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

Ali Arastehfard[1], Weiran Liu[2], Qixian Zhou[3], Zinan Shen[4], Liqiang Peng[2], Lin Qu[2], Shuya Feng[1,5], and Yuan Hong[1]

[1]*University of Connecticut* [2]*Alibaba Group* [3]*Ant International* [4]*Peking University*
[5]*University of Alabama at Birmingham*

Abstract—**Private Information Retrieval (PIR) enables clients to retrieve data from a server without revealing their query. Keyword PIR (KPIR), an extension for keyword-based queries that enables PIR using keywords, is crucial for privacy-preserving two-party analytics in unbalanced settings. However, existing KPIR solutions face two challenges in efficiently supporting arbitrary server-side computations and handling mismatched queries non-interactively.**

To our best knowledge, we take the first step to introduce Keyword PIR with Computation ("KPIR-C"), a novel PIR primitive that enables arbitrary non-interactive computation on responses while preserving query privacy. We overcome the arbitrary computation challenge by introducing TFHE into KPIR, which ensures efficient bootstrapping and allows arbitrary server-side computations. We address the mismatch challenge by identifying an important KPIR-C subroutine, referred to as KPIR with Default ("KPIR-D"), to remove disturbance of the computation caused by the mismatched responses. We instantiate KPIR-C with two constructions, one based on constant-weight codes and the other on recent LWE-based KPIR approaches. Both constructions enable efficient post-computation and offer trade-offs between communication overhead and runtime. Experiments show that our implemented constructions achieve competitive performance, and in some cases even outperform state-of-the-art KPIR solutions that do not support arbitrary computation.

1. 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]. Furthermore, recent works have demonstrated PIR's viability for real-world deployments in applications like password breach checkups [6], private web search [7], and live caller ID lookups [8].

In its original form, PIR requires the server to organize the database as an array, enabling the client to retrieve entries by their index. The original form of PIR is thus known as *index PIR*. However, this abstraction differs from real-world databases, where entries are typically indexed by arbitrary strings as keywords. This leads to a variant of PIR called *keyword PIR* (KPIR) [9]. Beyond its standard applications, KPIR has recently been shown to be a fundamental building block for circuit Private Set Intersection (circuit-PSI) in the *unbalanced setting* [10], [11], [12], where two parties want privacy-preserving data analysis based on the matching data and one party holds significantly more data than the other. Thanks to the communication efficiency of KPIR, one can construct solutions using KPIR with communication cost sublinear in the larger data size. Specifically, in the matching phase, two parties run KPIR, where the party with more data acts as the server while the other party acts as the client. The client uses its own matching entires as keywords to retrieve secret-shared payloads from the server. They transition to the analytics phase by executing either general secure two-party computation (2PC) for arbitrary data analytics [13], or protocols tailored to specific tasks [14], [15].

Existing unbalanced PSI solutions for arbitrary subsequent data analytics unavoidably impose a communication burden during the analytics phase. Although non-interactive approaches exist for the matching phase [11], the analytics phase still relies on general 2PC, making the communication cost inherently dependent on the complexity of the task. Such a burden is not acceptable when designing a non-interactive solution, especially for the client who owns less data and may have limited network bandwidth. Recall that most efficient KPIR solutions are based on Fully Homomorphic Encryption (FHE) [16], [17]. Roughly, the client sends an FHE-encrypted query to the server, which computes over the encrypted query and database entries, then returns an FHE-encrypted response that the client decrypts to obtain the result. Since FHE naturally supports arbitrary functions over encrypted data [18], a plausible approach is to replace the 2PC protocols in the analytics phase by outsourcing all computations on the encrypted response to the server.

Motivated by these challenges, we introduce a new concept, *keyword PIR with computation* ("KPIR-C"), which allows the server (or any other party) to perform arbitrary non-interactive computations over responses while preserving query privacy. KPIR-C enables outsourcing all computations to the server and, at the same time, opens up new application scenarios for KPIR. Here, we present two typical examples.

The first example is *predicate KPIR*, where a client wants to determine whether the retrieved entry satisfies a given predicate (i.e., a condition or property specified by the client). Although the client could retrieve the entry and compute the predicate locally, returning the (encrypted) entry incurs higher communication cost than returning only the (encrypted) predicate result. A naive solution for the server is to precompute the predicate over all entries and build a KPIR service where predicate results serve as entries. However, this becomes unsuitable *when the client needs to define the predicate at query time*. Moreover, the server must evaluate the predicate on the entire database, which has linear complexity. In contrast, KPIR-C supports predicate evaluation at query time with constant complexity, while returning minimal predicate responses.

Another example is *retrieving aggregated results from multiple keywords*. Assume a user utilizes a KPIR-based Bitcoin account lookup service to privately fetch the total number of Bitcoins across several accounts.[1] Instead of separately receiving and decrypting responses for each account, the user can request a single aggregated response, which reduces the number of decryptions and lowers communication and computation overhead.

1.1. Challenges in Constructing KPIR-C

There are two main challenges in constructing KPIR-C:

First, supporting arbitrary computations on responses is difficult for current KPIR solutions. Due to the concise functionality of KPIR, existing KPIR constructions have a relatively shallow circuit depth. Therefore, while KPIR can be constructed using leveled homomorphic encryption schemes such as BFV [19], [20] or BGV [21], these schemes become impractical for arbitrary computations on responses due to their inefficient bootstrapping [18], with a single bootstrapping operation taking approximately 35 seconds [22].

In contrast, third-generation FHE schemes such as GSW [23], FHEW [24], and TFHE [25], [26] offer significantly faster bootstrapping, which enables efficient computation at greater circuit depths. Specifically, in TFHE, a single bootstrapping operation takes only about 10 milliseconds, making it a promising candidate for building an efficient KPIR-C with support for complex server-side computation. Despite this, to the best of our knowledge, no (K)PIR scheme has been built on TFHE, and the efficiency of TFHE-based (K)PIR remains an open area for research and exploration.

TABLE 1: Overview of Prominent KPIR Schemes. Number of rounds is counted for a single query.

KPIR Scheme	KPIR-C	Return on Mismatch	#Rounds
[27]	✗	A random field value	1
[17], [16]	✗	Undefined ($\perp$)	2
[28], [29]	✗	Undefined ($\perp$)	2
[30], [31]	✗	Restricted to 0	1
TCWKPIR (ours)	✓	Any agreed-upon value	1
TCAKPIR (ours)	✓	Any agreed-upon value	2
TCAKPIR$^+$ (ours)	✓	Any agreed-upon value	2

Second, the "mismatch case" in KPIR-C, where the client's queried keyword is not present in the server's database, must be addressed, preferably in a non-interactive manner. Prominent KPIR schemes either return a random value [27] or are undefined (i.e., no meaningful response) [17], [16], [28] in the event of a mismatch. If subsequent computations are done, the response for the mismatched case would randomly disturb the computation result. Additionally, schemes that *inherently* support default values [30], [31] are typically restricted to a default value of 0, which is undesirable since 0 may be a valid entry. A more robust approach is to define a default value δ agreed upon by both parties to signal a mismatch (see Table 1).

To address this, one possible solution is to allow the server to obtain an encrypted bit indicating whether a match exists. This bit can then be used to obliviously retain the encrypted entry or replace it with a default value through a multiplexer operation [32]. While existing solutions [14] adopt this concept in an interactive setting, enabling the server to compute such indicator bits non-interactively requires homomorphic equality. However, as shown by Mahdavi and Kerschbaum [30], performing a single equality check under homomorphic encryption is impractical due to the high multiplicative depth of the equality circuit. To address this, they introduced a KPIR construction based on a specially designed constant-weight code that enables efficient homomorphic equality. Another approach is Pantheon [31], which leverages Fermat's little theorem to achieve homomorphic equality. Unfortunately, both methods rely on BFV rather than TFHE, so we still cannot enjoy arbitrary computations on responses with efficient bootstrapping.

1.2. Our Contributions

Below, we summarize our main contributions, which collectively establish the foundations of KPIR-C and its efficient realization over TFHE:

1) To the best of our knowledge, we take the first step in formally introducing KPIR-C, a new PIR primitive that enables the server to non-interactively and obliviously evaluate arbitrary circuits on client queries. We address the challenge of supporting arbitrary computation by employing TFHE as the main building block for KPIR. In addition, we tackle the mismatch challenge by introducing a subroutine of KPIR-C, namely keyword PIR with default ("KPIR-D"), also in the non-interactive setting. Informally, KPIR-D allows the server to obliviously assign a predefined default value δ to mismatched query responses, thereby eliminating the disturbances in computation that would otherwise result from such mismatches.
2) Arbitrary computations with efficient bootstrapping in TFHE naturally enable expressive, non-interactive computation over responses. However, bootstrapping remains the dominant cost. To mitigate this, we introduce a *KPIR-Specific Bootstrapping Strategy* (Section 3.6) that exploits the structure of KPIR to minimize the number of bootstrapping operations required in the KPIR component of KPIR-C.
3) Next, to realize KPIR-C, we must first construct a suitable KPIR scheme. However, extending prior KPIR

1. See https://github.com/blyssprivacy/clients as an example.

constructions directly to KPIR-C is not feasible. To bridge this gap, we design two KPIR constructions specifically tailored for KPIR-C, which we collectively refer to as TKPIR.

a) The first construction (TCWKPIR, Figure 2) builds on constant-weight encoding, previously used to realize KPIR over BFV/BGV [30]. By transitioning from BFV/BGV to TFHE, we obtain a substantial communication improvement for constant-weight KPIR schemes—a $9.7\times$ reduction for a database with 2^{16} entries—at the cost of only a $1.2\times$ runtime slowdown. This trade-off is acceptable given the scheme's support for post-computation. Crucially, our KPIR-specific bootstrapping strategy (Section 3.6) ensures runtime remains competitive, even though current TFHE implementations [33] do not yet support batching.
b) The second construction (TCAKPIR, Figure 3) departs from equality-based KPIR and introduces a client-aided protocol, where the client assists in generating the selection vector. This design enables an extremely fast protocol, achieving more than a $40\times$ speedup over TCWKPIR on a database with 2^{16} entries. Although the communication now scales linearly with the database size, the concrete costs remain acceptable due to the small size of TFHE ciphertexts. To further optimize efficiency, we shift most of the computation to an offline phase, yielding the TCAKPIR$^+$ variant, which provides more than a $9\times$ online runtime improvement over TCAKPIR without incurring any additional communication overhead.

4) Finally, to the best of our knowledge, no existing (K)PIR scheme has been instantiated over TFHE, and we are the first to investigate the efficiency of TFHE-based (K)PIR, which is essential for realizing KPIR-C.

1.3. Related Work

The idea of Keyword PIR dates back to the work of Chor et al. [9]. More recent works enable single-round KPIR via FHE. Specifically, [10], [34], [27] showed that symmetric (with server privacy) KPIR can be realized efficiently by asking the server to homomorphically evaluate two polynomials: a key-interpolated polynomial and a value-interpolated polynomial. In [17], Cuckoo hashing is used to map sparse keywords to an array, followed by multiple invocations of index PIR to realize KPIR. Subsequent work [35] improves the efficiency by replacing Cuckoo hashing with OKVS. Alternatively, Pantheon [31] and CWKPIR [30] use equality operators to perform KPIR. CWKPIR queries are no longer dependent on the entry length but instead on the Hamming weight, using a special encoding technique called constant-weight encoding. In another line of work, ChalametPIR [28] constructs the first hint-based KPIR. It allows the server to share a hint with the client so that the query phase is extremely efficient. ChalametPIR is concretely instantiated using FrodoPIR [36] and the unique OKVS named Binary Fuse Filters [37]. Most recently, Hao et al. [29] extend the idea of ChalametPIR into SimplePIR [38] to achieve sublinear query communication. In this work, we focus on keyword PIR with computation (KPIR-C)—a new primitive that can be realized through carefully designed KPIR schemes adhering to its definition (Section 3.1). A parallel goal of this work is to realize KPIR-C while keeping the additional overhead minimal.

2. Preliminaries

2.1. Notations

We use λ to denote the security parameter, and $\mathsf{negl}(\lambda)$ to represent a negligible function in λ. The notation $[n]$ denotes the set $\{1, 2, \ldots, n\}$. Similarly, $[a{:}b]$ denotes the set $\{a, a+1, \ldots, b\}$ for integers $a \leq b$. The expression $a \leftarrow\$ A$ indicates that a is sampled uniformly at random from the set A. If A is a distribution, then $a \leftarrow A$ denotes that a is sampled according to A. We write $a \leftarrow \mathrm{A}(x)$ to indicate that a is the output of the randomized algorithm A on input x, and $a := b$ to denote the assignment of the value of b to a. In addition to assignment, we use $:=$ to define constants, variables, functions, distributions, and other symbolic objects. We write $\mathbb{I}[\cdot]$ for the indicator function, which outputs 1 if the condition holds and 0 otherwise.

2.2. 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 [16], [17], [30], 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 containing m key-value pairs. Each keyword k belongs to the key space $\mathcal{K}$, and its associated value v is sampled from the value space $\mathcal{V}$. 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.
- $q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw})$: On input the public parameters pp, the client's query key qk, and a keyword kw, the query algorithm outputs a query q.
- $r \leftarrow \mathsf{Response}(\mathsf{pp}, q)$: On input the public parameters pp and a query q, the response algorithm outputs a response r corresponding to the query q.
- $\{v, \perp\} \leftarrow \mathsf{Recover}(\mathsf{qk}, r)$: On input the query key qk and a response r, the recover algorithm outputs a value v or a special symbol $\perp$.

Correctness. When the client and server honestly execute the KPIR scheme, the client retrieves either the intended value corresponding to the queried keyword in the database or $\perp$, indicating a mismatch. Formally, a KPIR scheme has a 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 every keyword $\mathsf{kw} \in \mathcal{K}$, the following probability is at least $1 - \mathsf{negl}(\lambda)$:

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

Security. The client's query should not reveal any information about the keyword being retrieved. Formally, a KPIR scheme is secure if, for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$ operating on a database of size m, and for all $\mathsf{kw}_i, \mathsf{kw}_j \in \mathcal{K}$,

$$\begin{aligned} &|\Pr[\mathcal{A}(1^m, q) = 1 : q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_i)] \\ &- \Pr[\mathcal{A}(1^m, q) = 1 : q \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_j)]| \le \mathsf{negl}(\lambda) \end{aligned}$$

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 a binary encrypted vector that allows the server to obliviously select the entries matching the client's query. It can be constructed either with the help of the client [28] (at the expense of communication) or fully offloaded to the server [30] (at the expense of computation).

2.3. Constant-Weight Code

A constant-weight code is an error-detecting code where all codewords have the same Hamming weight. Mahdavi and Kerschbaum [30] were the first to apply constant-weight codes in KPIR. They designed an encoding algorithm (Algorithm 1) 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 $\binom{\rho}{h} \ge 2^{\ell_k}$. They observed that equality between two FHE-encrypted constant-weight codes can be checked efficiently, effectively reducing the multiplicative depth of the equality circuit from $O(\log \ell_k)$ to $O(h)$.

2.4. Key-Value Stores

A key-value store (KVS) [39], [40] is a data structure that encodes a set $\mathcal{M}$ of key-value pairs $(k, v) \in \mathcal{K}\times\mathcal{V}$. Formally, a KVS is parameterized by a key space $\mathcal{K}$, a value space $\mathcal{V}$, the number of input key-value pairs m, the output length M, and the number of hash functions κ. It consists of two algorithms:

- $(F, H) \leftarrow \mathsf{write}(\mathcal{M})$: On input a set of key-value pairs $\mathcal{M} := \{(k_i, v_i)\}_{i\in[m]} \in (\mathcal{K}\times\mathcal{V})^m$, outputs a vector F and a set of hash functions $H := \{h_i\}_{i\in[\kappa]}$.

Algorithm 1 CWEncode

```
Input: Keyword kw, Codeword length $\rho$, Hamming weight $h$, s.t. $\binom{\rho}{h} \ge 2^{\ell_k}$
1: $\mathbf{c} := [0]^\rho$    ▷ Initialize codeword as a zero vector of length $\rho$
2: $r := \mathsf{kw}$
3: $h_r := h$
4: for $l = \rho - 1$ to $0$ do
5:     $t := \binom{l}{h_r}$
6:     if $r \ge t$ then
7:         $\mathbf{c}[l] := 1$
8:         $r := r - t$
9:         $h_r := h_r - 1$
10:        if $h_r = 0$ then
11:            break
Return: Codeword $c$ of length $\rho$ with Hamming weight $h$.
```

- $v \leftarrow \mathsf{recon}(H, k, F[H(k)])$: On input a set of hash functions $H := \{h_i\}_{i\in[\kappa]}$, a key $k \in \mathcal{K}$, and entries in the vector $F[H(k)] := \{F[h_i(k)]\}_{i\in[\kappa]}$, outputs a value $v \in \mathcal{V} \cup \{\perp\}$.

Since the set H contains κ hash functions, recon must access k entries in the vector F to reconstruct the value for the given key. Below, we demonstrate the correctness of both inclusion and non-inclusion [28]:

Correctness of Inclusion. A KVS is *correct for inclusion* if, for all $\mathcal{M} \in (\mathcal{K}\times\mathcal{V})^m$ with distinct keys and any $(k, v) \in \mathcal{M}$, the reconstruction algorithm satisfies $\mathsf{recon}(H, k, F[H(k)]) = v$ with probability 1, where $(F, H) \leftarrow \mathsf{write}(\mathcal{M})$.

Correctness of Non-inclusion. A KVS is *correct for non-inclusion* if, for all $\mathcal{M} \in (\mathcal{K}\times\mathcal{V})^m$ with distinct keys and any $k \notin K$, the reconstruction algorithm satisfies $\mathsf{recon}(H, k, F[H(k)]) =\perp$ with overwhelming probability, where $(F, H) \leftarrow \mathsf{write}(\mathcal{M})$ and K is the set of keys in $\mathcal{M}$.

Instantiation from Binary Fuse Filters. Binary Fuse Filters (BFFs), introduced by Graf and Lemire [37], provide an elegant instantiation of KVS with $k := 3$ hash functions. Compared to other constructions [39], [40], the size M does not exceed $1.156m$, resulting in a more compact encoding vector. In this paper, we use `BFF.write` to indicate that the key-value store is instantiated with BFF.

2.5. FHE over the Torus

Torus Fully Homomorphic Encryption (TFHE) is a leading third-generation FHE scheme built upon the GSW [23] and FHEW [24] schemes. Its key distinguishing feature is highly efficient programmable bootstrapping, which enables the evaluation of arbitrary functions while simultaneously refreshing ciphertexts by reducing noise.

TFHE is built on the torus $(\mathbb{T}, +)$, an Abelian group where elements lie in the set $[0, 1)$ of real numbers modulo 1 (i.e., $\mathbb{T} := \mathbb{R}/\mathbb{Z}$). Any element $t \in \mathbb{T}$ can be written as an infinite base-B expansion $(t_1, t_2, \dots)_B$, where $B \in \mathbb{Z}, B \ge 2$, and $t := \sum_{j=1}^{\infty} t_j B^{-j}$ with $t_j \in \{0, \dots, B-1\}$. In practice, torus elements are represented using finite precision

(32-bit or 64-bit) due to processor limitations. For binary representation, this gives $t := \sum_{i=1}^{\Omega} t_i \cdot 2^{-i} \bmod 1$, where $\Omega \in \{32, 64\}$ and $t_i \in \{0,1\}$. This finite representation replaces $\mathbb{T}$ with the submodule $\mathbb{T}_q := q^{-1}\mathbb{Z}/\mathbb{Z} \subseteq \mathbb{T}$, where $q := 2^{\Omega}$. The resulting elements take the form $\frac{i}{q} \bmod 1$ for $i \in \mathbb{Z}$, forming what is known as the *discretized torus*.

TFHE operates with four types of ciphertexts: TLWE, TGSW, TGLWE, and TGGSW.[2] In this section, we focus on TLWE ciphertexts. See Appendix A for details on other TFHE ciphertext types.

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 $\mathsf{TLWE}_{\mathbf{s}}(\mu)$.
- $\mu \leftarrow \mathsf{Decrypt}_{\mathbf{s}}(\mathsf{ct}, \mathsf{pp})$: On input $\mathsf{ct} := \mathsf{TLWE}_{\mathbf{s}}(\mu)$ 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 $v := \lfloor \mu \cdot p \rceil \bmod p \in \mathbb{Z}$.
- $\mathsf{TLWE}_{\mathbf{s}}(f(\mu)) \leftarrow \mathsf{PBS}_{\mathsf{evk}}(f, \mathsf{TLWE}_{\mathbf{s}}(\mu))$: On input a function $f : \mathbb{T}_p \to \mathbb{T}_p$ and ciphertext $\mathsf{ct} := \mathsf{TLWE}_{\mathbf{s}}(\mu)$ with noise e, returns $\mathsf{ct}' := \mathsf{TLWE}_{\mathbf{s}}(f(\mu))$ with reduced noise $e' < e$.
- $\mathsf{TLWE}_{\mathbf{s}}(\mu) \leftarrow \mathsf{Bootstrap}_{\mathsf{evk}}(\mathsf{TLWE}_{\mathbf{s}}(\mu))$: On input $\mathsf{ct} := \mathsf{TLWE}_{\mathbf{s}}(\mu)$, returns $\mathsf{PBS}_{\mathsf{evk}}(f, \mathsf{ct})$ with f as the identity.
- $\mathsf{TLWE}(\mu_1 + \mu_2) \leftarrow \mathsf{HomAdd}(\mathsf{TLWE}(\mu_1), \mathsf{TLWE}(\mu_2))$: On input TLWE ciphertexts of $\mu_1, \mu_2 \in \mathbb{T}_p$, returns $\mathsf{TLWE}(\mu_1 + \mu_2)$.
- $\mathsf{TLWE}(z \cdot \mu) \leftarrow \mathsf{ScalarMul}(z, \mathsf{TLWE}(\mu))$: On input a "small" $z \in \mathbb{Z}$ and a TLWE ciphertext of $\mu \in \mathbb{T}_p$, returns $\mathsf{TLWE}(z \cdot \mu)$.[3]

Remark 1. *While* $\mathsf{PBS}_{\mathsf{evk}}(f, \mathsf{ct})$ *is defined for univariate functions, it naturally extends to multivariate functions. By Kolmogorov's superposition theorem [42], any multivariate function can be expressed as a linear combination of univariate functions. Consequently, using* HomAdd *and* PBS*, we can implement arbitrary multivariate functions. We write* $\mathsf{PBS}_{\mathsf{evk}}(\mathcal{F}, \mathbf{c})$*, where* $\mathcal{F} : \mathbb{T}_p^{|\mathbf{c}|} \to \mathbb{T}_p$ *is a multivariate function, and* $\mathbf{c}$ *is a vector of ciphertexts.*

PBS thus enables the evaluation of circuits such as HomAND (homomorphic bitwise AND), HomMul (homomorphic multiplication), and HomEq (homomorphic equality check) [25]. See Appendix A for details on TFHE and its bootstrapping procedure.

2. The original TFHE work [26] uses the terms TLWE, TGSW, TRLWE, and TRGSW. We follow the naming conventions from [41].

3. The product $z \cdot \mu$ is well defined in the additive group $\mathbb{T}_p \subseteq \mathbb{T}$ [41].

3. Keyword PIR with Computation

3.1. Definition

Defining KPIR-C is not trivial, as it cannot be achieved by simply allowing the server to perform subsequent computations over the response. The difficulty lies in handling the keyword mismatch case, which may result in undefined outputs during subsequent computations.

KPIR with Default. To support subsequent computations over the response r, even in the mismatched case, we adopt the concept introduced by Lepoint et al. [14] by defining a variant of KPIR called *KPIR with default* (KPIR-D). This primitive ensures that the client always obtains a well-defined output: either the value corresponding to the queried keyword or a predetermined default value when no match exists. Crucially, the server learns nothing about the query or whether the returned result corresponds to an actual database entry or the default value.

To allow KPIR-D functionality, we assume that each client and the server can agree on some default value(s) $\delta \in \mathcal{V}$. We then modify the Recover algorithm by replacing $\perp$ with δ. Since δ is also a valid value in $\mathcal{V}$, it does not need to be explicitly distinguished from the "actual" value v in the matched case. 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 so that the client can decrypt and retrieve δ. To achieve this, the client is required to share an evaluation key evk corresponding to its query key qk with the server. This leads to additional modifications in the client setup algorithm and the response algorithm, as follows:

- $(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp})$: On input the public parameters pp, the client setup algorithm outputs an evaluation key evk and a query key qk.
- $r \leftarrow \mathsf{Response}(\mathsf{pp}, \mathsf{evk}, q)$: On input the public parameters pp, a query q from a client, and its evaluation key evk, the response algorithm outputs a response r.
- $v \leftarrow \mathsf{Recover}(\mathsf{qk}, r)$: On input the query key qk and a response r, the recover algorithm outputs a value v.

The *correctness* should ensure that the client obtains the default value in the mismatched case. Formally, for $i \in [m]$, the following probability is at least $1 - \mathsf{negl}(\lambda)$:

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

KPIR with Computation. We now define KPIR-C, which introduces a new algorithm named Evaluate run by the server to obliviously evaluate a circuit $\mathcal{F}$ on the response r, so that the client can retrieve $v' := \mathcal{F}(v)$.

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

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

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

Security for KPIR-C. We formally define security for KPIR-C (and similarly for KPIR-D) in Appendix B.1. Informally, a KPIR-C scheme is secure if the server cannot learn anything about the client's query, nor can it infer any additional information from the response (whether the protocol fails or a mismatch occurs).

Remark 2. *W.l.o.g., we can extend the KPIR-C definition to cross-query computations (i.e., evaluating a circuit over multiple queries). To see this, let n denote the number of queries. We can invoke $r_i \leftarrow \mathsf{Response}(\mathsf{pp}, q_i, \mathsf{evk})$, $i \in [n]$, and evaluate the circuit on the response vector $\mathbf{r} := (r_1, \ldots, r_n)$ by invoking $\mathsf{Evaluate}_{\mathcal{F}}(\mathsf{pp}, \mathsf{evk}, \mathbf{r})$. As noted in Remark 1, we can let* PBS *evaluate an arbitrary circuit $\mathcal{F}$ on any number of ciphertexts (i.e., responses).*

Remark 3. *With a slight modification to the definition of KPIR-C, the server can return δ directly instead of $\mathcal{F}(\delta)$. This is achievable by a multiplexer that replaces δ with $\mathcal{F}(\delta)$.*

3.2. Constructing KPIR-C in Practice

In Section 1, we discussed the main challenges in constructing a KPIR-C scheme. We explained why TFHE [33] is employed to address the first challenge (namely, supporting arbitrary computation). However, in its generic usage [33], TFHE ciphertexts are bootstrapped aggressively (see Section 4.3, Table 4) to control both noise growth and modular overflow. Although TFHE's bootstrapping is substantially faster than that of earlier generations, it still dominates the overall computational cost. Consequently, if KPIR-C were instantiated directly using TFHE in this generic mode, the resulting bootstrapping overhead would render the scheme less practical than existing KPIR constructions based on leveled HE [30], [17], [27] or LWE-based approaches [12], [28]. To overcome this limitation, we propose a *KPIR-Specific Bootstrapping Strategy* that leverages the structural properties of KPIR (Section 3.6) and minimizes the number of bootstrapping operations at the KPIR level within KPIR-C.

Next, we discussed the need to compute an indicator bit to obliviously detect mismatches and replace them with a default value. For this purpose, CWKPIR [30] and Pantheon [31] stood out as two viable candidates. Essentially, both schemes employ efficient oblivious equality techniques to construct a one-hot selection vector encrypting a single 1 at the queried keyword's index, which can later be used to construct KPIR-D (see Section 3.3). However, Pantheon constructs its selection vector using an oblivious equality check based on Fermat's Little Theorem, which is incompatible with practical implementations of TFHE [33] that rely on a power-of-two torus modulus (see Section 2.5) rather than a prime modulus. Consequently, CWKPIR appears to be the only viable option; however, it lacks support for default values and is built on BFV, whereas our setting requires TFHE. We therefore borrow the idea of constant-weight encoding to construct our first scheme, TCWKPIR, over the torus (see Section 3.3).

An appealing feature of this construction is that it offloads selection vector generation to the server and reduces query size by a factor of $O(m/\rho)$, where m is the database size and ρ the codeword length. TCWKPIR further reduces query size using compact TLWE ciphertexts (see Section 2.5). However, this comes at the cost of $O(h)$ homomorphic equality tests per entry on the server, where h denotes the Hamming weight. While acceptable for small keyword spaces, larger Hamming weights needed for sparse databases lead to significant computational overhead due to increased homomorphic operations (see Section 4.2).

This limitation motivates an alternative approach: having the client generate the selection vector in plaintext, encrypt it, and send it to the server. We refer to any scheme of this type as *Client-Aided*. While this increases query size by $O(m/\rho)$, the concrete communication cost remains acceptable when using LWE-based encryption [43]. Since TLWE ciphertexts can be viewed as a variant of LWE ciphertexts, it appears feasible to construct TFHE-based KPIR by following the LWE-based paradigm [36], [38], [28].

Two challenges arise in this context. First, the only known LWE-based KPIR construction at the time was ChalametPIR [28],[4] which requires appending the hash (digest) of each keyword to every database entry. In ChalametPIR, the key part is returned as part of the response and verified client-side; however, in KPIR-C, the server must instead evaluate an equality circuit between the queried digest and the response digest to generate an encrypted selector bit that enforces the default value upon mismatch.

Second, ChalametPIR leverages the fact that the "mask" part of the LWE ciphertext can be fixed across queries and sent to the client in the offline phase (as a "hint"), while only the "body" part depends on the query and is generated using a fresh query key in the online phase. This yields an extremely efficient online phase; however, to support arbitrary computations on the response (as required by KPIR-C), we cannot precompute any hint, since the circuit can be arbitrary. Additionally, the query key must remain fixed during the setup phase,[5] and the server must instead sample a fresh random mask for each query in the online phase. This is necessary because reusing the same mask can leak

4. Hao et al. [29] was not yet available. We benchmark their scheme and discuss its trade-offs in Section 5.1.

5. Cross-query computation under different keys requires server-side key switching, which is costly and adds noise. It also increases communication, as the client must send a separate evaluation key for each secret key. Hence, this approach is impractical for our construction.

information about the client's queries [44], [38], [41]. While this increases the server's online cost, it reverts to the standard KPIR setting and eliminates the need for a hint in the offline phase.

Building on these observations, we introduce our second non-interactive torus-based KPIR-C scheme: TCAKPIR (see Section 3.4). After formally describing this construction, we present a simple trick that allows us to achieve online performance comparable to "hint"-based KPIR schemes (e.g., [28]) without introducing additional communication rounds, and denote this optimized variant by TCAKPIR$^+$.

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

3.3.1. Construction Details. Here, we detail our TCWKPIR construction for a single query. As noted in Remark 2, it can be readily extended to support multiple queries. TCWKPIR requires only one round of communication: the client sends a query, and the server performs all computations and returns the *processed* value corresponding to the queried keyword.

Figure 1 outlines our TCWKPIR construction, which consists of five major steps: ① The client encodes its query (i.e., keyword kw) as a constant-weight codeword (Algorithm 1), encrypts and sends it to the server. ② As part of preprocessing, the server encodes all keywords into constant-weight codewords and performs an equality check (Algorithm 2) between the client's query and each encoded keyword to construct the selection vector sv. ③ The server computes the inner product (Algorithm 3) between sv and the database values to obliviously "select" the row corresponding to the encrypted 1's position in sv. ④ The server then uses a multiplexer to obliviously handle mismatches, returning a default value δ when they occur. ⑤ The client decrypts the response and recovers the result.

We formally describe our KPIR-C scheme in Figure 2. As per Remark 3, during the $\mathsf{Evaluate}_{\mathcal{F}}$ stage, the server can obliviously return δ instead of $\mathcal{F}(\delta)$ in case of a mismatch. This is feasible by evaluating a multiplexer that (re)computes each r'_j as $r'_j := \mathsf{sel} \cdot (r'_j - \delta_j) + \delta_j$, where $\mathsf{sel} := \sum_{i=1}^{m} \mathsf{sv}_i$, $j \in [\omega]$, and the values δ_j satisfy $\delta = \sum_{j=1}^{\omega} \delta_j \cdot p^{j-1}$. Interestingly, this can also be viewed as defining the circuit $\mathcal{F}$ to perform the multiplexing internally and output δ.

Algorithm 2 CWEq: Bitwise Constant-Weight Eq. Check

```
Input: (c: encoded query (ciphertext),
kw: encoded keyword (plaintext))
1: Initialize $e$
2: for $i = 1$ to $h$ do
3:     if $\mathsf{kw}[i] = 1$ then
4:         if $e$ is the initial value then
5:             $e \leftarrow \mathbf{c}[i]$
6:         else
7:             $e \leftarrow \mathsf{HomAND}(e, \mathbf{c}[i])$
Return: $e \in \mathbb{T}_q^{n+1}$
```

Supporting Default Values. To handle cases where no keywords match the client's query, we first present the pseudocode for the constant-weight equality check (CWEncode) in Algorithm 2. By construction, the algorithm encrypts 0 for each client-server encoded keyword pair that does not match. As a result, when no keywords in the database match the query, the selection vector consists entirely of 0s, and the resulting inner product evaluates to 0.

Algorithm 3 InnerProduct

```
Input: Selection vector sv, entries $D \in \mathbb{Z}_p^{m \times \omega}$, and evaluation key evk
1: $\mathbf{r} := (r_1, \ldots, r_\omega)$, where $r_i := \mathrm{TLWE}(0) \in \mathbb{T}_q^{n+1}$ for $i \in [\omega]$
2: for $i = 1$ to $\omega$ do
3:     for $j = 1$ to $m$ do
4:         $t' \leftarrow \mathsf{ScalarMul}(D_{ji}, \mathsf{sv}_j)$
5:         $r_i \leftarrow \mathsf{HomAdd}(r_i, t')$
6:         $r_i \leftarrow \mathsf{Bootstrap}_{\mathsf{evk}}(r_i)$          ▷ Perform as needed
Return: Response $\mathbf{r}$
```

Letting the default value $\delta := 0$, Algorithm 2 aligns, as is, with the definition of KPIR-D (Section 3.1). Let sv denote the selection vector constructed obliviously by the server. It computes $\mathsf{sel} := \sum_{i=1}^{m} \mathsf{sv}_i$ and returns the ciphertext $\mathsf{sel} \cdot r$. Upon decryption, the client receives 0 if no keyword matched. However, a more desirable approach is to allow δ to be set arbitrarily, according to the protocol's specifications. To achieve this, the server instead applies a homomorphic multiplexer and responds with $\mathsf{sel} \cdot (r - \delta) + \delta$. The client then receives either δ (no match) or the actual result.

Security and Correctness. We formally prove the security and correctness for TCWKPIR in Appendix B.3, Theorem 2, and Appendix B.4, Theorem 4, respectively.

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

3.4.1. Construction Details. In this section, we present our construction for TCAKPIR. Similar to the first construction, we assume that the server holds a database containing key-value pairs, and the client aims to retrieve the value (or a post-processed value) corresponding to a queried key, which may not exist in the key-value map. We formally describe our protocol in Fig. 3.

Supporting Default Values. In a standard KPIR scheme, the client decrypts the inner product result and checks whether the first few bytes of the plaintext (the digest) match the queried keyword. A mismatch may indicate that the keyword is absent or that the KPIR query failed. We incorporate this idea into our construction by enabling the server to perform the check obliviously and return an encrypted default value when no match is found. To do so, the client sends the encrypted digest of its query along with the encrypted selection vector. The server extracts and stores this encrypted digest, denoted $\mathbf{d}$, and performs a homomorphic equality test HomEq between $\mathbf{d}$ and the digest $\mathbf{d}'$ embedded in the inner product result. This yields a selector bit sel. The server then uses a multiplexer to compute $\mathsf{sel} \cdot (r - \delta) + \delta$, where

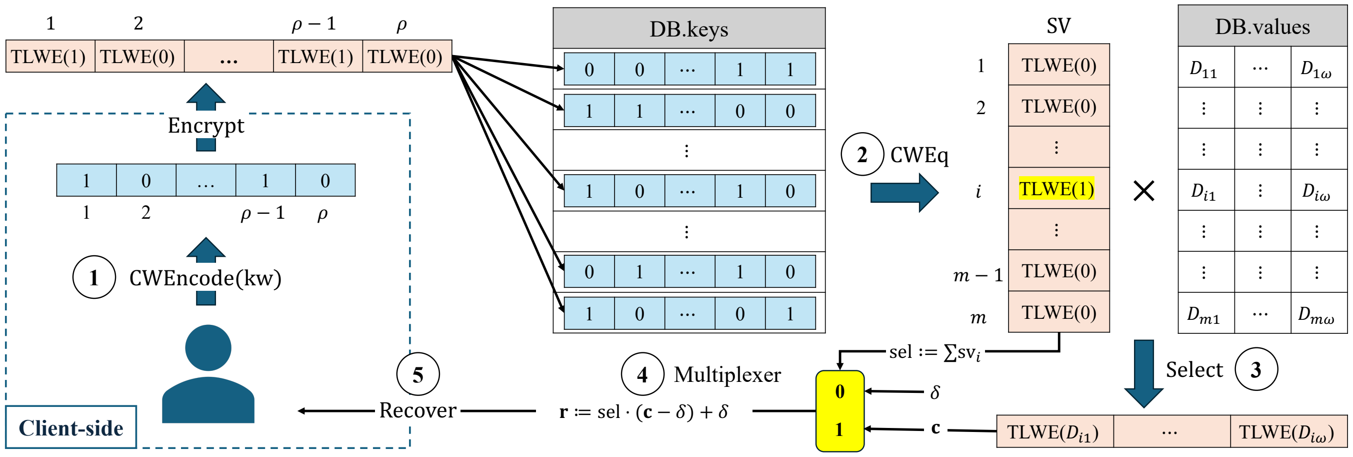


Figure 1: TCWKPIR Construction Overview. $\omega := \lceil \ell_d / \log p \rceil$ is the count of entry partitions, with ℓ_d as the max data bit length.

Protocol TCWKPIR

Description: A KPIR-C scheme $(\mathsf{Setup}, \mathsf{ClientSetup}, \mathsf{Query}, \mathsf{Response}, \mathsf{Recover}, \mathsf{Evaluate})$ 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 $\mathcal{F}(\mathsf{DB}[\mathsf{kw}]) \in \mathcal{V}$ if $\mathsf{kw} \in \mathsf{DB.keys}$, or otherwise a default value—either δ or $\mathcal{F}(\delta)$—where $\mathcal{F}$ is a post-processing circuit.
Parameters: Security parameter λ, Hamming weight h, database size m, data bit length ℓ_d, codeword length ρ, and a default value δ.
Protocol execution:
$\mathsf{pp} \leftarrow \mathsf{Setup}(\mathsf{DB})$:

- The server defines the public parameters $\mathsf{pp} := (n, \sigma, p, q)$.
- The server pre-processes the database 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.

$(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp})$:

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

$\mathbf{q} := (q_1, \ldots, q_\rho) \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw})$:

- 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 $q_i \leftarrow \mathsf{Encrypt}_{\mathsf{qk}}(\mathsf{Encode}(\mathsf{cw}_i)), i \in [\rho]$.

$\mathbf{r} \leftarrow \mathsf{Response}(\mathsf{pp}, \mathsf{evk}, \mathbf{q})$:

- 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 $\delta = \sum_{j=1}^{\omega} \delta_j \cdot p^{j-1}$.

$\mathbf{r}' \leftarrow \mathsf{Evaluate}_{\mathcal{F}}(\mathsf{pp}, \mathsf{evk}, \mathbf{r})$:

- The server evaluates the response $\mathbf{r}$ on some arbitrary function(s) $\mathcal{F}$ by invoking $\mathbf{r}' \leftarrow \mathsf{PBS}_{\mathsf{evk}}(\mathcal{F}, \mathbf{r})$.

$v' \leftarrow \mathsf{Recover}(\mathsf{qk}, \mathbf{r}')$:

- The client decrypts the response $\mathbf{r}'$ by invoking $v_i' \leftarrow \mathsf{Decode}(\mathsf{Decrypt}_{\mathsf{qk}}(r_i')), i \in [\omega]$.
- Next, the client reconstructs the value v' as $v' := \sum_{i=1}^{\omega} v_i' \cdot p^{i-1}$ and compares it to the default value δ (or $\mathcal{F}(\delta)$).

Figure 2: Our First Construction: TCWKPIR

r is the inner product result (or some post-processed value), and δ is the default.

Supporting Arbitrary Queries. Recall from Section 3.2 that supporting multiple queries—and thus cross-query computation—in TCAKPIR requires fixing the query key qk, which forces fresh random masks per query. This motivates

Protocol TCAKPIR

Description: A KPIR-C scheme (Setup, ClientSetup, Query, Response, Recover, Evaluate) 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 $\mathcal{F}(\mathsf{DB}[\mathsf{kw}]) \in \mathcal{V}$ if $\mathsf{kw} \in \mathsf{DB.keys}$, or otherwise a default value—either δ or $\mathcal{F}(\delta)$—where $\mathcal{F}$ is a post-processing circuit.
Parameters: Security parameter λ, data bit length ℓ_d, database size m, a hash function $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^\mu$, and a default value δ.
Protocol execution:
$\mathsf{pp} \leftarrow \mathsf{Setup}(\mathsf{DB})$:

- The server defines the public parameters $\mathsf{pp} := (n, \sigma, p, q)$ and 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 $(F, H) \leftarrow \texttt{BFF.write}(\mathcal{M})$, and obtains the filter storage $F \in \mathbb{Z}_p^{M \times \omega}$ and κ hash functions $H(x) := (h_1(x), \ldots, h_\kappa(x))$.

$(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp})$:

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

$\mathbf{q} := (q_1, \ldots, q_{M+\omega'}) \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw})$:

- 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 $q_{M+i} \leftarrow \mathsf{Encrypt}_{\mathsf{qk}}(\mathsf{Encode}(\overline{d}_i))$.

$\mathbf{r} \leftarrow \mathsf{Response}(\mathsf{pp}, \mathsf{evk}, \mathbf{q})$:

- 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 $\mathsf{r} := \mathsf{z}[\omega'{:}\omega]$.
- 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 $\bar{\omega} := \lceil \ell_d / \log p \rceil$.

$\mathbf{r}' \leftarrow \mathsf{Evaluate}_{\mathcal{F}}(\mathsf{pp}, \mathsf{evk}, \mathbf{r})$:

- The server evaluates the response $\mathbf{r}$ on some arbitrary function(s) $\mathcal{F}$ by invoking $\mathbf{r}' \leftarrow \mathsf{PBS}_{\mathsf{evk}}(\mathcal{F}, \mathbf{r})$.

$v' \leftarrow \mathsf{Recover}(\mathsf{qk}, \mathbf{r}')$:

- The client decrypts the response $\mathbf{r}'$ by invoking $\mathbf{v} := [\, \mathsf{Decode}(\mathsf{Decrypt}_{\mathsf{qk}}(r'_i)) \,]_{i \in [\bar{\omega}]}$.
- The client then reconstructs the value $v' := \sum_{i=1}^{\bar{\omega}} v[i] \cdot p^{i-1}$ and compares it to the default value δ (or $\mathcal{F}(\delta)$).

Figure 3: Our Second Construction: TCAKPIR

offloading more work to preprocessing to reduce online runtime. Ideally, we aim to achieve the same communication efficiency as ChalametPIR [28], while maintaining a comparably fast online phase. We now present an optimized variant of TCAKPIR, denoted as TCAKPIR⁺.

Optimization: Offloading More to Preprocessing. It is well known that a secure PRG allows the generation of a practically unlimited number of pseudorandom seeds. Therefore, the server needs to share a single uniformly random mask seed with the client only *once*, after which both parties can derive any number of masks. For practical reasons, we assume the client and server agree in advance on a large fixed number C, which represents the total number of queries supported in one preprocessing phase. Once they reach this limit, they rerun the preprocessing step without sharing a new seed.

Specifically, in the Setup phase (in Fig 3), the server performs two additional steps:

- The server generates $\theta \leftarrow\$ \{0,1\}^\lambda$. Then, it samples $t := M \cdot C + 1$ pseudorandom seeds $\tilde{\theta} := (\tilde{\theta}_1, \ldots, \tilde{\theta}_t) \leftarrow \mathsf{PRNG}(\theta)$. For each $i \in [C]$, the server samples a matrix $A_i \in \mathbb{T}^{M \times n}$, 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. The server stores A_i and shares θ with the client, who, upon receiving θ, performs the same computation to obtain and store $\tilde{\theta}$.
- Next, the server precomputes the masked portion of the inner product. 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 by the server.

Next, in the Query phase, after constructing the cleartext selection vector $\overline{\mathsf{sv}}$, 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 $B[i][j] :=$

TABLE 2: Comparison of TCAKPIR, TCAKPIR$^+$, and ChalametPIR; security parameter λ, LWE dimension n, number of queries C, ciphertext modulus q, storage length M, and $\omega := (\bar{\omega} + \omega')$; $\bar{\omega}$ and ω' denote the number of entry and digest partitions.

Scheme	Storage (bits)	Prep. Comm. (bits)		Online Comm. (bits)		Total Online
	Client	Client $\to$ Server	Server $\to$ Client	Client $\to$ Server	Server $\to$ Clinet	Time (complex.)
ChalametPIR [28]	$\omega n \log q + (C+1)\lambda$	-	$\omega n \log q + \lambda$	$CM \log q$	$\omega C \log q$	$O(\omega CM)$
TCAKPIR	$(C+1)\lambda$	$C\lambda$	-	$C(M+\omega')\log q$	$\bar{\omega}C(n+1)\log q$	$O(\omega CnM)$
TCAKPIR$^+$	$(\omega CM+1)\lambda$	-	λ	$C(M+\omega')\log q$	$\bar{\omega}C(n+1)\log q$	$O(\omega CM)$

$\sum_{k=1}^{n} A_i[j][k] \cdot \mathsf{qk}[k] + \mathsf{sv}[j] + e$, for $i \in [C]$ and $j \in [M]$.

In the Response phase, the server computes the matrix product $B' := B \times F \in \mathbb{T}_q^{C\times\omega}$, and prepends the precomputed mask part A'_i to construct the responses $\mathscr{R}_i := (A'_i[w], B'[i][w]) \in \mathbb{T}_q^{n+1}$ 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 2, we provide a comprehensive complexity analysis of TCAKPIR, TCAKPIR$^+$, and ChalametPIR. As discussed in Section 3.2, the first difference between ChalametPIR and TCAKPIR$^+$ is that, to satisfy KPIR-C, we cannot send any hint 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 total communication and runtime of TCAKPIR$^+$ remain the same as ChalametPIR (refer to Section 3.5.2 to see how we further reduce communication). However, we benefit from reduced communication over C queries, at the cost of the client's storage. In practice, this won't be a problem since C can be tuned based on available resources (e.g., client's storage), and preprocessing can be done in the background once the C limit is reached.

Security. We formally prove the security of TCAKPIR in Appendix B.3, Theorem 3. For TCAKPIR$^+$, recall that the random masks are already publicly known. Therefore, we establish its security via a reduction to TCAKPIR, arguing that the preprocessed randomness in TCAKPIR$^+$ perfectly simulates the randomness generated by TFHE. This holds as long as the PRG is secure and the same TFHE parameters are used (see Section 4.1) when sampling each A_{ij} for the jth row from a *distinct* pseudorandom seed.

Correctness. We argue correctness for TCAKPIR in Appendix B.4, Theorem 5. For TCAKPIR$^+$, note that the random masks can be preprocessed in the offline phase because (1) they are public, and (2) the inner product is a fixed circuit in KPIR-C, regardless of the evaluation circuit $\mathcal{F}$. Therefore, correctness follows directly from the observation that the inner product on the random masks can be precomputed during the offline phase.

3.5. Reducing Communication

3.5.1. Seed Compression. An immediate implication of having public random masks is that, instead of representing the ciphertext as $\mathsf{ct} := (a_1, a_2, \ldots, a_n, b)$, one can adopt a more compact form: $\mathsf{ct} := (\theta, b)$, where θ is a random λ-bit string for security parameter λ [41]. This technique, known as seed compression, leverages the deterministic nature of cryptographically secure pseudo-random number generators (CSPRNGs) used to produce the masks. By sharing the seed, both parties can expand it to regenerate the masks and recover the original ciphertext.

3.5.2. Modulus Switching. Modulus switching is a standard technique used to reduce the size of lattice-based encodings after homomorphic operations. As shown in [45], an LWE ciphertext with parameters (n, q) can be converted to one with (n, q') for $q' < q$. In TKPIR, after the evaluation phase ($\mathsf{Evaluate}_{\mathcal{F}}$), 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 A.5), which simultaneously evaluates the circuit and refreshes the ciphertext.

3.6. KPIR-Specific Bootstrapping Strategy

Recall from Figures 2 and 3 that throughout the protocols, data entries are partitioned into ω blocks and encrypted as TLWE ciphertext blocks. To perform a sequence of operations (additions, multiplications, etc.), one must detect carry (modular) overflows in each ciphertext block and propagate them to the next block that can accommodate additional bits. In the generic use of TFHE [33], this is handled via bootstrapping. Consequently, if we rely on the generic use of TFHE to perform arbitrary computations on ciphertexts, frequent bootstrapping is required to control both noise growth and carry propagation (see Section 4.3, Table 4). Applied directly to TKPIR, such aggressive bootstrapping would quickly dominate the overall cost. However, unlike arbitrary computation, the KPIR component in KPIR-C has a highly structured form, which enables the design of a more efficient construction.

In particular, the selection vector in KPIR is one-hot: it contains exactly one active entry encrypting 1, while all other entries encrypt 0. As a result, during the inner product (Algorithm 3), the protocol aggregates a single active ciphertext—composed of ω TLWE blocks—with many inactive ciphertexts. Since the inactive terms contribute encryptions of 0, these additions do not induce cross-block carry propagation in the response ciphertext. This leads to the following design principle: *When evaluating KPIR over TFHE, bootstrapping should be invoked only to bound noise growth, as the one-hot structure of the selection vector eliminates the need for bootstrapping to manage carry propagation.*

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 guarantee correctness under worst-case conditions, including the largest supported partition size and the most adversarial accumulation of noise. This ensures the strategy remains valid across all admissible data and noise values.

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 (e.g., arbitrary computation in KPIR-C).

Property 3 (Efficiency). A KBS-based KPIR construction must be no less efficient than a generic TFHE-based construction. Efficiency is measured by the number of bootstrappings required.

In the next section, we instantiate this strategy for TKPIR; put simply, we demonstrate how the number of bootstrappings can be reduced at the KPIR level while preserving all KBS properties.

4. Instantiation

4.1. Parameters for TFHE

We instantiate the TFHE scheme using TFHE-rs (v0.10.0) [33], 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 $p := p_m \times p_c$, 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 randomness is sampled from a Gaussian distribution with standard deviation $\sigma := 2.845 \cdot 10^{-15}$, yielding a decryption failure probability of approximately 2^{-64}. We refer readers to Appendix A.6 for additional parameters.

We acknowledge that these parameters do not satisfy the statistical correctness guarantees of our constructions (i.e., failure probability $\leq 2^{-128}$); however, at the time, TFHE-rs only guaranteed IND-CPA security with a decryption failure probability of 2^{-64}, which is reasonable for our experimental setting and yields slightly better performance. Throughout our instantiation of both constructions, we adhere to this failure probability.

4.2. Parameters for TKPIR

In TCWKPIR, we use a Hamming weight of 2 for our constant-weight encoding, which ensures "sufficient" coverage for distinct keywords while keeping the query size (codeword length) sublinear in the database size, i.e., $O(\sqrt{m})$. Table 3 illustrates this tradeoff more clearly for a fixed keyword range of m. Although a Hamming weight $h > 2$ also yields sublinear communication, the resulting increase in AND evaluations to construct the selection vector becomes impractical.

TABLE 3: Query size vs. AND gates count across varying Hamming weights h in a 2^{16} database

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

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

4.3. KBS for TKPIR

TFHE-rs [33] provides multiple API levels for ciphertexts. Higher-level APIs such as `FheUint` handle bootstrapping automatically, making them suitable for supporting arbitrary computation. However, as discussed in Section 3.6, 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 the KBS-based TKPIR, ensuring that all three KBS properties (Section 3.6) are satisfied, thereby maintaining correctness and efficiency.

In Appendix B.5, Theorems 6 and 7 show that τ_{bf} can be at most 214 and 2^{20} for TCWKPIR and TCAKPIR, respectively, while maintaining a failure probability of 2^{-64} (Property 2) under worst-case settings (Property 1). However, in practice, with uniformly random data (violating Property 1), we observe that performing bootstrapping once every 2^9 entries provides empirical correctness. For instance, we run 512 instances of TCWKPIR on a 2^{16}-entry database, bootstrapping every 2^9 entries, and report the results in Fig. 4. The left plot shows the results without bootstrapping, while the right plot illustrates the error when bootstrapping is applied once after every 2^9 entries during the inner product. Finally, Table 4 reports the number of bootstrappings in the

KPIR circuit of TKPIR (specifically, in the inner product) when instantiated with FheUint and our KBS strategy, using $\tau_{\mathsf{bf}} := 214$ for TCWKPIR and 2^{20} for TCAKPIR. As shown, both KBS-based schemes significantly reduce the number of bootstrappings (Property 3), with TCAKPIR eliminating them entirely for the KPIR component.

TABLE 4: Number of bootstrappings during the inner product in TKPIR instantiated with FheUint and with KBS for a database of length 2^{16}.

	Data Bitlength				
Strategy	**8**	**16**	**32**	**64**	**128**
FheUint (both)	2555904	9109504	33030144	122748928	470024192
KBS (TCWKPIR)	1212	2424	4848	9696	19392
KBS (TCAKPIR)	0	0	0	0	0

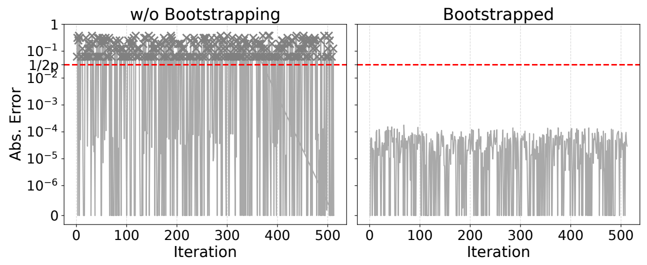


Figure 4: Absolute error from TCWKPIR over 512 iterations for $m := 2^{16}$, with bootstrapping performed every 2^9 entries. The dashed horizontal line indicates the maximum noise level required for correct decryption.

5. Evaluation

In this section, we detail the implementation and experimental evaluation of our constructions. The experiments were conducted on a server with an AMD Threadripper PRO processor, 512GB of RAM, and running Ubuntu 22.04.1 LTS. Unless otherwise specified, all evaluations were performed using a single thread and a single query, with the runtime averaged over 10 instances. The implementation is primarily written in Rust.

5.1. Keyword PIR

In this section, we benchmark our TKPIR schemes against the latest KPIR schemes. We note that TKPIR is not a KPIR but a KPIR-C. The goal of this section is to evaluate how competitive TKPIR is compared to its KPIR counterparts, while still satisfying the stricter requirements of KPIR-C (see Section 3.1).

vs. CWKPIR. In Table 5, we compare both of our constructions with CWKPIR [30]. CWKPIR uses the BFV homomorphic encryption scheme with a ciphertext modulus of 218 bits, a polynomial modulus degree of 8192, and a plaintext modulus of 21 bits, which results in a fixed

TABLE 5: Comparison with CWKPIR [30], $|\text{Entry}|$: 8 B.

DB Size	KPIR Scheme	Selection Vector (s)	Inner Product (s)	Total (s)	Comm. (KB)
2^{12}	CWKPIR [30]	48.25	4.78	53.44	319
	TCWKPIR	59.89	3.44	63.36	9
	TCAKPIR	1.55×10^{-5}	0.50	7.69	48
2^{14}	CWKPIR [30]	196.53	18.80	216.20	319
	TCWKPIR	240.57	15.13	255.83	17
	TCAKPIR	1.49×10^{-4}	2.42	10.77	164
2^{16}	CWKPIR [30]	782.96	79.58	864.21	319
	TCWKPIR	943.41	61.02	1004.51	33
	TCAKPIR	5.33×10^{-4}	6.92	22.41	612

communication cost of 319 KB, provided the available plaintext slots are not exceeded. In contrast, TCWKPIR is significantly more communication-efficient across all dimensions, as TLWE ciphertexts are inherently much more compact than BFV ciphertexts.

In addition, the computational bottleneck in both schemes lies in generating the selection vector, where CWKPIR performs marginally better across all dimensions. However, TCWKPIR achieves faster inner-product computation. Although TFHE-rs does not currently support batching (as SEAL does), we are able to achieve competitive performance for two reasons: (1) TLWE ciphertexts are significantly more compact, which makes per-ciphertext operations faster; and (2) we have optimized the number of bootstrapping operations at the KPIR level (Section 4.3).

On the other hand, TCAKPIR is expected to incur higher communication overhead than TCWKPIR, since the client sends a selection vector proportional to the database length. Moreover, while the overall communication remains manageable due to the compact size of TLWE ciphertexts, TCAKPIR still exceeds the communication of CWKPIR [30] once the total size of TLWE ciphertexts in the selection vector surpasses that of a single BFV ciphertext (which occurs for $m \gtrsim 15$).

In terms of runtime, TCAKPIR is expected to be much faster than TCWKPIR and CWKPIR, as the client encodes its selection vector in the clear using BFF and sends the encrypted filter to the server. The inner-product computation is also faster, as the freshly generated ciphertexts eliminate the need for bootstrapping (Section 4.3).

vs. Hint-based Benchmarks. In Table 6, we benchmark TCAKPIR and TCAKPIR$^+$ against the latest Hint-Based KPIR schemes [28], [29]. The reduced total communication compared to ChalametPIR [28] results from modulus switching (Section 3.5), together with the GLWE packing technique in TFHE-rs [33], which we apply to the response ciphertexts. Without these techniques, our total communication would be on par with ChalametPIR; however, our main advantage is that we satisfy the stricter KPIR-C requirements (Section 3.4) while still achieving comparable performance.

Additionally, we compare with Hao et al. [29], who propose a KPIR scheme that employs a Piecewise Linear Approximation (PLA) mapping [46] in place of Bloom filters.

This mapping enables very low online costs, with query and response sizes sublinear in the database size, while incurring relatively high preprocessing and storage costs. We emphasize that their construction satisfies KPIR, but not KPIR-C. Nevertheless, adapting their mapping approach to KPIR-C remains an interesting direction for future work.

TABLE 6: Hint-based KPIR benchmarks. |Entry|: 128 B.

Benchmark	#Entries	Runtime (s)		Comm. (KB)	
		Offline	Online	Offline	Online
ChalametPIR [20]	2^{12}	1.87	0.03	8708	48
Hao et al. [31]		1.96	0.02	9246	12
TCAKPIR		4.55	14.57	0.01	49
TCAKPIR$^+$		8.54	9.04	0.01	49
ChalametPIR [20]	2^{16}	25.36	0.4	8708	612
Hao et al. [31]		5.58	0.04	17977	29
TCAKPIR		4.89	77.53	0.01	613
TCAKPIR$^+$		75.92	9.07	0.01	613

All Together. In Table 7, we compare all benchmarks under more practical settings, including larger database sizes, varying payload lengths, and multithreading. The evaluation keys for CWKPIR [30] and TCWKPIR are 15 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 4.3). 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 significantly higher communication. In addition to communication efficiency, our schemes retain practical online latency, with TCAKPIR$^+$ performing comparably to Hao et al. [29] and typically achieving $<$ **1** s latency for databases up to 2^{20} entries.

6. Case Study: Towards Fuzzy PIR

Among the many applications of KPIR-C, we next present a case study on *Fuzzy PIR*, motivated from fuzzy matching [47]. This illustrates an application scenario in which the client submits a query (e.g., a GPS coordinate) to the server and seeks to retrieve values (e.g., nearby restaurants) associated with *matching* keywords (e.g., coordinates). The matching is based on a defined notion of closeness (e.g., L^p-distance) between the client's query and the server's keywords. After identifying the matched keywords, the server returns the values associated with the top k closest keys to the client's query. Here, we present a *relaxed* Fuzzy PIR implementation using our constructions. Optimizing Top-k selection or fuzzy matching is beyond the scope of this work and left for future research.

Fuzzy PIR over the Torus. We solve Fuzzy PIR using a fuzzy matching approach inspired by [47], where a sender

TABLE 7: 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.**

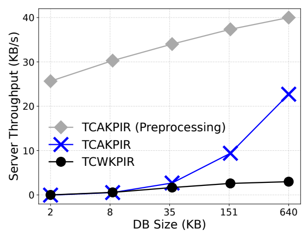
#Entries	Entry (B)	KPIR Benchmark	Upload (KB)	Download (KB)	Setup (s)	Online (s)
2^{16}	64 B	CWKPIR [30]	216	103	1.12	50.82
		Hao et al. [12]	8.29	12127.74	1.45	0.004
		ChalametPIR [28]	608	4612.5	–	–
		TCWKPIR (ours)	32.62	2.01	1.87	57.13
		TCAKPIR (ours)	608	3.63	2.10	11.81
		TCAKPIR$^+$ (ours)	608	3.63	14.14	0.22
	128 B	CWKPIR [30]	216	103	1.15	50.64
		Hao et al. [12]	11.39	18000.52	1.50	0.007
		ChalametPIR [28]	608	8712.5	–	–
		TCWKPIR (ours)	32.62	3.96	1.91	79.34
		TCAKPIR (ours)	608	5.57	2.24	13.22
		TCAKPIR$^+$ (ours)	608	5.57	15.74	0.41
	256 B	CWKPIR [30]	216	103	1.19	51.14
		Hao et al. [12]	16.25	28556.87	2.94	0.012
		ChalametPIR [28]	608	16912.5	–	–
		TCWKPIR (ours)	32.62	7.84	2	124.33
		TCAKPIR (ours)	608	9.46	2.47	17.62
		TCAKPIR$^+$ (ours)	608	9.46	19.33	0.81
2^{18}	64 B	CWKPIR [30]	216	103	4.14	200.87
		Hao et al. [12]	16.97	20842.98	2.63	0.006
		ChalametPIR [28]	2368	4612.5	–	–
		TCWKPIR (ours)	65.14	2.01	2.37	225.89
		TCAKPIR (ours)	2368	3.63	2.9	45.1
		TCAKPIR$^+$ (ours)	2368	3.63	46.93	0.26
	128 B	CWKPIR [30]	216	103	4.36	202.05
		Hao et al. [12]	23.32	30030.60	4.61	0.012
		ChalametPIR [28]	2368	8712.5	–	–
		TCWKPIR (ours)	65.14	3.96	2.7	331.30
		TCAKPIR (ours)	2368	5.57	3.48	53.71
		TCAKPIR$^+$ (ours)	2368	5.57	56.77	0.48
	256 B	CWKPIR [30]	216	103	4.53	204.95
		Hao et al. [12]	32.5	45528.30	9.12	0.021
		ChalametPIR [28]	2368	16912.5	–	–
		TCWKPIR (ours)	65.14	7.84	3.11	484.94
		TCAKPIR (ours)	2368	9.46	4.37	70.56
		TCAKPIR$^+$ (ours)	2368	9.46	76.63	1.02
2^{20}	64 B	CWKPIR [30]	216	103	OOM	OOM
		Hao et al. [12]	33.93	38371.75	9.50	0.018
		ChalametPIR [28]	9216	4612.5	–	–
		TCWKPIR (ours)	130.19	2.01	6.18	945.19
		TCAKPIR (ours)	9216	3.63	6.56	168.58
		TCAKPIR$^+$ (ours)	9216	3.63	185.79	0.45
	128 B	CWKPIR [30]	216	103	OOM	OOM
		Hao et al. [12]	46.67	54242.50	16.64	0.027
		ChalametPIR [28]	9216	8712.5	–	–
		TCWKPIR (ours)	130.19	3.96	7.02	1296.35
		TCAKPIR (ours)	9216	5.57	8.10	207.58
		TCAKPIR$^+$ (ours)	9216	5.57	219.57	0.82
	256 B	CWKPIR [30]	216	103	OOM	OOM
		Hao et al. [12]	64.98	79579.29	35.89	0.302
		ChalametPIR [28]	9216	16912.5	–	–
		TCWKPIR (ours)	130.19	7.85	8.84	1901.01
		TCAKPIR (ours)	9216	9.46	12.48	282.80
		TCAKPIR$^+$ (ours)	9216	9.46	296.81	1.72

and receiver check whether their d-dimensional hyperballs are within δ under L^p distance. We relax this to 1-dimensional hyperballs (keywords), where the receiver checks if its query lies within δ of any keyword in the sender's database. For 1-dimensional hyperballs, the problem is similar across all L^p distances and reduces to finding all keywords $k \in \mathcal{K}$ such that $|q - k| \leq \delta$. This corresponds to the client sending $c := 2\delta + 1$ queries, $q_j := q + j$ for $j \in \{-\delta, \ldots, \delta\}$, and retrieving the matched entries. Instead of returning all matches, the sender then runs an oblivious Top-k algorithm to identify the k closest keywords and their values.

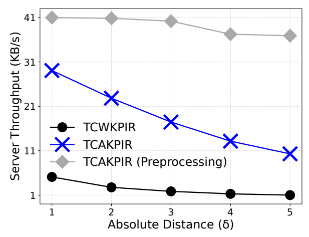
Technical Overview. For this application, we implement a naive Top-k using selection sort. With TKPIR, Fuzzy PIR incurs no extra communication or round complexity, as Top-k is offloaded to the server—enabled by TFHE's efficient PBS. After executing KPIR on c queries, the server sorts the matched pairs (k_i, v_i) for $i \in [c]$ by distance $\delta_i \in \{-\delta, \ldots, \delta\}$ from the query and returns the top k. Although δ_i is in plaintext, the server does not know which values matched, so it uses an oblivious multiplexer with a large default value to de-prioritize unmatched entries.

Fig. 5 compares the server throughput of the TKPIR constructions, including preprocessing throughput for TCAKPIR. In Fig. 5(a), we fix δ and measure server throughput as the database size varies. In Fig. 5(b), we fix the database size at 640 KB and measure throughput as δ changes. The unoptimized sorting significantly impacts performance, as seen in Fig. 5(b). However, in higher dimensions with fixed δ, this effect diminishes, since sorting is only performed on a fixed set of c pairs.

TFHE's PBS allows us to eliminate both the communication and round overhead of state-of-the-art mixed Top-k protocols [48], [49] by offloading the entire computation to the server. For example, Panther [48] requires $O(n \log^2 k)\Pi_{\text{cmp}}$ bits and up to $O(\log \ell)$ rounds, depending on the comparison protocol. In contrast, TKPIR adds no communication or round overhead for Top-k or any other post-computation. If Panther used selection sort, the cost would rise to $O(n^2)\Pi_{\text{cmp}}$ bits and still require $O(\log \ell)$ rounds.



(a) $\delta := 2$.



(b) DB size: 640 KB.

Figure 5: Fuzzy PIR over the torus. Server throughput (KB/s) for varying database sizes (Fig. 5(a)) and δ values (Fig. 5(b)).

7. Conclusion

In this work, we constructed efficient solutions for KPIR-C that enable oblivious post-processing on responses, while ensuring support for default values in cases where the queried keyword does not exist in the database. For future work, we aim to extend our constructions to support symmetric KPIR-C that imposes a stronger security requirement to ensure the privacy of both the server's data and the client's query. Additionally, we plan to explore the possibility of integrating BFV and TFHE [50] to leverage the efficiency of BFV's SIMD operations (e.g., during the selection vector generation phase) while converting responses to TFHE to benefit from its efficient programmable bootstrapping for arbitrary circuit computation. We are further interested in exploring the possibility of implementing an authenticated [51], [52] KPIR-C, which would also ensure the *integrity* of the server's response.

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*.

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

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

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

[7] A. Henzinger, E. Dauterman, H. Corrigan-Gibbs, and N. Zeldovich, "Private web search with tiptoe," in *SOSP 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," *IACR Cryptology ePrint Archive*, p. 3, 1998.

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

[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)*.

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

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

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

[17] 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*.

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

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

[20] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptology ePrint Archive*, p. 144, 2012.

[21] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Innovations in Theoretical Computer Science (TICS 2012)*, S. Goldwasser, Ed. ACM, 2012, pp. 309–325.

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

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

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

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

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

[27] 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*.

[28] 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.

[29] 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," *IACR Cryptol. ePrint Arch.*, p. 210, 2025.

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

[31] 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.

[32] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *CCS 2020*.

[33] 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.

[34] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled PSI from fully homomorphic encryption with malicious security," in *CCS 2018*.

[35] S. Patel, J. Y. Seo, and K. Yeo, "Don't be dense: Efficient keyword PIR for sparse databases," in *USENIX Security 2023*.

[36] A. Davidson, G. Pestana, and S. Celi, "Frodopir: Simple, scalable, single-server private information retrieval," *PETs 2023*.

[37] 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.

[38] 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*.

[39] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "PSI from paxos: Fast, malicious private set intersection," in *EUROCRYPT 2020*.

[40] G. Garimella, B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "Oblivious key-value stores and amplification for private set intersection," in *CRYPTO 2021*.

[41] 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.

[42] A. N. Kolmogorov, "On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition," *Doklady Akademii Nauk*, vol. 114, no. 5, pp. 953–956, 1957.

[43] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *STOC 2005*.

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

[45] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM J. Comput.*, vol. 43, no. 2, pp. 831–871, 2014.

[46] 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.

[47] A. van Baarsen and S. Pu, "Fuzzy private set intersection with large hyperballs," in *EUROCRYPT (5)*. Springer, 2024, pp. 340–369.

[48] J. Li, Z. Huang, M. Zhang, J. Liu, C. Hong, T. Wei, and W. Chen, "PANTHER: private approximate nearest neighbor search in the single server setting," *IACR Cryptol. ePrint Arch.*, p. 1774, 2024.

[49] H. Chen, I. Chillotti, Y. Dong, O. Poburinnaya, I. P. Razenshteyn, and M. S. Riazi, "SANNS: scaling up secure approximate k-nearest neighbors search," in *USENIX Security 2020*.

[50] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "CHIMERA: combining ring-lwe-based fully homomorphic encryption schemes," *Journal of Mathematical Cryptology*, vol. 14, no. 1, pp. 316–338, 2020.

[51] S. Colombo, K. Nikitin, H. Corrigan-Gibbs, D. J. Wu, and B. Ford, "Authenticated private information retrieval," in *USENIX Security 2023*.

[52] M. Dietz and S. Tessaro, "Fully malicious authenticated PIR," in *CRYPTO 2024*.

[53] N. Genise, D. Micciancio, and Y. Polyakov, "Building an efficient lattice gadget toolkit: Subgaussian sampling and more," in *EUROCRYPT*, 2019.

[54] O. Bernard, M. Joye, N. P. Smart, and M. Walter, "Drifting towards better error probabilities in fully homomorphic encryption schemes," in *EUROCRYPT (8)*. Springer, 2025, pp. 181–211.

[55] B. Li and D. Micciancio, "On the security of homomorphic encryption on approximate numbers," in *EUROCRYPT (1)*. Springer, 2021, pp. 648–677.

[56] 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.

Appendix A. More on TFHE

A.1. TGLWE Ciphertexts

TGLWE ciphertexts (General TLWE) extend TLWE to torus polynomials over $\mathbb{T}_{N,q}[X]$,[6] where $\mathsf{pp} = (N, k, \sigma, p, q)$ are public parameters: N is the (power-of-2) polynomial degree, k the GLWE dimension, σ the noise standard deviation, and $p \mid q$ are the plaintext and ciphertext moduli. Algorithms for TGLWE extend naturally from the corresponding TLWE algorithms discussed in Section 2.5, 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:

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

We refer readers to [41] for more details.

A.2. Gadget Decomposition over the Torus

Definition 1. *For any finite additive group A, an A-gadget of size w and quality β is a vector $\mathbf{g} \in A^w$ such that any group*

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

element $v \in A$ can be written as an integer combination $v = \sum_{i=1}^{w} g_i \cdot u_i$ where $\mathbf{u} := (u_1, \ldots, u_w) \in \mathbb{Z}^w$ has norm at most $||u|| \leq \beta$ [53].

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 $G := I_n \otimes \mathbf{g}^\intercal \in A^{n \times nw}$ 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 $\mathbf{g} := (1/\beta, \ldots, 1/\beta^w) \in \mathbb{T}_q^w$, such that for every $v \in \mathbb{T}_q$, there exists $\mathbf{u} \in \mathbb{T}_q^w$ satisfying $v = \sum_{i=1}^{w} g_i \cdot u_i$. To ensure this holds for all $v \in \mathbb{T}_q$, we require $\beta^w = q$ [41]. 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 representation $\mathbf{u} \in \mathbb{Z}^{n\ell}$ satisfying $\mathbf{v} \approx \mathbf{u} \times G^\intercal$, where ℓ denotes the precision with $\ell \leq w$ and $\beta^\ell \mid q$. The matrix G is referred to as the *gadget matrix* over the torus, and $\mathbf{u}$ as the *gadget decomposition* of $\mathbf{v}$. As in [41], the inverse $G^{-1} : \mathbb{T}_q^n \to \mathbb{Z}^{n\ell}$ maps $\mathbf{v}$ to a "small" decomposition $G^{-1}(\mathbf{v})$ such that $G^{-1}(\mathbf{v}) \times G^\intercal \approx \mathbf{v}$. These arguments extend naturally to torus polynomials over $\mathbb{T}_{N,q}[X]^k$.

A.3. T(G)GSW Ciphertexts

TGSW ciphertexts are the torus-based variant of Gentry–Sahai–Waters (GSW) ciphertexts [23]. Let G be the gadget matrix over $\mathbb{T}$, β the quality, and ℓ the precision, with $p := \beta^\ell$. The TGSW encryption of a message $m \in \mathbb{Z}_p$ under secret key $\mathbf{s} \in \mathbb{B}^n$ is defined as $\text{TGSW}_\mathbf{s}(m) := Z^\intercal + m \cdot G^\intercal \in \mathbb{T}_q^{(n+1)\ell \times (n+1)}$, where $Z := \left[\text{TLWE}_\mathbf{s}(0) \mid \cdots \mid \text{TLWE}_\mathbf{s}(0)\right] \in \mathbb{T}_q^{(n+1) \times (n+1)\ell}$.

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

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

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 $C := \text{TGGSW}(m)$ and $c := \text{TGLWE}(\mu)$. Then:

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

Notably, we need these ciphertexts because the gadget matrix G controls noise growth in both products, as $G^{-1}(C_2)$ and $G^{-1}(c)$ are small by definition (Appendix A.2).

A.4. Controlled Multiplexer (CMUX)

The Controlled Multiplexer (CMUX) is an application of the external product in TFHE [41]. Let $c_0 := \text{TGLWE}(\mu_0)$, $c_1 := \text{TGLWE}(\mu_1)$, and $B := \text{TGGSW}(b)$, 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 $\text{TGLWE}(\mu_0 - \mu_1)$. Specifically, we have: $\mathsf{CMUX}(B, c_0, c_1) := B \boxdot (c_1 - c_0) + c_0$.

A.5. Bootstrapping and Programmable Bootstrapping (PBS)

In TFHE, *bootstrapping* refreshes a ciphertext by homomorphically evaluating its decryption circuit. Let $\mathbf{c} := \text{TLWE}_\mathbf{s}(\mu)$ 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 $b := \sum_{i=1}^{n} s_i \cdot a_i + e$ 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: $v := \sum_{j=0}^{N-1} f\left(\frac{\lfloor pj/(2N) \rceil \bmod p}{p}\right) \cdot X^j$. 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.

Algorithm 4 Bootstrap [41]

Params: LWE dimension n, polynomial degree N, ciphertext modulus q, plaintext modulus p, quality β, precision ℓ
Input: TLWE ciphertext $\mathbf{c} := (a_1, \ldots, a_n, b) \in \mathbb{T}_q^{n+1}$ of μ under secret key $\mathbf{s} := (s_1, \ldots, s_n) \in \mathbb{B}^n$; bootstrapping keys (i.e., evaluation keys) $\mathsf{bsk}[j] := \text{TGGSW}_{\mathbf{s}'}(s_j)$ for $j \in [n]$, where $\mathbf{s}' := (\mathbf{s}'_1, \ldots, \mathbf{s}'_k) \in \mathbb{B}_N[X]^k$ is the TGGSW secret key

```
1: [Modulus Switching] Scale $\mathbf{c}$ by the rounding factor $2N$ to obtain $\tilde{\mathbf{c}} := (\tilde{a}_1, \ldots, \tilde{a}_n, \tilde{b}) \in \mathbb{T}_{2N}^{n+1}$, which encrypts $\tilde{\mu} \in \mathbb{T}_p$
2: [GLWE Encoding] Construct test polynomial $v := \sum_{j=0}^{N-1} \left(\left\lfloor \frac{pj}{2N} \right\rceil \bmod p\right)/p \cdot X^j$ and set $\text{GLWE}(v) := (0, \ldots, 0, v) \in \mathbb{T}_{N,q}[X]^{n+1}$
3: [Blind Rotation] Initialize $\mathbf{c}'_0 := \mathsf{PolyMul}(X^{-\tilde{b}}, \mathsf{GLWE}(v))$
4: for $j = 1$ to $n$ do
5:     $\mathbf{c}'_j := \mathsf{CMUX}(\mathsf{bsk}[j], \mathbf{c}'_{j-1}, \mathsf{PolyMul}(X^{\tilde{a}_j}, \mathbf{c}'_{j-1}))$
6: [Sample Extraction] We have $\mathbf{c}'_n := \text{GLWE}_{\mathbf{s}'}(\mu + \cdots)$. Set $\mathbf{c}' := \mathbf{c}'_n$, and write $\mathbf{c}' := (a'_1, \ldots, a'_k, b')$ with $b' := \sum_{i=0}^{N-1} b'_i X^i$; extract $\bar{b} := b'_0$. Let $\mathbf{a}' := (a'_1, \ldots, a'_k)$. Flatten[7] $\mathbf{s}'$ and $\mathbf{a}'$ into $\bar{\mathbf{s}} := (\bar{s}_1, \ldots, \bar{s}_{kN}) \in \mathbb{B}^{kN}$ and $\bar{\mathbf{a}} := (\bar{a}_1, \ldots, \bar{a}_{kN}) \in \mathbb{T}_q^{kN}$, respectively. Set $\bar{\mathbf{c}} := (\bar{\mathbf{a}}, \bar{b})$
7: [Key Switching] Let $\mathbf{g} := (1/\beta, \ldots, 1/\beta^\ell) \in \mathbb{T}_q^\ell$ denote the $\mathbb{T}_q$-gadget. For each $\bar{a}_i$, compute its decomposition $\bar{\bar{a}}_i := \mathbf{g}^{-1}(\bar{a}_i) \in \mathbb{Z}^\ell$. Define $\mathsf{ksk}[i,j] := \text{TLWE}_\mathbf{s}(\bar{s}_i \cdot g_j)$ for $1 \leq i \leq kN$, $1 \leq j \leq \ell$. Let $\text{TLWE}(\bar{b}) := (0, \ldots, 0, \bar{b})$. Then compute: $\mathbf{c}'' := \text{TLWE}(\bar{b}) - \sum_{i=1}^{kN} \sum_{j=1}^{\ell} (\bar{\bar{a}}_i)_j \cdot \mathsf{ksk}[i,j]$
```

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

A.6. Instantiation from TFHE-rs [33]

In Section 4.1, we briefly discuss the parameter set used to instantiate TFHE from TFHE-rs [33], which suffices to

7. 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 $\langle \mathbf{a}', \mathbf{s}' \rangle \bmod (X^N + 1)$.

understand the main constructions. However, to fully follow the proofs in Appendix B 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 4.1), the encryption key is parameterized as a *Big* encryption key, where the LWE dimension is defined as $n_{\text{Big}} := N \cdot k$. Here, $N := 2048$ and $k := 1$ 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 $n := n_{\text{Big}} = 2048$ 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 input 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 enables ciphertexts to remain in the large LWE dimension without compromising bootstrapping performance. To reproduce our correctness results, the remaining parameters can be found under the parameter set `PARAM_MESSAGE_2_CARRY_2_KS_PBS` in TFHE-rs version 0.10.0 [33].

Appendix B. Security and Correctness

B.1. Security for KPIR-C

For a KPIR-C scheme (and similarly for KPIR-D) to be secure, it must satisfy the standard KPIR security requirements (Section 2.2) and ensure that the server learns neither if the evaluated ciphertext corresponds to the default value δ, nor any additional information about the query through arbitrary circuit evaluations. To capture this, we define the query indistinguishability game (Definition 3) and show in Theorem 1 that it suffices to meet the full security requirements of a KPIR-C scheme.

Definition 3 (Query Indistinguishability $\mathsf{IND\text{-}Q}^{\mathcal{A}}_{\text{KPIR-C}}$)**.** *We say that a KPIR-C scheme (see Section 3.1) is query indistinguishable under the security parameter λ if, for any PPT adversary $\mathcal{A}$ participating in the query indistinguishability game $\mathsf{IND\text{-}Q}^{\mathcal{A}}_{\mathit{KPIR\text{-}C}}$, the adversary wins the game with at most negligible advantage; that is:* $\left|\Pr[\mathsf{IND\text{-}Q}^{\mathcal{A}}_{\mathit{KPIR\text{-}C}}(\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$.

Theorem 1. *Any KPIR-C scheme that satisfies query indistinguishability is a secure KPIR-C scheme.*

Proof Sketch. 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 $\mathcal{A}$ also selects a circuit $\mathcal{F}$ and invokes $r \leftarrow \mathsf{Evaluate}_{\mathcal{F}}(\mathsf{pp}, \mathsf{evk}, q_b)$. It outputs 1 if it learns r corresponds to q_1, and 0 otherwise.

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

1. Adversary $\mathcal{A}$ sends $(\mathsf{pp}, \mathsf{kw}_0, \mathsf{kw}_1)$ to challenger $\mathcal{C}$:
- Initializes $\mathsf{DB} \in (\mathcal{K} \times \mathcal{V})^*$
- $\mathsf{pp} \leftarrow \mathsf{Setup}(\mathsf{DB})$
- Chooses $\mathsf{kw}_0, \mathsf{kw}_1 \in \mathcal{K}$

2. Challenger $\mathcal{C}$ responds with (evk, q_b):
- $(\mathsf{qk}, \mathsf{evk}) \leftarrow \mathsf{ClientSetup}(\mathsf{pp})$
- $b \leftarrow\$ \{0, 1\}$
- $q_b \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_b)$

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

Figure 6: Query Indistinguishability game for KPIR-C

Game 2: Same as Game 1, except the database DB is fixed in advance. The adversary chooses keywords $\mathsf{kw}_0, \mathsf{kw}_1 \in \mathcal{K}$ such that $(\mathsf{kw}_0, v) \in \mathsf{DB}$ for some $v \in \mathcal{V}$, and no such pair exists for kw_1. It then selects a circuit $\mathcal{F}$, computes: $r \leftarrow \mathsf{Evaluate}_{\mathcal{F}}(\mathsf{pp}, \mathsf{evk}, q_b)$, and outputs 1 if it learns that r corresponds to the default value, and 0 otherwise.

Note that Game 1 is stricter than Game 0, as the adversary is limited to guessing b based on its evaluation of a circuit $\mathcal{F}$ (capturing post-computation security). Game 2 is stricter than Game 1, as the adversary now enforces a mismatch and must guess whether the response indicates such a mismatch, which is then used to infer b (capturing indistinguishability in the presence of mismatches). Therefore, if every PPT adversary has negligible advantage in Game 0, then no adversary can gain a non-negligible advantage in Game 1 or Game 2; otherwise, one could construct an adversary that breaks query indistinguishability. □

B.2. IND-CPAD Security for TFHE

We assume that TFHE satisfies IND-CPAD security (Indistinguishability under Chosen Plaintext Attack with decryption correctness), a stronger notion than the IND-CPA security (Indistinguishability under Chosen Plaintext Attack) [54], which has recently been established for TFHE-rs [33]. We use $\mathsf{IND\text{-}CPAD}^{\mathcal{A}}_{\text{TFHE}}$ (Figure 7) to denote the IND-CPAD security game for TFHE. It is known that any encryption scheme that is both statistically correct (i.e., with failure probability $\leq \mathsf{negl}(\lambda)$; $\lambda := 128$) and IND-CPA secure is also IND-CPAD secure [55]. Informally, the goal of this game is to construct a PPT adversary that either breaks the IND-CPA security of TFHE or causes the decryption algorithm to fail with probability greater than $\mathsf{negl}(\lambda)$ [54].

To fully capture the security of TKPIR, it is necessary to ensure that no adversary—including failure-aware adversaries capable of breaking the IND-CPA security of TFHE can gain extra advantage in the query indistinguishability game for KPIR-C (Appendix B.1). Since TKPIR relies on the security of TFHE, the existence of such an adversary would imply one that can win the KPIR-C query indistinguishability game. To this end, in Appendix B.3, we use IND-CPAD security for TFHE to establish the security of TKPIR. Then, in Appendix B.4, we show that TKPIR must also be statistically correct (i.e., with failure probability $\leq 2^{-128}$). We note that the goal is to show that TKPIR can theoretically achieve both security and statistical correctness; however, in our implementation, we adhere to a failure probability of $\leq 2^{-64}$.

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

1. Generate keys $(\mathsf{sk}, \mathsf{evk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$**; let** pp **denote the public parameters.**

2. $\mathcal{A}$ **is given access to three oracles with a shared state** $S := \emptyset$**:**

1) $c \leftarrow \mathcal{O}_{\mathsf{Enc}}(m_0, m_1)$:
   - $c \leftarrow \mathsf{Encrypt}_{\mathsf{sk}}(m_b)$
   - Updates S as: $S \leftarrow S \cup \{(m_0, m_1, c)\}$
2) $c \leftarrow \mathcal{O}_{\mathsf{Eval}}(\mathcal{F}, j_1, \ldots, j_k)$:
   - $c \leftarrow \mathsf{Evaluate}_{\mathsf{evk}}(\mathcal{F}, S[j_1].c, \ldots, S[j_k].c)$
   - Updates S as:
   $$S \leftarrow S \cup \big\{ \big(\mathcal{F}(S[j_1].m_0, \ldots, S[j_k].m_0), \mathcal{F}(S[j_1].m_1, \ldots, S[j_k].m_1),\ c\big) \big\}.$$
3) $m \leftarrow \mathcal{O}_{\mathsf{Dec}}(j)$:
   - If $S[j].m_0 = S[j].m_1$, then $m \leftarrow \mathsf{Decrypt}_{\mathsf{sk}}(S[j].c)$.

3. Adversary $\mathcal{A}$ **interacts with the oracles and eventually outputs a guess** $\bar{b} \in \{0,1\}$**, and wins if** $\bar{b} = b$**.**

Figure 7: IND-CPAD game for FHE

B.3. Security for TKPIR

We prove the security of the TKPIR constructions via a reduction to the IND-CPAD security of TFHE (Appendix B.2), captured by the game in Figure 7. We note that a reduction to the IND-CPA security of TFHE is sufficient to establish security for our constructions. In fact, in our proofs, the adversary in the IND-CPAD game only interacts with the encryption oracle, which is identical to the IND-CPA game. However, we adopt the stronger IND-CPAD notion because it additionally enforces statistical correctness for TFHE, which we rely on in Appendix B.4. If we can tolerate a higher failure probability (e.g., $\leq 2^{-64}$; see Appendix B.5), then a reduction to the IND-CPA game for TFHE suffices. The reduction to the classic IND-CPA game follows similarly to Theorems 2 and 3, with IND-CPAD replaced by IND-CPA.

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

Proof. Suppose, for contradiction, that TCWKPIR is not secure. Then, there exists a PPT adversary $\mathcal{B}$ that can win the query indistinguishability game $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\text{TCWKPIR}}$ 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 $\mathsf{IND\text{-}CPAD}^{\mathcal{A}}_{\text{TFHE}}$.

Let $\mathcal{C}$ be the challenger in the $\mathsf{IND\text{-}CPAD}^{\mathcal{A}}_{\text{TFHE}}$ 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 $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\text{TCWKPIR}}$ game. To simulate the query $\mathbf{q}_b \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{qk}, \mathsf{kw}_b)$ in the $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\text{TCWKPIR}}$ game, $\mathcal{A}$ runs:

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

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 $\mathcal{C}$ runs:

$$\mathsf{ct}_{0i} \leftarrow \mathsf{Encrypt}_{\mathsf{sk}}(\mathsf{Encode}(\mathsf{cw}_{0i})), \text{ and}$$
$$\mathsf{ct}_{1i} \leftarrow \mathsf{Encrypt}_{\mathsf{sk}}(\mathsf{Encode}(\mathsf{cw}_{1i})), \text{ for } i \in [\rho].$$

It then flips a coin to choose $b \leftarrow\$ \{0,1\}$ and forwards ct_b to the adversary $\mathcal{A}$. The adversary $\mathcal{A}$ then sets $\overline{\mathbf{q}_b} := \mathsf{ct}_b$ and sends $\overline{\mathbf{q}_b}$ to $\mathcal{B}$ in the $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\text{TCWKPIR}}$ 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 $\mathsf{IND\text{-}Q}^{\mathcal{B}}_{\text{TCWKPIR}}$ 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 $\mathsf{IND\text{-}CPAD}^{\mathcal{A}}_{\text{TFHE}}$ game. It is clear that if $\mathcal{B}$ can guess b with non-negligible advantage, then so can $\mathcal{A}$, thereby breaking the IND-CPAD security of TFHE. $\square$

Theorem 3. *TCAKPIR construction (Figure 3) is secure under Definition 3, assuming that TFHE is IND-CPAD secure (Appendix B.2).*

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 $(\overline{d_0}, \overline{\mathsf{sv}_0})$ and $(\overline{d_1}, \overline{\mathsf{sv}_1})$ for both keywords as per Figure 3 and submits the pair $(\mathbf{m_0}, \mathbf{m_1})$ to the challenger, where $\mathbf{m_0} := (\overline{d_0}, \overline{\mathsf{sv}_0}) \in \mathbb{Z}_p^{\omega+\omega'}$, and $\mathbf{m_1} := (\overline{d_1}, \overline{\mathsf{sv}_1}) \in \mathbb{Z}_p^{\omega+\omega'}$. The challenger $\mathcal{C}$, playing in the $\mathsf{IND\text{-}CPAD}^{\mathcal{A}}_{\text{TFHE}}$ game, responds with:

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

The adversary $\mathcal{A}$ then sets $q_b := \mathsf{ct}_b$ 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 $\mathcal{A}$. $\square$

B.4. Statistical Correctness for TKPIR

We say that our constructions are statistically correct if their failure probability is $\mathsf{negl}(\lambda)$, for $\lambda := 128$. In the following theorem, we provide a proof outline showing how the TKPIR construction can theoretically satisfy statistical correctness.

Theorem 4. *TCWKPIR is a correct KPIR-C scheme (Section 3.1; $\lambda := 128$), provided that TFHE is IND-CPAD secure (Appendix B.2).*

Proof Sketch. To prove correctness for TCWKPIR, observe that it depends solely on the client's ability to decrypt correctly, as the client has white-box access to the entire protocol. Thus, the only possible source of error is incorrect decryption by TFHE. We can therefore reduce the correctness of TCWKPIR to that of TFHE. Specifically, we construct an adversary $\mathcal{A}$ that plays in the $\mathsf{IND\text{-}CPAD}^{\mathcal{A}}_{\mathsf{TFHE}}$ game (Figure 7), selecting an arbitrary circuit $\mathcal{F}$ and querying the decryption oracle to attempt to break TFHE's statistical correctness. Assuming TFHE is IND-CPAD secure and thus statistically correct, no such adversary $\mathcal{A}$ can exist. $\square$

Theorem 5. *TCAKPIR is a correct KPIR-C scheme (Section 3.1; $\lambda := 128$) if TFHE is IND-CPAD secure (Appendix B.2) and the random oracle hash function* H *(Figure 3) has a false positive probability of at most* $\mathsf{negl}(\lambda)$.

Proof Sketch. The correctness of TCAKPIR follows from an argument similar to that of TCWKPIR, with the key difference that TCAKPIR additionally depends on the false positive probability (FPP) of the hash function H, which maps keywords to digests (see Figure 3). The FPP of H is at most $2^{-\mu}$, where μ is the digest length. As long as $\mu \geq \lambda$, this error is negligible and does not affect the statistical correctness of TFHE.

Note that the false positive rate of BFF in TCAKPIR does not affect correctness. On a false positive, the server returns the default value δ (using the client's encrypted digest shared at query time; see Figure 3), which the client uses to verify correctness. $\square$

B.5. Failure Probability of $\leq 2^{-64}$ for TKPIR

Theorem 6. *Let d be the maximum partition entry. TCWKPIR guarantees a failure probability of $\leq 2^{-64}$ by bootstrapping once every $m := 214$ entries in the inner product.*

Proof. In TFHE-rs [33] under the chosen parameter set (Section 4.1), bootstrapping consists of key switching, modulus switching, and blind rotation. To ensure correctness, the total noise after modulus switching must remain below the threshold Δ.

We define a tighter threshold as $\Delta' := \sqrt{\Delta^2 - c^2 v_{\text{ks}} - mc^2 v_{\text{bs}} - v_{\text{ms}}}$, where v_{bs} is the variance of the error from bootstrapping (Theorem 4 in [56]), v_{ks} is from key switching (Theorem 4.1 in [26]), and v_{ms} is from modulus switching (Remark 4.2 in [54]). The constant $c := 2N/q$ (or $2N$ in torus representation) accounts for scaling during modulus switching.

With parameters (Section 4.1), we get $\Delta' \approx \sqrt{2.08 - 2.07m} \cdot 10^{-3}$. To ensure failure probability $F \leq 2^{-64}$, we model the accumulated error as $S := \sum_{i=1}^{m} cde_i \sim \mathcal{N}(0, mc^2d^2\sigma^2)$, and apply the tail bound: $Q\left(\frac{\Delta'}{\sqrt{m}dc\sigma}\right) \leq 2^{-65}$, implying $\frac{\Delta'}{\sqrt{m}dc\sigma} \geq 9.42$, or equivalently, $m \leq \left(\frac{\Delta'}{9.42dc\sigma}\right)^2$. Numerical evaluation yields $m \leq 214$. $\square$

Theorem 7. *Let d be the maximum partition entry. TCAKPIR achieves a failure probability of $\leq 2^{-64}$ for an inner product of depth $m := 2^{20}$, without bootstrapping.*

Proof. Let $E := \sum_{i=1}^{m} d \cdot |e_i|$, where $e_i \sim \mathcal{N}(0, \sigma^2)$ and $d := 3$. Using the union bound, bounding each $|e_i| \leq 9.16 \cdot \sigma$ ensures $E \leq 9.16 \cdot 3 \cdot \sigma \cdot m$ with probability at least $1 - m \cdot Q(9.16) \geq 1 - 2^{-64}$. To preserve correctness, we require $E \leq \Delta$, yielding $m \leq \Delta/(9.16 \cdot 3 \cdot \sigma)$. Plugging in values from Section 4.1 gives $m \leq 2^{38} \gg 2^{20}$. $\square$