OnionPIR: Response Efficient Single-Server PIR

Muhammad Haris Mughees
University of Illinois at Urbana-Champaign
mughees2@illinois.edu

Hao Chen*
Facebook
haoche@fb.com

Ling Ren
University of Illinois at Urbana-Champaign
renling@illinois.edu

ABSTRACT

This paper presents OnionPIR and stateful OnionPIR, two single-server PIR schemes that significantly improve the response size and computation cost over state-of-the-art schemes. OnionPIR scheme utilizes recent advances in somewhat homomorphic encryption (SHE) and carefully composes two lattice-based SHE schemes and homomorphic operations to control the noise growth and response size. Stateful OnionPIR uses a technique based on the homomorphic evaluation of copy networks. OnionPIR achieves a response overhead of just 4.2x over the insecure baseline, in contrast to the 100x response overhead of state-of-the-art schemes. Our stateful OnionPIR scheme improves upon the recent stateful PIR framework of Patel et al. and drastically reduces its response overhead by avoiding downloading the entire database in the offline stage. Compared to stateless OnionPIR, Stateful OnionPIR reduces the computation cost by 1.8 ∼ 22x for different database sizes.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; **Management and querying of encrypted data**; *Symmetric cryptography and hash functions*; *Block and stream ciphers.*

KEYWORDS

Privacy; Private information retrieval; Homomorphic encryption

ACM Reference Format:
Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response Efficient Single-Server PIR . In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3460120.3485381

1 INTRODUCTION

Protecting user privacy is becoming a critical concern to cloud applications and service providers. *Private information retrieval* (PIR) is an important cryptographic primitive to protect user privacy when fetching data from the cloud. Informally, PIR allows a user to retrieve a particular entry from a public database without revealing the identity of the entry to the database server. Recently, PIR has been suggested for various applications, including anonymous communication [5, 51], privacy-preserving media streaming [40], ad delivery [39], location routing [32], contact discovery [11], password-checking [2], and safe-browsing [46].

At a high level, PIR schemes can be classified into *single-server* [2, 4, 14, 19, 33, 47, 49, 50, 53] ones and *multi-server* ones [7, 8, 23, 24, 30, 30, 36, 60, 61, 61]. Multi-server schemes are generally more efficient but they need the stronger trust assumption of multiple non-colluding servers. This requires coordination from multiple organizations makes them hard to deploy in practice. In this paper, we will focus on single-server PIR, which has thus far been quite inefficient. The central goal of this project is to substantially improve the efficiency of single-server PIR schemes and enable them for practical adoption.

In PIR, there are three main performance measures: the *request size*, the *response size*, and the server's *computation cost*. There are often trade-offs between these measures. For example, a trivial (singer-server) PIR scheme is to download the whole database. This trivial scheme involves no server computation and has almost zero request size, but it incurs a huge response size. State-of-the-art single-server PIR schemes have achieved a reasonably small request size. For example, SealPIR's request size can be made only 32 KB (after applying a standard optimization discussed in Section 4.3) for a database with up to four million entries [4]. But they are still very expensive in terms of response size and server computation, as we elaborate below.

- *Large response.* State-of-the-art single-server PIR schemes incur around 100x overhead in response size. That means to fetch a 30 KB entry (e.g., an ad), the client needs to download 3 MB of data.
- *Heavy computation.* State-of-the-art single-server PIR schemes require heavy computation on the server. For example, SealPIR requires about 400 seconds of server computation to fetch a 30 KB file from a database with one million entries.

This paper addresses the above two performance issues of single-server PIRs.

Main contribution 1. We first present a new single-server PIR scheme we call OnionPIR that significantly improves the response size. The main technique is to carefully control the noise growth from the ciphertext operations on the server with the help of recent advances in homomorphic encryption schemes. OnionPIR has a mere 4.2x response size overhead (over the insecure baseline), in contrast to the 100x overhead in state-of-the-art schemes like SealPIR. Concretely, to download a 30 KB entry, the client in OnionPIR only needs to download 128 KB of data.

OnionPIR maintains a similar computation cost as SealPIR. The small downside of OnionPIR is a slight increase in the client request size for small databases. Specifically, for a database with one million entries, the request size in OnionPIR is 64 KB, which is about twice as large as SealPIR's request size of 32 KB. However, we note that

*The work was partially done when Hao Chen was at Microsoft Research.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3485381

the request size for OnionPIR remains 64 KB even for all realistic database sizes in practice. In contrast, the request size for SealPIR starts to increase quickly once the database size exceeds four million: it becomes 64 KB for a database with 16 millions entries and around 512 KB for one billion entries.

Main contribution 2. Next, to address the computation issue, we improve the *Stateful* PIR paradigm of Patel et al. [54] and integrate it with OnionPIR. In stateful PIR, the client has a local state and uses that state to make PIR queries cheaper [26, 54]. The original stateful PIR scheme of Patel et al. can already improve computation but it requires the client to download the whole database in the offline phase, which drastically increases the amortized response size. We develop a new technique based on homomorphic evaluation of *copy networks* [28, 48] to reap the computation savings of stateful PIR while increasing the amortized response size by only a factor of two over stateless OnionPIR. The benefit of our stateful OnionPIR scales with the size of the database. For database sizes from 2^{16} to 2^{24}, the reduction in computation over stateless OnionPIR ranges from 1.8x to 22x. Similarly, compared to the original stateful PIR scheme of Patel et. al., the response size is reduced by $27 \sim 3,900$x. These reductions also translate into better monetary costs of stateful OnionPIR, which are $1.3 \sim 22$x less than stateless OnionPIR and $10 \sim 107$x less than Patel et al..

Concurrent works. Concurrent works of Park and Tibouchi [53] and Ali et al. [2] also study how to reduce response size in (stateless) single-server PIR. At a high level, they used different techniques from us and also achieved a substantial reduction in response size over SealPIR. But OnionPIR is more efficient than their scheme in all three metrics as we elaborate in Section 7.

2 BACKGROUND AND PRELIMINARY

2.1 Somewhat Homomorphic Encryption

State-of-the-art single-server PIR schemes rely on lattice-based *somewhat homomorphic encryption* (SHE). The security of lattice-based SHE is based on the hardness of Learning With Errors (LWE) or its variant on the polynomial ring (RLWE). We will use RLWE-based SHE schemes, in particular, BFV [31] and RGSW [22, 34].

As its name suggests, a SHE scheme supports a limited number of homomorphic addition and multiplication operations on the ciphertexts. All known constructions of SHE produce *noisy* ciphertexts. Homomorphic operations on the ciphertexts increase the noise level in the resulting ciphertext. After a certain number of operations, the noise in the ciphertext would become too large and the ciphertext could no longer be decrypted. It is important to note that ciphertext multiplications result in the noise to multiply and hence blow up rapidly. Thus, to keep the noise growth under control, existing PIR schemes have to introduce expensive techniques. This is a major source of their inefficiency that we will address in this work.

BFV encryption. The BFV SHE scheme is defined over a fixed polynomial ring $R = \mathbb{Z}/(x^n + 1)$. Here, n is the degree of the polynomial and is usually a power of two. In the BFV SHE scheme, the secret key s is a polynomial sampled from a distribution of "small" (e.g., with binary coefficients) polynomials in R. Let q and t denote the coefficient modulus for the ciphertext and plaintext, respectively. A plaintext message m is a polynomial in $R \mod t$. Each ciphertext consists of two polynomials in $R \mod q$, and is given as $(c_0, c_1) = (a, b + e + m)$ where a is sampled uniformly at random from $R \mod q$, $b = a \cdot s + e$, and e is a noise polynomial with coefficients sampled from a bounded *Gaussian* distribution. The message m is encoded in the most significant bits of the coefficients of the second polynomial. A ciphertext can be decrypted by computing $\mu = c_1 - c_0 \cdot s = e + m$. Since the message is encoded in the most significant bits and the noise e is small, rounding μ recovers m.

Ciphertext expansion factor. An efficiency metric critical to our purpose is the *ciphertext expansion factor*, which is denoted as F and defined as the ratio between the ciphertext size and the plaintext size. For BFV,

$$F = \frac{2 \log q}{\log t}$$

because the ciphertext is a pair of polynomials modulo q whereas the plaintext is a polynomial modulo t. The ciphertext expansion factor F directly affects the response size of the PIR protocol. One of the main tasks in this paper is to reduce F.

RGSW encryption. We will use a second SHE scheme called RGSW [20]. Given a base B and a parameter l, a RGSW scheme has a *gadget vector* defined as:

$$g^{(l \times 1)} = (B^{\log q / \log B - 1}, B^{\log q / \log B - 2}, \cdots, B^{\log q / \log B - l})$$

The base B and the gadget vector length l give a trade-off between efficiency and noise growth. The gadget vector then gives a gadget matrix as follows:

$$\mathbf{G} = \mathbf{I}_2 \vee g = \begin{bmatrix} g & 0 \\ 0 & g \end{bmatrix} \in R^{(2l \times 2)}.$$

A RGSW encryption of a plaintext polynomial $m \in R$ is

$$\mathbf{C} = \mathbf{Z} + m \cdot \mathbf{G}$$

where each row of $\mathbf{Z} \in R^{(2l \times 2)}$ is a BFV encryption of 0. Following the BFV decryption, $\mathbf{Z}$ satisfies that $\|\mathbf{Z} \cdot (-s, 1)\|_\infty$ is small. Note that the bottom half of the matrix $\mathbf{C}$ consists of l BFV ciphertexts encrypting base-B decompositions of the plaintext m.

2.2 Noise Growth and Computational Cost of Homomorphic Operations

As we have mentioned before, each homomorphic operation in SHE increases the noise in the output ciphertext. Different operations result in drastically different noise growth, and this will significantly impact our design decisions. These operations also have different computation costs, usually dominated by the number of polynomial multiplications required. We elaborate below on the approximate noise growth and computation costs of different operations and summarize them in Table 1. Let $\mathrm{Err}(\mathrm{ct})$ denote the variance of the noise term in a ciphertext ct.

BFV **ciphertext addition.** This operation adds two BFV ciphertexts c_1 and c_2, and outputs a BFV ciphertext encrypting the plaintext sum. The noise in the output ciphertext grows *additively*, i.e., the noise of the output is the sum of the noise terms from the two inputs. This operation does not involve polynomial multiplication and its cost is very small compared to the other operations below.

Operation	Cost	Noise Growth
BFV ciphertext addition	–	$O(\mathsf{Err}(\mathsf{ct}_1) + \mathsf{Err}(\mathsf{ct}_2))$
BFV ctxt-ptxt mult.	2	$O(\mathsf{Err}(\mathsf{ct}) \cdot \lvert m \rvert)$
BFV ciphertext mult.	$4 + 2l$	$O(t \cdot (\mathsf{Err}(\mathsf{ct}_1) + \mathsf{Err}(\mathsf{ct}_2)))$
External product	$2l$	$O(B \cdot \mathsf{Err}(C) + \mathsf{Err}(d))$

Table 1: Comparison of computational costs and noise growths of homomorphic operations. The computational cost of BFV ciphertext multiplication and external product depends on l. Typically, l is set to 5. The noise growth is multiplicative in BFV ciphertext and ctxt-ptxt multiplications. In contrast, the noise growth in the external product is additive, which allows the evaluation of deeper circuits.

BFV **ciphertext-plaintext multiplication.** This operations takes as input a plaintext polynomial m and a BFV ciphertext ct encrypting m'. The output is a BFV ciphertext encrypting the product $m \cdot m'$. The noise term is multiplied with the plaintext [31]. This operation requires two polynomial multiplications.

BFV **ciphertext multiplication.** This operations takes as input two BFV ciphertexts c_1 and c_2, and outputs a BFV ciphertext encrypting the plaintext product. This operation increases the noise by a factor of t (the plaintext modulus). This operation also requires an expensive relinearization step and its computation cost is about $4 + 2l$ polynomial multiplications, where l is usually the same as the decomposition factor l in RGSW. Note that this operation is expensive in terms of both noise growth and computation cost, and it is the main culprit for the inefficiency of existing PIR schemes. We will not use this operation and will not go into details about it.

External Product. The external product operation takes as input a BFV ciphertext $\mathbf{d}$ encrypting m_d, and a RGSW ciphertext $\mathbf{C}$ encrypting m_C, with respect to same secret key s. The output is a BFV ciphertext encrypting their plaintext product $m_C \cdot m_{\mathbf{d}}$.

It is not important to understand the details of the external product operation for the purpose of understanding our PIR schemes. But we give a brief description of the external product below for completeness. Readers can refer to [22] for more details. We first define a vector $\mathbf{v}$'s gadget decomposition, denoted as $G^{-1}(\mathbf{v}) \in R^{2l}$. Intuitively, the gadget decomposition of a vector has small coefficients and when multiplied by the matrix G, gives an approximation of the original vector. More precisely, $G^{-1}(\mathbf{v})$ has coefficients in $(-B/2, B/2]$ and the decomposition error $\left\|G^{-1}(\mathbf{v}) \cdot G - \mathbf{v}\right\|_\infty$ is upper bounded by $B^{\log q/\log B - l}$. Also, note that the result is a BFV ciphertext and we never need to decrypt RGSW ciphertexts in this paper, which is why we did not discuss RGSW decryption.

The noise after external product is bounded by $O(\left\|G^{-1}(d)\right\|_\infty \cdot \mathsf{Err}(C) + \lvert m_C \rvert \cdot \mathsf{Err}(d))$. In our PIR schemes, m_C will always be a single bit (i.e., either 0 or 1). Also note that $\left\|G^{-1}(d)\right\|_\infty$ is $B/2$. Thus, the resultant noise term is roughly $O(B \cdot \mathsf{Err}(C) + \mathsf{Err}(d))$.

It is important to note that external product operations increase noise *additively*. That is to say, if we perform a series of L external products, the final noise will be roughly L times larger. In sharp contrast, if we apply the previous two types of multiplication operations L times in a row, the final noise term will be exponential in L. This is why we will use external products in most steps of our OnionPIR schemes.

Inputs: The client inputs an index idx $\in [N]$ and the server inputs database DB of size N.

(1) **Pre-processing database** DB: The server database is encoded in an amenable format.
(2) **Query Generation**: For i from 1 to N, the client sets c_i, the i-th encrypted query ciphertext, to $\mathsf{Enc}(pk, 1)$ if $i =$ idx and $\mathsf{Enc}(pk, 0)$ if $i \neq$ idx. The client sends encrypted query vector c to the server.
(3) **Response Generation**: The server computes $r = \sum_{i=1}^{N} c_i \cdot \mathsf{DB}_i$ using homomorphic ciphertext-plaintext multiplications and ciphertext additions. The server then returns r to the client..
(4) **Output** The client decrypts r to get database entry corresponding to idx.

Figure 1: Basic single-server PIR protocol.

3 OVERVIEW AND LIMITATIONS OF CURRENT PROTOCOLS

The most basic single-server PIR scheme is given in Figure 1. The database is represented as an array of size N. To access an entry, the client generates a query vector of N ciphertexts. The ciphertext corresponding to the target entry encrypts 1 whereas all the other ciphertext encrypts 0. The server *homomorphically* computes a dot-product between the query vector and the plaintext database to generate a response. The client decrypts the response to get the desired entry in the database.

The above basic PIR has a request size linear in the database size. To reduce the request size, three techniques have been suggested by existing PIR schemes. The first technique is *hierarchical query*, which dates back to the earliest works on PIR [25, 59]. It represents a database as a multi-dimensional hypercube. To access a database entry, the client now sends d query vectors, each consisting of $\sqrt[d]{N}$ ciphertexts, where d is the number of dimensions. In all existing protocols, d is set to 2, and this reduces request size to $2\sqrt[2]{N}$ ciphertexts. Another method to reduce PIR request size is *query compression*, proposed recently by SealPIR [4]. Instead of encrypting one bit per ciphertext, the client packs many bits within one (BFV) ciphertext. The server can then unpack this ciphertext into a list of ciphertexts, each encrypting a single bit. The total number of bits packed in a single ciphertext is equal to the degree of ciphertext polynomial n. In SealPIR, n is set to 2048, so for a database with up to four million entries, the $2\sqrt[2]{N}$-sized query can be packed into two ciphertexts. The third method is to send only the second component of a fresh BFV ciphertext together with the seed used to generate the first component pseudorandomly [21]. This method reduces the size of the request in half. We will further discuss these optimizations in Section 4.3.

SealPIR is a state-of-the-art single-server PIR scheme. Combining the above techniques (SealPIR did not incorporate the third technique but could easily do), SealPIR would achieve a request size of only 32 KB for databases with up to four million entries. After that, the request size will increase proportionally to $\sqrt[2]{N}$.

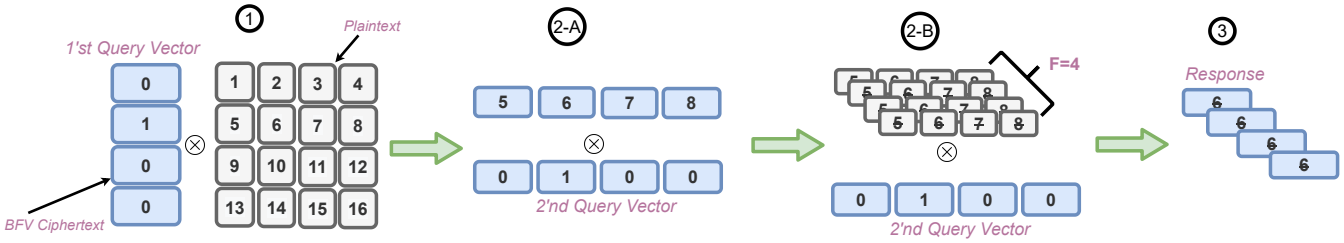

Figure 2: A basic hirarchical PIR in two dimension.

However, as mentioned earlier, existing single-server PIR schemes including SealPIR still suffer from large response size and high computation cost, which we explain in more detail below.

Why response size is large. The cause for the large response size lies in the way state-of-the-art schemes use homomorphic operations in hierarchical PIR. Figure 2 illustrates state-of-the-art schemes like XPIR [50] and SealPIR [50] for a database of size $n = 16$. The database is represented as a two-dimensional matrix of size 4×4 and the query consist of 2 vectors each consisting of 4 BFV ciphertexts (1). As shown in the figure, for the first dimension, the server performs a dot-product between each column of the plaintext database and the query vector. The output of this dot-product is a vector of ciphertexts corresponding to the matrix row containing the requested entry (2A). However, these ciphertexts are not directly used in the dot product with the second query vector. Instead, each of these ciphertexts is first *split* into F chunks where F is the *ciphertext expansion factor* as described in Section 2. (Recall that a ciphertext is F times larger than a plaintext.) Then, each chunk is treated as a plaintext in the dot-product with the second query vector (2B). The reason behind this ciphertext split design is to avoid homomorphic ciphertext multiplications that would have added very large noise to the resulting ciphertext as discussed in Section 2. Of course, the downside of this design is that now the response consists of F BFV ciphertexts (3). Although we used $F = 4$ as an example, the actual SealPIR implementation has $F = 10$. The overall response overhead would be F^2, which is around 100x.

Why computation cost is high. The computation cost is $O(N)$ since every entry in the database is involved in a homomorphic operation. In fact, one can argue that this is a fundamental barrier in the standard PIR model rather than a drawback of SealPIR or any particular scheme. If some entries are not involved in the computation, it would reveal to the server that these entries are not what the client is interested in, which violates the privacy guarantee of PIR. Thus, there seems to be little room for computation reduction in the standard PIR model. Looking ahead, we will incorporate the stateful PIR framework [54] to reduce computation.

4 RESPONSE EFFICIENT PIR

4.1 A Warm-up Protocol

We first present a warm-up protocol that drastically reduces the response size at the expense of higher computation. This basic protocol will serve as a stepping stone to introduce our main OnionPIR protocol, which will improve both response size and computation. We adopt the SHE and the hierarchical query framework. The top part of Figure 3 illustrates a sample execution of the warm-up protocol. Compared to prior works such as SealPIR and XPIR, we make three key changes.

Use of external products. The first change is that the client query vectors now consist of RGSW ciphertexts and the server uses *external product* (instead of ciphertext-plaintext multiplication) (1). Recall that SealPIR had to split the intermediate ciphertexts after the first multiplication because BFV ciphertext multiplications increase noise rapidly. In contrast, as mentioned in Section 2, the noise only grows *additively* after each *external product* operation. Therefore, we no longer have to split the intermediate ciphertexts and can use them directly for the multiplications in the second (2) and the third (3) dimensions. As a result, the response, which is the output of the third multiplication, is only a single BFV ciphertext (4), rather than F BFV ciphertext. In other words, the response overhead is now simply the ciphertext expansion factor F, down from F^2.

Parameterization for smaller ciphertext expansion factor F. The smaller noise growth from external product already gives an improvement in F over prior works: for a fixed ciphertext modulus q, if the noise growth is smaller, we can leave less room for noise growth and use a larger plaintext modulus t. But we can optimize the parameter choices of BFV to further reduce F. We will use a larger ciphertext modulus q. One can see from Table 1 that the noise growth does not depend on q. Thus, increasing q allows more room for a larger t, and hence decreases F. However, a larger q means a bigger ciphertext, which in turn implies a larger request size. Therefore, we use a moderately large q in our implementation; concretely, we use 124-bit q and it allows a 60-bit t.

Higher-dimensional cube. While prior works represent the database as a two-dimensional matrix, we can represent the database as a higher-dimensional cube, again thanks to the smaller noise growth of the external product. Using a higher dimension will help us further decrease the ciphertext expansion factor F for reasons that will become clear later in Section 4.3.

Limitation of the warm-up protocol. Unfortunately, the warm-up protocol suffers from higher computational costs. Recall from Section 2 that the computational cost of the external product is $2l$ polynomial multiplications. Thus, the total computation (measured

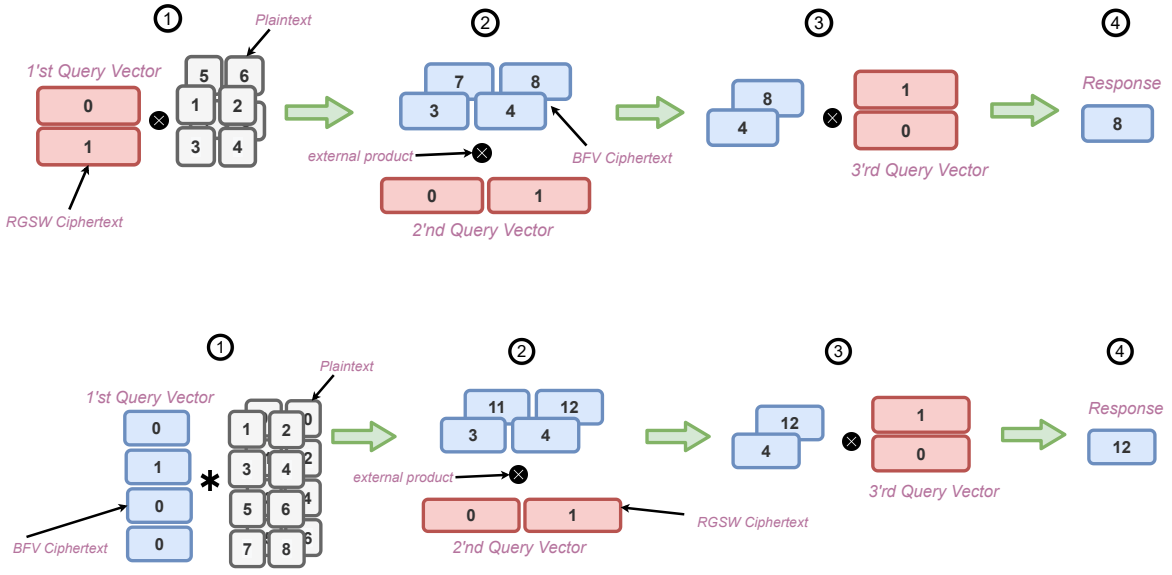

Figure 3: The warm-up protocol (top) and the improved protocol (bottom) using RGSW ciphertexts and external products. In the warm-up protocol, the client queries are RGSW ciphertexts for all dimensions and the output of each dimension are BFV ciphertexts. In the improved protocol, the first dimension's query vector consists of BFV ciphertexts and the remaining dimensions are RGSW ciphertexts; furthermore, the first dimension is slightly larger than the other dimensions so that the first dimension dominates the computation cost.

by the number of polynomial multiplications) of the warm-up protocol from the first dimension alone is $2lN$. In comparison, SealPIR's computation bottleneck lies in its first dimension, which involves N BFV ciphertext-plaintext multiplications. Each of these operations requires only 2 polynomial multiplications, giving a total computational cost of $2N$. Typically $l = 5$, so the computational cost of the warm-up protocol is at least 5x higher than SealPIR.

4.2 Optimizing the Computation

As mentioned, the warm-up protocol achieves a small noise growth and response size, at the expense of higher computation. Here is a simple method to improve the computation cost: revert to BFV ciphertext-plaintext multiplication in the first dimension and make the first dimension slightly larger than the remaining dimensions. Let N_i denote the size of i-th dimension. With some foresight, we will set the first dimension size to be $N_1 = 128$ and subsequent dimension sizes to $N_2 = N_3 = \ldots = 4$. This way, the total computation cost is once again dominated by the first dimension and will be comparable to the prior art.

But as mentioned in Section 2, BFV ciphertext-plaintext multiplication introduce large noise, on the order of the plaintext modulus t. This will force us to reduce t which hurts the ciphertext expansion factor. Therefore, we need a scheme that strikes a balance between noise growth and computational cost for the first dimension.

Inspired by the external product technique which reduces noise growth by decomposing a ciphertext into smaller parts, our solution is to *decompose* the plaintext before multiplying them with the encrypted query vector. A similar approach is proposed in [35].

(1) $\text{pt}' = \{\text{pt}_1, \text{pt}_2\} \leftarrow \text{DecompPlain}(\text{pt})$: Decompose input pt into two parts each of size $\log(t)/2$ bits.
(2) $\text{ct}' = \{\text{ct}_1, \text{ct}_2\} \leftarrow \text{DecompEncrypt}(b)$: Takes as input a query bit b, and output two BFV ciphertexts encrypting $\{b \cdot 2^{\log(t)/2}, b\}$.
(3) $\text{ct} \leftarrow \text{DecompMul}(\text{pt}', \text{ct}')$: Computes the dot-product between pt' and ct' and outputs a BFV ciphertext.

Figure 4: Decomposed ciphertext-plaintext product. This operation increases noise by about $\log(t)/2$ bits.

The details of this technique are given in Figure 4. We found that decomposing into two components gives us good enough noise growth with our parameter choices. In this case, each DecompMul operation adds about $\log(t)/2$ bits of noise. The server first uses the DecompPlain function to decompose each database entry. Similarly, the client encrypts the first query vector using DecompEncrypt for each bit. The server then performs the first dimension of dot product using DecompMul operations. All subsequent dimensions will use external products to control noise growth.

4.3 Query Compression

Sending one ciphertext per query bit results in a large request size. Previous works have proposed the query compression technique [4, 20] to reduce the query size. The high-level idea is that the client can pack many *independent* bits into a single ciphertext. The server then obliviously unpacks this ciphertext into encryptions of single

Algorithm 1: QueryPack algorithm used in OnionPIR

Input: $\{b_i\}_{i=1}^{d}$, a set of plaintext query vectors one for each dimension.

Output: $\tilde{c}$, a single BFV ciphertext packing all the query vectors.

Notation:

- d, number of dimensions.
- N_i, size of i-th dimension.
- f, number of components for each kind of encryption.
- $b_{i,j}$, j-th bit in i-th vector.

$ptr = 0$
Sets plaintext pt as follows:
for $i = 1 : d$ **do**
  for $j = 1 : N_i$ **do**
    for $k = 1 : f$ **do**
      ▷ $f = 2$ for first-dimension and $f = l$ for rest.
      $\text{pt}_{ptr} = b_{i,j}[k]$
      $ptr = ptr + 1$
    end
  end
end
BFV-encrypts pt to get $\tilde{c} = \text{BFV}(\text{pt})$ and outputs $\tilde{c}$.

bits. In OnionPIR we adopted the query compression algorithm given by Chen et al. [20].

Query packing. Algorithm 1 is the query packing algorithm in OnionPIR. All the query vectors are packed into a single plaintext pt, which is then encrypted using BFV encryption. Chen et al. [20] show that for RGSW encryption of a query bit b, it is sufficient to only pack l values, corresponding to first l rows of RGSW ciphertext. Similarly, for the first dimension, we pack 2 values for each bit in the first query vector. Later in the section, we show that in OnionPIR a single plaintext is sufficient to pack all the query vectors.

Query unpacking. Algorithm 2 is the query unpacking algorithm (also called query expansion) in OnionPIR. The algorithm first calls the BFV expansion procedure given in Algorithm 3 of [20]. This procedure outputs an array of BFV ciphertexts, encrypting individual values. The algorithm sets the first query vector directly from the output array. For the remaining query vectors, BFV expansion only gives the bottom l rows of each RGSW ciphertext. To get the top l rows for each RGSW ciphertext, the algorithm performs external products between the RGSW encryption of the client secret-key and the bottom l rows. We refer readers to Section 4.3 of [20] for further explanation on this trick.

As it turns out, query compression increases noise in the output ciphertext. The noise growth is multiplicative to the number of entries packed in one ciphertext. So, it is desirable to have fewer entries to pack. This is why we opt to represent the database as a high-dimensional hypercube (cf. Section 3) and set all dimensions small, i.e., $N_2 = N_3 = \ldots = 4$, after the first dimension of $N_1 = 128$ (cf. Section 4.2). This makes the number of dimensions d logarithmic in the database size, or concretely, $d = 1 + \lceil \log_4(N/128) \rceil$.

Algorithm 2: QueryUnpack algorithm adopted from algorithm 4 in [20]. We assume that secret-key encryption A is provided by the client at the time of initialization. This algorithm outputs a single encrypted query vector for each dimension.

Input: $(\tilde{c})$, a single BFV ciphertext packing all the query vectors; $A = \text{RGSW}(-s)$, RGSW encryption of the client's secret key.

Output: $(\text{C}_{\text{BFV}}, \{\text{C}_{\text{RGSW}}{}^{i}\}_{i=1}^{d-1})$, unpacked encrypted query vectors.

Notation:

- C_{BFV}, query vector of BFV ciphertexts for the first dimension.
- $\text{C}_{\text{RGSW}}{}^{i}$, query vector of RGSW ciphertexts for the i-th dimension.
- expandRlwe, BFV expansion procedure in Algorithm 3 of [20].
- Remaining as defined in Algorithm 1.

$\text{c} = \text{expandRlwe}(\tilde{c})$ ▷ a flat array of all expanded ciphertexts.
for $j = 0 : N_1 - 1$ **do**
  ▷ setting first query vector.
  $\text{C}_{\text{BFV}\,j}[0] = c[2j + 1]$
  $\text{C}_{\text{BFV}\,j}[1] = c[2j + 2]$
end
$ptr = 2N_1$
for $i = 1 : d - 1$ **do**
  ▷ setting higher query vectors.
  for $j = 0 : N_{i+1} - 1$ **do**
    for $k = 1 : l$ **do**
      $\text{C}_{\text{RGSW}}{}^{i}_{j}[k + l] = \text{c}[ptr + jl + k]$
      $\text{C}_{\text{RGSW}}{}^{i}_{j}[k] = \text{externalProduct}(A, \text{c}[ptr + jl + k])$
    end
  end
  $ptr = ptr + lN_i$
end
Outputs $(\text{C}_{\text{BFV}}, \{\text{C}_{\text{RGSW}}{}^{i}\}_{i=1}^{d-1})$.

We pack two values for each query bit, hence, in total 256 values for the first dimension (cf. Section 4.2). Likewise, for the remaining $d - 1$ dimensions combined, we have $4l(d - 1)$ values to pack. Concatenating them gives a plaintext vector of size $256 + 4l(d - 1)$. This means that for a database with one million entries and $l = 5$, a total of 386 values will be packed. In our implementation, each ciphertext has $n = 4096$ plaintext slots, so we pack all these plaintexts into a *single* BFV ciphertext. Unpacking these entries will add only a small amount of noise in the resulting ciphertexts. We remark that even for very large databases (with up to 2^{390} entries) we can still pack all the query vectors in a single BFV ciphertext.

Pseudorandom first component in ciphertexts. We can further reduce the request size by using a simple optimization from [21]. Recall that each BFV ciphertext consists of two components (c_0, c_1), and in a fresh ciphertext, the first component c_0 is sampled uniformly randomly from $R \mod q$. Thus, instead of sending a truly

Algorithm 3: OnionPIR Protocol.

Input:

- DB server database of size N.
- id, the index of the client's desired entry.

Notation:

- Notations used in Algorithm 2.
- N, database size.
- DB_i, i-th entry.
- DB', intermediate database.
- All the notations defined in Algorithm 1 and 2.
- shaded part is executed by server.

Server computes $\{\text{pt}_j\}_{j=1}^{N} = \{\text{DecompPlain}(\text{DB}_j)\}_{j=1}^{N}$

Client converts idx into a vector $(i_1, \cdots i_d)$, where i_j is the position of idx entry in j-th dimension of the hypercube.

Client generates query vectors $\{b_j\}_{j=1}^{d}$ corresponding to $(i_1, \cdots i_d)$, such that $b_j[i_j]$ is 1 and rest are 0.

Client computes $\tilde{c} = \text{QueryPack}(\{b_j\}_{j=1}^{d})$, and sends $\tilde{c}$ to the server.

Server computes: $(\text{C}_{\text{BFV}}, \{\text{C}_{\text{RGSW}}{}^{i}\}_{i=1}^{d-1}) = \text{QueryUnpack}(\tilde{c})$.

for $j = 0 : N/N_1 - 1$ **do**

  ▷ first dimension dot-product.

  $\text{DB}'_j = \sum_{k=1}^{N_1} \text{DecompMul}(\text{C}_{\text{BFV}k}, \text{pt}_{k+(j*N_1)})$

end

for $i = 2 : d$ **do**

  ▷ remaining dot-products.

  for $j = 0 : |\text{DB}'|/N_i - 1$ **do**

    $\widetilde{\text{DB}}_j = \sum_{k=1}^{N_i} \text{externalProduct}(\text{C}_{\text{RGSW}}{}_{k}^{i-1}, \text{DB}'_{k+(j*N_i)})$

  end

  $\text{DB}' = \widetilde{\text{DB}}$

end

Server sends $r = \text{DB}'$ (a single entry now) to the client.

Client decrypts r to get data of record id.

random c_0, the client can generate a pseudorandom c_0 from a short random seed and send the seed to the server. This optimization reduces the request size in half.

4.4 OnionPIR Full Protocol

The final OnionPIR protocol is given in Algorithm 3. We have introduced all the components of the algorithm separately in previous sections. Below we describe the protocol putting together all the techniques.

The database is represented as a hypercube of d dimensions. The size of the first dimension is $N_1 = 128$ and each of the remaining dimensions is of size 4. The total number of dimensions is thus $d = 1 + \lceil \log_4(N/N_1) \rceil$.

As a pre-processing step of the protocol, the server decomposes each entry of the database into two parts. The client represents the desired index idx into d query vectors, one for each dimension of the hypercube. The client then packs all of the query bits into a single BFV ciphertext and sends the ciphertext to the server using Algorithm 1. The server unpacks this ciphertext into separate encrypted query vectors using Algorithm 2. Each entry in the first encrypted query vector consists of two BFV ciphertext and each entry in subsequent encrypted query vectors is a RGSW ciphertext.

For the first dimension, the server performs a dot-product (using the DecompMul operation) between the first query vector and each (plaintext) column of the hypercube. The output is an *encrypted* hypercube of one fewer dimension. The server then continues to process higher dimensions in the same manner but now using external products. After the dot-product at each dimension, the output is an intermediate hypercube of one fewer dimension and it is used as the input to the next dot-product. The final output after the last dot-product is a single BFV ciphertext encrypting the desired entry. This is sent back to the client as the response and the client decrypts it to get the desired database entry.

Request size. The request of OnionPIR is a single BFV ciphertext. Using the pseudorandom seed optimization discussed in Section 4.3, the request size is 64 KB.

Response size. We set the ciphertext modulus q to 124 bits (padded to 128 bits in the implementation). The plaintext modulus t is set to 60 bits. This gives a ciphertext expansion factor $F \approx 4.2$. The response is thus only 4.2x larger than the plaintext entry.

Computational cost. Query unpacking requires around $w \cdot l^2$ polynomial multiplications where w is the total number of packed bits [20]. Because of the high dimensions, only a logarithmic number of bits are packed. Therefore, query unpacking is not the computation bottleneck.

The total number of polynomial multiplications required by the dot product operations is about $2 \cdot N + 4 \cdot l \cdot (\frac{N}{N_1} + \frac{N}{4N_1} + \frac{N}{16N_1} + \cdots)$. Recall that $N_1 = 128$ is the size of the first dimension. Thus, the term N/N_1 is very small, and the computational cost is dominated by the $2N$ polynomial multiplications in the first dimension.

Due to the larger t and the larger polynomial degree $n = 4096$, each ciphertext in our protocol contains 30 KB of plaintext data. This is 10 times more than SealPIR. On the other hand, the first dimension in our protocol uses decomposition and is hence twice as expensive as SealPIR. Furthermore, each polynomial multiplication in our protocol is about 4x more expensive because of our doubled values of $\log q$ and n. Therefore, in theory, the computation cost of our protocol will be about 1.25x better than SealPIR. In our actual implementation and experiments, we found that the computational costs of OnionPIR and SealPIR are almost identical.

Noise growth estimate. In OnionPIR, the noise in the output ciphertext largely results from the query unpacking and the ciphertext-plaintext multiplications in the first dimension.

The noise in the unpacked ciphertext (RGSW and BFV both) is bounded by [20]:

$$\text{Err}(\text{ct}_{exp}) \leq O(w^2) \cdot \text{Err}(\text{BFV})$$

Here, Err(BFV) is the initial noise in the packed input ciphertext and w is the number of packed bits. Since fewer bits need to be

packed in OnionPIR, query expansion adds less noise than prior art.

In the dot-product of the first dimension, the noise increases by a factor of $O(N_1 B')$. Here, $N_1 = 128$ is the size of the first dimension and it appears due to the homomorphic additions; B' is the maximum value of the decomposed plaintext. Therefore, the estimated total noise after the first dimension is around:

$$\mathsf{Err}(\mathsf{ct}_1) = O(w^2 N_1 B') \cdot \mathsf{Err}(\mathsf{BFV})$$

Subsequent dimensions use external products and the noise increase is *additive* and insignificant.

From the above analysis, the total noise in the response ciphertext is bounded by:

$$\mathsf{Err}(\mathsf{ct}_{resp.}) \leq \mathsf{Err}(\mathsf{ct}_1) + O(d) \cdot \mathsf{Err}(\mathsf{ct}_{exp})$$

As a comparison, we remark that had we used BFV ciphertext multiplications instead of external products, the noise in the output ciphertext would have grown exponentially to $\mathsf{Err}(\mathsf{ct}_{resp.}) \leq O(t^d \cdot N) \cdot \mathsf{Err}(\mathsf{BFV})$. This noise grows too fast with the number of dimensions d, which is why prior works were limited to $d = 2$.

5 STATEFUL PIR

Although OnionPIR has very small response size and request size, the *computational* burden on the server is still quite large (about the same as the prior art SealPIR). Note that the server has to perform at least one *ciphertext-plaintext* multiplication per database entry, resulting in $O(N)$ computation cost on the server. This is a somewhat fundamental barrier in computation in the standard PIR model: if some entries are not involved in the computation, it would reveal to the server that these entries are not what the client is interested in, which violates the privacy guarantee of PIR.

To overcome this computation bottleneck, Patel et. al [54] proposed an elegant framework called *Private Stateful Information Retrieval* (PSIR). PSIR significantly outperforms prior best single-server PIR schemes in terms of computation. The main idea of the PSIR framework is that the client is often *stateful* and can store some helper data retrieved in an offline phase. Then, in the online phase, the client uses its state (helper data) to make cheaper PIR queries.

The challenge is how to retrieve the required state privately in the offline phase. The approach recommended by Patel et al. is to simply download the entire database, which is clearly impractical for many applications.

To address the above limitation and make the PSIR framework practical, we propose a technique that allows the client to efficiently and privately retrieve the required state. We further integrated OnionPIR with our proposed offline technique into a stateful PIR framework. The resulting scheme achieves about 1.3 $\sim$ 22x reduction in computation cost over stateless OnionPIR for different database sizes. Compared to the Patel et al. scheme, our stateful scheme reduces the amortized response size by 27 $\sim$ 3900x at the expense of a slight increase in request size and a moderate increase in the computation.

In the remainder of this section, we will first provide a high-level overview of the PSIR framework of Patel et al. and then present our improved offline phase.

5.1 Private Stateful Information Retrieval

At a high level, the PSIR protocol by Patel et al. [54] has an offline phase and an online phase:

Offline phase. In the offline phase, the client privately retrieves some states from the server. This step is defined as *Private batched sum retrieval* (PBSR) problem in [54], which is defined as follows: Given c subsets $S_1, \cdots, S_c$ where each subset consists of k' random indices, privately fetch the sum of all the entries in each subset. The privacy of PBSR requires that the server does not learn anything about the c subsets $S_1, \cdots, S_c$.

StreamPBSR. To perform PBSR, the main protocol of Patel et al. ultimately decides that the server simply streams the *entire* database to the client. For many applications, streaming the entire database to the client is impractical. For example, for private video streaming application with database sizes in terabytes downloading the entire database for millions of users is essentially impractical.

BatchCodePBSR. In Appendix E.3 of Patel et al. [54], the authors also sketch a construction based on batch codes and homomorphic encryption. In this construction, the database is encoded using batch codes and the client runs a batched PIR protocol given in [4] to privately retrieve the subset sums. Although this construction avoids streaming the entire database, the authors found that the computational overhead of this construction is so high that it nullifies any computation improvement of stateful PIR over stateless PIR.

Online phase. In the online phase, the client uses the subset sums she obtained from the server to retrieve the entries. In this paper, we will not modify the online phase of Patel et al., so it is not important to understand its details. But we still briefly describe it below for completeness.

Suppose in the online phase the client wants to retrieve an entry i, the client will find an *unused* subset in the local storage that does not contain i. Let that subset and its corresponding sum be denoted as S' and s'. After that, the client generates a random ordered partition of the database such that (i) there are $m = N/(k'+1)$ partitions $P_1, P_2, \cdots, P_m$, each of size $k'+1$ and (ii) one partition P_r is equal to $S' \cup i$, where r is picked uniformly randomly from $[m]$. The client then sends a *succinct* description of the partition to the server. The server then represents the database in the form of a matrix where each row contains entries corresponding to a partition and add up each row. The client then performs a stateless PIR to retrieve the r-th sum. The client can now recover the i-th entry by subtracting s' from it. Once the client runs out of subset sums to use, it will perform the offline phase again. The privacy of the protocol is based on the privacy of the offline PBSR and the online PIR.

Observe that in the online phase, the client's PIR query is evaluated on a database of size N/m where $m = k'+1$. This results in a factor of $m = k'+1$ reduction in server computation. The online phase is hence quite efficient. In the next subsection, we will provide an efficient construction for the offline PBSR phase.

5.2 Efficient Private Batch Sum Retrieval

In this subsection, we introduce a novel PBSR construction. Although we motivated our construction for stateful PIR, it can be of

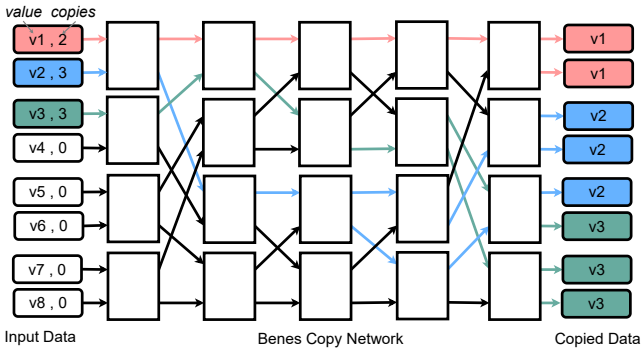

Figure 5: Beneš copy network. Each intermediate node is a a switch that either pass through or swap the incoming packets or replicate one of them on the output links. In this example, first element is copied twice while element two and three are each copied three times.

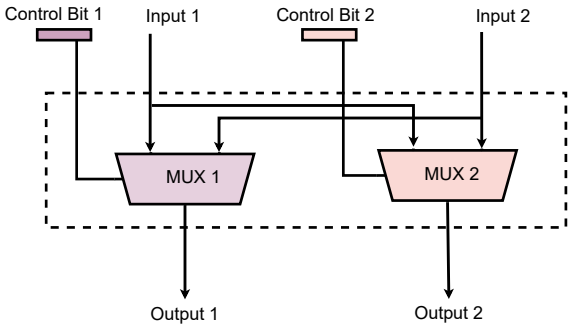

Figure 6: A 2×2 switch using two mux gates. The control bits decide if the switch will; pass through or swap inputs, or replicate one of the inputs on both outputs.

independent interest. Our key observation is that the PBSR problem has a similar interface to *copy networks*. We will also use a variant of PIR called *batched PIR*.

Batched PIR. Batched PIR allows the server to answer a batch of PIR queries at a lower cost than answering each query independently. Angel et al. gives an efficient framework to transform any single-query PIR to a batched PIR. In their scheme, to retrieve a batch of queries the total server computation is 3x of the single-query PIR. Therefore per query computation cost is significantly smaller. We provide a brief overview of the scheme below and refer readers to [4] for details.

Batched PIR can be defined using three functions. In the setup stage, the server calls BatchPIR.Setup, which encodes the database into b buckets randomly where $b = 1.5r$ and r is the number of entires the client wishes to retrieve. Each bucket is treated as a smaller independent database on which the client can perform PIR queries. The client who wishes to retrieve entries at indexes $I = \{i_1, \cdots, i_r\}$ from the server can locally call BatchPIR.QueryGen which provides encrypted PIR queries such that all the desired entries are retrieved. BatchPIR.QueryGen guaranteed that no bucket is queried more than once. The server then calls BatchPIR.Response, which uses each of these queries to run a separate PIR on the corresponding buckets. This results in b PIR responses that the server can forward to the client. The client decrypts the responses to retrieve the desired entries. In [4], the authors used SealPIR as the underlying PIR scheme. Naturally, we will plug in our OnionPIR to get a more efficient batched PIR.

Copy Networks. A copy network is a computer network that can replicate input packets from various sources simultaneously. More concretely, a copy network can be configured on the fly to copy each input value for a desired number of times to the destinations.

For our PBSR construction, we are going to use the Beneš copy network. It is a $N \times N$ interconnection network with $2 \log N - 1$ stages. Each stage contains $N/2$ nodes where each node is a 2×2 switch. Each switch can be configured to pass through or swap the incoming packets, or replicate one of the incoming packets. The only restriction of Beneš copy network is that the total number of desired copies should not exceed the number of destinations. Figure 5 depicts an example of a five-stage Beneš copy network. The network replicates the first input packet twice and the second and third input packets three times each.

Deng et. al. [28] provides an efficient algorithm to find the configuration of the Beneš copy network so that all the copy requests are satisfied. Specifically, their configuration algorithm takes as input a set of indexes and the desired number of corresponding copies and outputs configurations of all the network switches. We refer interested readers to [28] for further details on the Beneš copy networks and their configuration algorithm.

Note that the network structure of the Beneš copy network is independent of the input values or the copy requests. Therefore as long as the switching logic is evaluated homomorphically, the server does not learn any information. Observe that each switch in the Beneš copy network could be configured in one of the four configurations. In Figure 6 we show that such a switch can be constructed using two mux gates. Chillotti et al. [22] use external product to construct a homomorphic mux gate. Specifically, each input is a BFV ciphertext and the control bit is a RGSW ciphertext.

Therefore, to homomorphically evaluate the copy network, the client can send two encrypted control bits (RGSW ciphertexts) for each switch and the server will use these bits to homomorphically evaluate each switch. Once again, the client will use the query compression technique to pack these encrypted bits into as few BFV ciphertexts as possible. Each input to the copy network passes through $2 \log N - 1$ switches, so the noise in the output ciphertext only increases logarithmically in N.

PBSR construction. Putting batched PIR and homomorphic copy network together, our final PBSR protocol is presented below.

(1) The client picks c random subsets of size k', such that $ck' \leq N$.
(2) The client union these c subsets into a set I and pad I to have size ck' by adding dummy indices.
(3) The client and server run Batched PIR with I as the client input and the database DB as the server input.
One difference is that the batched PIR response will be kept at the server side (denoted as an array A of size $b = 1.5ck'$) instead of being sent back to the client.
(4) For each index in I, the client counts the number of subsets that include the index. The client inputs the indices and their

	StreamPBSR	BatchCodePBSR	Our PBSR
Response	$O(N)$	$O(c)$	$O(c)$
Request	–	$O(c^2k' + N)$	$O(ck' \log(ck') + N)$
Computation	–	$O(c^2k' + N)$	$O(ck' \log(ck') + N)$

Table 2: Comparison of response, request and computation of our proposed PBSR scheme with StreamPBSR and BatchCodePBSR. Our PBSR has significantly smaller response than StreamPBSR and much better request size and computation than BatchCodePBSR (since $c \gg \log(ck')$).

counts to the copy network routing algorithm to obtain the configurations of all the switches.

(5) The client and the server then homomorphically evaluate the copy network on the encrypted set A.
At the end of this step, the server holds $b = 1.5ck'$ encrypted entries containing the desired number of copies for each element in A.

(6) The client and the server homomorphically permute the output using a permutation network [20].

(7) The client asks the server to homomorphically add these copies into c subset sums and return the results. If an entry is included in multiple subsets, the client will pick a different copy for each subset sum.

Note that after step (5), the server knows that the adjacent entries are likely copies of the same database entry. This may leak information to the server about how subsets overlap. This is why we need the permutation step. After the permutation step, the server has no information about the subsets and whether/how they overlap.

Comparison. Table 2 compares the asymptotic complexity of our proposed PBSR with the two PBSR schemes given by Patel et al. [54]. Our construction has a significantly smaller response size than StreamPBSR. The response size of BatchCodePBSR is similar to our construction, but its request size and server computation are quadratic in the number of subsets c; in our construction, they are quasi-linear in c.

6 IMPLEMENTATION AND EVALUATIONS

6.1 Implementation Details

We implemented OnionPIR atop the SEAL Homomorphic Encryption Library[1]. SEAL only provides a BFV encryption scheme. So we implemented RGSW and external products in SEAL. We also implemented the CRT representation of RGSW encryption, which is more efficient than using multi-precision arithmetic operations.

Optimizing polynomial multiplications. In SEAL the polynomial multiplications are performed using number-theoretic transformation (NTT). Each NTT operation has a complexity of $n \log n$, where n is the size of the polynomial. However, we notice that the NTT implementation in SEAL is quite slow, which hurts the computation time of our protocol. Thus, for NTT, we instead use NFLlib [55], an efficient library that uses several arithmetic optimizations and AVX2 specialization for arithmetic operations over polynomials. The NTT implementation in NFLlib is 2 − 3x faster than SEAL. We integrated NFLlib's NTT into SEAL. In total, our modifications consist of around 3000 lines of C++ code.

[1]We have used Version 3.5.1 of SEAL library.

6.2 Experimental Setup

We run our experiments on Amazon EC2 instances. Specifically, we used a t2.2xlarge instance with 32 GB ram and 8 CPU cores with AVX enabled. We have not implemented batched PIR, so the offline phase computation cost is simulated. All the other results are obtained by running each experiment 10 times and taking the average. We also report server monetary cost, which is the sum of CPU cost for server computation and the server-side cost of network traffic. These costs were computed using standard rates from Amazon EC2 Instance prices [3], which at the time of writing are one cent per CPU-hour and nine cents per GB of Internet traffic.

Parameters. We set the polynomial degree n to 4096 and the size of coefficient modulus q to 124 bits. We use SEAL's default values for standard deviation error and secret key distribution. The LWE estimator by Albrecht et al. [1] suggests these parameters yield about 111 bits of computational security. Due to the lower noise growth, we can set plaintext modulus t to 60 bits. This gives a ciphertext expansion factor $F = 4.2$x. SealPIR, in comparison, sets n to 2048 and q to 60 bits, which provides 115 bits of security. They set t to 12 bits, which gives a ciphertext expansion factor of $F = 10$.

In our experiments, we set each database entry to be 30 KB. With $n = 4096$ and 60-bit t, 30 KB of plaintext data can fit in a single ciphertext. In SealPIR, each ciphertext could accommodate only 3 KB of plaintext data, so each database entry is split into 10 chunks. We will evaluate OnionPIR and stateful OnionPIR with database sizes ranging from 2^{16} to 2^{24}.

For stateful OnionPIR, recall that c is the number of subset sums the client retrieves in each offline phase (also the size of the client state); k' is the size of each subset. The larger k' saves more computation in the online phase but for stateful OnionPIR it increases the computation in the offline phase. We set c to 500 for all the database sizes. For Patel et al., we set $k' = \sqrt{N}/2$ following their original paper. For stateful OnionPIR, we picked k' that gives us the best results. Specifically, for databases of size 2^{16}, 2^{18}, 2^{20}, 2^{24} the value of k' is set to 8, 16, 16, 64, respectively.

6.3 Evaluation Results of OnionPIR

We evaluate OnionPIR with different database sizes, report the computational cost, request size, and response size and compare with SealPIR in Table 3.

Computational. In OnionPIR and SealPIR, the server mainly performs two tasks: query unpacking and dot-products between the query vectors and the (intermediate) database. Query unpacking in OnionPIR takes much less time than SealPIR because we pack only a logarithmic number of query bits (q.v. Section 4.3), while in SealPIR $2\sqrt[2]{N}$ bits are packed in query ciphertexts. Overall, the dot-products account for most of the server computation. The computational cost of OnionPIR is almost identical to SealPIR across all database sizes.

For both SealPIR and OnionPIR, the computation time increases linearly with the database size. This results in a quite high computation time for large databases. For example, for a database with 16 million entries, the server computation time is around 1.7 hours.

	SealPIR				OnionPIR			
	$N = 2^{16}$	$N = 2^{18}$	$N = 2^{20}$	$N = 2^{24}$	$N = 2^{16}$	$N = 2^{18}$	$N = 2^{20}$	$N = 2^{24}$
Response size (KB)	3,200	3,200	3,200	3,200	128	128	128	128
Request size (KB)	32	32	32	64	64	64	64	64
Query Unpack (sec)	5.4	10.7	21.5	86.3	3.6	4.1	4.6	5.5
Dot-Products (sec)	20.7	91.2	381.6	6,362.1	21.3	97.0	396.3	6,410.7
Total Computation (sec)	26.1	101.9	403.1	6,448.4	24.9	101.1	400.9	6,416.2
Server cost (US cents)	0.034	0.055	0.139	1.818	0.008	0.029	0.112	1.792

Table 3: Performance comparison of OnionPIR and SealPIR for different database sizes. *Red* boxes represent worse efficiency and blue boxes represent better efficiency. OnionPIR has significantly smaller response size and computation comparable SealPIR. Regarding request size, for database until one million entries request size in SealPIR is half of OnionPIR. But for database with 16 million entries request size of OnionPIR and SealPIR is equal.

		Stateful OnionPIR				Patel et al. Scheme			
		$N = 2^{16}$	$N = 2^{18}$	$N = 2^{20}$	$N = 2^{24}$	$N = 2^{16}$	$N = 2^{18}$	$N = 2^{20}$	$N = 2^{24}$
Online	Response size (KB)	128	128	128	128	3,200	3,200	3,200	3,200
	Request size (KB)	64.1	64.2	64.2	64.5	34	36	40	64
	Computation (sec)	3.1	6.3	25.1	200.5	0.1	0.4	0.8	3.1
Offline	Response size (KB)	128	128	128	128	3,932	15,728	62,914	1,006,632
	Request size (KB)	11.0	23.1	24.6	66.5	–	–	–	–
	Computation (sec)	10.6	23.2	25.0	87.1	–	–	–	–
Total (amortized)	Response size (KB)	256	256	256	256	7,132	18,928	66,114	1,009,832
	Request size (KB)	75.1	87.3	88.8	131.0	34	36	40	64
	Computation (sec)	13.7	29.5	50.1	287.6	0.1	0.4	0.8	3.1
	Server Cost (US cents)	0.006	0.010	0.016	0.081	0.061	0.162	0.567	8.668

Table 4: Comparison of stateful OnionPIR with Patel et al. scheme for various database sizes. Each database entry is 30 KB. The number of entries in hint $c = 500$. In all cases, the client storage is only 14.6 MB. For Patel et al. scheme, subset size k' is set to $\sqrt{N}/2$ for all databases and for stateful OnionPIR, subset size ranges between $8 \sim 64$ for different database sizes.

Request Size. For databases with up to four million entries, the request size in OnionPIR is twice as large as SealPIR. This is because each ciphertext in OnionPIR is four times bigger than the SealPIR ciphertext. But for larger databases, the request size of OnionPIR will remain 64 KB while the request size of SealPIR will start to increase and eventually exceed OnionPIR. For example, for a database with one billion entries (not shown in the table) the request size of SealPIR will be 512 KB.

Response Size. OnionPIR shines in response size. Specifically, the response size is only 128 KB where the response size in SealPIR is 3,200 KB.

Server monetary Cost. The server monetary cost heavily depends on the database size. For smaller databases, the server monetary cost is dominated by network traffic, and OnionPIR is orders of magnitude cheaper than SealPIR. But for bigger databases, computation becomes the dominating factor in the server monetary cost, and the two schemes become almost equal. As an example, for a database with 65,536 entries, the server cost of OnionPIR is four times less than SealPIR; but for a database with one million entries, OnionPIR's server cost is just 19% lower.

6.4 Evaluation Results of Stateful OnionPIR

In Table 4, we compare the performance of stateful OnionPIR with the Patel et al. scheme using StreamPBSR [54] in the offline phase and SealPIR in the online phase.

Comparison with OnionPIR. Stateful OnionPIR significantly reduces the computation cost over stateless OnionPIR. The reduction in computational time scales with the database size, ranging

from 1.8x to 22x in our experiments. The trade-offs are request size and response size. Specifically, response size is doubled and the request size increased by 1.1 ∼ 2x for different database sizes. In terms of monetary cost, stateful OnionPIR is 1.3 ∼ 22x cheaper over stateless OnionPIR.

Comparison with Patel et al.. The Patel et al. scheme has quite a large amortized response size due to downloading the entire database in the offline phase. With our proposed PBSR scheme, the amortized response size in stateful OnionPIR is significantly reduced. For all the databases in Table 4, the amortized response size of stateful OnionPIR is only 256 KB, which is a reduction of $27 \sim 3,900$x compared to Patel et al.. A trade-off here is that Patel et al. have very small computation. This is because their offline phase does not require any computation and we picked a much bigger subset size k' for their scheme, which significantly reduces their online computation. Despite the better computation, their significantly larger response size results in a higher monetary cost. Overall, the stateful OnionPIR has around 10 ∼ 107x cheaper monetary cost than Patel et al..

7 RELATED WORK

Early single-server PIR schemes. Some of the early single-server PIR protocols are based on *additively homomorphic encryption* (AHE). These schemes followed the blueprint of Kushilevitz and Ostrovsky [47]: the database is represented as a high-dimensional hypercube and the client's request is encrypted under an AHE. The original protocol of the Kushilevitz and Ostrovsky scheme has a request size of $O(\sqrt{N}\log N)$ and a response size of $O(\sqrt{N})$. Cachin et al. [14] proposed a PIR protocol based on the ϕ-Hiding assumption with a request size of $O(\log^4 N)$ and a response size of $O(\log^d N)$. Gentry and Ramazan [33] generalized Cachin et al.'s approach and proposed a communication-efficient PIR protocol with a request size of $O(\log^{3-o(1)} N)$. Chang [19] follows the Kushilevitz-Ostrovsky scheme but uses Pailer homomorphic encryption to construct PIR with $O(\sqrt{N}\log N)$ request size and $O(\log N)$ response size. Lipmaa [49] generalizes it to the Damgard-Jurik encryption [27] to achieve $O(\log^2 N)$ request size and $O(\log N)$ response size.

Unfortunately, Sion and Carbunar [57] observe that these schemes in practice often perform slower than downloading the entire database when the network bandwidth is just a few hundred Kbps. The poor performance is because, in all of these schemes, the server needs to perform at least N big-integer modulus multiplications or modulus exponentiations. The computation cost of these operations is often higher than simply sending the data to the client.

Recent practical single-server PIR schemes. Recent single-server PIR constructions are based on lattice-based cryptography, and in particular, Ring Learning with error (RLWE) encryption. Aguilar-Melchor et al. [50] present XPIR. To retrieve a 30 KB entry from a database with one million entries, their protocol takes around 383 seconds of server computation, which is slightly less than OnionPIR. However, the downside of their protocol is that the request size is 17 MB and the response size overhead is $100x$. SealPIR [4] addresses the request size bottleneck by introducing the query compression technique. This results in a significant reduction in request size (to 32 KB) at a cost of a slight increase in overall computation. But the response size is still $100x$, similar to XPIR.

Concurrent works. Very recently, Park and Tibouchi [53] present a construction based on external products that improve the response overhead to 16x; but their computation cost more than doubled compared to SealPIR. Ali et al. [2] also gives a protocol that improves upon SealPIR's response size. Their main technique is to use BFV ciphertext multiplication in the second dimension followed by modulus switching to reduce the response size. To handle the higher noise growth from BFV ciphertext multiplication, their protocol requires larger FHE parameters, which increases server computation cost. Overall, our OnionPIR performs better than their scheme in all the metrics. Concretely, to retrieve 60 KB entry[2] from a database with one million entries requires around 900 seconds of server computation, 357 KB response size, and 119 KB request size. In comparison, for the same setting, OnionPIR requires 800 seconds of computation, 256 KB response size, and 64 KB request size.

Multi-server PIR. While the focus of our paper is single-server PIR, we mention that there also exist many PIR protocols based on multiple non-colluding servers [6–8, 23, 24, 30, 36, 60, 61]. The first multi-server PIR schemes are proposed by Chor *et al.* [24] and they provide information-theoretic security. At a high level, the client sends XOR-based secret shares of the query to each server and the server performs plaintext XOR operations. The request size is $O(\sqrt{N})$ with two servers. Protocols with better request sizes are known using three or more non-colluding servers. The best existing three-server schemes have a request size of $2^{O(\sqrt{\log N \log\log N})}$ [30, 61]. Gilboa et al. [36] proposed a two-server computationally secure PIR scheme with a poly-logarithmic request size based on *distributed point functions*. The server computation consists of $O(N)$ PRG evaluations and XOR operations. Overall, these multi-server schemes have superior computational efficiency than single-server schemes because their server computation does not involve costly public-key operations.

Stateful PIR. Patel et al. [54] introduced stateful PIR where the client retrieves some helper data in the offline phase and uses them to make the online phase cheaper. The construction of Patel et al. uses a single server. The amortized computation cost of their framework is still linear in the database size, but most of the operations involve only symmetric-key cryptography. The number of public-key operations dropped to sub-linear, which leads to a substantial reduction in amortized computation cost over stateless PIR. However, their scheme requires the client to download the entire database in the offline phase. For applications with large database sizes downloading the entire database is not practical.

In recent pioneering work, Corrigan-Gibbs and Kogan have proposed two-server stateful PIR schemes with amortized sublinear computation complexity [26, 46]. This two-server PIR scheme shows promising efficiency in both theory and practice. Corrigan-Gibbs and Kogan also proposed a single-server variant of their stateful PIR utilizing FHE. This single-server variant, however, is much less efficient. Specifically, the single-server variant needs to

[2] Ali et al.'s scheme work best when the entry size is a multiple of 20 KB while OnionPIR works best when the record size is multiple of 30 KB. This is why we chose 60 KB for a fair comparison.

run the offline phase again after every single online query. Therefore, it only reduces the online cost while the overall cost is actually much worse than stateless PIR.

Orthogonal directions to improve PIR computation. We mention two orthogonal directions to reduce server computation in PIR. One direction is batched PIR. This general strategy has been adopted in a setting where the queries comes from a single client [4, 42, 43] or multiple clients [9, 44]. OurOnionPIR can be extended to support batched queries and we have used it in our PBSR construction. But we remark that batched PIR is not always applicable because, in many scenarios, the client has only one query to make at a time.

Another direction is PIR with preprocessing, first proposed by Beimel et al. [9]. In their scheme, the server first performs a linear preprocessing step; after that, the server's work per query is sub-linear. Their protocol requires multiple non-colluding servers. Recently, Canetti et al. [15] and Boyle et al. [13] constructed single-server PIR with preprocessing, which is also called doubly efficient PIR. These schemes have been proposed in both symmetric-key and public-key settings. In the symmetric-key variant, the database can only be accessed by a single client, which does not fully match the public database model of PIR. In other words, this would require the server to store a separate copy of the prepossessed database for each client. On the other hand, the current public key variant requires strong cryptographic assumptions such as obfuscation, which makes them impractical at the moment.

Related privacy-preserving primitives. Oblivious RAM (ORAM) is another primitive that provides access pattern privacy [37, 38, 58]. It solves a related but different problem since it is designed for a *private* database that can only be accessed by a single client. Conventional ORAM constructions must incur a logarithmic response overhead. To reduce this overhead, SHE-based ORAM constructions have been proposed [20, 29]. These works have also partially inspired the design in this paper.

Even though ORAM has sublinear computation and constant bandwidth. These schemes could not be used for PIR because they do not support multiple clients [10, 45]. Several works have considered extending ORAM schemes to enable access to a large group of clients [12, 16–18, 52] but these works have limitations; they either require inter-client communication, a trusted proxy that manages client-server communication, or the computation increases with the number of clients.

Hamlin et al. recently introduced Private Anonymous Data Access (PANDA) [41]. PANDA is built on symmetric-key doubly efficient PIR, with the additional feature that the server is stateful and maintains information between multiple requests.

The scheme guarantee privacy if the number of corrupted clients is below a certain threshold but the downside is that the client and server computation is linear in the number of colluding clients.

8 CONCLUSION

In this paper, we have proposed a OnionPIR, response-efficient single-server PIR scheme with a response overhead of just 4.2x of an insecure baseline. The computation cost of OnionPIR is comparable or slightly better than the prior art. We improve the stateful PIR framework of [54] by introducing a novel and efficient offline phase. We integrate OnionPIR into the stateful PIR framework and achieve a 1.8 $\sim$ 22x improvement in computation time over stateless OnionPIR.

Future Directions. Even with all our improvements, single-server PIR (both stateless and stateful) still requires considerable server computation for large databases. It is interesting to explore further improvements to stateless single-server PIR (which is used in both the offline and online phases of our stateful PIR) as well as the PBSR problem.

One potential avenue is through better implementation or hardware acceleration. In our experiments, we noticed that over 80% of the server compute time is due to number-theoretic transformation (NTT) (which is the bottleneck of polynomial multiplication). In our implementation of OnionPIR, we have used the NFLlib library that has implemented NTT using AVX2 specialization. Recent research efforts have demonstrated that GPU and FPGA can significantly speed up polynomial multiplications [56]. An interesting future direction is to integrate them into PIR.

Another direction is to try to get rid of the expensive public-key operations in the online phase of stateful PIR. One such construction is given in [26] and it has a very cheap online phase. But the client has to rerun the offline phase after every online query. Finding an efficient online phase that does not require public-key operations or rerunning the offline phase every time is a promising future direction.

One limitation of the stateful PIR framework is that it currently only applies to the static database. An interesting direction is to explore how to support updates to the database in stateful PIR.

REFERENCES

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of Learning with Errors. *J. Math. Cryptol.* 9, 3 (2015), 169–203.

[2] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. 2019. Communication-Computation Trade-offs in PIR. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1483.

[3] Amazon. 2021. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/ Accessed: 2021-07-13.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society, San Francisco, California, USA, 962–979.

[5] Sebastian Angel and Srinath T. V. Setty. 2016. Unobservable Communication over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. USENIX Association, Savannah, GA, USA, 551–569.

[6] Omer Barkol, Yuval Ishai, and Enav Weinreb. 2010. On Locally Decodable Codes, Self-Correctable Codes, and t-Private PIR. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 10th International Workshop, APPROX* 4627 (2010), 311–325.

[7] Richard Beigel, Lance Fortnow, and William I. Gasarch. 2006. A tight lower bound for restricted pir protocols. *Comput. Complex.* 15, 1 (2006), 82–91.

[8] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Ilan Orlov. 2012. Share Conversion and Private Information Retrieval. In *Proceedings of the 27th Conference on Computational Complexity, CCC*. IEEE Computer Society, Porto, Portugal, 258–268.

[9] Amos Beimel, Yuval Ishai, and Tal Malkin. 2004. Reducing the Servers' Computation in Private Information Retrieval: PIR with Preprocessing. *J. Cryptol.* 17, 2 (2004), 125–151.

[10] Erik-Oliver Blass, Travis Mayberry, and Guevara Noubir. 2017. Multi-client Oblivious RAM Secure Against Malicious Servers. In *Applied Cryptography and Network Security - 15th International Conference, ACNS (Lecture Notes in Computer Science)*. Springer, Kanazawa, Japan, 686–707.

[11] Nikita Borisov, George Danezis, and Ian Goldberg. 2015. DP5: A Private Presence Service. *Proc. Priv. Enhancing Technol.* 2015, 2 (2015), 4–24.

[12] Elette Boyle, Kai-Min Chung, and Rafael Pass. 2016. Oblivious Parallel RAM and Applications. In *Theory of Cryptography - 13th International Conference, TCC (Lecture Notes in Computer Science, Vol. 9563)*. Springer, Tel Aviv, Israel, 175–204.

[13] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. 2017. Can We Access a Database Both Locally and Privately?. In *Theory of Cryptography - 15th International Conference, TCC*. Springer, Baltimore, MD, USA, 662–693.

[14] Christian Cachin, Silvio Micali, and Markus Stadler. 1999. Computationally Private Information Retrieval with Polylogarithmic Communication. In *Advances in Cryptology - EUROCRYPT, International Conference on the Theory and Application of Cryptographic Techniques*. Springer, Prague, Czech Republic, 402–414.

[15] Ran Canetti, Justin Holmgren, and Silas Richelson. 2017. Towards Doubly Efficient Private Information Retrieval. In *Theory of Cryptography - 15th International Conference, TCC*. Springer, Baltimore, MD, USA, 694–726.

[16] Anrin Chakraborti and Radu Sion. 2019. ConcurORAM: High-Throughput Stateless Parallel Multi-Client ORAM. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society, San Diego, California, USA, 23–63.

[17] T.-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. 2017. On the Depth of Oblivious Parallel RAM. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, (Lecture Notes in Computer Science, Vol. 10624)*. Springer, Hong Kong, China, 567–597.

[18] T.-H. Hubert Chan and Elaine Shi. 2017. Circuit OPRAM: Unifying Statistically and Computationally Secure ORAMs and OPRAMs. In *Theory of Cryptography - 15th International Conference, TCC 2017 (Lecture Notes in Computer Science, Vol. 10678)*. Springer, Baltimore, MD, USA, 72–107.

[19] Yan-Cheng Chang. 2004. Single Database Private Information Retrieval with Logarithmic Communication. In *Information Security and Privacy: 9th Australasian Conference, ACISP*. Springer, Sydney, Australia, 50–61.

[20] Hao Chen, Ilaria Chillotti, and Ling Ren. 2019. Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, London, UK, 345–360.

[21] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology - ASIACRYPT- 22nd International Conference on the Theory and Application of Cryptology and Information Security*. eprint, Hanoi, Vietnam, 3–33.

[22] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *J. Cryptol.* 33, 1 (2020), 34–91.

[23] Benny Chor and Niv Gilboa. 1997. Computationally Private Information Retrieval (Extended Abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing*. ACM, El Paso, Texas, USA, 304–313.

[24] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private Information Retrieval. In *36th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Milwaukee, Wisconsin, USA, 41–50.

[25] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *J. ACM* 45, 6 (1998), 965–981.

[26] Henry Corrigan-Gibbs and Dmitry Kogan. 2020. Private Information Retrieval with Sublinear Online Time. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Zagreb, Croatia, 44–75.

[27] Ivan Damgård and Mads Jurik. 2001. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC (Lecture Notes in Computer Science, Vol. 1992)*. Springer, Cheju Island, Korea, 119–136.

[28] Yun Deng et al. 2006. Crosstalk-free conjugate networks for optical multicast switching. *Journal of lightwave technology* 24, 10 (2006), 3635–3645.

[29] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. In *Theory of Cryptography - 13th International Conference, TCC 2016 (Lecture Notes in Computer Science, Vol. 9563)*. Springer, Tel Aviv, Israel, 145–174.

[30] Klim Efremenko. 2009. 3-query locally decodable codes of subexponential length. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC*. ACM, Bethesda, MD, USA, 39–44.

[31] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[32] Eric Fung, Georgios Kellaris, and Dimitris Papadias. 2015. Combining Differential Privacy and PIR for Efficient Strong Location Privacy. In *Advances in Spatial and Temporal Databases - 14th International Symposium, SSTD (Lecture Notes in Computer Science, Vol. 9239)*. Springer, Hong Kong, China, 295–312.

[33] Craig Gentry and Zulfikar Ramzan. 2005. Single-Database Private Information Retrieval with Constant Communication Rate. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP*. Springer, Lisbon, Portugal, 803–815.

[34] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference*. Springer, Santa Barbara, CA, USA, 75–92.

[35] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML (JMLR Workshop and Conference Proceedings, Vol. 48)*. JMLR.org, New York City, NY, USA, 201–210.

[36] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen (Lecture Notes in Computer Science, Vol. 8441)*. Springer, Copenhagen, Denmark, 640–658.

[37] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. ACM, New York, New York, USA, 182–194.

[38] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

[39] Matthew Green, Watson Ladd, and Ian Miers. 2016. A protocol for privately reporting ad impressions at scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Vienna, Austria, 1591–1601.

[40] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath T. V. Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and Private Media Consumption with Popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI*. USENIX Association, Santa Clara, CA, USA, 91–107.

[41] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. 2019. Private Anonymous Data Access. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Lecture Notes in Computer Science, Vol. 11477)*. Springer, Darmstadt, Germany, 244–273.

[42] Ryan Henry. 2016. Polynomial Batch Codes for Efficient IT-PIR. *Proc. Priv. Enhancing Technol.* 2016, 4 (2016), 202–218.

[43] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2004. Batch codes and their applications. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*. ACM, Chicago, IL, USA, 262–271.

[44] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2006. Cryptography from Anonymity. In *47th Annual IEEE Symposium on Foundations of Computer Science FOCS*. IEEE Computer Society, Berkeley, California, USA, 239–248.

[45] Nikolaos P. Karvelas, Andreas Peter, and Stefan Katzenbeisser. 2016. Blurry-ORAM: A Multi-Client Oblivious Storage Architecture. *IACR Cryptol. ePrint Arch.* 2016 (2016), 1077.

[46] Dmitry Kogan and Henry Corrigan-Gibbs. 2021. Private Blocklist Lookups with Checklist. *IACR Cryptol. ePrint Arch.* 2021 (2021), 345.

[47] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is NOT Needed: SINGLE Database, Computationally-Private Information Retrieval. In *38th Annual Symposium on Foundations of Computer Science, FOCS*. IEEE Computer Society, Miami Beach, Florida, USA, 364–373.

[48] Tony T. Lee. 1988. Nonblocking copy networks for multicast packet switching. *IEEE J. Sel. Areas Commun.* 6, 9 (1988), 1455–1467.

[49] Helger Lipmaa. 2005. An Oblivious Transfer Protocol with Log-Squared Communication. In *Information Security, 8th International Conference, ISC (Lecture Notes in Computer Science, Vol. 3650)*. Springer, Singapore, 314–328.

[50] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR : Private Information Retrieval for Everyone. *Proc. Priv. Enhancing Technol.* 2016, 2 (2016), 155–174.

[51] Prateek Mittal, Femi G. Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. 2011. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *20th USENIX Security Symposium*. USENIX Association, San Francisco, CA, USA, 295–312.

[52] Kartik Nayak and Jonathan Katz. 2016. An Oblivious Parallel RAM with O($\log^2$ N) Parallel Runtime Blowup. *IACR Cryptol. ePrint Arch.* 2016 (2016), 1141.

[53] Jeongeun Park and Mehdi Tibouchi. 2020. SHECS-PIR: Somewhat Homomorphic Encryption-Based Compact and Scalable Private Information Retrieval. In *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security*. Springer, Guildford, UK, 86–106.

[54] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2018. Private Stateful Information Retrieval. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, Toronto, ON, Canada, 1002–1019.

[55] Quarkslab. 2016. quarkslab/NFLlib. https://github.com/quarkslab/NFLlib

[56] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne, Switzerland, 1295–1309.

[57] Radu Sion and Bogdan Carbunar. 2007. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*. Internet Society, San Diego, California, USA, 2006–06.

[58] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, Berlin, Germany, 299–310.

[59] Julien P. Stern. 1998. A New Efficient All-Or-Nothing Disclosure of Secrets Protocol. In *Advances in Cryptology - ASIACRYPT '98, International Conference on the Theory and Applications of Cryptology and Information Security (Lecture Notes in Computer Science, Vol. 1514)*. Springer, Beijing, China, 357–371.

[60] Stephanie Wehner and Ronald de Wolf. 2005. Improved Lower Bounds for Locally Decodable Codes and Private Information Retrieval. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP (Lecture Notes in Computer Science, Vol. 3580)*. Springer, Lisbon, Portugal, 1424–1436.

[61] Sergey Yekhanin. 2007. Towards 3-query locally decodable codes of subexponential length. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*. ACM, San Diego, California, USA, 266–274.