfloor \frac{pj}{2N} \right\rceil \bmod p\right)/p \cdot X^j$ and set $\mathrm{TGLWE}(v) \coloneqq (0, \ldots, 0, v) \in \mathbb{T}_{N,q}[X]^{n+1}$

3: **[Blind Rotation]** Initialize $\mathbf{c}'_0 \coloneqq \mathsf{PolyMul}(X^{-\tilde{b}}, \mathrm{TGLWE}(v))$

4: **for** $j = 1$ **to** n **do**

5: $\quad \mathbf{c}'_j \coloneqq \mathsf{CMUX}(\mathsf{bsk}[j], \mathbf{c}'_{j-1}, \mathsf{PolyMul}(X^{\tilde{a}_j}, \mathbf{c}'_{j-1}))$

6: **[Sample Extraction]** We have $\mathbf{c}'_n \coloneqq \mathrm{TGLWE}_{\mathbf{s}'}(\mu + \cdots)$. Set $\mathbf{c}' \coloneqq \mathbf{c}'_n$, and write $\mathbf{c}' \coloneqq (a'_1, \ldots, a'_k, b')$ with $b' \coloneqq \sum_{i=0}^{N-1} b'_i X^i$; extract $\bar{b} \coloneqq b'_0$. Let $\mathbf{a}' \coloneqq (a'_1, \ldots, a'_k)$. Flatten[8] $\mathbf{s}'$ and $\mathbf{a}'$ into $\bar{\mathbf{s}} \coloneqq (\bar{s}_1, \ldots, \bar{s}_{kN}) \in \mathbb{B}^{kN}$ and $\bar{\mathbf{a}} \coloneqq (\bar{a}_1, \ldots, \bar{a}_{kN}) \in \mathbb{T}_q^{kN}$, respectively. Set $\bar{\mathbf{c}} \coloneqq (\bar{\mathbf{a}}, \bar{b})$

7: **[Key Switching]** Let $\mathbf{g} \coloneqq (1/\beta, \ldots, 1/\beta^\ell) \in \mathbb{T}_q^\ell$ denote the $\mathbb{T}_q$-gadget. For each $\bar{a}_i$, compute its decomposition $\bar{\bar{a}}_i \coloneqq \mathbf{g}^{-1}(\bar{a}_i) \in \mathbb{Z}^\ell$. Define $\mathsf{ksk}[i, j] \coloneqq \mathrm{TLWE}_{\mathbf{s}}(\bar{s}_i \cdot g_j)$ for $1 \le i \le kN$, $1 \le j \le \ell$. Let $\mathrm{TLWE}(\bar{b}) \coloneqq (0, \ldots, 0, \bar{b})$. Then compute: $\mathbf{c}'' \coloneqq \mathrm{TLWE}(\bar{b}) - \sum_{i=1}^{kN} \sum_{j=1}^{\ell} (\bar{\bar{a}}_i)_j \cdot \mathsf{ksk}[i, j]$

Return: TLWE ciphertext $\mathbf{c}''$ of μ under key $\mathbf{s}$ with refreshed noise

Second, the KPIR-C security definition (Section IV-B) subsumes the additional mismatch-security requirement of KPIR-D (Section IV-A).

For the first result, we define query indistinguishability for KPIR-C (Definition 3) as Game 0 and KPIR-C security as Game 1, and show in Theorem 1 that security in Game 0 implies security in Game 1. We then restrict Game 1 to the case where one challenge keyword is present in the database and the other is absent, yielding Game 2, which captures the mismatch-indistinguishability requirement of KPIR-D. We show that security in Game 1 implies security in Game 2.

Definition 3 (Query Indistinguishability $\mathsf{IND\text{-}Q}^{\mathcal{A}}_{\text{KPIR-C}}$)**.** *We say that a KPIR-C scheme (see Section IV-B) is query indistinguishable under security parameter λ if every PPT adversary $\mathcal{A}$ has negligible advantage in the game $\mathsf{IND\text{-}Q}^{\mathcal{A}}_{\textit{KPIR-C}}$; that is, $\left|\Pr[\mathsf{IND\text{-}Q}^{\mathcal{A}}_{\textit{KPIR-C}}(\lambda) = 1] - \frac{1}{2}\right| \le \mathsf{negl}(\lambda)$.*

Theorem 1. *Any KPIR-C scheme that satisfies query indistinguishability is secure according to the KPIR-C security definition (Section IV-B).*

Proof. We define a sequence of games:

Game 0: The query-indistinguishability game $\mathsf{IND\text{-}Q}^{\mathcal{A},}_{\text{KPIR-C}}$.

Game 1: Same as Game 0, except the adversary $\mathcal{A}_1$ additionally selects a circuit $\mathcal{F}$ and computes $r \leftarrow \mathsf{Response}(\mathsf{pp}, \overline{\mathsf{DB}}, \mathsf{evk}, \delta, q_b)$ and $r' \leftarrow \mathsf{Evaluate}(\mathsf{pp}, \mathsf{evk}, \mathcal{F}, r)$. It outputs a guess $\bar{b} \in \{0, 1\}$ and wins if $\bar{b} = b$. This captures the KPIR-C security definition (Section IV-B).

Game 2: Same as Game 1, except the adversary $\mathcal{A}_2$ chooses the challenge keywords such that $(\mathsf{kw}_0, v) \in \mathsf{DB}$ for some $v \in \mathcal{V}$, while $(\mathsf{kw}_1, v) \notin \mathsf{DB}$ for all $v \in \mathcal{V}$. It outputs a guess $\bar{b} \in \{0, 1\}$ and wins if $\bar{b} = b$. This captures the mismatch-indistinguishability requirement of KPIR-D (Section IV-A) when KPIR-D is used as a subroutine of KPIR-C under the selected circuit $\mathcal{F}$.

Suppose a PPT adversary $\mathcal{A}_1$ distinguishes Game 1 with non-negligible advantage. We construct an adversary $\mathcal{B}_0$ against Game 0 that, given q_b, computes the corresponding r and r' as in Game 1 and forwards them to $\mathcal{A}_1$. The simulation is identical, so $\mathcal{B}_0$ achieves the same advantage.

Similarly, any PPT adversary $\mathcal{A}_2$ against Game 2 yields an adversary $\mathcal{B}_1$ against Game 1 by using the same database and challenge keywords; these are valid choices in Game 1. Thus, $\mathcal{B}_1$ has the same advantage as $\mathcal{A}_2$.

Therefore, security in Game 0 implies security in Game 1, which in turn implies security in Game 2. □

$\mathsf{IND\text{-}Q}^{\mathcal{A}}_{\text{KPIR-C}}$

1. Challenger $\mathcal{C}$ initializes the scheme:
- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
- $(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp})$
- Give $(\mathsf{pp}, \mathsf{evk})$ to $\mathcal{A}$

2. Adversary $\mathcal{A}$ chooses a database $\mathsf{DB} \in (\mathcal{K} \times \mathcal{V})^*$ and challenge keywords $\mathsf{kw}_0, \mathsf{kw}_1 \in \mathcal{K}$, and sends them to $\mathcal{C}$.

3. Challenger $\mathcal{C}$ generates the challenge:
- $\overline{\mathsf{DB}} \leftarrow \mathsf{Encode}(\mathsf{pp}, \mathsf{DB})$
- $b \leftarrow\$ \{0, 1\}$
- $q_b \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_b)$
- Give $(\overline{\mathsf{DB}}, q_b)$ to $\mathcal{A}$

4. $\mathcal{A}$ outputs a guess $\bar{b} \in \{0, 1\}$ and wins if $\bar{b} = b$.

Fig. 7. Query-indistinguishability game for KPIR-C.

Corollary 1. *Any secure KPIR-C scheme satisfies the mismatch-security requirement of KPIR-D.*

Proof. This follows directly from the Game 1 to Game 2 reduction in the proof of Theorem 1. □

$\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\text{FHE}}$

1. Generate $(\mathsf{sk}, \mathsf{evk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$ and give $(\mathsf{pp}, \mathsf{evk})$ to $\mathcal{A}$.

2. $\mathcal{A}$ outputs

$$\mathbf{m}_0 = (m_{0,1}, \ldots, m_{0,\ell}), \qquad \mathbf{m}_1 = (m_{1,1}, \ldots, m_{1,\ell}),$$

where $\ell = \mathrm{poly}(\lambda)$ and $|m_{0,i}| = |m_{1,i}|$ for all $i \in [\ell]$.

3. Sample $b \leftarrow\$ \{0, 1\}$ and compute

$$\mathbf{c} \leftarrow \left(\mathsf{Encrypt}_{\mathsf{sk}}(m_{b,1}), \ldots, \mathsf{Encrypt}_{\mathsf{sk}}(m_{b,\ell})\right),$$

using fresh randomness for each encryption. Give $\mathbf{c}$ to $\mathcal{A}$.

4. $\mathcal{A}$ outputs $\bar{b} \in \{0, 1\}$ and wins if $\bar{b} = b$.

Fig. 8. Multi-message IND-CPA game for FHE.

B. Security of Our Constructions

We prove the security of our constructions via a reduction to the IND-CPA security of TFHE, that can be captured by the

game in Figure 8. We use the multi-message variant of IND-CPA, which is equivalent to the standard single-message notion for polynomially many messages via a hybrid argument [56, Section 5.2.4]. This formulation directly captures our setting, since each query consists of multiple ciphertexts, and also extends to polynomially many queries by viewing all ciphertext components across the queries as a single polynomial-length message vector.

Theorem 2. *The TCWKPIR construction (Figure 2) is secure under Definition 3, assuming that TFHE is IND-CPA secure.*

Proof. Suppose, for contradiction, that TCWKPIR is not secure. Then, there exists a PPT adversary $\mathcal{B}$ that can win the query indistinguishability game $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\mathrm{TCWKPIR}}$ with a non-negligible advantage. We show that this implies the existence of a PPT adversary $\mathcal{A}$ that can win the ciphertext indistinguishability game $\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathrm{TFHE}}$.

Let $\mathcal{C}$ be the challenger in the $\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathrm{TFHE}}$ game. We construct the adversary $\mathcal{A}$ such that it internally runs $\mathcal{B}$ as a subroutine to win this game. The adversary $\mathcal{B}$ begins by submitting two keywords $\mathsf{kw}_0, \mathsf{kw}_1 \in \mathcal{K}$ to $\mathcal{A}$ in the $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\mathrm{TCWKPIR}}$ game. To simulate the query $\mathbf{q}_b \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_b)$ in the $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\mathrm{TCWKPIR}}$ game, $\mathcal{A}$ runs:

$$\mathsf{cw}_0 \coloneqq (\mathsf{cw}_{01}, \ldots, \mathsf{cw}_{0\rho}) \leftarrow \mathsf{CWEncode}(\mathsf{kw}_0, h, \rho), \text{ and}$$
$$\mathsf{cw}_1 \coloneqq (\mathsf{cw}_{11}, \ldots, \mathsf{cw}_{1\rho}) \leftarrow \mathsf{CWEncode}(\mathsf{kw}_1, h, \rho),$$

where h and ρ denote the Hamming weight and the codeword length, respectively. $\mathcal{A}$ then submits cw_0 and cw_1 to the challenger $\mathcal{C}$. The challenger $\mathcal{C}$ samples $b \leftarrow\$ \{0,1\}$ and returns

$$\mathbf{ct}_b \coloneqq (\mathsf{Encrypt}_{\mathsf{sk}}(\mathsf{Encode}(\mathsf{cw}_{bi})))_{i\in[\rho]}\,.$$

The adversary $\mathcal{A}$ then sets $\overline{\mathbf{q}_b} \coloneqq \mathsf{ct}_b$ and sends $\overline{\mathbf{q}_b}$ to $\mathcal{B}$ in the $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\mathrm{TCWKPIR}}$ game. It is easy to see that $\overline{\mathbf{q}_b}$ correctly simulates $\mathbf{q}_b$. Upon receiving the query $\mathbf{q}_b$ in the $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\mathrm{TCWKPIR}}$ game, $\mathcal{B}$ outputs a guess $\bar{b} \in \{0,1\}$ and submits it to $\mathcal{A}$, who then outputs $\bar{b}$ as a response to its challenger $\mathcal{C}$ in the $\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathrm{TFHE}}$ game. It is clear that if $\mathcal{B}$ can guess b with non-negligible advantage, then so can $\mathcal{A}$, thereby breaking the IND-CPA security of TFHE. □

Theorem 3. *TCAKPIR construction (Figure 4) is secure under Definition 3, assuming that TFHE is IND-CPA secure.*

Proof. The proof follows the same logic as in Theorem 2. However, this time, upon receiving two keywords $\mathsf{kw}_0, \mathsf{kw}_1$, the adversary $\mathcal{A}$ computes $(\overline{\mathsf{sv}_0}, \overline{d_0})$ and $(\overline{\mathsf{sv}_1}, \overline{d_1})$ for both keywords as per Figure 4 and submits the pair $(\mathbf{m_0}, \mathbf{m_1})$ to the challenger, where $\mathbf{m_0} \coloneqq (\overline{\mathsf{sv}_0}, \overline{d_0}) \in \mathbb{Z}_p^{M+\omega'}$ and $\mathbf{m_1} \coloneqq (\overline{\mathsf{sv}_1}, \overline{d_1}) \in \mathbb{Z}_p^{M+\omega'}$. The challenger $\mathcal{C}$, playing in the $\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathrm{TFHE}}$ game, responds with:

$$\mathsf{ct}_{bi} \leftarrow \mathsf{Encrypt}_{\mathsf{sk}}(\mathsf{Encode}(m_{bi})), \text{ where}$$
$$b \leftarrow\$ \{0,1\} \text{ and } i \in [M+\omega'].$$

The adversary $\mathcal{A}$ then sets $q_b \coloneqq \mathsf{ct}_b$ and sends it to $\mathcal{B}$, and submits what $\mathcal{B}$ outputs. If $\mathcal{B}$ can guess the bit b with non-negligible advantage, so can $\mathcal{A}$, thereby breaking the IND-CPA security of TFHE. □

C. Correctness of Our Constructions

Correctness of TCWKPIR. The correctness of TCWKPIR reduces to the correctness of TFHE decryption, as the sole source of failure is noise accumulation in the TFHE ciphertexts. The selected TFHE-rs parameter set targets a failure probability of approximately 2^{-64} for each programmable bootstrapping (PBS) operation.[9] Selection-vector generation directly follows the bootstrapping strategy of TFHE-rs [38]. In the inner-product phase, we reduce the bootstrapping frequency using our KBS strategy (Section V-E), requiring us to bound the noise accumulated between consecutive bootstrappings. Theorem 4 establishes that an accumulation block containing at most 214 entries has failure probability at most 2^{-64}. The failure probability of a complete execution is then bounded by applying a union bound over its PBS operations and inner-product blocks.

Theorem 4. *Let $d \coloneqq p_m - 1$ be the maximum partition entry. An inner-product accumulation block of depth at most $m \coloneqq 214$ in TCWKPIR has failure probability at most 2^{-64} without intermediate bootstrapping.*

Proof. In TFHE-rs [38], under the chosen parameter set (Section VI-A), bootstrapping consists of key switching, modulus switching, and blind rotation. To ensure correctness, the output after aggregating m entries must remain bootstrappable, meaning that the total error after modulus switching remains below the decryption threshold Δ. Since the selection vector consists of freshly bootstrapped ciphertexts, we use the bootstrapping error variance v_{bs} (Theorem 4 in [36]) as the input noise variance of each selection-vector entry, and write $\sigma_{\mathsf{bs}} \coloneqq \sqrt{v_{\mathrm{bs}}}$.

We define the tighter threshold $\Delta' \coloneqq \sqrt{\Delta^2 - c^2 v_{\mathrm{ks}} - v_{\mathrm{ms}}}$, where v_{ks} is the variance of the key-switching error (Theorem 4.1 in [34]), v_{ms} is the variance of the modulus-switching error (Remark 4.2 in [57]), and $c \coloneqq 2N/q$ (or $2N$ in torus representation) accounts for scaling during modulus switching.

Let $d_i \in \{0, \ldots, p_m - 1\}$ denote the i-th partition entry, and let $d \coloneqq p_m - 1$. The accumulated error after m entries is $S \coloneqq \sum_{i=1}^{m} d_i e_i$, where $e_i \sim \mathcal{N}(0, \sigma_{\mathsf{bs}}^2)$. Conservatively considering worst-case error accumulation, if $|e_i| \le t\sigma_{\mathsf{bs}}$ for all i, then $|S| \le mdt\sigma_{\mathsf{bs}}$.

By a union bound, $\Pr[\exists i : |e_i| > t\sigma_{\mathsf{bs}}] \le 2mQ(t)$. Thus, to achieve a failure probability of at most 2^{-64}, it suffices to choose $t = Q^{-1}(2^{-65}/m)$ and require $md\sigma_{\mathsf{bs}}Q^{-1}\left(\frac{2^{-65}}{m}\right) \le \Delta'$. Numerical evaluation with the parameters from Section VI-A yields $m \le 214$. Therefore, each accumulation block containing at most 214 entries satisfies the failure bound. □

Correctness of TCAKPIR. For the correctness analysis, we model Hash as a random oracle with μ-bit outputs. The correctness of TCAKPIR follows from an argument similar to that of TCWKPIR, with two key differences. First, TCAKPIR additionally depends on the collision probability of Hash, which maps keywords to digests (see Figure 4). The collision

[9] At implementation time, TFHE-rs provided no parameter set targeting 2^{-128} failure per PBS; a parameter set with such a target can nevertheless be used with our design for stronger execution-level reliability.

probability of Hash is at most $2^{-\mu}$, where μ is the digest length. Assuming a correctness parameter λ, choosing $\mu \geq \lambda$ ensures that the collision probability is at most $2^{-\lambda}$. The false positive rate of BFFs does not introduce an additional correctness error: on a false positive, the digest comparison yields $\mathsf{sel} = 0$ except with the digest-collision probability, causing the reconstructed response to be replaced with δ.

Second, the only source of noise accumulation in TCAKPIR is the inner-product phase. Theorem 5 establishes that a single inner-product output ciphertext computed over at most 2^{38} entries has failure probability at most 2^{-64} without any intermediate bootstrapping.

Theorem 5. *Let $d \coloneqq p_m - 1$ be the maximum partition entry. An inner product of depth at most $m \coloneqq 2^{38}$ in TCAKPIR has failure probability at most 2^{-64} without bootstrapping.*

Proof. In TCAKPIR, the selection vector is generated client-side, so bootstrapping noise does not contribute to the noise accumulated during the inner product. Each freshly encrypted ciphertext has error $e_i \sim \mathcal{N}(0, \sigma^2)$.

Let $d_i \in \{0, \ldots, p_m - 1\}$ denote the i-th partition entry, and let $d \coloneqq p_m - 1 = 3$. The accumulated error after an inner product of depth m is $S \coloneqq \sum_{i=1}^{m} d_i e_i$. Conservatively considering worst-case error accumulation, if $|e_i| \leq t\sigma$ for all i, then $|S| \leq mdt\sigma$.

By a union bound, $\Pr[\exists i : |e_i| > t\sigma] \leq 2mQ(t)$. Thus, to achieve a failure probability of at most 2^{-64}, it suffices to choose $t = Q^{-1}(2^{-65}/m)$ and require $md\sigma Q^{-1}\left(\frac{2^{-65}}{m}\right) \leq \Delta$. Plugging in the parameters from Section VI-A, namely $\Delta = 0.03125$, $d = 3$, and $\sigma = 2.845 \times 10^{-15}$, gives $m \leq 2^{38.19}$. Therefore, each inner product containing at most 2^{38} entries satisfies the stated failure bound. □