SHECS-PIR: Somewhat Homomorphic Encryption-Based Compact and Scalable Private Information Retrieval

Jeongeun Park[1] and Mehdi Tibouchi[2(✉)]

[1] Department of Mathematics, Ewha Womans University, Seoul, Republic of Korea
jungeun7430@ewhain.net
[2] NTT Corporation, Tokyo, Japan
mehdi.tibouchi.br@hco.ntt.co.jp

Abstract. A Private Information Retrieval (PIR) protocol allows a client to retrieve arbitrary elements from a database stored in a server without revealing to the server any information about the requested element. PIR is an important building block of many privacy-preserving protocols, and its efficient implementation is therefore of prime importance. Several concrete, practical PIR protocols have been proposed and implemented so far, particularly based on very low-depth somewhat homomorphic encryption. The main drawback of these protocols, however, is their large communication cost, especially in terms of the server's reply, which grows like $O(d\sqrt[d]{n})$ for an n-element database, where d is a parameter typically chosen as 2 or 3.

In this paper, we describe an efficient PIR protocol called SHECS-PIR, based on deeper circuits and GSW-style homomorphic encryption. SHECS-PIR reduces the communication cost down to $O(\log n)$ removing all other factors apart from database size while maintaining a high level of efficiency. In fact, for large databases, we achieve faster server processing time in addition to more compact queries.

Keywords: PIR · Privacy-preserving technique · Homomorphic encryption · TFHE

1 Introduction

Retrieving data even from a public database can be a privacy-sensitive operation, which may reveal unwanted information about the client to the database operator: this could be the case for example for databases of patents, stock quotes, medical conditions, compromised passwords and more. As a result, clients may request that the content of their queries be protected from the database server. This can be achieved using *private information retrieval* (PIR) protocols, as introduced by Chor et al. [16].

J. Park—This work was partially carried out while the first author was a research intern at NTT Corporation, Japan.

In PIR, the database is modeled as an array of elements[1], and clients are allowed to retrieve those elements by querying their indices in the array. The required security property is that those queries remain hidden from the database operator(s). A consequence of that property is that, in order to answer a query, the server has to process the entire database, making the protocol computationally heavy on the server side.

We can ask the security to hold either in a statistical sense or in a computational sense: this corresponds to two classes of protocols, called information-theoretic PIR (IT-PIR) on the one hand [7,16,19,20,25], and computational PIR (cPIR) on the other [1,5,8,10,18,21,23,27,28,30,33]. IT-PIR offers unconditional security guarantees, and is usually more computationally efficient, since it usually involves simple bit operations on the database. However, any non-trivial IT-PIR requires multiple non-colluding servers (as Chor et al. [16] proved that the trivial protocol in which clients are sent the entire database is communication optimal in the single-server setting), which is often not achievable in practical scenarios. On the other hand, cPIR can achieve sublinear communication with a single server, but is typically more computationally expensive as it usually involves cryptographic operations based on public-key primitives to be carried out on each element of the database (and its security guarantees rely on some hardness assumption).

Standard PIR schemes do not normally offer any guarantee regarding the privacy of the server, in the sense that a client may learn information about elements of the database other than just the requested one. A PIR protocol which ensures that a client only learns the desired element and no more is called a symmetric PIR, and can be seen as a single-server, multi-client variant of oblivious transfer.

The focus of this paper is (single-server) cPIR based on somewhat homomorphic encryption.

1.1 Achieving Efficient cPIR

Recent constructions of cPIR all rely on broadly similar approaches based on homomorphic encryption. Since homomorphic encryption makes it possible to compute on encrypted data, it is a natural fit for PIR.

In fact, cPIR can be achieved with asymptotically essentially optimal efficiency (both in terms of communication and computation) using *fully* homomorphic encryption (FHE): the client sends as its query an encryption of its desired index using the FHE scheme, and the server homomorphically applies to this ciphertext the function mapping an index to the corresponding database element and sends the result back to the client. For an n-element database, this protocol has an optimal query size of $O(\log n)$, an optimal server computation complexity of $O(n)$ (since the function can be represented as a circuit of size

[1] This basic building block enabling private queries to a contiguous array can then be combined with techniques like cuckoo hashing to achieve private queries to more advanced data structures like key-value stores.

$O(n)$) and the reply size is again optimal, linear in the size of a database entry. Unfortunately, those nice asymptotic formulas tend to hide impractically large constants corresponding to the considerable overhead of FHE, in terms of ciphertext expansion and in computation cost, due to the expensive *bootstrapping* step required to homomorphically evaluate large circuits.

Protocols suggested so far for practical cPIR have therefore been substantially more complicated than this simple description, so as to circumvent the large overhead of FHE and rely instead on more efficient *somewhat* homomorphic encryption schemes (SHE), that only support the homomorphic evaluation of circuits of limited depth. For instance, one of the first practical cPIR protocol, XPIR [1] is based on the BV somewhat homomorphic encryption scheme [9]. Several subsequent works [5,23] then considered other SHE primitives to achieve better efficiency in terms of communication or computation cost.

The basic underlying technique in those works can be described as follows: if we represent the database as an n-dimensional vector, and the query for the i-th database element as the vector of size n with all zeroes and a 1 in the i-th component, the desired element is simply the inner product between those two vectors. If the query vector is encrypted componentwise using an (at least) additively homomorphic scheme, the inner product can easily be evaluated in encrypted form and returned to the client. An obvious difficulty, however, is that the query itself consists of n ciphertexts, and hence communication is no longer sublinear. This can be solved using a technique due to J.P. Stern [33] in which the database is structured as a d-dimensional hypercube. With this structure, $d\sqrt[d]{n}$ ciphertexts are needed as query vectors rather than n. Computing the reply then involves the homomorphic evaluation of an arithmetic circuit of depth d instead of just a linear function: this is the basic structure of XPIR.

SealPIR improves upon XPIR in terms of query size at the cost of additional work on the server side. Instead of sending d query vectors of length $\sqrt[d]{n}$, the client sends d ciphertexts containing the information on the desired index, and the server expands those ciphertexts into ciphertext vectors in a homomorphic way. Further optimizations of this technique have recently been proposed in [3], in order to further reduce communication at the cost of increased computation and noise on the server side.

1.2 Our Contribution

The main observation of this work is that the basic FHE approach to cPIR described at the beginning of the previous section can in fact be instantiated in practice, *without bootstrapping*, and achieve the same level of efficiency as state-of-the-art schemes like SealPIR or even better, and with lower communication cost overall.

To do so, we rely on the TFHE homomorphic encryption scheme [12–14], which is an efficient implementation of the GSW [24] approach to homomorphic encryption. With respect to suitably structured circuits, GSW enjoys a slow (additive rather than multiplicative) noise growth, and can therefore evaluate relatively deep circuits without bootstrapping. This is in particular the case for the circuits representing a large lookup table, which is exactly what we want to evaluate in PIR. This lookup table circuit consists of a binary tree of depth $O(\log n)$ of multiplexer gates (CMUX gates in TFHE; see Fig. 1), and can be evaluated homomorphically without bootstrapping using the basic TFHE parameters even for very large database sizes.

We also use suitable key-switching techniques in order to efficiently implement the query expansion, whereby the packed query of the client, containing all the bits of the index in a single ciphertext, is decomposed bitwise into several ciphertexts to be fed into the CMUX tree. Since there are fewer resulting ciphertexts than in SealPIR ($O(\log n)$ compared to $O(d\sqrt[d]{n})$), this step is also more efficient in an asymptotic sense, although the implied constant in the big-O is actually larger in our case.

The resulting scheme, which we call SHECS-PIR (Somewhat Homomorphic[2] Encryption-based Compact and Scalable PIR), is competitive with SealPIR in terms of computation cost, and achieves better communication cost (particularly for the server's reply, where we are essentially optimal). In addition, SHECS-PIR scales better to larger databases: thanks to slower noise growth, no increase in parameters is needed until a much larger database size than SealPIR. In addition, our query ciphertext can contain multiple indices up to the point not exceeding the dimension of plaintext degree without increasing query size. Therefore, SHECS-PIR can be combined with all the efficient (cheap computation cost) multi-query PIR techniques using probabilistic batch codes [5] or just batch codes [26,32] for better performance on server's computation time with much lower network cost increase and query generation time.

1.3 A Note on Communication Cost

We mentioned earlier that the FHE approach to cPIR achieves *essentially* optimal complexity since the query size is $O(\log n)$ and the answer size is linear in the size of database entries. The caveat implied by "essentially" here is that, while that bound certainly holds if size is measured in terms of numbers of ciphertexts, there can be some additional overhead due to the *ciphertext expansion factor*, namely the ratio F between the size of ciphertexts and plaintexts in the underlying homomorphic encryption scheme. In fact, that expansion factor is an even larger contributor to communication cost in schemes like XPIR, since answer size incurs an overhead of F^{d-1}, which can be large when d grows (i.e. for larger database sizes).

One can mention recent efforts to reduce this expansion factor F down to a constant close to 1, e.g. in [23], which proposes novel techniques to achieve an asymptotically close to optimal communication complexity even when ciphertext

[2] We stress that SHECS-PIR uses *somewhat* homomorphic (or arguably "leveled fully homomorphic") encryption in the sense that it does not rely on bootstrapping. This is despite the fact that the underlying homomorphic encryption TFHE is bootstrappable, and hence an FHE scheme. Not using bootstrapping is simply better for efficiency.

expansion is taken into account. Those efforts, however, are largely orthogonal to the line of work in which this paper fits: while they do obtain better communication rate in an asymptotic sense, they have a substantial fixed cost. For instance, query size in [23] is around 200 MB for typical parameters, so the scheme only offers an attractive communication rate when database entries themselves have sizes in the hundreds of megabytes, and server computation time is accordingly large. This can be relevant in specific settings, but for more common cPIR use cases where database entries have sizes in kilobytes or less, it is not very practical.

Regarding the underlying encryption scheme of SHECS-PIR itself, it satisfies $F \approx 4$ for the security level and the large database sizes we consider, so the corresponding overhead is small (and communication cost is effectively smaller than the state of the art for this range of parameters). In an asymptotic sense, F would increase very slightly with both database size (in order to accommodate noise growth) and security level (to ensure the hardness of the underlying lattice problem), but the scaling is an iterated logarithm, so practically speaking, F can be considered a constant.

Along similar lines as [23], a previous paper due to Kiayias et al. [27] achieves cPIR with optimal communication rate for databases with large entries, in the sense that the total size of communication asymptotically approaches the size of the unencrypted database entry alone. Moreover, it does so by relying on leveled homomorphic encryption, and thus does not require bootstrapping, similarly to the present work. While this is an important feasibility result, it again has limited practicality, however, due to the heavy computational cost of the underlying encryption scheme, as the authors themselves underscore. Moreover, as in [23], there are substantial fixed communication costs that limit the applicability of the scheme to only databases with very large entries (the authors consider the retrieval of movie files of several gigabytes), which is again a different setting as the one we focus on.

Another recent work discussing various approaches to reducing communication costs for PIR in a range of parameters more in line with the focus of this work is Ali et al.'s paper [3]. It presents a number of ways to optimize concrete cPIR schemes for lower communication, a number of which are largely independent of this work, and in fact compatible (e.g., modulus switching in queries). It does however introduce a new cPIR scheme called MulPIR, which is slower than SealPIR but more compact. We do not include a detailed comparison with MulPIR, due to the lack of a readily available implementation; however, since it has larger query size than SealPIR and since replies consist of multiple ciphertexts, it should be less efficient than SHECS-PIR in terms of both communication (by comparison of query and answer sizes) and computation (because we perform similarly to SealPIR or better).

2 Basic Tools (Homomorphic Encryption Scheme)

2.1 Homomorphic Encryption

Our PIR protocol is constructed by a somewhat homomorphic encryption scheme which allows limited number of operations on ciphertexts. Homomorphic Encryption (HE) allows a computation on encrypted data, where PIR scenario wants to do. We give properties of our base homomorphic encryption scheme first and concrete algorithms next. Homomorpic encryption scheme consists of four algorithms (KeyGen, Enc, Dec, Eval). It is an encryption scheme having additional Eval algorithm to evaluate arbitrary function on ciphertexts. Our protocol uses the full power of homomorphic encryption (multiplication, addition on ciphertexts) to evaluate a homomorphic mux gate (data selector).

– Homomorphic mux gate: Given two encrypted data d_0, d_1 and an encryption of $b \in \{0, 1\}$, say C, it outputs d_0 if $C = \mathsf{Enc}(0)$, otherwise d_1.

It is easy to construct homomorphic mux gate using standard FHE schemes. However, the most concern is the efficiency in terms of error growth and computational time for a practical use. The less noise overhead after any operation of an FHE scheme, the more operations are possible with it, i.e. the deeper circuit can be constructed from it. The ciphertext of all existing FHE schemes contains a noise component in it. The noise grows with homomorphic operation with regard to Euclidean norm. GSW-style homomorphic encryption [24] which keeps noise overhead additive after homomorphic multiplication has deeper depth by utilizing asymmetric noise propagation. Furthermore, its multiplication is natural i.e. just multiplication over ciphertexts avoiding other additional algorithms (relinearization, key switching, modulus switching e.t.c). To obtain a ciphertext (usually a vector) encrypting multiplication of plaintexts using homomorphic operation in other non-GSW style FHE schemes, tensor product of ciphertexts vectors are done at first. The product of vectors causes size of vector quadratic so that extra algorithms such as relinearization are required to reduce the size as original ciphertext. TFHE [14] adapts GSW encryption over Torus, but makes multiplication faster preserving GSW property using its algebraic fact. From this reason, we can eventually implement an efficient PIR protocol so we introduce this TFHE scheme below. We implemented our protocol based on TFHE library [15].

2.2 TLWE and TRLWE

Notation: We denote λ as the security parameter. We define vectors and matrices in lowercase bold and uppercase bold, respectively. Dot product of two vectors $\mathbf{v}, \mathbf{w}$ is denoted by $<\mathbf{v}, \mathbf{w}>$. For a vector $\mathbf{x}$, $\mathbf{x}_i$ denotes the i-th component scalar. We denote that $\mathbb{B}$ as the set $\{0, 1\}$ and $\mathbb{T}$ as the real torus $\mathbb{R}/\mathbb{Z}$, the set of real number modulo 1. We denote $\mathbb{Z}_N[X]$ and $\mathbb{T}_N[X]$ by $\mathbb{Z}[X]/(X^N + 1)$ and $\mathbb{R}[X]/(X^N + 1)$ mod 1, respectively. $\mathbb{B}_N[X]$ denotes the polynomials in $\mathbb{Z}_N[X]$

with binary coefficients. The norm notation $\|\cdot\|$ denotes infinity norm. $\log(\cdot)$ is binary logarithm. We use the same notation as [14] for better understanding.

The TFHE scheme [14] is working entirely on real torus $\mathbb{T}$ and $\mathbb{T}_N[X]$ based on TLWE problem and TRLWE problem which are torus variant of LWE problem and RLWE problem respectively, where N is a power of two. It is easy to see that $(\mathbb{T},+,\cdot)$(resp. $(\mathbb{T}_N[X],+,\cdot)$) is $\mathbb{Z}$(resp. $\mathbb{Z}_N[X]$) module.

A TLWE (resp. TRLWE) sample is defined as $(\mathbf{a},b) \in \mathbb{T}^{kn+1}$ (resp. $\mathbb{T}_N[X]^{k+1}$) for any $k > 0$, where $\mathbf{a}$ is chosen uniformly over $\mathbb{T}^{kn}$(resp. $\mathbb{T}_N^k$) and $b = {<}\mathbf{a},\mathbf{s}{>}{+}e$. The vector $\mathbf{s}$ is a secret key which is chosen uniformly from $\mathbb{B}^{kn}$(resp. $\mathbb{B}_N[X]^k$) and the error e is chosen from Gaussian distribution with standard deviation $\delta \in \mathbb{R} > 0$. Furthermore, we follow the definition of trivial sample in [14]. as having $\mathbf{a} = \mathbf{0}$ and noiseless sample as having the standard deviation $\delta = 0$. Throughout this paper, we set $k = 1$ and $n = N$. Here, we denote the message space to $\mathcal{M} \subseteq \mathbb{T}$. A TLWE ciphertext of $\mu \in \mathcal{M}$ is constructed by adding a trivial TLWE message sample $(0,\mu)$ to a non-trivial TLWE sample. Therefore, the TLWE ciphertext of μ, say $\mathfrak{c}$, which we will interpret as a TLWE sample (of μ) is $(\mathbf{a},b) \in \mathbb{T}^{k+1}$, where $b = {<}\mathbf{a},\mathbf{s}{>} + e + \mu$. To decrypt it correctly, we use a linear function $\varphi_{\mathbf{s}}$ called *phase*, which results in $\varphi_{\mathbf{s}}(\mathfrak{c}) = b - {<}\mathbf{a},\mathbf{s}{>} = \mu + e$ and we round it to the nearest element in $\mathcal{M}$. For a TRLWE encryption, it follows the same way over $\mathbb{T}_N$ but a message μ is a polynomial of degree N with coefficients $\in \mathcal{M}$.

2.3 TRGSW and CMUX Gate

As we can see, TLWE and TRLWE samples have additive homomorphic property. In order to support multiplication, the authors of [14] define TGSW ciphertext which supports external product with TLWE ciphertext to get a TLWE sample encrypting multiplication of messages. It is possible to be extended to polynomials. In this paper, since we only use TGSW samples in ring mode, we use the notation TRGSW which is working with TRLWE and also give the definition of a TRGSW sample only.

For any positive integer $B_g \geq 2, \ell, k$, a TRGSW sample is a matrix $\mathbf{C} = \mathbf{Z} + \mu \cdot \mathbf{H} \in \mathbb{T}_N[X]^{(k+1)\ell \times (k+1)}$, where each row of $\mathbf{Z}$ is a TRLWE sample of zero and $\mathbf{H}$ is a gadget matrix which is defined by $\mathbf{H} = \mathbf{I}_{k+1} \otimes \mathbf{g} \in \mathbb{T}_N[X]^{(k+1)\ell \times (k+1)}$, where $\mathbf{g} = (1/B_g, \ldots, 1/B_g^{\ell})$.

The message μ is in $\mathbb{Z}_N[X]$. In this paper, we restrict the message space of TRGSW to $\{0,1\}$ and set $k = 1$ as we mentioned above. We denote $\mathsf{TLWE}(\mu), \mathsf{TRLWE}(\mu)$, and $\mathsf{TRGSW}(\mu)$ as a ciphertext of each proper message μ of $\mathsf{TLWE}, \mathsf{TRLWE}$, and TRGSW, respectively. An external product between a TRGSW sample and a TRLWE sample, denoted as $\boxdot$, is defined as $\mathbf{A} \boxdot \mathbf{b} = \mathbf{H}^{-1}(\mathbf{b}) \cdot \mathbf{A}$, where $\mathbf{A}$ is a TRGSW sample of μ_A, $\mathbf{b}$ is a TRLWE sample of μ_b and $\mathbf{H}^{-1}(\cdot)$ is the gadget decomposition function $Dec_{\mathbf{H},\beta,\epsilon}$ of [14] with different notation.

This external product outputs a TRLWE sample of $\mu_A \cdot \mu_b$. With the homomorphic operations, we can construct a small circuit which is called CMUX gate. It outputs one of two TRLWE samples depending on a message of TRGSW sample without decrypting it. To be concrete, $\mathsf{CMUX}(C, \mathbf{d}_0, \mathbf{d}_1) = C \boxdot (\mathbf{d}_1 - \mathbf{d}_0) + \mathbf{d}_0$, where $C = \mathsf{TRGSW}(\mu_C), \mathbf{d}_0 = \mathsf{TRLWE}(\mu_{d_0})$, and $\mathbf{d}_1 = \mathsf{TRLWE}(\mu_{d_1})$. Since we restricted the message space of TRGSW to $\{0,1\}$, if $\mu_C = 0$, CMUX gate outputs $\mathsf{TRLWE}(\mu_{d_0})$ otherwise, $\mathsf{TRLWE}(\mu_{d_1})$ is the output. We refer to [14] for more detail.

2.4 Basic Algorithms for TFHE

We introduce basic algorithms $\mathsf{SampleExtract}$ and PrivKS, which we use in our PIR protocol. $\mathsf{SampleExtract}$ converts TRLWE samples of polynomial with message coefficient under a key K (denoted as $\mathsf{TRLWE}_K(\sum_{i=0}^{N-1} \mu_i X^i)$) into $\mathsf{TLWE}(\mu_i)$ under a key $\mathfrak{K}$ (denoted as $\mathsf{TLWE}_{\mathfrak{K}}(\mu_i)$), where $\mu_i \in \mathbb{T}$ for $\forall i \in [0, N-1]$. It is possible because we can extract a coefficient of a polynomial (viewed as slots) as a scalar with algebraic operation and it works on the FHE ciphertext. This algorithm does not add any noise.

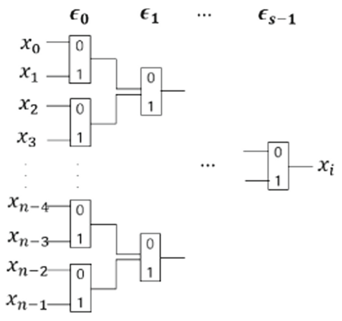
There is an algorithm called the Private Functional Key Switching (PrivKS) which allows to switch the message space from $\mathbb{T}$ to $\mathbb{T}_N[X]$. In other words, it can convert a TLWE sample under a key $\mathfrak{K}$ into a TRLWE sample under a key K. We use this algorithm for unpacking query step. This function takes a key switching key $\mathsf{KS}_{i,j^{(f)}} \in \mathsf{TRLWE}_K(f_u(\frac{\mathfrak{K}_i}{2^j}))$ and a $\mathsf{TLWE}_{\mathfrak{K}}(\mu)$ on input and outputs $\mathsf{TRLWE}_K(f_u(\mu))$. One can use the function f_u mapping from $\mathbb{T}^p$ to $\mathbb{T}_N[X]$ with p TLWE samples, however, $p = 1$ is enough for our protocol. Furthermore, we use two kinds of function f_u where u indicates the position where the input is added in a TRLWE sample. In detail, $\mathsf{TRLWE}_K(f_0(x)) = (a + x, b)$, $\mathsf{TRLWE}_K(f_1(x)) = (a, b + x)$, where $(a, b) \in \mathsf{TRLWE}_K(0)$, $x \in \mathbb{T}_N[X]$.

3 Overall Description

A PIR protocol consists of three basic procedure: query generation, response encoding(main computation), and response decoding [31]. Our PIR protocol requires a somewhat homomorphic encryption (SHE) scheme consists of four algorithms ($\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval}$). Unlike other basic cPIR protocols based on SHE, we use full power of homomorphic encryption i.e., multiplication over ciphertexts. Basically, multiplication is the most tricky step as we mention in Sect. 2, since it is usually followed by additional steps such as relinearization, modulus switching, key switching etc., furthermore, large noise growth is another trouble. However, GSW-style schemes support simple multiplication (with no other additional steps) and additive noise growth. So one of GSW-style scheme, TFHE, is adequate for instantiating our protocol. We introduce our protocol below.

3.1 Our PIR Protocol

Query Generation. A client chooses an index i to retrieve the ith item out of n data from server's DB and encrypts each bit of the index as $\log n$ ciphertexts. Then it sends to a server. Therefore, the query complexity is $O(\log n)$.



ϵ_{s-1}	$\ldots$	ϵ_0	Data
0	$\ldots$	0	x_0
0	$\ldots$	1	x_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	$\ldots$	1	$x_{\mathfrak{n}-1}$

Fig. 1. The Cmux binary decision tree (left figure) and Look Up Table (right table): the database with $\mathfrak{n}$ elements ($\mathfrak{n} = 2^s$). The figure represents how a server computes the desired i-th item from whole database. x_j stands for each data elements $\forall j \in [0, \mathfrak{n} - 1]$ and each ϵ_d is an binary element of the index $i = \sum_{d=0}^{s-1} \epsilon_d 2^d$, $\forall d \in [0, s-1]$.

Response Encoding. As a preprocessing, server saves a database as the look up table (See Fig. 1). The server runs $n - 1$ times MUX gate (a homomorphic mux gate) where it selects one of the input elements to output the encrypted i-th data. In our case, a MUX gate takes two data elements and one query ciphertext which encrypts a bit of the index. Depending on a query ciphertext, it obliviously selects one of two data inputs. After running MUX gates $n/2$ times (let's say it is the first level), all the outputs are ciphertexts so that the server does not know which items are chosen. It is possible thanks to homomorphic encryption. The server does the second level with the next query ciphertext and previous outputs running $n/4$ times MUX gates. Finally, after $\log n$ levels, it gives the output to the server. The total number of MUX gates for a server to evaluate is $n-1$, then the server's computational complexity is $O(n)$. This process is done via a look up table and binary decision tree (see Fig. 1).

Response Decoding. The client decrypts the ciphertext given by the server with his secret key. Unlike the previous efficient protocols (XPIR [1], SealPIR [5]), the complexity of PIR response does not depend on the expansion factor of cryptosystem, $F = |\text{ciphertext size}|/|\text{plaintext size}|$, in our approach. Note that constructing a look up table (LUT) is inefficient with traditional schemes like BV [9] or FV [22] in XPIR and SealPIR respectively, due to their structure with high noise level for every multiplication and large parameter. However, TFHE is suitable to construct our protocol with concrete parameters due to its nature. We give the concrete protocol, SHECS-PIR, from TFHE.

3.2 Concrete PIR Protocol (SHECS-PIR) from TFHE

We assume that a server has $\mathfrak{n} = 2^s$ data with β bit size (for convention, we set $\mathfrak{n}$ is a power of 2) and a client wants to retrieve the i-th data from server's database for $i \in [0, \mathfrak{n}-1]$. Before sending a query, each client registers its own key switching input set $\mathsf{KS}^{f_u} = \{\mathsf{KS}_{a,b}^{(f_u)}\}_{a \in [N+1], b \in [t]}$ which is a set of TRLWE ciphertexts of a secret key bits to a server as a setup process, for $u \in [0, 1]$. This seems quite necessary for every somewhat homomorphic encryption schemes as SealPIR also requires cryptographic material for substitution operation (key switching) from each client as a setup. However, It is sent only once by each client and the server uses it for the computation, every time a client who registered those. In fact, a server sets the database as a Look Up Table (LUT). It contains a list of indices from 0 to $\mathfrak{n}$-1 with binary representation in the left column and corresponding data represented as TRLWE message polynomial of degree at most N in the right column, where each coefficient is in $\mathbb{T}$ (see Fig. 1). Moreover, server can pack the database as much as possible by storing bit length of plaintext modulus ($\log p$) of the data element into a coefficient of message polynomial. Then, the database size can be decreased to $m = \mathfrak{n}/P_d$, where $P_d = \frac{N \log p}{\beta}$.

Query Generation:

(1) Choose an index $i \in [\mathfrak{n}]$ and represent $i = \sum_{j=0}^{j=s-1} \epsilon_j 2^j$ for $\epsilon_j \in \{0, 1\}$.
(2) A client encrypts each bit ϵ_j as a TRGSW ciphertext. $\rightarrow \log \mathfrak{n}$TRGSW ciphertexts as a query.
(3) Send them to a server.

Response Encoding: The server starts the main computation with its $\mathfrak{n}$ database. The server converts data element (as Fig. 1) into as a trivial TRLWE sample, $(0, D_j)$, where $D_j \in \mathbb{T}[X]/(X^N + 1)$ for $j \in [0, \mathfrak{n} - 1]$. He makes a binary tree with these data and runs CMUX gates $\mathfrak{n} - 1$ times via the binary CMUX tree to evaluate one TRLWE sample which contains the desired data. Note that one CMUX gate contains 2ℓ times ring multiplication.

Response Decoding: After receiving the answer from the server, the client can get the i-th data by decrypting the answer with the TRLWE secret key. In this case, the client only gets one ciphertext which is a TRLWE sample.

4 Implementation Details

4.1 Reducing Communication Cost

Packing and Unpacking Query. It is possible to compress query size as a ciphertext encrypting all the bits of the index i. In other words, a client packs all the bits of the index in a plaintext polynomial batching each bit into a coefficient then encrypt it. To unpack a query to $\log \mathfrak{n} (= s)$ ciphertexts encrypting each bit, we let the server do additional work which is called query unpacking step. Then the number of ciphertext for query is reduced to $\lceil \log \mathfrak{n}/N \rceil$ from $\log \mathfrak{n}$. As soon as a server gets a query ciphertext from a client, he unpacks the query as $\log \mathfrak{n}$ ciphertexts of each binary element of i. For example, let $i = 3, \mathfrak{n} = 16$ then the binary representation of 3 is 0011. A client gives an output of $\mathsf{Enc}(0011)$ to the server and he unpacks it to outputs of $\mathsf{Enc}(0), \mathsf{Enc}(0), \mathsf{Enc}(1)$, and $\mathsf{Enc}(1)$, where Enc is an encryption algorithm. If there exists an algorithm extracting (unbatching) one bit obliviously, a server runs it $\log \mathfrak{n}$ times, hence this step has $O(\log \mathfrak{n})$ computational complexity.

We show how to construct all the procedure with TFHE. Due to TRGSW sample's structure, client needs ℓ number of ciphertexts to pack query. Client gives TRLWE samples as a query then server unpacks it to TRGSW samples. A query consists of $\ell\mathsf{TRLWE}$ ciphertexts under the key K having $\log \mathfrak{n}$ binary elements of i. It is unpacked as $\ell \times \log \mathfrak{n}$ TLWE samples under the key $\mathcal{K}$ then converted to $\log \mathfrak{n}$ TRGSW samples under the key K.

[Query Generation]

(1) Choose an index $i \in [\mathfrak{n}]$ and represent $i = \sum_{j=0}^{j=s-1} \epsilon_j 2^j$ for $\epsilon_j \in \{0,1\}$.
(2) Set ℓ message polynomials as $\sum_{j=0}^{j=s-1} \frac{\epsilon_j}{B_g} X^j, \dots \sum_{j=0}^{j=s-1} \frac{\epsilon_j}{B_g^\ell} X^j$ for a positive integer $B_g > 2$.
(3) A client encrypts these polynomials as ℓ TRLWE samples.
$\rightarrow$ $\mathsf{TRLWE}_K(\sum_{j=0}^{j=s-1} \frac{\epsilon_j}{B_g} X^j), \dots, \mathsf{TRLWE}_K(\sum_{j=0}^{j=s-1} \frac{\epsilon_j}{B_g^\ell} X^j)$, where K is a TRLWE secret key of the client and $\epsilon_j \in \{0,1\}$ for $j \in [0, s-1]$.
(4) Send them to the server.

So a query consists of $\ell\lceil \frac{\log \mathfrak{n}}{N} \rceil\mathsf{TRLWE}$ ciphertexts in $\mathsf{SHECS\text{-}PIR}$. Since $\log \mathfrak{n}$ is much smaller than N, in general, just ℓ ciphertexts are required.

[Query Unpacking: Converting ℓ TRLWE samples to $\log \mathfrak{n}$ TRGSW samples.]

(1) Run $\mathsf{SampleExtract}(\mathsf{TRLWE}_K(\sum_{j=0}^{j=s-1} \frac{\epsilon_j}{B_g^w} X^j)) \rightarrow \{\mathsf{TLWE}_{\mathfrak{K}}(\frac{\epsilon_j}{B_g^w})\}_{j\in[0,s-1], w\in[1,\ell]}$ for $w \in [1, \ell]$
(2) For $j \in [0, s-1]$, $u \in [0,1]$ and $w \in [1, \ell]$, run $\mathsf{PrivKS}(\mathsf{KS}^{f_u}, \mathsf{TLWE}_{\mathfrak{K}}(\frac{\epsilon_j}{B_g^w})) \rightarrow \{\mathsf{TRGSW}_K(\epsilon_j)\}_{j\in[0,s-1]}$.

In total, server runs $\mathsf{SampleExtract}$ $\ell \log \mathfrak{n}$ times and PrivKS $2\ell \log \mathfrak{n}$ times. Essentially, $\mathsf{SampleExtract}$ is free since it just extracts coefficients from a polynomial, but PrivKS has a large constant (at most Nt) times ciphertext addition itself, where N is the dimension of ciphertext polynomial and t is a parameter of PrivKS. Usually $N = 1024$ or $N = 2048$, $t = 12$. To optimize query size, the client can concatenate all the $\ell \log \mathfrak{n}$ bits in one polynomial then only one ciphertext a query but the server's unpacking time is twice as a trade off.

Using Random Oracle. TFHE is basically a symmetric key encryption scheme so that a client can give just seed of uniformly random part of a TRLWE sample using random oracle. Then the server generates the exact value using the same oracle. Roughly, the query size is reduced by half since the seed size is $\{0,1\}^\lambda$. In general, LWE based symmetric key encryption scheme can use random oracle to reduce the communication cost.

Table 1. Communication and computation complexity, n = database size

	`Query`	`Answer`	`First-Step`	`Main`
XPIR	$O(d\sqrt[d]{n})$	$O(F^{d-1})$	N/A	$\Omega(n+F\sqrt{n})$
SealPIR$(d=2)$	$O(d\lceil\sqrt[d]{n}/N\rceil)$	$O(F^{d-1})$	$O(d\sqrt[d]{n})$	$\Omega(n+F\sqrt{n})$
SHECS-PIR w query unpacking	$O(\lceil\log n/N\rceil)$	$O(1)$	$O(\log n)$	$O(\ell n)$
SHECS-PIR w/o query unpacking	$O(\log n)$	$O(1)$	N/A	$O(\ell n)$

Table 2. Communication cost of SHECS-PIR and SealPIR for the same $\mathfrak{n}$, N, and β.

	`With First-step`		`Without First-step`	
	SHECS-PIR	SealPIR$(d=2)$	SHECS-PIR	SealPIR$(d=2)$
`Query[ctxt]`	$\ell\lceil\frac{\log\mathfrak{n}}{N}\rceil$	$d\lceil\frac{\sqrt{\mathfrak{n}}}{N}\rceil$	$\log\mathfrak{n}$	$d\sqrt{\mathfrak{n}}$
`Answer[ctxt]`	1	$\lceil\frac{2\log q}{\log p}\rceil$	1	$\lceil\frac{2\log q}{\log p}\rceil$

Modulus Switching for Answer Ciphertext. In order to reduce the answer size, a naive approach is modulus switching. Other homomorphic encryption primitives [9,22] use this technique either to reduce the noise contained in a ciphertext or to reduce the size of a ciphertext. We can easily employ it since both the ciphertext modulus and plaintext modulus can be set as a power of 2.

4.2 Comparison with Other Protocols

Communication and Computation Cost: We give a complexity comparison among previous works below in Table 1 (First-step is query unpacking in SHECS-PIR and query expansion in SealPIR). The complexity of main computation is expressed in polynomial multiplication unit so that SHECS-PIR has other factor ℓ since one CMUX gate consists of 2ℓ polynomial multiplication. The server does $2\ell(\mathfrak{n}-1)$ polynomial multiplication finally. This is because the schemes TFHE and FV work over different algebraic structure. The elements over the torus $\mathbb{T}_N[X]$ is rescaled by a factor 2^{64} to be mapped to 64 bit integers for implementation. Then we can view the ciphertext modulus as $q = 2^{64}$ and plaintext modulus as $p(<q)$. However, FV (or BGV) works over $\mathbb{Z}_q[X]/(X^N+1)$ (q is a prime s.t. $q = 1 \bmod 2N$) so that they can use NTT operation while TFHE uses FFT operation for ring multiplication. Furthermore, FFT can be more scaled than NTT in general. Therefore, the actual cost comparison does not seem proper. In SealPIR and XPIR's main computation, server does $2(\mathfrak{n}+F\sqrt{\mathfrak{n}})$ ring multiplication when $d=2$. Roughly, SHECS-PIR's server seems to work twice since we set $\ell = 2$. However, the actual cost is similar because the FFT operation in TFHE library [15] is more scaled than NTT used in SealPIR library [29].

SealPIR can also use some our optimization technique such as random oracle (when it uses symmetric key version) and modulus switching, hence, we can have similar ciphertext size in both protocols. Therefore, how many ciphertexts are needed for query and answer is important for communication cost. Although SHECS-PIR doesn't run query unpacking, we can see that the query size becomes smaller than the size of SealPIR with query expansion at some point since our query size complexity is $O(\log \mathfrak{n})$. Table 2 shows the exact number of ciphertexts of a query and an answer in SHECS-PIR and SealPIR. A query which is a TRLWE ciphertext can represent 2^N indices and usually $N \geq 1024$ so that we can say that the query size actually does not increase for realistic size of database. For $\mathfrak{n} = 2^{32}$, $N = 2048$, one ciphertext (=16 kB) is required for SHECS-PIR with query unpacking while 64 ciphertexts (=2048 kB) are needed for SealPIR with query expansion and database dimension $d = 2$ (so the database is a $2^{16} \times 2^{16}$ matrix). This is because SealPIR represents an index using $2^{16}/2048 (= \lceil \sqrt{\mathfrak{n}}/N \rceil) = 32$ for each database dimension. This size is as same as SHECS-PIR's just giving all $\log \mathfrak{n} (= 32)$ TRGSW ciphertexts (=2048 kB) in SHECS-PIR without query unpacking so that the server's running time would be much smaller also. In fact, the query unpacking would be faster than expansion if $\sqrt{\mathfrak{n}}$ is much larger than $N \log \mathfrak{n}$. For noise issue, SealPIR may increase N and decrease p, while we do not need to do. Moreover, the answer size does not increase since it does not depend on the expansion factor F. Therefore, we can achieve better performance on both total communication cost and server's computation for large database.

Noise Growth: Somewhat homomorphic encryption supports limited number of operation over ciphertexts, hence, the deeper depth a scheme has, the larger database its application can support without bootstrapping. Since bootstrapping takes relatively long time and require other material (quite large size) as an input, it is important not to use it as much as possible. Multiplication over ciphertexts, in general, incurs large noise growth. In fact, noise growth is depending on the size of plaintext in FV so that SealPIR keeps downsizing the plaintext modulus to achieve more depth. But it causes the factor F large, hence, it has an influence on server's answer size and main computation time as well. However, TFHE has larger depth since it has additive noise growth for both addition and multiplication and also the noise growth of it does not depend on plaintext modulus.

We show heuristic noise bound after server's computation of SealPIR and SHECS-PIR then how much noise has left until decryption will fail using noise budget defined in [11]. First, we can redefine TFHE ciphertext with rescaled version for integer representation of implementation ($q = 2^{64}$).

Definition 1 (Rescaled TFHE for implementation). *Let* $\mathbf{ct} = (c_0, c_1)$, *where* $c_i \in \mathbb{Z}_q[X]/(X^N + 1), i \in \{0, 1\}$ *be an* TRLWE *cipertext encrypting a message* $m \in \mathbb{Z}_p[X]/(X^N + 1)$. *Its scaled inherent noise* v *is the polynomial with the smallest infinity norm such that,*

$$\frac{p}{q}\mathbf{ct}(s) = \frac{p}{q}(c_0 + c_1 s) = m + v + ap,$$

where a *is a polynomial with integer coefficient.*

Table 3. TFHE error growth ($N = 2048, p = 2^{12}, q = 2^{64}$, the number of trial = 10000)

	Fresh Ciphertext	Addition	Multiplication
mean(bit)	11	12	42
standard deviation	0.12	0	0.12

Lemma 1. *A* TRLWE *ciphertext* **ct** *encrypting a message m can be correctly decrypted if the scaled inherent noise v satisfies*

$$\|v\| < \frac{1}{2}$$

Noise budget for rescaled TFHE is actually as same as FV's [11], where q is ciphertext modulus and p is plaintext modulus and v is the invariant noise contained in a ciphertext. In TFHE, p divides q since the two are both powers of 2 so that it causes less noise than the case $p \nmid q$ of SealPIR. The noise budget of both schemes is $-\log 2v$ A ciphertext is decryptable only when the noise budget of it is positive (>0). Now we can observe that how fast the noise budget contained in the reply ciphertext reaches to 0 in parameter database size $\mathfrak{n}$.

SealPIR(based on FV) error growth. Let v_{in} be the initial error, which is an error of a query essentially, and v_s be the error contained in a ciphertext which is generated after server's computation. We set $\|v_{out}\| = \|(\lfloor \frac{p}{q} v_s \rceil) \bmod p\|$, where $\|v_s\| \leq Np^2\mathfrak{n}\sqrt{\mathfrak{n}}(\|v_{in}\| + B)$ [5], where N is the dimension of plaintext, p is plaintext modulus, $\mathfrak{n}$ is the number of database, and B is a constant error generated from query expansion step. We assume the database dimension $d = 2$. Since the noise budget of this result ciphertext is $-\log \|2v_{out}\|$, it decreases with $O(\log \mathfrak{n})$ complexity.

SHECS-PIR(based on TFHE) error growth. Let v_{in} and v_{out} be the same notation defined above. Then we observe the final error based on TFHE noise analysis [14]. It satisfies $\|v_{out}\| \leq \log \mathfrak{n}((k+1)\ell N\beta(\|v_{in}\| + (N+1)2^{-(t+1)} + t(N+1)\|v_{ks}\|)$, where N is the dimension of plaintext, p is plaintext modulus, $\mathfrak{n}$ is the number of database, ℓ, t, β are constant of TRGSW sample and v_{ks} is key switching error (encryption of secret key). Then the noise budget of this result ciphertext decreases with $O(\log\log \mathfrak{n})$ complexity. Table 3 shows how much TFHE noise is added after addition and multiplication to the original fresh ciphertext having noise 11 bits (for 120 bits of security). All the operation is done over fresh ciphertext (non-evaluated) with the same noise distribution. Since query unpacking step which consists of addition does not add much error, we focus on multiplication error growth. According to our noise estimation above, we can see that $\log\log \mathfrak{n} + 42$ bits are the final error contained in server's reply. To decrypt it correctly (the noise budget > 0), $\log\log \mathfrak{n}$ should be smaller than 9. which means, $\mathfrak{n} < 2^{512}$. As a result, we are able to run large enough database without changing parameter using only somewhat homomorphic encryption functionalities. We can expect that the noise budget of the reply ciphertext would still remain positive with large enough data while SealPIR may not be able to support.

4.3 Security

The security of our cPIR scheme follows directly from the IND-CPA (i.e., semantic) security of the underlying homomorphic encryption scheme TFHE [14]. Indeed, the query consists of TFHE ciphertexts, and semantic security ensures that the server cannot learn any information about the underlying plaintexts, which encode the queried database index. Therefore, SHECS-PIR is a secure cPIR protocol.

The assumptions for security are slightly different in the version of the protocol with query compression and the version without: this is because in the latter one, the key material sent to the server consists of just the evaluation key, allowing the semantic security of TFHE to be proved under plain Ring-LWE). On the other hand, in the former case, the server is also provided with key-switching material, encrypting key-dependent information; the security proof for TFHE then relies on an additional circular security assumption, as is always the case for FHE schemes. This discrepancy, however, is not believed to have any impact on concrete security, since no attack is known on circular security.

As usual for lattice-based cryptographic schemes, we can estimate concrete security by evaluating the cost of the best possible attack against the proposed parameters (which in our case are selected as $N = 2048$, $q = 2^{64}$, and $\alpha = 6.957 \cdot 10^{-17}$ for the error magnitude, corresponding to our error distribution with standard deviation 2^{-55}). Albrecht et al.'s LWE estimator [2] shows that the best attack is then the primal uSVP attack [4,6], which yields 121 bits of security. As a comparison, SealPIR achieves 115 bits of security with their choice of parameters ($N = 2048$, $q = 2^{60} - 2^{18} + 1$, and $\alpha = 8/q$).

5 Experimental Result

Implementation Setup. All experiments are performed on a single core of a server with Xeon Platinum 8160 @ 2.10 10 GHz CPUs. In the concrete protocol SHECS-PIR, we set $k = 1$, then TRLWE sample consists of two polynomials, $(a, b) \in \mathbb{T}_N[X]^2$, where a is chosen uniformly. For TRGSW sample, we set $\ell = 2$, $B_g = 2^{15}$.

Communication and Computation Cost. Table 4 shows the actual cost using each library (SHECS-PIR based on TFHE [15] and SealPIR [29]). We stress that those numbers corresponds to the case when only one database element is stored in a given plaintext. It is possible to pack multiple database elements per plaintext in order to support larger databases.

We set $N = 2048$ and ciphertext modulus $q \approx 2^{60}$ for both protocols. Since the FFT multiplication in TFHE library performs better than SEAL's NTT, our main computation time is similar to SealPIR. However, the First step (query

Table 4. Computation cost of SHECS-PIR and SealPIR for the same $\mathfrak{n}$, $q \approx 2^{60}$, $N = 2048$.

DB size $\mathfrak{n}$	SHECS-PIR			SealPIR ($d = 2$)		
	2^{16}	2^{18}	2^{20}	2^{16}	2^{18}	2^{20}
Query[kB]	32	32	32	32	32	32
Answer[kB]	32	32	32	320	320	320
Server preprocessing[ms]	0	0	0	5733	23101	92944
First-step[ms]	4507	5073	5846	187	422	840
Main[ms]	2282	9024	35902	1935	7025	26833
NB	8	7	7	7	6	5
NB (w/o unpack)	8	7	7	N/A		

expansion, consisting of mainly polynomial additions) of SHECS-PIR is more expensive than SealPIR's for the database sizes considered in the table. It scales slower with database size, however (logarithmically rather than in the square root), so becomes negligible for larger databases.

Both protocols are based on "symmetric key" homomorphic encryption, so that they use the random oracle model to reduce the query size by half. We observe how much signal is left after the noise increase in homomorphic operations. NB represents the "noise budget" after server's computation in the table, namely the number of bits of plaintext recoverable above the noise in each of the N coefficients of the plaintext. For example, for $\mathfrak{n} = 2^{16}$ in SHECS-PIR, each coefficient of the reply can store up to 8 bits of information, for a total bandwidth of $8N = 16384$ bits of information (2048 bytes) per plaintext: this means that if database entries are $\beta = 288$ bytes long, we can store 7 of them per plaintext, and hence support database of size $\approx 2^{19}$ in that case). NB(w/o unpack) denotes the noise budget after server's computation without query unpacking step. As we can see that, query unpacking has very small error growth so that it has little impact on the noise budget. The noise growth in SHECS-PIR is in $\log \log \mathfrak{n}$ compared to SealPIR's $\log \mathfrak{n}$, so the noise budget is higher in SHECS-PIR, and we can support very large databases before this budget is reduced significantly. In SealPIR on the other hand, parameters have to be increased somehow to support large databases; there is a complicated set of trade-offs between the data element size β, the plaintext modulus p, the polynomial degree N and the array size $\mathfrak{n}$, with an increase in one resulting in a decrease on another, making parameter selection somewhat tricky. Comparatively, SHECS-PIR is relatively free of trade-offs as $\mathfrak{n}$ increases.

For the computation time of the database in the applicable range, the server processing time (main computation) scales very close to linearly with $\mathfrak{n}$ (the database size) and it is similar to SealPIR. We have a small overhead over SealPIR due to the choice of avoiding any database preprocessing, more precisely, storing database elements as NTT/FFT form in advance. Note that a plaintext is a

Table 5. Comparison between SHECS-PIR and SealPIR for large $\mathfrak{n}$, $q \approx 2^{60}$, $N = 2048$.

DB size $\mathfrak{n}$		2^{22}	2^{24}	2^{25}	2^{26}	2^{27}
SHECS-PIR	`Compressed-Query[#]`	2	2	2	2	2
	`Query[#]`	2×22	2×24	2×25	2×26	2×27
	`Answer[#]`	1	1	1	1	1
SealPIR ($d = 3$)	`Compressed-Query[#]`	3	3	3	3	3
	`Answer[#]`	100	100	100	100	100
SHECS-PIR	`First-step[s]`	15	16	17	18	18
	`Main[s]`	143	574	1167	2327	4645
	`NB`	7	6	6	6	6
SealPIR ($d = 3$)	`Server preprocessing[s]`	291	1192	[*out of memory*]		
	`Server time[s]`	132	489			

polynomial of 12 bit coefficients, while ciphertext consists of 64 bit coefficient. As a result, we have almost no storage overhead for the database in memory, compared to an overhead of more than 5 (=64/12) in SealPIR. This lets us support very large databases up to 2^{27} (corresponding to 2^{30} entries of 384 KB each, 384 GB of data in total), while the same could not be achieved with SealPIR on commodity hardware (See Table 5). In addition, for large databases, SealPIR makes it necessary to increase d, which results in a larger response size. Specifically, SealPIR simply fails if d is set to 2 for $\mathfrak{n} \geq 2^{22}$, so we have to set d to at least 3, and get a response consists of a hundred ciphertexts or more; due to memory constraints, we could run it only up to $\mathfrak{n} = 2^{24}$, with larger instances too big to fit in memory our relatively high-end server.

The communication cost (query and answer size) is expressed in the number of ciphertexts. `Compressed-Query[#]` denotes an optimization of query size (query unpacking in SHECS-PIR, query expansion in SealPIR). We can see that our total communication cost *even without query unpacking* is actually lower than SealPIR with query expansion for large database sizes, due to the much larger response size in SealPIR. Nevertheless, query unpacking becomes relatively negligible for large database sizes, so it would seem natural to use it as well and enjoy our close to optimal communication complexity.

The server computation time may seem large, but it is almost completely embarrassingly parallel, so on our 48-core server the total server computation time can be brought down to less than 100 s for $\mathfrak{n} = 2^{27}$, say, by using multithreading.

A Optimization Options of Reducing Communication Cost

We explain modulus switching technique which is widely used in several homomorphic encryption schemes as one of optimization options. It changes the ciphertext space to lower space by switching modulus, hence it makes answer size smaller in PIR protocol. Another factor which has an effect on communication cost is ciphertext polynomial degree. In fact, it directly affects not only the size of query and answer but also computation time.

As stated in Sect. 5, there is a complicated relation on multiple factors of both computation and communication cost. One change of the factors results in small or big trade-offs in many cPIR protocols. There are several reasons to increase the polynomial degree such as controlling error growth, handling larger database e.t.c. However, we show that we can keep ciphertext polynomial size lower dealing with larger database and having no noise problem.

A.1 Modulus Switching

We just set a new ciphertext modulus $\bar{p}$ such that $p < \bar{p} < q$. Then modulus switching takes original TRLWE ciphertext $\mathsf{ct} = (c_0, c_1)$ gives a new TRLWE ciphertext $\bar{\mathsf{ct}} = (\bar{c_0}, \bar{c_1})$, where $\left[\lceil \bar{c_0} = \frac{\bar{p}}{q} c_0 \rfloor\right] \bmod \bar{p}$, $\left[\lceil \bar{c_1} = \frac{\bar{p}}{q} c_1 \rfloor\right] \bmod \bar{p}$. This is almost free in implementation since it just shifts all the coefficients. Furthermore, it causes fairly small noise growth comparing to FV ciphertext [17] since all the modulus $p, \bar{p}, q$ are power of 2. As a result, we can reduce the communication cost without increasing the server's computation cost.

A.2 Smaller Polynomial Degree

As a ciphertext of query in $\mathsf{SHECS\text{-}PIR}$ has just bit length information which is usually much smaller than polynomial degree N, there is no need to keep the polynomial degree large. It may hardly happen that $\log \mathfrak{n} > N$, hence, we could keep the same modulus q and N. Larger N may contain large data element size in one ciphertext but decreases the efficiency of protocol having more computation and noise. Therefore, $\mathsf{SHECS\text{-}PIR}$ has a benefit on maintaining smaller query size not increasing other factors (no trade-offs), while more ciphertexts are required for a query in $\mathsf{SealPIR}$ as $\mathfrak{n}$ increases.

B Multi-query PIR

Our protocol with packed query naturally supports multi-query scenario where the same client wants to retrieve multiple elements from the same server or multiple indices are asked to a server for one answer. For the former, we can obtain single query size cost even for realistic large enough number of database and the answer size is linear on the number of indices. For the latter, the communication cost is as same as single query version. The reason is that we only use $\log \mathfrak{n}$ coefficients of polynomial to generate the single query ciphertext for fixed the number of data $\mathfrak{n}$ and the degree N. Then it is possible to have at most $\lfloor N/\log \mathfrak{n} \rfloor$ indices in one polynomial as a multi query without increasing query size. It just maintains the communication cost of single query.

There are some multi query protocols to improve CPU costs. SHECS-PIR gives a benefit on communication cost if it is applied to any computationally efficient technique (batch codes [26,32], probabilistic batch codes [5]) of multi query PIR protocol. Comparing to the previous work in [5], SealPIR requires each query ciphertext to be expanded to each dimension's query vector by expand algorithm for an index. Therefore, a query ciphertext cannot contain more information apart from the desired index using their way. It implies that a client has to encrypts b times which outputs b ciphertexts to request b items from a server's DB. However, unpacking query step in our protocol is only dependent on coefficient of polynomial. For example, to retrieve 64 items out of 2^{20}, SealPIR requires more than 64 query ciphertexts (using probabilistic batch codes, they require $b(= 1.5 \times 64)$ query). But our approach requires only one query ciphertext having b indices for fixed $N = 2048, \mathfrak{n} = 2^{20}$ having the same efficient computational cost. Furthermore, for the server's reply, only one ciphertext per query is given by server with SHECS-PIR, while F^{d-1} ciphertexts are required per query to answer for a server in SealPIR and usually $F \geq 4, d \geq 2$.

References

1. Aguilar Melchor, C., Barrier, J., Fousse, L., Killijian, M.O.: XPIR: private information retrieval for everyone. PoPETs **2016**(2), 155–174 (2016)
2. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. J. Math. Cryptol. **9**(3), 169–203 (2015). http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml
3. Ali, A., et al.: Communication-computation trade-offs in PIR. Cryptology ePrint Archive, Report 2019/1483 (2019). https://eprint.iacr.org/2019/1483
4. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - a new hope. In: Holz, T., Savage, S. (eds.) USENIX Security 2016, pp. 327–343. USENIX Association, August 2016
5. Angel, S., Chen, H., Laine, K., Setty, S.T.V.: PIR with compressed queries and amortized query processing. In: 2018 IEEE Symposium on Security and Privacy, pp. 962–979. IEEE Computer Society Press, May 2018
6. Bai, S., Galbraith, S.D.: Lattice decoding attacks on binary LWE. In: Susilo, W., Mu, Y. (eds.) ACISP 2014. LNCS, vol. 8544, pp. 322–337. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08344-5_21
7. Beimel, A., Ishai, Y., Kushilevitz, E., Raymond, J.F.: Breaking the $O(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval. In: 43rd FOCS, pp. 261–270. IEEE Computer Society Press, November 2002
8. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. In: Ostrovsky, R. (ed.) 52nd FOCS, pp. 97–106. IEEE Computer Society Press, October 2011
9. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 505–524. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_29
10. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 402–414. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48910-X_28

11. Chen, H., Laine, K., Player, R.: Simple encrypted arithmetic library - SEAL v2.2. Technical report (2017)
12. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 3–33. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53887-6_1
13. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10624, pp. 377–408. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_14
14. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. J. Cryptol. **33**(1), 34–91 (2020)
15. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/
16. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: 36th FOCS, pp. 41–50. IEEE Computer Society Press, October 1995
17. Costache, A., Laine, K., Player, R.: Homomorphic noise growth in practice: comparing BGV and FV. Cryptology ePrint Archive, Report 2019/493 (2019). https://eprint.iacr.org/2019/493
18. Dams, D., Lataille, J., Sanchez, R., Wade, J.: WIDESEAS: a lattice-based PIR scheme implemented in EncryptedQuery. Cryptology ePrint Archive, Report 2019/855 (2019). https://eprint.iacr.org/2019/855
19. Demmler, D., Herzberg, A., Schneider, T.: Raid-PIR: practical multi-server PIR. In: Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security, CCSW 2014, pp. 45–56. ACM, New York (2014)
20. Devet, C., Goldberg, I., Heninger, N.: Optimally robust private information retrieval. In: Kohno, T. (ed.) USENIX Security 2012, pp. 269–283. USENIX Association, August 2012
21. Dong, C., Chen, L.: A fast single server private information retrieval protocol with low communication cost. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8712, pp. 380–399. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11203-9_22
22. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012). http://eprint.iacr.org/2012/144
23. Gentry, C., Halevi, S.: Compressible FHE with applications to PIR. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019. LNCS, vol. 11892, pp. 438–464. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-36033-7_17
24. 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
25. Goldberg, I.: Improving the robustness of private information retrieval. In: 2007 IEEE Symposium on Security and Privacy, pp. 131–148. IEEE Computer Society Press, May 2007
26. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Batch codes and their applications. In: Babai, L. (ed.) 36th ACM STOC, pp. 262–271. ACM Press, June 2004
27. Kiayias, A., Leonardos, N., Lipmaa, H., Pavlyk, K., Tang, Q.: Optimal rate private information retrieval from homomorphic encryption. PoPETs **2015**(2), 222–243 (2015)

28. Kushilevitz, E., Ostrovsky, R.: Replication is NOT needed: SINGLE database, computationally-private information retrieval. In: 38th FOCS, pp. 364–373. IEEE Computer Society Press, October 1997
29. Laine, K., et al.: SealPIR: a computational PIR library that achieves low communication costs and high performance. https://github.com/microsoft/SealPIR
30. Lipmaa, H., Pavlyk, K.: A simpler rate-optimal CPIR protocol. In: Kiayias, A. (ed.) FC 2017. LNCS, vol. 10322, pp. 621–638. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70972-7_35
31. Olumofin, F.G., Goldberg, I.: Revisiting the computational practicality of private information retrieval. In: Danezis, G. (ed.) FC 2011. LNCS, vol. 7035, pp. 158–172. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27576-0_13
32. Paterson, M.B., Stinson, D.R., Wei, R.: Combinatorial batch codes. Adv. Math. Commun. **3**(1), 13–27 (2009)
33. Stern, J.P.: A new efficient all-or-nothing disclosure of secrets protocol. In: Ohta, K., Pei, D. (eds.) ASIACRYPT' 1998. LNCS, vol. 1514, pp. 357–371. Springer, Heidelberg (1998)