An Efficient Private Information Retrieval Protocol Based on TFHE

Haibo Tian[1,2(✉)] and Yini Lin[1,2]

[1] School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China
linyn29@mail2.sysu.edu.cn

[2] Guangdong Province Key Laboratory of Information Security Technology, Guangzhou 510006, China
tianhb@mail.sysu.edu.cn

Abstract. Private Information Retrieval (PIR) allows a user to query an entry from a database without revealing the index of the entry to the database owner. It is a building tool for many privacy enhancement applications, such as compromised credential checking and stock value query. However, the efficiency of PIR protocols has always been a bottleneck in their practical deployment. To address the challenge, we propose leveraging the fully homomorphic encryption over the torus (TFHE) scheme, which supports implementation on 32-bit unsigned integers and features a fast controlled selector (CMux) gate. The CMux gate can naturally construct a table lookup algorithm to serve PIR. We introduce compression and expansion algorithms for a PIR query, resulting in an encrypted query only a few hundred bytes in size. Further, we present a parallel PIR protocol and four simultaneous table lookup (STLU) algorithms based on TFHE, which are implemented on graphics processing units (GPUs) and make a distributed PIR service with a fast response time. On four servers with five GPUs, we obtain a PIR service with a response time of less than 2 seconds and a query size of less than 200 bytes on a database containing 2^{22} entries, with each entry encoding a message of at least 256 bytes.

Keywords: private information retrieval · fully homomorphic encryption over the torus · simultaneous table lookup · graphics processing units

1 Introduction

In many situations, individuals or organizations may want to access data from a database without revealing their identity or the specific data they are interested in. *Private Information Retrieval* (PIR) allows a user to query a database or storage system without revealing any information about the data being accessed. It is an important area in privacy-preserving research and has a variety of applications. Examples include anonymous communication [3], stock value query [4,7],

H. Duan et al. (Eds.): SecureComm 2023, LNICST 567, pp. 476–495, 2025.
https://doi.org/10.1007/978-3-031-64948-6_24

and *compromised credential checking* (C3) services provided by "HaveIBeenPwned" and "Google Password Checkup" [13].

A trivial solution to PIR is to download the entire database, but this can be costly when the database is large. In 1995, Chor et al. [7] introduced the first PIR protocol. This protocol, known as CGKS, was based on the idea of random sampling to hide the identity of the desired index and used some replicated copies of the database to reduce communication cost. Since then, PIR has undergone significant development and inspired two main branches: information-theoretic PIR (IT-PIR) and computational PIR (cPIR).

IT-PIR schemes offer unconditional security guarantees and typically involve multiple non-colluding database replicas. RAID-PIR [8] improves over CGKS [7] on the number of replicas and the communication cost. CIP-PIR [11] further improves over RAID-PIR [8] by moving part of the user's query generation to the server side, reducing the online computation required. While IT-PIR schemes are computationally efficient due to cheap bit operations, their key challenge is the impracticality of finding non-colluding servers in real-world scenarios.

On the other hand, cPIR relies on cryptographic hardness assumptions, and the user's privacy is guaranteed if an adversary could be modeled as a probabilistic polynomial time (PPT) algorithm. The computation time of cPIR is usually linear in the database size. To improve the efficiency, researchers have employed recursion, single instruction multiple data (SIMD), and modulus switching techniques to reduce the server response time and bandwidth [2,3]. In 2000, Beimel et al. [4] proposed a preprocessing step to break the linear computation limit. More recently, in 2021, Shi et al. [18] proposed a 2-server PIR with preprocessing that achieves $O(\sqrt{|\boldsymbol{db}|})$ online computation per query where $|\boldsymbol{db}|$ is the number of entries in the target database.

Park et al. [16] proposed SHECS-PIR in the class of cPIR. Their approach shows the possibility of utilizing the fully homomorphic encryption over the torus (TFHE) scheme in PIR. There are three types of ciphertexts in TFHE: TLWE, TRLWE and TRGSW. A table lookup algorithm is constructed based on controlled selector (CMux) gates. The CMux gate takes as inputs a TRGSW ciphertext and two TRLWE ciphertexts, one of which is served as an output controlled by the encrypted bit in the TRGSW ciphertext. If we encrypt the query index in PIR bit-by-bit as TRGSW ciphertexts and encode a database as trivial TRLWE ciphertexts, we obtain a PIR scheme. Park et al. [16] also presented packing and unpacking algorithms to compress a PIR query, where a query consisting of ℓ TRLWE ciphertexts are converted into $log_2|\boldsymbol{db}|$ TRGSW ciphertexts by a private functional key switching algorithm [6]. The motivation to use TFHE in SHECS-PIR is to reduce the communication cost from $O(d\sqrt[d]{|\boldsymbol{db}|})$ to $O(log_2|\boldsymbol{db}|)$, where d is a parameter typically chosen as 2 or 3 [16].

There left some room for improvement on the work of Park et al. [16]. Firstly, we can optimize their packing algorithm by replacing the TRLWE ciphertexts with TLWE ciphertexts using a random seed. This modification reduces the query size from approximately $8 \times 2N \times \ell$ bytes to around $8 \times \ell \times \lceil \log_2 |\boldsymbol{db}| \rceil$ bytes, where N and ℓ are both TFHE parameters. Secondly, their implementation is

based on 64-bit integers, resulting in their query and response sizes that are approximately twice as large as implementation on 32-bit integers. Thirdly, it is important to note that the table lookup algorithm described in [6,16] is designed for a single server operating in a single thread. Consequently, the database size is constrained by the computational capacity of a server if we desire a reasonable response time for PIR queries.

This paper shows a full-fledged efficient PIR protocol on distributed servers with optimal query and response size based on TFHE. We give new compression and expansion algorithms based on TFHE, suitable parameter sets for 32-bit implementation, and serial simultaneous table lookup algorithms for GPUs in multiple servers. Note that the implementation of TFHE on 32-bit unsigned integers makes it easily runnable on GPUs, compared to other fully homomorphic encryption (FHE) schemes that operate on integers of several hundred bits. Our experiments show that on four servers with five GPUs, the server response time could be reduced to approximately 2 s for a database with 2^{22} entries, each at least 256 bytes in size.

1.1 Related Works

We provide a brief overview of existing works in the class of cPIR that have a linear computation cost.

XPIR [1] was one of the first practical proposals to query a single database using FHE schemes [5,9]. SealPIR [3] improved upon XPIR by introducing a query compression and expansion algorithm to reduce the query size. MulPIR [2] built upon SealPIR to further reduce communication cost. It introduces techniques such as using a random seed to shorten queries, modulus switching to shorten responses, and an improved expansion algorithm to pack multiple queries into a single ciphertext. SHECS-PIR [16] introduces TFHE [6] to cPIR. They demonstrated a query packing technique, where queries are packed into TRLWE ciphertexts and then unpacked using a private functional key-switching algorithm. OnionPIR [15] combines FHE schemes in [6,9,10] to control noise growth and response size. SPIRAL [14] combines FHE schemes in [10,17] to achieve a better trade-off between computation and communication. Based on the performance data from these works, we give Table 1 to show their typical performance, where "RTT" denotes round trip time. Note that XPIR [1] is excluded since it is for a database where the number of entries is small while the size of an entry is large.

Our work improves the current record on both communication cost and response time. Specifically, we achieve a substantial reduction in the PIR query size, which has been reduced from the level of "KB" to the level of bytes. The PIR response size is about 8 KB. Our RTT is about 2 seconds on four servers for a database with 2^{22} entries, each at least 256 bytes in size. It is important to note that the response time can be further improved by increasing the number of servers dedicated to providing the PIR service.

Table 1. Performance comparison of state-of-the-art PIR schemes with experimental data from the original papers, for a database with 2^{20} entries.

Schemes	Entry size (B)	RTT (seconds)	Query size (B)	Response size (KB)
SealPIR [3]	288	2.24	64K	256
MulPIR [2]	288	6	122K	119
SHECS-PIR [16]	1.75K	42	32K	32
OnionPIR [15]	30K	400.9	64K	128
SPIRAL [14]	256	1.69	14K	21
This work	256	0.5	164	8

1.2 Contributions

The main contribution of this paper is a PIR scheme with minimal communication cost and a fast response time. It includes four aspects.

A Parallel PIR Protocol . We propose a parallel private information retrieval (PPIR) protocol, which utilizes multiple Database Servers (DBSs) and a Map Server (MS) to process queries in parallel. The key observation here is that a database could be divided into several sub-databases, and the sub-database holders could collaboratively compute a response for a user query.

Simultaneous Table Lookup (STLU) Algorithms . We propose four STLU algorithms based on the table lookup algorithm in [6]. These algorithms are used to deal with the problems of whether the GPU can preload the whole database and whether the database size is a power of two. The main observation of STLU is that the CMux gate in TFHE could be computed on pairs of entries in a database simultaneously.

Query Compression and Expansion Algorithms. We propose using TLWE as the user query and then convert it into TRGSW at the server side. The TLWE ciphertexts are generated by a pseudorandom generator with a random seed, so the user query is compressed as a seed and some 32-bit integers. On the server side, we expand user queries by a private functional key-switching algorithm on 32-bit integers directly, resulting in a fast expansion.

Implementation . The STLU algorithms are implemented by CUDA C++ and carefully evaluated. They are integrated into the PIR protocol implemented by C++ to build a PIR service.

2 Preliminaries

We give our notations in Sect. 2.1 and review relevant algorithms of TFHE in Sect. 2.2.

Table 2. TFHE Parameter Symbols and Descriptions

Symbols	Descriptions
n	The dimension of the TLWE key
η	The standard deviation of Gaussian error for TLWE encryption
α	The standard deviation of Gaussian error for TRLWE/TRGSW encryption
N	A power of 2 and $N-1$ is the degree of polynomial
ϵ	The precision of gadget decomposition
β	The quality of gadget decomposition
k	The dimension of TRLWE/TRGSW keys and we set $k=1$ in this paper
B_g	A positive integer and is the base of gadget decomposition
ℓ	A positive integer and is the depth of gadget decomposition
$\mathbf{H}$	A gadget matrix defined by $\mathbf{I}_{k+1} \otimes \mathbf{g}$, where $\mathbf{g}^{\mathsf{T}} = (1/B_g, 1/B_g^2, \ldots, 1/B_g^{\ell})$
t	A positive integer and is the precision of binary decomposition
γ	The standard deviation of the key-switching key

2.1 Notations

We denote matrices in uppercase bold and vectors in bold italics. Let $\mathbf{A}_{i,j}$ represent the i-th row and j-th column element of a matrix $\mathbf{A}$. Let a_i be the i-th element of the vector $\boldsymbol{a}$ and $\boldsymbol{a}_{n_1:n_2}$ a vector of elements $(a_{n_1}, \ldots, a_{n_2-1})$. The dot product of two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ will be denoted as $\boldsymbol{v} \cdot \boldsymbol{w}$. We denote as E^m and $E^{p \times q}$ the vectors of dimension m and the matrices of dimension $p \times q$ with elements in a set E, respectively. Let $\mathbf{I}_k$ be an identity matrix of size $k \times k$.

Let λ be a security parameter. We denote the set of binary bits $\{0,1\}$ by $\mathbb{B}$ and a discrete torus over L-bit unsigned integers by $\mathbb{T}$. Let $\mathbb{Z}_N[X]$ be the ring of polynomials $\mathbb{Z}[X]/(X^N+1)$, $\mathbb{B}_N[X]$ the polynomials in $\mathbb{Z}_N[X]$ with binary coefficients, and $\mathbb{T}_N[X]$ the ring of polynomials in $\mathbb{T}[X]/(X^N+1)$.

Let $\widetilde{a} \xleftarrow{U} \mathcal{S}$ be the process of sampling $\widetilde{a}$ uniformly at random over $\mathcal{S}$. Let $\mathbb{T}^{\ell} \leftarrow \mathsf{PRG}(s, \ell)$ be a pseudorandom generator that takes as input a seed value s and a length parameter ℓ and produces a vector of dimension ℓ in $\mathbb{T}$.

Table 2 shows the TFHE parameter symbols used in this paper. The symbols n, α, N, k, B_g, ℓ, $\mathbf{H}$, ϵ are used for level 1 parameters of TRLWE/TRGSW encryptions, and the same symbols with under bars/upper bars for level 0/2. Let $\underline{\mathfrak{K}} \in \mathbb{B}_{\underline{n}}, \mathfrak{K} \in \mathbb{B}^n$ and $\bar{\mathfrak{K}} \in \mathbb{B}^{\bar{n}}$ be level 0, 1 and 2 TLWE secret keys, and $\underline{K}, K, \bar{K}$ their respective TRLWE interpretation.

2.2 TFHE

Our PPIR protocol is based on the TFHE scheme of Chillotti et al. [6]. We introduce seven algorithm interfaces with some definitions and lemmas in [6].

TFHE Algorithm Interfaces. The seven algorithm interfaces are as follows.

– $(sk, pk) \leftarrow \mathsf{KeyGen}(\lambda)$. It takes as input a security parameter λ and outputs a secret key sk and a public key pk. The secret key includes $(\underline{\mathfrak{K}}, K, \bar{K})$ for

TLWE, TRLWE and TRGSW encryptions. The public key includes a private key-switching key $\mathsf{KS}_{\tilde{\mathfrak{K}} \to K}$.

- $\mathfrak{c} \leftarrow \mathsf{TLWE}_{\underline{\mathfrak{K}},\eta}(\mu)$. It takes as input a message $\mu \in \mathbb{T}$, and produces a TLWE ciphertext $\mathfrak{c}$ under an encryption key $\underline{\mathfrak{K}} \in \mathbb{B}^{\underline{n}}$ and a standard deviation of Gaussian error η.
- $\boldsymbol{c} \leftarrow \mathsf{TRLWE}_{K,\alpha}(\mu)$. It takes as input a message $\mu \in \mathbb{T}_N[X]$ and produces a TRLWE ciphertext $\boldsymbol{c}$ under an encryption key $K \in \mathbb{B}_N[X]$ and a standard deviation of Gaussian error α.
- $\mathbf{C} \leftarrow \mathsf{TRGSW}_{K,\alpha,\mathbf{H}}(\mu)$. It takes as input a message $\mu \in \mathbb{Z}_N[X]$ and produces a TRGSW ciphertext $\mathbf{C}$ under an encryption key $K \in \mathbb{B}_N[X]$, a standard deviation of Gaussian error α and a canonical gadget matrix $\mathbf{H}$.
- $\boldsymbol{c} \leftarrow \mathsf{ExternalProduct}(\mathbf{A}, \boldsymbol{b})$. It takes as inputs a TRGSW ciphertext $\mathbf{A} \leftarrow \mathsf{TRGSW}_{K,\alpha,\mathbf{H}}(\mu_{\mathbf{A}})$ and a TRLWE ciphertext $\boldsymbol{b} \leftarrow \mathsf{TRLWE}_{K,\alpha}(\mu_{\boldsymbol{b}})$, and outputs a TRLWE ciphertext $\boldsymbol{c}$ of message $\mu_{\mathbf{A}} \cdot \mu_{\boldsymbol{b}}$ under the same key K. The operation is denoted as $\boxdot$, and $\boldsymbol{c} = \mathbf{A} \boxdot \boldsymbol{b} = \mathsf{Decomp}(\boldsymbol{b}) \cdot \mathbf{A}$, where Decomp is a decomposition algorithm (Algorithm 1 in [6]).
- $\boldsymbol{d} \leftarrow \mathsf{CMux}(\mathbf{C}, \boldsymbol{d_1}, \boldsymbol{d_0})$. It takes as inputs two TRLWE ciphertexts $\boldsymbol{d_0}$ and $\boldsymbol{d_1}$, and a TRGSW ciphertext $\mathbf{C}$, and outputs a TRLWE ciphertext $\boldsymbol{d} = \mathbf{C} \boxdot (\boldsymbol{d_1} - \boldsymbol{d_0}) + \boldsymbol{d_0}$.
- $\boldsymbol{c} \leftarrow \mathsf{TableLookup}(\boldsymbol{db}, \boldsymbol{q})$. It takes a database $\boldsymbol{db}$, which is a list of trivial TRLWE, and a query $\boldsymbol{q}$, which is a list of TRGSW encryptions of each bit of the desired index $\widetilde{i}$, as inputs. It outputs a TRLWE sample $\boldsymbol{c}$ of the $\widetilde{i}$-th item of the database.

In more detail, a TLWE ciphertext $\mathfrak{c}$ of message μ under the key $\mathfrak{K}$ is in the form of $(\mathfrak{a}, \mathfrak{b})$ where $\mathfrak{a} \xleftarrow{U} \mathbb{T}^n$, $b = \mu + e + \mathfrak{a} \cdot \mathfrak{K}$, $e \leftarrow \chi$ and χ is a Gaussian distribution with a standard deviation η. A TRLWE ciphertext $\boldsymbol{c}$ of a message $\mu \in \mathbb{T}_N[X]$ is represented as (a, b), where $a, b \in \mathbb{T}_N[X]$. A TRGSW ciphertext $\mathbf{C}$ is constructed by $\mathbf{C} = \mathbf{Z} + \mu \cdot \mathbf{H} \in \mathbb{T}_N[X]^{2\ell \times 2}$, where $\mathbf{Z}$ is a list of 2ℓ TRLWE encryptions of zero, $\mu \in \mathbb{Z}_N[X]$ and $\mathbf{H}$ is the canonical gadget matrix.

TFHE Definitions. Let I be an ideal of $\mathbb{Z}[X]$, $\mathfrak{R} = \mathbb{Z}[X]/I$ and $\mathbb{T}_I[X] = \mathbb{T}[X]/I$. Let a phase function φ be a Lipschitz morphism from a $\mathfrak{R}$-module M to $\mathbb{T}_I[X]$. Let χ be an error distribution on M and $(\varphi_s)_{s \in \mathcal{S}}$ be a family of phase functions indexed by a secret s in the secret key space $\mathcal{S}$. Then there is an abstract TLWE problem definition.

Definition 1 (Abstract TLWE problem [6]). *A homogeneous* TLWE *distribution for the secret s is defined as the sum of the uniform distribution over $ker(\varphi_s)$ and an error from a Gaussian distribution χ. Then abstract* TLWE *is λ-secure if neither of the following two problems can be solved in less than 2^λ bit operations:*

- TLWE *decision problem: Given samples in M, distinguish if they come from the uniform distribution on M or from the homogeneous* TLWE *distribution with the unknown secret s.*
- TLWE *search problem: Given samples from the homogeneous* TLWE *distribution for a particular secret s, find s.*

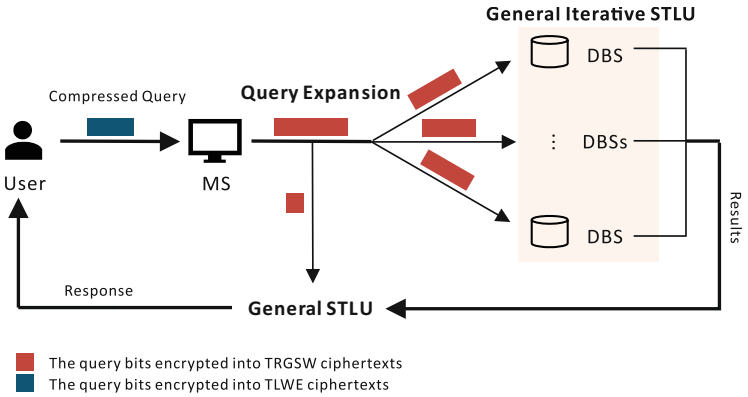

Fig. 1. The architecture of PPIR.

In the abstract TLWE problem, if $I = X + 1$, $M = \mathbb{T}^{n+1}$, $\mathfrak{c} = (\mathfrak{a}, \mathfrak{b})$ and $\varphi_{\mathfrak{K}}(\mathfrak{c}) = \mathfrak{b} - \mathfrak{a} \cdot \mathfrak{K}$, the definition is a concrete scale-invariant LWE [6], whose average case is asymptotically as hard as worst-case lattice problems. In other words, the semantic security of TLWE encryption is asymptotically equivalent to worst-case lattice problems.

If $I = X^N + 1$, $M = \mathbb{T}_N[X]^2$, $\boldsymbol{c} = (a, b)$ and $\varphi_K(\boldsymbol{c}) = b - a \cdot K$, the definition is a concrete ring version of LWE. The semantic security of TRLWE encryption is based on the ring version of LWE.

TFHE Lemmas. The error of a TLWE ciphertext $\mathsf{Err}(\mathfrak{c})$ of a message μ is equal to $\varphi_s(\mathfrak{c}) - \mu$. The variance of $\mathsf{Err}(\mathfrak{c})$ is denoted by $\mathsf{Var}(\mathsf{Err}(\mathfrak{c}))$. The error and their variance of TRLWE and TRGSW ciphertexts are denoted by $\mathsf{Err}(\boldsymbol{c})$, $\mathsf{Err}(\mathbf{C})$, $\mathsf{Var}(\mathsf{Err}(\boldsymbol{c}))$ and $\mathsf{Var}(\mathsf{Err}(\mathbf{C}))$, respectively. Then, the error variance of a CMux output is estimated in Lemma 1.

Lemma 1 (CMux gate variance [6]). *Let* $\boldsymbol{d_0}, \boldsymbol{d_1}$ *be* TRLWE *ciphertexts and* $\mathbf{C}$ *be a* TRGSW *ciphertext. Then,*

$$\mathsf{Var}(\mathsf{Err}(\mathsf{CMux}(\mathbf{C}, \boldsymbol{d_1}, \boldsymbol{d_0})) \leq \tau(\boldsymbol{d_0}, \boldsymbol{d_1}) + \vartheta(\mathbf{C}), \tag{1}$$

where $\tau(\boldsymbol{d_0}, \boldsymbol{d_1}) = \max(\mathsf{Var}(\mathsf{Err}(\boldsymbol{d_0})), \mathsf{Var}(\mathsf{Err}(\boldsymbol{d_1})))$, $\vartheta(\mathbf{C}) = (kN + 1)\epsilon^2 + (k + 1)\ell N \beta^2 \mathsf{Var}(\mathsf{Err}(\mathbf{C}))$ *where* ϵ *and* β *are parameters related to the canonical gadget decomposition algorithm.*

3 PPIR

In the proposed model, a single database is partitioned into multiple segments, each stored on an individual server. Figure 1 illustrates this configuration, where a Map Server (MS) is typically responsible for receiving user queries and delivering responses. The specific algorithms depicted in Fig. 1 will be elaborated on in subsequent sections. A PPIR protocol involves the following four key stages:

- **Preprocessing**. A database manager splits the single database into 2^{n_m} parts and stores each partition on a Database Server (DBS). Each DBS formats its database for the query. A user generates her key materials. If the user needs to frequently query a database on a mobile phone, she should register public keys.
- **Query Generation**. The user encrypts the desired index $\widetilde{i}$ and sends the ciphertext as a query to the MS.
- **Response Computation**. On receiving a query from a user, the MS sends partial queries to all the DBSs. It then collects the results from the DBSs and combines them with the remaining query to generate a response. The MS then returns this response to the user.
- **Response Decryption**. The user decrypts the response to get the $\widetilde{i}$-th entry of the database.

All the servers are honest but curious in that they will collaboratively learn the user query. The security of PPIR is then defined as follows.

Definition 2 (Security of PPIR). *A PPIR protocol with $2^{n_m}+1$ servers is secure if for any two queries, q_0 and q_1, no PPT adversary could distinguish the view of all the servers produced after they receive the query q_0 from the view of all the servers produced after they receive the query q_1. Note that a view of a server includes all the inputs, outputs, and internal states.*

The correctness and communication efficiency of PPIR is defined as follows.

Definition 3 (Correctness). *A PPIR protocol is correct if the answer obtained by a user is the $\widetilde{i}$-th entry in the single database.*

Definition 4 (Communication efficiency). *A PPIR protocol is communication efficient if the overall communication of a user is less than downloading the database itself.*

3.1 Query Compression and Expansion

The communication cost is a major concern in PIR schemes. To compress the query size, [16] presents a packing and unpacking procedure. Specifically, the user packs the κ-bit index into ℓ polynomials as $\sum_{j=0}^{\kappa} \frac{i_j}{B_g} X^j, \ldots, \sum_{j=0}^{\kappa} \frac{i_j}{B_g^\ell} X^j$, and then encrypts them as ℓ TRLWE ciphertexts. In the unpacking process, the server employs SampleExtract and PrivKS (Algorithm 3 in [6]) to obtain κ TRGSW ciphertexts.

In TFHE, the CircuitBootstrapping algorithm (Algorithm 11 in [6]) is to convert a TLWE ciphertext into a TRGSW ciphertext. This involves 2ℓ PrivKS operations and ℓ GateBootstrapping operations (Algorithm 9 in [6]). Given a TLWE ciphertext of message μ, the GateBootstrapping algorithm generates a TLWE ciphertext of μ/B_g^i, where $i \in [1, \ell]$. Due to the small value of noise variance, this operation changes the torus representation from 32-bit integers to 64-bit integers. The PrivKS algorithm then adds the message in ℓ TLWE ciphertexts to either a part or b part, where (a, b) is a TRLWE sample of zero.

Algorithm 1: Query Compression

Input: A security parameter λ, the desired index $\widetilde{i} = (i_0, \ldots, i_{\kappa-1})$ and the secret key $\mathfrak{K}$

. **Output**: A compressed represntation of $\kappa\ell$ TLWE ciphertexts.

Sample a random seed $seed \xleftarrow{U} \{0,1\}^\lambda$ and run $\mathsf{PRG}(seed, \kappa\ell n)$ to produce $\kappa\ell$ mask vectors $\mathfrak{a}_{p,z} \in \mathbb{T}^n$, where $p \in [0, \kappa-1]$ and $z \in [1, \ell]$

Sample $\kappa\ell$ error values $e_{p,z} \leftarrow \chi$ from a Gaussian distribution over $\mathbb{T}$

Compute $\mathfrak{b}_{p,z} = \frac{i_p}{B_g^z} + e_{p,z} + \mathfrak{a}_{p,z} \cdot \mathfrak{K}$

return $(\mathfrak{b}_{0,1}, \ldots, \mathfrak{b}_{\kappa-1,\ell}, seed)$.

Algorithm 2: Query Expansion (Calling Algorithm 3 in [6])

Input: A query $(\mathfrak{b}_{0,1}, \ldots, \mathfrak{b}_{\kappa-1,\ell}, seed)$, a key-switching key $\mathsf{KS}_{i,j}^{(f_u)} \in \mathsf{TRLWE}_{K,\gamma}(f_u(\frac{\mathfrak{K}_i}{2^j}))$ where $i \in [0, n-1]$, $j \in [1, t]$, $f_u : \mathbb{T} \to \mathbb{T}_N[X]$ is a secret R-Lipschitz morphism for $u \in \{0,1\}$. Specifically, $\mathsf{TRLWE}_{K,\gamma}(f_0(x)) = (a + x, b)$ and $\mathsf{TRLWE}_{K,\gamma}(f_1(x)) = (a, b + x)$, where (a, b) is a TRLWE sample of 0.

Output: κ TRGSW ciphertexts of the desired index $\widetilde{i} = (i_0, \ldots, i_{\kappa-1})$ under the secret key K.

Run $\mathsf{PRG}(seed, \kappa\ell n)$ to obatin $\kappa\ell$ mask vectors $\boldsymbol{a}_{p,z} \in \mathbb{T}^n$ for $p \in [0, \kappa-1]$ and $z \in [1, \ell]$

Compose $\kappa\ell$ TLWE ciphertexts $\mathfrak{c}_{p,z} = (\mathfrak{a}_{p,z}, \mathfrak{b}_{p,z}) \in \mathsf{TLWE}_{\mathfrak{K},\eta}(\frac{i_p}{B_g^z})$

for $p = 0$ to $\kappa - 1$ **do**

  for $z = 1$ to ℓ **do**

    for $u = 0$ to 1 **do**

      $\boldsymbol{c}_{p,z+u\cdot\ell} \leftarrow \mathsf{PrivKS}(\mathsf{KS}^{(f_u)}, \mathfrak{c}_{p,z})$

return κ TRGSW $\mathbf{C}_p = [\boldsymbol{c}_{p,w}]$, where $p \in [0, \kappa-1]$ and $w \in [1, 2\ell]$.

Our starting point is that TRLWE ciphertext is still costly compared to TLWE. Algorithm 1 shows the query compression procedure. Suppose the query has κ bits, then $\ell\kappa$ TLWE ciphers should be generated. Since in TLWE, $\mathfrak{a} \xleftarrow{U} \mathbb{T}^n$, one could generate $\ell\kappa n$ random elements in $\mathbb{T}$ using the pseudorandom generator PRG with one seed. Table 1 clearly shows the benefits of this approach in query and response sizes.

Algorithm 2 shows the query expansion procedure. It uses the PrivKS algorithm in [6]. The correctness is shown in Theorem 1.

Theorem 1. *The Query Expansion Algorithm produces* κ TRGSW *ciphertexts* $\mathbf{C}_p = [\boldsymbol{c}_{p,w}]$, *satisfying Eq. (2) for* $p \in [0, \kappa-1]$ *and* $z \in [1, \ell]$*:*

$$\begin{cases} \boldsymbol{c}_{p,z} = (a_{p,z} + \dfrac{i_p}{B_g^z}, b_{p,z}) \\ \boldsymbol{c}_{p,z+\ell} = (a_{p,z+\ell}, b_{p,z+\ell} + \dfrac{i_p}{B_g^z}) \end{cases}, \tag{2}$$

where $(a_{p,j}, b_{p,j})_{p\in[0,\kappa-1],j\in[1,2\ell]}$ *is a* TRLWE *sample of* 0 *under the key* K*. And the error and error variance directly follow from Algorithm 3 in [6], which are given by*

- $\|\mathsf{Err}(\mathbf{C}_p)\|_\infty \leq \|\mathsf{Err}(\mathfrak{c})\|_\infty + (n+1)2^{-(t+1)} + t(n+1)\mathcal{A}_{\mathsf{KS}}$ *(worst case),*
- $\mathsf{Var}(\mathsf{Err}(\mathbf{C}_p)) \leq \mathsf{Var}(\mathsf{Err}(\mathfrak{c})) + (n+1)2^{-2(t+1)} + t(n+1)\vartheta_{\mathsf{KS}}$ *(average case),*

where $\mathcal{A}_{\mathsf{KS}}$ *and* $\vartheta_{\mathsf{KS}} = \gamma^2$ *are, respectively, the amplitude and the variance of the error of* KS*. And if* $\|\mathsf{Err}(\mathbf{C}_p)\|_\infty \leq \frac{1}{4}$*, we are guaranteed that* $\mathbf{C}_p$ *is a valid* TRGSW *sample of* i_p *under the key* K*.*

Proof. It is easy to prove the correctness of Eq. (2). According to Algorithm 3 in [6], we know that the $\mathsf{PrivKS}(\mathsf{KS}^{(f)}, \mathfrak{c}^{(1)}, \ldots, \mathfrak{c}^{(m)})$ algorithm outputs a TRLWE sample of message $f(\mu_1, \ldots, \mu_m)$, where μ_i is the message of $\mathfrak{c}^{(i)}$ and m is the number of TLWE ciphertexts. In our case, we set $m = 1$. Therefore, $\mathsf{PrivKS}(\mathsf{KS}^{(f_u)}, \mathfrak{c}_{p,z})$ outputs a TRLWE sample of $f_u(\frac{i_p}{B_g^z})$, which is $\boldsymbol{c}_{p,z} = (a_{p,z} + \frac{i_p}{B_g^z}, b_{p,z})$ when $u = 0$, and $\boldsymbol{c}_{p,z+\ell} = (a_{p,z+\ell}, b_{p,z+\ell} + \frac{i_p}{B_g^z})$ otherwise. As we call 2ℓ PrivKS to generate 2ℓ TRLWE ciphertexts respectively for each TRGSW ciphertext, the error and the error variance remain the same as in Algorithm 3 in [6]. □

Since we represent torus elements using 32-bit integers and do not apply binary decomposition. Therefore, the inequality for the error variance should be formulated as:

$$\mathsf{Var}(\mathsf{Err}(\mathbf{C})) \leq \mathsf{Var}(\mathsf{Err}(\mathfrak{c})) + (n+1)2^{-2(t'+1)} + t(n+1)\vartheta_{\mathsf{KS}}. \tag{3}$$

Here, t' is the product of t and the number of bits in the decomposition base. Since in TFHE, the security parameter is a function of n and standard deviation for LWE samples (Fig. 9 in [6]), we set $\eta = 2^{-28}$, $n = 2^{10}$, $t = 10$, $t' = 30$, and $\gamma = 2^{-28}$. Note that η is the standard deviation of Gaussian error for TLWE encryption. The security parameter for this set of parameters is greater than 128. Using these values, we estimate the error variance of the final output TRGSW to be approximately equal to 2^{-43}. Therefore, the standard deviation should be $2^{-21.5}$, and the noise bit length is about 13.

3.2 STLU Algorithms

We show our Simultaneous Table Lookup (STLU) algorithms step by step.

A Server with One GPU. Consider a server with a database, $\boldsymbol{db}$, containing $|\boldsymbol{db}| = 2^\kappa$ entries. Our protocol includes a **Preprocessing** step where each entry is taken as a trivial TRLWE $(0^N, b)$, where an entry is encoded to the coefficients of b. For example, the coefficients of the plaintext polynomial use the message space $\{0, 1, 2, 3\}$, which means that an entry may have $2N$ bits.

Then, for the first query bit, there are $2^{\kappa-1}$ pairs of inputs to the CMux gates. A core observation is that these CMux gates could be run independently,

Algorithm 3: Simultaneous Table Lookup (STLU)

Input: a database $\boldsymbol{db}$ with 2^{κ} entries where each entry is a trivial TRLWE in $\mathbb{T}_N[X]^2$, a vector of TRGSW ciphertexts $\boldsymbol{v}$ with dimension κ for a query index $\widetilde{i}$.
Output: The $\widetilde{i}$-th entry in the databse $\boldsymbol{db}$.

Set $\boldsymbol{db}^{(0)} = \boldsymbol{db}$
for $j = 0$ **to** $\kappa - 1$ **do**
  The even-numbered entries of the database $\boldsymbol{db}^{(j)}$ form a vector $\boldsymbol{dy}$, and the odd-numbered entries form $\boldsymbol{dx}$
  Compute parallelly $\boldsymbol{df} = \boldsymbol{dx} - \boldsymbol{dy}$
  Compute parallelly $\boldsymbol{db}^{(j+1)} = \boldsymbol{df} \boxdot \boldsymbol{v}_j + \boldsymbol{dy}$
return $\boldsymbol{db}^{(\kappa)}$.

Algorithm 4: Iterative STLU

Input: a database $\boldsymbol{db}$ with $2^{n_r+n_t}$ entries where each entry is a trivial TRLWE in $\mathbb{T}_N[X]^2$, a vector of TRGSW ciphers $\boldsymbol{v}$ with dimension $(n_r + n_t)$ for $\widetilde{i}$.
Output: The $\widetilde{i}$-th entry in the databse $\boldsymbol{db}$.

for $j = 0$ *to* $2^{n_t} - 1$ **do**
  Run STLU with $\boldsymbol{db}_{j \cdot 2^{n_r}:(j+1)\cdot 2^{n_r}}$ and $\boldsymbol{v}_{0:n_r}$, and store the STLU output locally
Form the STLU outputs as a database with 2^{n_t} entries and run STLU with the database and the query vector $\boldsymbol{v}_{n_r:n_r+n_t}$ to get an output
return *the output of step 3.*

and the TRGSW for the CMux gates are the same, which enables the gates to be executed simultaneously on a GPU. Let $\boldsymbol{db}^{(j+1)}$ be the output of the j-th iteration. We present a basic STLU algorithm in Algorithm 3.

When the database is larger, it may not be possible to preload all of the data into the GPU memory. In this case, we can divide the database into blocks and run STLU on each block individually. The outputs of STLU can be used to create a smaller table, on which STLU can be run again to obtain a response.

To illustrate this process, consider a scenario where the GPU memory can only hold up to 2^{n_r} entries once. If the database has a size of $2^{n_r+n_t}$, we can transfer 2^{n_r} entries to the GPU and run STLU with a vector of query ciphers $\boldsymbol{v}_{0:n_r}$ to obtain an output. We can then iteratively execute this process 2^{n_t} times using the same vector of query ciphers. This will produce a smaller database with 2^{n_t} entries, on which STLU can be run again with new query ciphers $\boldsymbol{v}_{n_r:n_r+n_t}$. This process is outlined in the Algorithm 4.

A Server with Multiple GPUs. It is natural to employ the Iterative STLU algorithm in a database server with multiple GPUs. Suppose the server has 2^{n_c} GPUs, and the database has $2^{n_r+n_t+n_c}$ entries. To perform the computation concurrently, we can use $2^{n_c} + 1$ processes. Each process is in charge of a GPU, and the remaining process is responsible for gathering the results from the other processes. More concretely, we partition the database into 2^{n_c} blocks. Each process runs the Iterative STLU algorithm on a block with a vector of query ciphers

Algorithm 5: General STLU

Input: a database $\boldsymbol{db}$ with $2^{s_i} r_i \leq 2^{n_{r_i}}$ entries where each entry is a trivial TRLWE cipher in $\mathbb{T}_N[X]^2$, a vector of TRGSW ciphers $\boldsymbol{v}$ with dimension $\kappa = s_i + \lceil log_2(r_i) \rceil$ for a query index $\widetilde{i}$.

Output: The $\widetilde{i}$-th entry in the database $\boldsymbol{db}$.

```
for (j = 0, s = 2^{s_i} r_i, db^{(0)} = db; j < κ; j++, s /= 2) do
    if 2 ∤ s then
        Append a trivial TRLWE cipher 0^{2N} to db^{(j)}, and set s = s + 1
    The even-numbered entries of the database form a vector dy, and the
    odd-numbered entries form dx
    Compute parallelly df = dx − dy
    Form a database with s/2 entries by computing parallelly db^{(j+1)} = df ⊡ v_j + dy
return db^{(κ)}.
```

$\boldsymbol{v}_{0:n_r+n_t}$. The process that gathers the results constructs a small database using the outputs from the other processes. Since n_c is small, the process can directly run TableLookup on the small database using the query vector $\boldsymbol{v}_{n_r+n_t:n_r+n_t+n_c}$.

Multiple Servers with GPUs. If we have multiple servers, we can use them simultaneously to query a massive database. The basic idea is depicted in Fig. 1. The massive database is divided into 2^{n_m} blocks with sizes $S_1, \ldots, S_{2^{n_m}}$, where 2^{n_m} is the number of DBS. The MS receives the user query and sends it to the 2^{n_m} DBSs, which search for the requested data in their respective blocks. The DBSs then return their responses to the MS, which combines them to find the final response.

Since the servers may have different computation abilities, the sizes of the blocks $S_1, \ldots, S_{2^{n_m}}$ could be different. And in particular, when $S_i = 2^{s_i} r_i$ and $2 \nmid r_i$, STLU is not applicable. To handle this case, we propose two general STLU algorithms that can be used to query a database of arbitrary size. Specifically, if $S_i \leq 2^{n_{r_i}}$, where $2^{n_{r_i}}$ is the maximum number of entries that can be preloaded into the GPU memory of the i-th DBS, we show a General STLU procedure in Algorithm 5 to find the desired entry in the i-th block.

If $S_i > 2^{n_{r_i}}$, let $S_i = q_i \cdot 2^{n_{r_i}} + R_i$ where $R_i < 2^{n_{r_i}}$. We could run STLU q_i times and General STLU once, which gives us $q_i + 1$ outputs to form a small database. We then run General STLU one more time on the small database to get a final output. The procedure is shown in Algorithm 6.

3.3 The PPIR Protocol

Putting everything together, we now describe how our PPIR protocol works as a whole. Consider a database $\boldsymbol{db}$ with 2^{κ} entries. Suppose there is an MS and 2^{n_m} DBSs available, and a user wants to retrieve the $\widetilde{i}$-th entry privately.

Preprocessing. The user runs $\mathsf{KeyGen}(\lambda)$ to produce a secret key sk and a public key pk and registers pk to the MS.

Algorithm 6: General Iterative STLU

Input: a database $\boldsymbol{db}$ with $2^{s_i} r_i > 2^{n_{r_i}}$ entries where each entry is a trivial TRLWE cipher in $\mathbb{T}_N[X]^2$, a vector of TRGSW ciphers $\boldsymbol{v}$ with dimension $s_i + \lceil log_2(r_i) \rceil$ for an index $\widetilde{i}$.

Output: The $\widetilde{i}$-th entry in the database $\boldsymbol{db}$.

1 Set $s = 2^{s_i} r_i$, $q = \lfloor s/2^{n_{r_i}} \rfloor$, $r = s \% 2^{n_{r_i}}$

2 $t = (r = 0)?n_{r_i} : \lceil log_2(r) \rceil$

3 **for** $j = 0$ *to* $q - 1$ **do**

4 Run STLU with $\boldsymbol{db}_{j \cdot 2^{n_{r_i}} : (j+1) \cdot 2^{n_{r_i}}}$ and $\boldsymbol{v}_{0:t}$, and store the STLU output locally

5 **if** $r! = 0$ **then**

6 Run General STLU with $\boldsymbol{db}_{q \cdot 2^{n_{r_i}} : s}$ and $\boldsymbol{v}_{0:\lceil log_2(r) \rceil}$

7 Run General STLU with the above STLU (and General STLU) outputs using the remaining query ciphers

8 **return** *the output of step 7.*

The database manager divides the whole database into 2^{n_m} partitions according to the computation abilities of the DBSs and decides a **query depth** d, which indicates how many query ciphers should be sent to each DBS. Each partition is stored on a separate DBS. Then, each DBS prepares its data as trivial TRLWE in the same way as the **Preprocessing** procedure in Sect. 3.2.

Query Generation. The user runs Query Compression described in Algorithm 1 to get a query $\boldsymbol{q}$ for the desired index $\widetilde{i}$.

Response Computation. On a user query $\boldsymbol{q}$, the MS and DBSs do the following in parallel, as shown in Fig. 2.

- The MS is responsible for cipher expansion and choosing the final response.
  (1) The MS runs Query Expansion described in Algorithm 2 to obtain κ TRGSW. Once MS produces one TRGSW, it immediately sends the query cipher to each DBS. This process repeats d times.
  (2) When all DBSs return responses, the MS runs General STLU with these responses and the remaining query cipher to get a final result $\boldsymbol{c}$, which is sent back to the user.
- Each DBS executes General Iterative STLU with the TRGSW given by the MS. Since the expansion process is far quicker than the table lookup, the DBS should receive the quired query ciphers before using them (except the first bit) so that it could perform table lookup almost continuously.

Response Decryption. The user decrypts the response $\boldsymbol{c}$ from the MS with sk to get a message, which is decoded as the $\widetilde{i}$-th entry of the database.

Remark 1. We assign different amounts of data to each DBS according to its computing ability, but we give them the same number of query bits. That is to

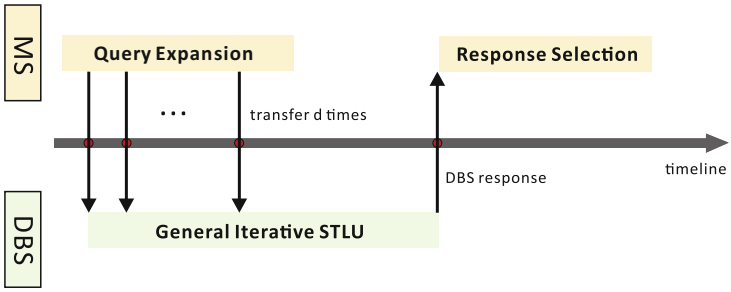

Fig. 2. Response Computation.

say, the query depth of each DBS should be consistent. So, for some DBSs, the response it returns may contain multiple database entries, while others may only return a single entry. These responses together form a small database, which the MS then uses for further table lookup.

3.4 Security Analysis

Security of PPIR. The security proof of our PPIR scheme is similar to SHECS-PIR [16] since the ciphertexts used in our PPIR also rely on the TFHE [6]. In the random oracle model, a TLWE expanded on the server side is indistinguishable from a standard TLWE in [6]. And the semantic security of the TFHE ensures that the servers can not distinguish the two views generated by two queries. So no information about the user index is leaked for a PPT adversary.

Correctness. To prove the correctness of PPIR, we use the symbol ϑ to denote the error variance and consider the error variance $\vartheta(\boldsymbol{c})$ of the final output $\boldsymbol{c}$.

Theorem 2. *Given a query* $\boldsymbol{q} = (\mathfrak{b}_{0,1}, \ldots, \mathfrak{b}_{\kappa-1,\ell}, seed)$*, the PPIR protocol outputs a* TRLWE *cipher* $\boldsymbol{c}$*. Let* $\boldsymbol{db}^{(j+1)}$ *be the output for the* j*-th iteration, where* $0 \leq j < \kappa$*. Apparently, the final output* $\boldsymbol{c}$ *is equal to* $\boldsymbol{db}^{(\kappa)}$*. Then the variance of* $\boldsymbol{c}$ *could be recursively computed as follows:*

$$\vartheta(\boldsymbol{db}^{(j+1)}) \leq \begin{cases} (k+1)\ell N\beta^2\vartheta(\mathbf{C}_j)) + (1+kN)\epsilon^2, & j = 0, \\ \vartheta(\boldsymbol{db}^{(j)}) + (k+1)\ell N\beta^2\vartheta(\mathbf{C}_j)) + (1+kN)\epsilon^2, & 0 < j < \kappa. \end{cases} \tag{4}$$

where $\vartheta(\mathbf{C}_j))$ *is given by Theorem 1.*

We give a proof sketch. Note that the table lookup algorithm involves merely CMux, so we only need to work out how the variance of each CMux gate goes. Since the inputs of CMux have the same error variance, by Lemma 1,

$$\vartheta(\boldsymbol{db}^{(j+1)}) \leq \vartheta(\boldsymbol{db}^{(j)}) + (k+1)\ell N\beta^2\vartheta(\mathbf{C}_j)) + (1+kN)\epsilon^2, \tag{5}$$

Table 3. Configuration of four DBSs

Device	CPU and GPU
DBS_1 (MS)	Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz × 64, 256GB GPU_0 / GPU_1: Nvidia RTX A6000, 48GB
DBS_2	Intel(R) Core(TM) i9-10900U CPU @ 2.80GHz × 20, 64GB GPU_2: GeForce GTX 1660, 6GB
DBS_3	Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz × 80, 128GB GPU_3: GeForce RTX 3090, 24GB
DBS_4	Intel(R) Core(TM) i9-10900U CPU @ 2.80GHz × 20, 64GB GPU_4: GeForce GTX 1660, 6GB

where $0 \leq j < \kappa$. Since the databases in DBSs are in the form of trivial TRLWE, we could define $\vartheta(\boldsymbol{db}^{(0)}) = 0$, which leads to the above inequalities.

To apply Theorem 2, we give a concrete parameter set by the estimator code in [12] where $\lambda = 128$. Let $N = 1024$, $k = 1$, $\ell = 2$, $\beta = 2^9$, $\epsilon = 2^{-21}$, $\vartheta(\mathbf{C}_j)) = 2^{-43}$ (the same parameters as we described in Sect. 3.1). We could figure out the variance of the final output $\boldsymbol{c}$ is about $\kappa \cdot 2^{-13}$. That means, for a database with 2^{25} entries, the variance of the final output is about $2^{-8.4}$. In other words, the standard deviation of $\boldsymbol{db}^{(\kappa)}$ is about $2^{-4.2}$. Furthermore, for $L = 32$, the noise bit length is less than 30 bits. As a result, under this parameter set, one could encode 2 bits in a torus element, which means that the entry size is $2 \times 2^{10-3} = 256$ bytes. Since $N = 1024$, the capacity of the database is about $2 \times 2^{10+25-3} = 2^{33}$ bytes.

Communication Efficiency. For a database with 2^κ entries, suppose each has 2^{n_d} bytes. Then the whole database is $2^{\kappa+n_d}$ bytes. That is, the communication cost of a user to download the database is $2^{\kappa+n_d}$ bytes. In PPIR, the query $\boldsymbol{q} = (\mathfrak{b}_{0,1}, \ldots, \mathfrak{b}_{\kappa-1,\ell}, seed)$. So, the communication cost of a user to send a query is $4(\kappa\ell + 1)$ bytes. There is an extra TRLWE to be received by the user. So the overall communication cost of a user is $4(\kappa\ell + 1) + |\mathsf{TRLWE}|$ bytes (Let $|\mathsf{TRLWE}|$ be the size of TRLWE in bytes).

4 Performance

Implementation Setup. We conducted experiments using four DBSs, with the configuration for each DBS shown in Table 3. Note that DBS_1 is also treated as the MS and has two GPU cards, while the other DBSs only have one. The experiments were implemented on Ubuntu 20.04 using CUDA C++. We set up a cluster environment using the high-performance portable message passing interface (MPICH) on the four DBSs for communication and used multi-processing to make them work in parallel. For convenience, we used the **mkl** library provided by Intel to implement Fast Fourier Transform (FFT) on the CPU, and the cufft library that comes with CUDA on the GPU. We set the security parameter λ

Table 4. The size of TFHE ciphers and keys

Ciphers and Keys	without PRG	with PRG
TLWE	2.46 KB	20 B
TRLWE	8 KB	4.0 KB
TRGSW	32 KB	16.0 KB
KS	320 MB	160 MB

Table 5. The running time of TFHE algorithms

Algorithm	Time
TLWE	0.01 ms
TRLWE	1.6 ms
TRGSW	4.16 ms
ExternalProduct	0.08 ms

Table 6. Computation time of TableLookup and STLU, and database transfer time (ms)

DB Size	TableLookup	STLU	Transfer
2^{10}	156.46	5.67	0.69
2^{11}	315.67	7.67	1.45
2^{12}	635.12	10.93	2.88
2^{13}	1076.22	16.41	5.74
2^{14}	2175.75	27.26	11.30
2^{15}	4310.16	49.08	22.34
2^{16}	8569.18	90.91	44.71
2^{17}	20371	169.26	92.46
2^{18}	40906.2	318.65	194.90
2^{19}	81821.2	724.48	380.77

Table 7. Computation time of Iterative STLU with $n_r = 19$ (s)

DB Size	**Iterative STLU**
2^{20}	2.21
2^{21}	4.43
2^{22}	8.82
2^{23}	17.69
2^{24}	36.38

for the TFHE scheme to 128. Let $k = 1$, $N = 1024$, and the coefficients of the plaintext polynomials use the message space $\{0, 1, 2, 3\}$.

4.1 TFHE Performance

Using the parameter set in Sect. 3.1, we give Table 4 to show the size of ciphers with or without PRG. We also evaluate basic TFHE algorithms on DBS_1 to test their running time on the CPU. The running time is given in Table 5.

4.2 Table Lookup Performance

In this section, we present the performance of STLU algorithms on GPU with database transfer time between CPU and GPUs. We also show the running time of TableLookup algorithm in [6] on CPU for comparison. All experiments were performed on DBS_1.

Table 6 shows the running time of the TableLookup and STLU algorithms with a preloaded database. From the table, we can see that as the database size doubles, the running time of both algorithms also doubles, with STLU growing slower than TableLookup. For the same database size, the running time of TableLookup is about 30 to 130 times that of STLU. The larger the database size is, the larger the gap in running time between the two algorithms is. Due to the

limitation in GPU memory size, we can only preload 2^{19} entries on the GPU. It is worth noting that the gadget decomposition algorithm, FFT, and polynomial multiplication also require GPU memory.

The results in Table 6 are mainly due to the ability of GPUs to efficiently handle parallel tasks. For a CMux gate, we use $2N$ threads to run ExternalProduct, with one thread for each coefficient of a TRLWE ciphertext. Then, for a database of size 2^κ, we need $2^{\kappa-1} \times 2N$ threads to select $2^{\kappa-1}$ TRLWE ciphertexts from the entire database in parallel using the first bit of the encrypted index. In contrast, on a CPU, we run ExternalProduct on the database in sequence, resulting in the long execution time of TableLookup. We also note that our TableLookup implementation is about 4 times slower than that in [16]. We stress that we run the TableLookup algorithm in a single thread.

In Table 6, we also include the database transfer time. It is clear that the database transfer time increases proportionally with the amount of data. When the database size reaches 2^{19}, database transfer time is nearly half of the STLU execution time. However, we want to stress that although GPU implementation requires additional database transfer time, the overall overhead is relatively modest, making it still preferable to use STLU rather than TableLookup.

As the size of the database increases, we need to apply Iterative STLU. Table 7 shows the running time of Iterative STLU with 2^{19} entries per iteration. On average, the cost for database transfer time is about 0.38 s and 0.72 s for STLU. So for a database with 2^{24} entries, it needs 32 iterations, which adds up to around 35.2 s. The results calculated using Table 6 are in good agreement with Table 7. Based on our experiments, it takes about 36 s to get a response from a database with 2^{24} entries. If we use two GPUs, the response time is about 21 s with 2^{23} entries for each GPU.

4.3 PPIR Performance

Parameters and Evaluation Metrics. In this paper, we consider the use of Iterative STLU for table lookup. The total time for database transfer and STLU execution is included in the running time for Iterative STLU. We run Iterative STLU on the four DBSs with varying values of n_r and n_t, where 2^{n_r} is the table lookup length of each iteration and 2^{n_t} is the number of iterations. The following are some important observations we got.

- Due to the difference in the GPU memory, we could only preload 2^{19} entries on GPU_0 and GPU_1 for running STLU once, 2^{17} on GPU_3, and 2^{15} on GPU_2 and GPU_4.
- For GPU_0, the performance of Iterative STLU is better when n_r is set to 17, compared to setting it to 18 or 19. The same is true for GPU_1.
- For the same database size, the running time of Iterative STLU of GPU_2 (also GPU_4) is about twice that of GPU_0 (also GPU_1, GPU_3).

These observations suggest that we should assign computation tasks based on the ability of each DBS, allowing them to collaborate and complete the table

Table 8. Computation time (s) and communication cost (bytes) for different database sizes (Each entry size = 256 bytes). The columns labeled GPU_i show the running time of Iterative STLU on GPU_i. The "Overall" column shows the response time in the parallelized mode. The "Query" and "Response" show the query and response size.

DB Size	DBS_1		DBS_2	DBS_3	DBS_4	Overall	Query	Response
	GPU_0	GPU_1	GPU_2	GPU_3	GPU_4			
	$n_r = 17$	$n_r = 17$	$n_r = 15$	$n_r = 17$	$n_r = 15$			
2^{20}	0.524	0.524	0.508	0.489	0.511	0.526	164	8K
2^{21}	1.056	1.065	1.019	0.994	1.024	1.067	172	8K
2^{22}	2.080	2.112	2.052	2.010	2.064	2.114	180	8K
2^{23}	4.204	4.246	4.088	4.033	4.109	4.248	188	8K
2^{24}	8.492	8.592	8.225	8.208	8.287	8.594	196	8K
2^{25}	16.476	16.670	16.321	16.227	16.612	16.688	204	8K

Table 9. Communication cost comparison (bytes)

	Query	Response
SHECS-PIR	$64K \times \kappa$	32K
SHECS-PIR*	32K	32K
PPIR	$4(\kappa\ell + 1)$	8K

* indicates that the SHECS-PIR uses compressed queries.

Table 10. Computation time comparison (s)

DB size	PPIR	SHECS-PIR
2^{16}	0.09	6.79
2^{18}	0.32	14.10
2^{20}	0.53	41.75
2^{22}	2.11	158
2^{24}	8.59	590
2^{25}	16.69	1184

lookup task in a relatively short time. Therefore, we divide the entire database into five parts in a ratio of 2 : 2 : 1 : 2 : 1 and assign the data to the five GPUs based on their computing abilities. For GPU_0, GPU_1, and GPU_3, we set n_r to 17, and for GPU_2 and GPU_4, we set n_r to 15. This enables us to fully utilize the computation ability of each GPU and speed up the calculation. In practice, we calculate the response time for each GPU individually and then combine the results using STLU to obtain the overall response time. The running time for each GPU is listed in Table 8. Despite the uneven workload distribution, the response time of each GPU is so consistent that there is almost no time wasted.

From Table 8, we can see that the Iterative STLU algorithm performs well and achieves similar run times on each DBS. However, the overall response time is slightly longer due to the communication costs between the MS and DBSs, as well as the STLU algorithm. In summary, the overall response time scales linearly with the size of the database. For a database with 2^{25} entries, the overall response time is about 17 s. However, the test was limited by CPU memory and could not be performed on larger databases. To handle larger databases, we would need additional devices (Table 9).

Compared with SHECS-PIR. We also compare our protocol with SHECS-PIR proposed in [16]. The parameter set used in [16] is $N = 2048$ and $L = 64$, which leads to different communication costs of TRLWE ciphers. For computation cost, we summarize the experimental results in Table 10. We calculate the computation cost of SHECS-PIR by adding the time costs of the First-step and Main, given in their Tables 4 and 5 [16], which mainly consist of query unpacking and table lookup work on the server side. The table lookup time for SHECS-PIR increases roughly exponentially with the increase of κ. In contrast, our computation time is significantly lower due to the design of parallel PIR.

5 Conclusion

We show a PPIR protocol with minimal communication cost and a fast response time. The benefits of communication cost come from our compression and expansion algorithms for a PIR query and the underlying table lookup algorithm of TFHE. The fast response time comes from the parallel PIR protocol design, including parallel DBSs, parallel GPUs, parallel execution of expansion, and table lookup. It is natural to apply the parallelization idea in other PIR schemes to further improve their response time on a larger database. The compression and expansion algorithms could also benefit the SHECS-PIR proposal. However, exploring and implementing other PIR schemes require significant engineering work, which we may undertake in the future.

Acknowledgements. This work is supported by the Guangdong Major Project of Basic and Applied Basic Research (2019B030302008) and the National Natural Science Foundation of China (No. 62272491, and No. 61972429).

References

1. Aguilar-Melchor, C., Barrier, J., Fousse, L., Killijian, M.O.: XPIRâĂŕ: private information retrieval for everyone. Proc. Priv. Enhancing Technol. **2016**(2), 155–174 (2015)
2. Ali, A., et al.: Communication-computation trade-offs in PIR. In: Bailey, M., Greenstadt, R. (eds.) 30th USENIX Security Symposium, USENIX Security 2021, August 11–13, 2021, pp. 1811–1828. USENIX Association (2021). https://www.usenix.org/conference/usenixsecurity21/presentation/ali
3. Angel, S., Chen, H., Laine, K., Setty, S.: Pir with compressed queries and amortized query processing. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 962–979 (2018). https://doi.org/10.1109/SP.2018.00062
4. Beimel, A., Ishai, Y., Malkin, T.: Reducing the servers computation in private information retrieval: PIR with preprocessing. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 55–73. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44598-6_4
5. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Rogaway, P. (ed.) Advances in Cryptology - CRYPTO 2011, pp. 505–524. Springer, Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_29

6. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. J. Cryptology **33**(1), 34–91 (2019). https://doi.org/10.1007/s00145-019-09319-x
7. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: Proceedings of IEEE 36th Annual Foundations of Computer Science, pp. 41–50 (1995). https://doi.org/10.1109/SFCS.1995.492461
8. Demmler, D., Herzberg, A., Schneider, T.: Raid-PIR: practical multi-server PIR. In: CCSW 2014: Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security, pp. 45–56. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2664168.2664181
9. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012). https://eprint.iacr.org/2012/144
10. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 75–92. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_5
11. Günther, D., Heymann, M., Pinkas, B., Schneider, T.: GPU-accelerated PIR with client-independent preprocessing for large-scale applications. Cryptology ePrint Archive, Report 2021/823 (2021). https://ia.cr/2021/823
12. Ilaria, C., Nicolas, G., Mariya, G., Malika, I.: Tfhe: fast fully homomorphic encryption library over the torus. Github (2021). https://github.com/tfhe/tfhe
13. Li, L., Pal, B., Ali, J., Sullivan, N., Chatterjee, R., Ristenpart, T.: Protocols for checking compromised credentials. In: CCS 2019: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 1387–1403. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3354229
14. Menon, S.J., Wu, D.J.: SPIRAL: fast, high-rate single-server PIR via FHE composition. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022, pp. 930–947. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833700
15. Mughees, M.H., Chen, H., Ren, L.: OnionPIR: response efficient single-server PIR. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS 2021: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15–19, 2021, pp. 2292–2306. ACM (2021). https://doi.org/10.1145/3460120.3485381
16. Park, J., Tibouchi, M.: SHECS-PIR: somewhat homomorphic encryption-based compact and scalable private information retrieval. In: Chen, L., Li, N., Liang, K., Schneider, S. (eds.) Computer Security - ESORICS 2020, pp. 86–106. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-59013-0_5
17. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: STOC 2005: Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing, pp. 84–93. Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1060590.1060603
18. Shi, E., Aqeel, W., Chandrasekaran, B., Maggs, B.: Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12828, pp. 641–669. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84259-8_22