OnionPIR: Response Efficient Single-Server PIR

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

Hao Chen
Facebook, USA
sxxach@gmail.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 reduces its response overhead by 22x by avoiding downloading the entire database in the offline stage. Compared to state-of-the-art stateless PIR schemes including OnionPIR, stateful OnionPIR reduces the computation cost by 7x.

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 database while guaranteeing that the identity of the entry remains hidden from the database server. Recently, PIR has been suggested for various applications, including privacy-preserving media streaming [35], anonymous communication [45], metadata private messaging [6], privacy preserving ad delivery [34], contact discovery [12], safe-browsing [41], password-checking [4], and privacy preserving location routing [27].

At a high level, PIR schemes can be classified into *single-server* [4, 5, 14, 16, 28, 42–44, 46] ones and *multi-server* ones [8, 9, 19, 20, 25, 25, 31, 52, 53, 53]. Multi-server schemes are generally more efficient but they need the stronger trust assumption of multiple non-colluding servers. The need for coordination from multiple organizations makes them hard to deploy in practice. In this paper, we will focus on single-server PIR. Single-server PIR schemes have thus far been quite inefficient. This limits their use to proof-of-concept systems or very small databases. The central goal of this paper is to substantially improve single-server PIR efficiency.

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 smaller response sizes than the trivial scheme but they are still very expensive in terms of response size and server computation, as we elaborate below.

Performance bottlenecks of single-server PIR. State-of-the-art single server PIR schemes such as SealPIR [5] have made significant progress in reducing the request size. For example, SealPIR's request size is only 32 KB [1] for a database with up to a million entries. However, they suffer from high costs in the other two metrics.

- *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 4030 seconds of server computation to fetch a 300 KB file (e.g. a song) from a database with a 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 and maintains a similar computation cost. 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. Concretely, to download a 30 KB entry, the client in OnionPIR only needs to download 128 KB of data.

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 request size for OnionPIR remains 64 KB for databases with up to one billion entries and increases only logarithmically with the database size after that. In contrast, the request size for SealPIR starts to increase quickly once the database size exceeds four million, e.g., it becomes 64 KB for a database with 16 millions entries and around 1 MB for one billion entries.

Main contribution 2. Next, to address the computation issue, we improve the *Stateful* PIR paradigm and integrate it with OnionPIR. In stateful PIR, the client has a local state and uses that state to make cheaper PIR queries [22, 47]. We remark that the original stateful PIR scheme of Patel et al. [47] 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* [24], which allow us to reap the computation savings of stateful PIR while keeping the amortized response roughly the

[1] after applying a standard optimization discussed in Section 4.3

same. The amortization benefit of stateful PIR is more prominent when accessing relatively larger data entries and larger databases. Concretely, the amortized computational cost to access a 300 KB entry in a database with one million entries is 578 seconds on a single CPU core, which is 7x better than vanilla OnionPIR. Compared to the original stateful PIR scheme of Patel et. al., stateful OnionPIR achieves around 22x reduction in amortized response size at the expense of a moderate increase in computational cost and request size. *Overall, the monetary cost of stateful OnionPIR is 6x less than stateless OnionPIR and 3x less than Patel et. al.* Hence, our scheme provides a more practical trade-off for single-server PIR.

Independent work of Ali et al. A concurrent work of Ali et al. [4] also studies how to reduce response size in (stateless) single-server PIR. At a high level, they used different techniques from us (we will elaborate on this in Section 7) and also achieved 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 [26] and RGSW [18, 29].

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 and 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 to avoid ciphertext multiplications altogether. 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 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 $R \mod q$ and $b = a \cdot s + e$. 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 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 in particular, $F = \frac{2 \log q}{\log t}$ 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 and a main task in this paper is to reduce F.

RGSW encryption. We will use a second somewhat homomorphic encryption scheme called RGSW [17]. A RGSW scheme has a *gadget vector* $g^{(l \times 1)} = (B^{(k-1)}, \cdots, B^{(k-l)})$ where B is called the decomposition *base* and $k = \log q / \log B$. The parameter B and the size of the gadget vector l give a trade-off between efficiency and noise growth. The gadget vector then gives a gadget matrix as follows:

$$\mathsf{G} = \mathsf{I}_2 \vee g = \begin{matrix} g & 0 \\ 0 & g \end{matrix} \in R^{(2l \times 2)}. \quad (1)$$

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

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

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

2.2 Noise Estimate and Computational Cost of Homomorphic Operations

As we have mentioned before, each homomorphic operation in SHE increases the noise in the output ciphertext. It turns out different operations result in drastically different noise growth, and this will significantly impacts our design decisions. These operations also have different computation costs, usually dominated by the number of polynomial multiplications required. We elaborate below on the noise growth and computation costs of different operations and summarize them in Table 3. Let $\mathsf{Err}(c)$ denote the variance of the noise term in a ciphertext c.

BFV **ciphertext addition.** This operation adds two BFV ciphertexts and outputs a BFV ciphertext encrypting their 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.

BFV **ciphertext-plaintext multiplication.** This operations takes as input a plaintext polynomial m and a BFV ciphertext, c encrypting m'. The output is a BFV ciphertext encrypting the product $m \cdot m'$. The noise term is multiplied with the plaintext [26]. In terms of computation, this operation requires 2 polynomial multiplications.

BFV **ciphertext multiplication.** This operations takes as input two BFV ciphertexts, and outputs a BFV ciphertext encrypting their product. The noise in the output ciphertext gets multiplied by the plaintext modulus t. This operation is also costly in terms of computation as it requires a relinearization step. The total cost is about $4 + l$ polynomial multiplications, where l is usually the same as the decomposition factor l in external products. Note that this operation is also expensive in terms noise growth. And it is the main culprit for the inefficiency of existing PIR schemes. We will not use this operation and hence will not go into details about it.

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

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 external product is additive only which allow evaluating deeper circuits.

External Product. The external product operation takes as input a BFV ciphertext d, encrypting μ_{d}, and a RGSW ciphertext C, encrypting μ_C, with respect to same secret key s. The output is a BFV ciphertext encrypting their plaintext product $\mu_C \cdot \mu_{\mathsf{d}}$.

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 external product below for completeness. Readers can refer to [18] for more details. We first define a vector v's gadget decomposition, denoted as $G^{-1}(\mathsf{v}) \in R^l$. Intuitively, the gadget decomposition of a vector has small coefficients and when multiplied by the matrix G, gives an approximation of the original vector. In more detail, $G^{-1}(\mathsf{v})$ has coefficients in $(-B/2, B/2]$, such that decomposition error is upper bounded by $B^{\log q/\log B - l}$. (Note that the result is a BFV ciphertext and we never need to decrypt RGSW ciphertexts in this paper.)

The noise after external product is bounded by $G^{-1}(d) \cdot \mathsf{Err}(C) + |\mu_C| \cdot \mathsf{Err}(d)$. In our paper, μ_C will always be a single bit (i.e., either 0 or 1). Also note that the coefficients of $G^{-1}(d)$ are bounded by $B/2$. Thus, the resultant noise term is roughly $B \cdot \mathsf{Err}(C) + \mathsf{Err}(d)$.

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.

3 OVERVIEW AND LIMITATIONS OF CURRENT PROTOCOLS

The most basic single-server PIR scheme is given in the 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 item encrypts 1 whereas all the other ciphertext encrypts either 0. The server *homomorphically* computes a dot-product between the query vector and the plaintext database to generate a response.

The above basic PIR has a request size linear in the the database size. To reduce the request size, two techniques have been suggested and adopted by existing PIR schemes. The first technique is *hierarchical query*, which dates back to the earliest works on PIR [21, 51]. 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 [5]. Instead of encrypting one bit per ciphertext, the client packs many bits within a single ciphertext using *lattice-based* BFV *homomorphic encryption*. The server can then expand this packed ciphertext into a list of ciphertexts, each encrypting a single bit. The third method involves sending only the second component of fresh BFV ciphertext together with the seed used to generate the first component (SealPIR did not incorporate this trick but could easily do). The server then uses the seed to generate the first component pseudorandomly [2]. This method reduces the size of the request in half. We will further discuss this optimization in Section 4.3.

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

1. **Pre-processing database** DB: The server database is encoded amenable format.
2. **Query Generation**(idx): For i from 1 to N, the client sets q_i (i-th encrypted query bit) to $\mathsf{Enc}(pk, 1)$ if $i = \mathsf{idx}$ and $\mathsf{Enc}(pk, 0)$ if $i < \mathsf{idx}$.
3. **Response Generation**($\{q_i\}_{[N]}$): The server computes $r = \sum_{i=1}^{N} q_i \cdot \mathsf{DB}_i$.
4. **Output** $\{r\}$.

Figure 1: Basic single-server PIR protocol. Step 3 uses plaintext-ciphertext multiplication and homomorphic addition.

Combining the above techniques, SealPIR would be able to 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 $\sqrt[2]{N}$.

To the best of our knowledge, SealPIR is currently the state-of-the-art single-server PIR. However, SealPIR still suffers 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 the hierarchical PIR idea in state-of-the-art PIR protocols [5, 44] 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 dot-product between each column of *plaintext* database and the query vector. The output of this multiplication is a vector of ciphertexts corresponding to the matrix row selected by the first query vector (2A). However, these ciphertexts are not directly used in the dot product with the second query vector. Instead, each of these ciphertexts are first *split* into 4 chunks and then each chunk is treated as plain-text 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 ciphertext split design is that now the response consists of F BFV ciphertexts (3) where F is the *ciphertext expansion factor* as described in Section 2. Although we used $F = 4$ as an example in the figure, in the actual SealPIR

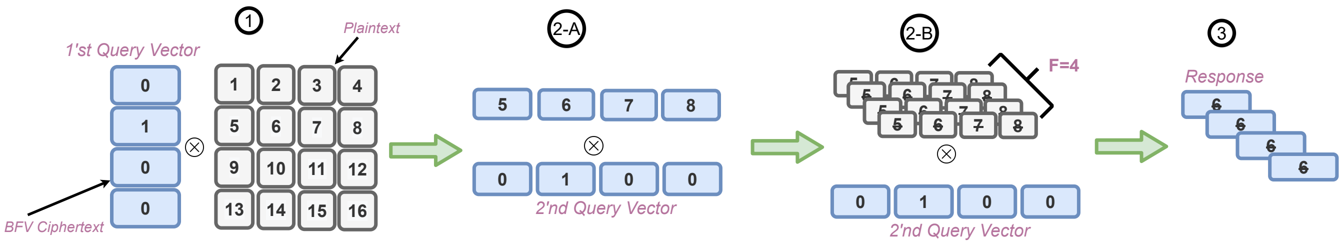

Figure 2: A basic hirarchical PIR in two dimension.

protocol, F is 10. Recall that the network overhead of a single ciphertext is equal to the expansion factor F. As the above scheme sends F ciphertexts, the overall response overhead would become 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 is little room for computation reduction in the standard PIR model. Looking ahead, we will incorprate the stateful PIR framework [47] 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.

Our warm-up protocol will adopt SHE and the hierarchical query framework. The top part of Figure 3 illustrates a sample execution of the 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, i.e., in a multiplicative fashion. In contrast, as mentioned in Section 2, the noise only grows *additively* after each *external product*. Therefore, we no longer have to split the intermediate ciphertexts and can use them directly for the second dimension of multiplication 2. As a result, the response, which is the output of the second multiplication, is only a single BFV ciphertext 3, 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. Next, we optimize the parameter choices of BFV SHE to reduce the ciphertext expansion factor F. Note that the smaller noise growth from external product already gives a minor improvement in F to our warm-up protocol over prior works. For a fixed ciphertext modulus q, if the noise growth is smaller, we can leave less room to noise growth and use a larger plain-text modulus t. This will reduce the ciphertext expansion factor F.

Furthermore, we will use a larger q to further reduce F. As one can see from Table 3, the noise growth does not depend on q. Thus increasing q allows more room for a larger t, and this helps decrease F even further. 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 us to set 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 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. For now, we remark that even with more dimensions, the response will still be a single BFV ciphertext in our protocol.

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 is dominated by the total number of polynomial multiplication operations. Observe that in the warm-up protocol the server performs at least one external product for each database entry. Each external product requires $2l$ polynomial multiplications due to the size of the RGSW ciphertext. So the number of polynomial multiplications from the first dimension alone is $2lN$. In comparison, SealPIR's computational bottleneck lies in its first dimension of N BFV ciphertext-plaintext multiplication. Each BFV ciphertext-plaintext multiplication requires only 2 polynomial multiplications, giving a total computational cost of $2N$ polynomial multiplications. 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 good ciphertext expansion factor, at the expense of higher computation. Here is a simple method to improve the computation cost: revert back 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.

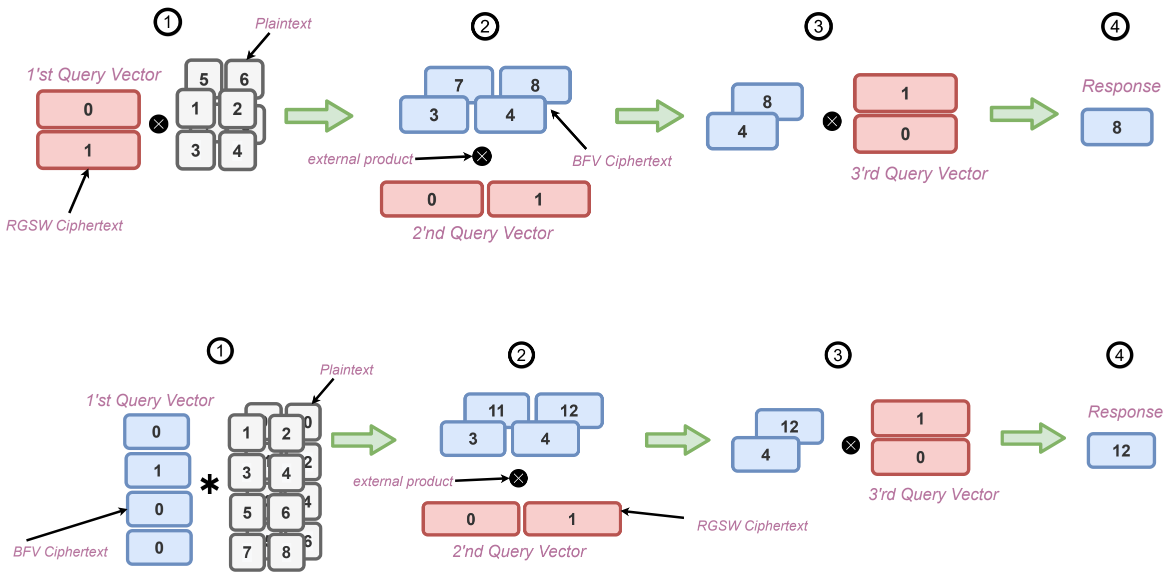

Figure 3: Examples of different variants of three dimensional hierarchical PIR. In the top figure the client queries are composed of all RGSW ciphertexts and the output of each dimension is BFV ciphertexts. In the bottom figure, first dimension's query vector consists of BFV ciphertexts, remaining query vectors are still RGSW ciphertexts and the first dimension is slightly larger than the other dimension.

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

Figure 4: *Decomposed* ciphertext-plaintext product. Output of DecompMul is a single BFV ciphertext encrypting $q \cdot \mathsf{pt}$. The noise in this output ciphertext is only increased by $\log(t)/2$ bits.

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 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 BFV ciphertexts. A similar approach is proposed in [30].

The details of this technique are 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. As mentioned, we also make the first dimension slightly larger. The server first uses 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.

With this technique, each DecompMul operation is about twice as expensive as BFV ciphertext-plaintext multiplication. But note that we have a much better ciphertext expansion factor F, which means each ciphertext in OnionPIR contains a lot more plaintext data than prior works. Overall, we actually expect to see an small improvement in computation cost.

4.3 Query Compression

Sending one ciphertext per query bit results in a large request size. As discussed earlier, previous works have proposed the query compression technique [5, 17] 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 expands this ciphertext into an encryption of each bit separately. In OnionPIR we adopted the query compression algorithm given by Chen et al. [17].

Query packing. Algorithm 1 represents query packing in OnionPIR. All the query vectors are packed into a single plaintext pt, which is then encrypted using BFV encryption. Chen et al. [17] show that for each query bit q , 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. There is a chance that query vectors may not fit into a single plaintext. However, later in the section, we show that a single plaintext is sufficient in OnionPIR.

Algorithm 1: □eryPack algorithm used in OnionPIR

Input: $\{q_i\}_{i=1}^{d}$ a set of plaintext query vectors one for each dimension
Output: $\tilde{q}$ a single BFV ciphertext packing all the query vectors
Notation:
- d,- number of dimensions
- N_i, size of i-th dimension
- e, number of components for each kind of encryption
- $q_{i,j}$, j-th bit in i-th vector

$p = 0$
Sets plaintext pt as follows:
for $i = 1 : d$ **do**
  for $k = 1 : e$ **do**
    • $e = 2$ for first-dimension and $e = l$ for rest.
    for $j = 1 : N_i$ **do**
      $\mathsf{pt}_{p+j+k} = q_{i,j}[k]$
    end
  end
  $p = p + N_i + e$
end
BFV-encrypts pt to get $\tilde{q} = \mathsf{BFV}(\mathsf{pt})$ and outputs q

Query unpacking. Algorithm 2 represents query unpacking in OnionPIR. The algorithm first calls BFV expansion procedure given in Algorithm 3 of [17]. This procedure outputs an array of BFV ciphertexts, encrypting individual values. The algorithm parses the first query vector. For the remaining query vectors, BFV expansion only gives the first l rows of each RGSW ciphertexts. To get second l rows for each RGSW ciphertext, the algorithm performs external products between RGSW encryption of client secret-key and first l rows. We refer the user to Section 4.3 of [17] for further explanation on this trick.

But 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 that need 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 $d = 1 + \lceil \log_4(N/128) \rceil$.

We pack two values for each query bit therefore in total 256 values. 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.

Pseudorandom first component in ciphertexts. We can further reduce the request size by using the optimization given in [2]. Recall that each BFV ciphertexts consists of two components (c_0, c_1),

Algorithm 2: □eryUnpack algorithm adopted from algorithm 4 in [17]. We assume that A is provided by each client at the time of initialization. This algorithm outputs a single encrypted query vector for each dimension.

Input: $(\tilde{q})$ a single BFV ciphertext packing all the query vectors, $A = \mathsf{RGSW}(-s)$ rgsw encryption of client secret key
Output: $\{\mathfrak{q}_j\}_{j=1}^{d}$ set of encrypted query vectors
Notation:
- $\mathfrak{q}_{i,j}$, j-th ciphertext component of i-th query vector.
- $c[]$, single BFV ciphertext.
- Remaining as defined in Algorithm 1.

Subroutines:
- expandRlwe, BFV expansion procedure given in algorithm 3 of [17]

$\mathsf{c} = \mathsf{expandRlwe}(\tilde{q})$ • flat array of all expanded ciphertexts
$\mathfrak{q}_{1,:}[0] = c[1 : N_1]$ • setting first query vector
$\mathfrak{q}_{1,:}[1] = c[N_1 + 1 : 2N_1]$
$ptr = 2N_1$
for $i = 2 : d$ **do**
  • setting higher query vectors
  for $k = 0 : l - 1$ **do**
    for $j = 1 : N_i$ **do**
      $\mathfrak{q}_{i,j}[k] = \mathsf{c}[ptr + kN_i + j]$
      $\mathfrak{q}_{i,j}[k + l] = \mathsf{ExtProd}(A, \mathsf{c}[ptr + kN_i + j])$
    end
  end
  $ptr = ptr + lN_i$
end
Outputs $\{\mathfrak{q}_j\}_{j=1}^{d}$

where in a fresh ciphertext the first component is sampled uniform randomly from $R \bmod q$. Thus, instead of sending a truly random c_0, the client use a pseudorandom c_0 can send a short random seed instead, and the server can derive c_0 from the seed. This optimization reduces the request size in half.

4.4 OnionPIR Full Protocol

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

The database is represented as a hypercube of d dimensions. The size of the first dimension $N_1 = 128$ 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. The server unpacks this ciphertext into separate encrypted query vectors. Each

Algorithm 3: OnionPIR Protocol.

Input: DB server database of size N

Notation:

- N, database size
- DB_i, i-th record
- id, client's index
- $\mathfrak{q}_i$, encrypted query vector for i-th dimension dimension
- DB', intermediate database
- All the notations defined in Algorithm 1 and 2
- shaded part is executed by server

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

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

Client generates query vectors $\{q_j\}_{j=1}^{d}$ corresponding to $(i_1, \cdots i_d)$, such that only $q_j[i_j]$ is equal to 1 and remaining all 0.

Client computes $\tilde{q} = \mathsf{QueryPack}(\{q_j\}_{j=1}^{d})$, and sends q to server

Server computes:

$\{\mathfrak{q}_j\}_{j=1}^{d} = \mathsf{QueryUnpack}(\tilde{q})$ • query expand

$s = N/N_1$

for $j = 1 : s$ **do**

  $\mathsf{DB}'_j = \sum_{k=1}^{N_1} \mathsf{DecompMul}(\mathfrak{q}_{1,k}, \mathsf{pt}_{k+(j*N_1)})$

  • *first dimension*

end

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

  $s = \mathsf{DB}'/N_i$

  for $j = 1 : s$ **do**

    $\widetilde{\mathsf{DB}}_j = \sum_{k=1}^{N_i} \mathsf{ExtProd}(\mathfrak{q}_{i,k}, \mathsf{DB}'_{k+(j*N_i)})$

  end

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

  • *remaining dimensions*

end

Server sends $r = \mathsf{DB}'$ (a single record now) to the client

Client decrypts r to get data of record id

entry in the first query vector consists of two BFV ciphertexts and each entry in subsequent dimensions is a RGSW ciphertexts.

For the first dimension, the server performs a dot-product (using $\mathsf{DecompMul}$ operation) between the first query vector and each (plaintext) column of the hyper cube. 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 treated 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 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 optimization discussed in Section 4.3, the request size is 64 KB.

Response size. We set the ciphertext modulus q to 124 bits (in the implementation, we pad it to 128 bits). As the plaintext modulus, t has 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 expansion requires around $w \cdot l^2$ polynomial multiplications [34] where w is the total number of packed bits. Because of the high dimensions, only a logarithmic number of bits are packed. Therefore, query expansion is not the computation bottleneck.

The total number of polynomial multiplications required by the dot product operations is about $2 \cdot N + 4 \cdot l(\frac{N}{N_1} + \frac{N}{4N_1} + \frac{N}{16N_1} + \cdots)$. 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 hence twice as many polynomial multiplications. Furthermore, each polynomial multiplication in our protocol is about 4.2x 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 expansion and the ciphertext-plaintext multiplications in the first dimension.

The noise in the expanded ciphertext (RGSW and BFV both) is bounded by [17]:

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

Here, $\mathsf{Err}(\mathsf{BFV})$ is the initial noise in the packed input ciphertext and w is the number of packed bits. As mentioned earlier, there are fewer bits to be packed in OnionPIR, so query expansion adds less noise than prior art.

In the first dimension, the noise increases by a factor of $O(N_1 B')$. Here, N_1 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

$$\mathsf{Err}(\mathsf{ct}_{resp.}) \leq \mathsf{Err}(\mathsf{ct}_1) + O(d) \cdot \mathsf{Err}(\mathsf{ct}_{exp}) \tag{3}$$

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 a 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 [47] 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 results in large offline response size, 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 7x reduction in computation cost over vanilla OnionPIR for relatively large databases and entry sizes. Compared to Patel et al.'s scheme, our stateful scheme reduces the offline response size by 22x at the expense of 2.2x increase in request size and 9x increase in computation.

In the remainder of this section, we will first provide a high-level overview of the PSIR framework of [47] and then present our improved offline phase.

5.1 Private Stateful Information Retrieval

At a high level, the PSIR protocol by Patel et al. [47] 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 [47], which is defined as follows: Given c subsets $S_1, \cdots, S_c$, where each subset consists of k' random indexes. Privately fetch c values corresponding to the sum of all the entries in each subset. The client stores the subset and their corresponding sums locally. The privacy of PBSR requires that the server should not learn the c input subsets.

StreamPBSR. To perform PBSR, the main protocol of Patel et al. ultimately proposes that the server streams the *entire* database to the client. Every time the client runs out of local state or a new client joins, the server has to stream the database again. For any practical application streaming the entire database to each client is not plausible. 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 [47], the authors also sketch a construction based on batch codes and homomorphic encryption. In this construction, the server encodes the database into batch codes and the client then runs a private batch retrieval protocol given in [5] to privately retrieve the subset sums. Although this construction gets rid of database streaming. The authors found that the computational overhead of this construction is so high that the computation of stateful PIR becomes comparable to vanilla 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.'s PSIR protocol [47], 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 the 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 there are $m = N/(k'+1)$ partitions $P_1, P_2, \cdots, P_m$, each of size $k'+1$ and a random partition P_r is equal to $S' \cup i$, where r is randomly picked 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 PIR query 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 subsets it will perform the offline phase again to get new states. 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 m which is $k'+1$ times smaller than the original database. This results in a factor of $k'+1$ reduction in server computation. The online phase of 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 PSIR, it can be of independent interest. Our key observation is that the PBSR problem has a similar interface to *copying networks*.

Copy Networks PBSR. A copy network is a special kind of computer network, that replicates input packets from various sources simultaneously. At a high level, the copy network can be configured on the fly to copy each input packet for a requested number of times to the destinations. Precisely, a copy network 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. Packets arriving at each switch can either be routed on one of the output links or replicated and sent out on both of the links. Figure 5 depicts an example of five-stage copy network. White switches route the incoming packets on output wires and the grey switches replicate

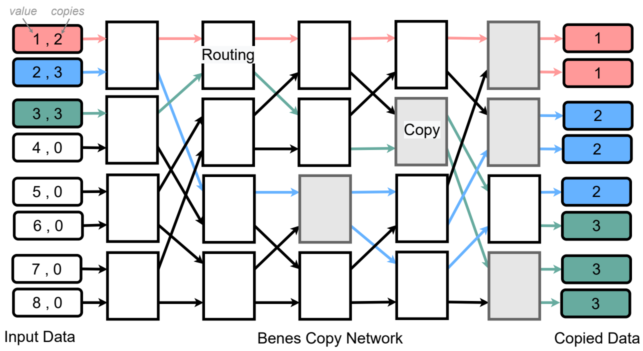

Figure 5: Benes copy network. Each intermediate node is a controlled swap gate that either routes the inputs or copies one of the inputs to output wires. Each input node has a value and number of copies. First element is copied twice while element two and three are each copied three times.

them. The network replicates the first input packet twice and the second and third input packets three times each.

Deng et. al. [24] showed that it is possible to find control bits for each switch that satisfies all the copy requests. The only restriction is that the total number of requested copies does not exceed the number of destinations. They provide an efficient routing algorithm that outputs control bits for the switches. We refer interested readers to [24] for further details on the copy networks and the routing algorithm.

If we evaluate the copy network homomorphically, the client could use it to perform PBSR by the following approach:

(1) The client picks c random subsets of size k', such that $ck' = N$.
(2) For each database entry, the client counts the number of subsets that include the entry.
(3) The client uses these counts as copy requests in the routing algorithm to generate control bits.
(4) The client then encrypts and sends these bits to the server.
(5) The server uses these encrypted control bits to homomorphically evaluate the copy network on the database.
(6) The client then asks the server to homomorphically add outputs of the copy network into c sums and return the results.

In Step 6 the client will ask the server to add values corresponding to each subset together. If multiple subsets include the same entry, the client will pick a different copy for each sum. The server gains no information about the elements added together from the encrypted output in Step 5.

Homomorphic evaluation of copy network. Note that the network structure of the 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. In [17], Chen et al. use external products to construct an encrypted version of the controlled mux gates. Specifically, they encrypted each input as BFV ciphertexts and the swap bits as RGSW ciphertexts. They used these gates to homomorphically evaluate a permutation network. We note that evaluating the copy network is quite similar to the permutation network. The only difference is that each switch either route or replicate the incoming messages. In Figure 6 we show that such a switch can by constructed using

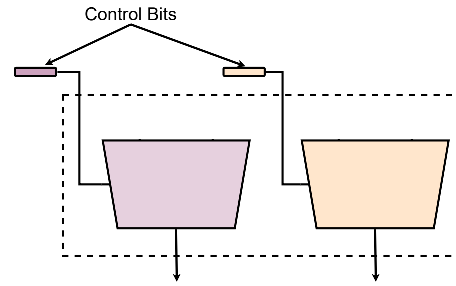

Figure 6: 2×2 switch using two mux gates. Inputs are not shown.

	StreamPBSR	BatchCodePBSR	Our PBSR
Response	$O(N)$	$O(c)$	$O(c)$
Request	–	$O(Nc)$	$O(N \log N)$
Computation	–	$O(Nc)$	$O(N \log N)$

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. Also, in Patel et al. scheme $c >> \log N$, therefore Our PBSR has better request and computation than BatchCodePBSR.

two mux gates. Each input ciphertext in the copy network passes through $2 \log N - 1$ switches and each switch evaluates two external products. As a result, the noise in the output ciphertext only logarithmically increases.

One remaining minor issue is that the server knows that the adjacent outputs from the copy network are likely copies of the same element. That may leak information to the server about subset overlapping. Thus, after evaluating the copy network in Step 5, the client and the server first homomorphically permute the output using a permutation network [17] and then perform the Step 6.

Table 2 compares the asymptotic complexity of our proposed PBSR with the two PBSR schemes given by Patel et al [47]. Our construction has a significantly smaller response size than StreamPBSR. The response size of BatchCodePBSR is similar to our construction, but the request size and server computation are linear in the number of subsets c. In our construction, they do not depend on c. Therefore, for large c, our construction has better asymptotic efficiency than BatchCodePBSR.

6 IMPLEMENTATION AND EVALUATIONS

6.1 Implementation Details

We implemented OnionPIR atop the SEAL Homomorphic Encryption Library. 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 implementation of NTT in SEAL is quite slow, which in turn hurts the computation time of our protocol. Thus, for NTT, we instead use NFLlib [48], an efficient library that uses several arithmetic

	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)	5.49	10.79	21.59	86.36	3.67	4.13	4.6	5.5
Dot-Products (sec)	20.51	91.21	381.41	6, 361.64	21.33	96.87	396.4	6, 410.4
Total Computation (sec)	26	102	403	6, 448	25	101	401	6, 416
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 half of OnionPIR. But for database with 16 million entries request size of OnionPIR and SealPIR is equal.

		Stateful OnionPIR		OnionPIR		Patel et al. Scheme	
		$N = 2^{20}$	$N = 2^{24}$	$N = 2^{20}$	$N = 2^{24}$	$N = 2^{20}$	$N = 2^{24}$
Online	Response size (KB)	1, 280	1, 280	1, 280	1, 280	32, 000	32, 000
	Request size (KB)	162	431	64	64	130	399
	Computation (sec)	62	253	4, 010	64, 480	63	264
Offline	Response size (KB)	1, 280	1, 280	–	–	24, 769	107, 088
	Request size (KB)	128	576	–	–	–	–
	Computation (sec)	516	2, 664	–	–	–	–
Total (amortized)	Response size (KB)	2, 560	2, 560	1, 280	1, 280	56, 769	139, 088
	Request size (KB)	291	1, 007	64	64	130	399
	Computation (sec)	578	2, 917	4, 010	64, 480	63	264
	Server Cost (US cents)	0.18	0.83	1.12	17.92	0.54	1.26
	Client Storage (GB)	3.8	14	–	–	3.8	14

Table 4: Comparison of stateful OnionPIR with vanilla OnionPIR construction for databases of size 314 GB and 5, 033 GB. Each database entry is of 300 KB. For database with one million entries, number of entries in hint $c = 12,700$ and for 16 million entries, $c = 47,000$. In both cases, the client storage is less than 2% of total server storage.

optimizations and AVX2 specialization for arithmetic operations over polynomials. We note that the NTT implementation in NFLlib is 2 – 3x faster than SEAL. We integrated NFLlib's NTT with SEAL. In total, our modifications consist of around 3000 lines of C++ code.

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 run each experiment 10 times and report the averages. Monetary costs is the sum of CPU cost for server computation and the cost of network traffic. These costs were computed using standard rates from Amazon EC2 Instance prices [1], which at the time of writing were at one cent per CPU-hour and 9 cents per GB of internet traffic.

Parameters. We choose security parameters based on the FHE standard [3]. We set the polynomial degree n to 4096, and size of coefficient modulus q to 124 bits. This provides us around 128 bits of security. In SealPIR, n is set to 2048 and q is set to 60 bits, which provides 128 bits of security.

For basic OnionPIR experiments, we set the size of each database entry to 30 KB and for stateful OnionPIR experiments we consider each entry to be of 300 KB. As mentioned before, due to the lower noise growth, we can set plaintext modulus t to 60 bits. This means that in OnionPIR 30 KB of data can fit in a single plaintext. In contrast, in SealPIR each plaintext could accommodate only 3 KB of data.

6.3 Evaluation Results of OnionPIR

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

Computational. In OnionPIR and SealPIR, the server mainly performs two tasks: Query unpacking and the Dot-products between the query vectors and the database. Query unpacking in OnionPIR takes less time than SealPIR because we pack only a logarithmic number of query bits (q.v. Section 4.3), while in SealPIR sub-linear number of bits are packed in each query ciphertext. Also, we observe that dot-products take a significant proportion of total server computation. Polynomial multiplication is a core operation used in each dot-product. Therefore, improving the performance of polynomial multiplications could significantly improve the performance of both schemes.

In the theoretical analysis in Section 4.4, we estimated that OnionPIR is 1.25x better than SealPIR in terms of computation. But in the actual experiments, we observe that the computational cost of OnionPIR is almost identical to SealPIR across all database sizes. This is in part due to the cost from RGSW decomposition.

Request Size. For databases with up to one 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 a database with 16 million entries, request size of SealPIR and OnionPIR is equal.

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. For applications with an even bigger entry size, OnionPIR is an ideal candidate as it significantly reduces the network overhead.

Server Cost. For smaller databases, the server cost in OnionPIR is orders of magnitude smaller than SealPIR. But for bigger databases, the cost of both schemes becomes almost equal. In the case of a smaller database, the server computation is small and the response size is the bottleneck. With the increase in database size, the server computation becomes the bottleneck which is comparable in both schemes. As an example, for a database with 65, 536 entries, the server cost of OnionPIR is four times less than SealPIR. But OnionPIR only has 19% less cost for a database with one million entries.

6.4 Evaluation Results of stateful OnionPIR

For stateful OnionPIR, we integrate PBSR construction given in Section 5.2, into the stateful PIR framework along with OnionPIR. In Table 4 we compare the performance of stateful OnionPIR with vanilla OnionPIR and Patel et al. scheme. Patel et al. scheme uses StreamPBSR [47] in the offline phase and SealPIR in the online phase.

Amortization of offline phase. As discussed in Section 5.2, for stateful PIRs, the client interacts with the server to run the offline phase once and retrieve c subset sums. The client then uses this state to make c *cheaper* queries in the online phase. In other words, each offline phase is amortized to c queries. In our experiments, for a database with one million entries, c is set to 12, 700 and for a database of 16 million entries, c is set to 47, 000.

Comparison with OnionPIR. As discussed before, vanilla OnionPIR has a high computational cost. Specifically, for our experimental database sizes, the computation times would be 1.1 hours and 18 hours. It is quite clear that for any application, 18 hours of computation is impractical. Stateful OnionPIR significantly reduces these times to 9 and 48 minutes, which is a factor of 7x and 22x improvement over vanilla SealPIR. The trade-offs are request and response sizes. Specifically, the response size increases by a factor of two, and the request size increases by a factor of 4.5 and 15, respectively.

Comparison with Patel et al.. The amortized response size in Patel et al. scheme is 55 MB and 136 MB for the two databases. With our proposed PBSR scheme, the amortized response size in stateful OnionPIR is reduced to 2.5 MB for both database sizes. A trade-off here is that our request size is 2 $\sim$ 2.5x larger and the computation is 9 $\sim$ 11x higher. This is because the StreamPBSR in their scheme does not require any server computation or client input.

Summary. Stateful OnionPIR provides a nice middle ground between computation and communication. Even though vanilla OnionPIR has moderately better request and response sizes, its computation cost is too high for large databases. Similarly, Patel et al. has better request size and computation, but downloading the entire database results in very high response size for large databases. Overall, stateful OnionPIR has moderate costs across all the performance metrics. This reflects in the smaller monetary cost of stateful OnionPIR, which is 6x less than vanilla OnionPIR and 3x less than Patel et al. scheme.

7 RELATED WORK

Early single-server PIR schemes. Some of the early single-server PIR protocols are based on *Additive homomorphic encryption* (AHE). These schemes followed the blueprint of Kushilevitz and Ostrovsky [42]: the database is represented as 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 response size of $O(\sqrt{N})$. Cachin et al. [14] proposed a PIR protocol based on ϕ-Hiding assumption. The protocol has request size of $O(\log^4 N)$ and polylogarithm response size $O(\log^d N)$. Gentry and Ramazan [28] 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 [16] 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 [43] uses the Damgard-Jurik encryption scheme [23] to achieve $O(\log^2 N)$ request size and $O(\log N)$ response size.

Sion and Carbunar [50] observe that these schemes in practice often perform slower than downloading the entire database when the network bandwidth is a few hundred Kbps. The poor performance is due to the fact that , in all of the these schemes, the server needs to perform at least N big-integer modulus multiplications or modulus exponentiations. The computational cost of such an operation 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. [44] present XPIR with good computation cost. Specifically, to retrieve a 30 KB entry from a database with one million entries, their protocol takes around 383 seconds, which is slightly less than our OnionPIR. However, the downside of their protocol is that the request size is 17 MB and the response size overhead is $100x$. SealPIR [5] addresses the request size bottleneck by introducing a novel 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; their response size is similar to XPIR. Very recently, Park and Tibouchi [46] present a construction based on TFHE [18] that improves the response overhead of SealPIR to 16x; but their computation cost more than doubled compared to SealPIR.

Concurrent work. Ali et al. [4] 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 (cf. Table 3), their protocol requires larger FHE parameters, which lead to higher sever 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 takes around 900 seconds with the response size of 357 KB and request size of 119 KB. In comparison, for the same database, OnionPIR requires 800 seconds with the response size of 256 KB and request size of 64 KB.

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 [7–9, 19, 20, 25, 31, 52, 53]. The first multi-server PIR schemes are proposed by Chor *et al.* [20] 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 size are known using three or non-colluding more servers. The best known three-server schemes have a request size of $2^{O\sqrt{\log N \log\log N}}$ [25, 53]. Gilboa et al. [31] proposed a two-server PIR 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 cryptographic operations.

Stateful PIR. Patel et al. [47] introduced stateful PIR where the client retrieves some helper data in the offline phase and use 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 substantially reduction in amortized computation cost over vanilla 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 a recent pioneering work, Corrigan-Gibbs and Kogan have proposed two-server stateful PIR schemes with amortized sublinear computation complexity [22, 41]. This two-server PIR scheme shows promising efficiency in both theory and in 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 run the offline phase again after every single online query. Therefore, it only reduces the online cost while the overall cost is actually even worse than vanilla stateless PIR.

Orthogonal directions to improve PIR computation. We mention two orthogonal directions to reduce server computation in PIR.

One direction is called batched PIR, which allows the server to answer a batch of PIR queries with lower cost than answering each query separately. This general strategy has been adopted in a setting where the queries comes from single client [5, 37, 38] or multiple clients [10, 39]. Our protocol can also be extended to support batched queries but we remark that this approach is not always applicable as 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. [10]. 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 also means they are still impractical at the moment.

Related privacy-preserving primitives. *Oblivious Ram* (ORAM) provides access pattern privacy for a private database [32, 33]. The client state contains a secret key that is used to decrypt the data. The secret key enables the ORAM to use sublinear computation and bandwidth. However, these schemes could not be used for PIR because they do not support multiple clients [11, 40]. Several works have considered extending ORAM schemes to enable access to a large group of clients. But these approaches are highly inefficient because they either require running a separate ORAM for each client or multiple non colluding servers.

Hamlin et al. recently introduced Private Anonymous Data Access (PANDA) [36]. PANDA is build on symmetric-key DEPIR, with additional feature that the server is stateful and maintains information between multiple requests. The schemes guarantee privacy if the number of corrupted clients is below a certain threshold. The downside of their scheme is that the client and server computation is linear in the number of colluding clients.

8 CONCLUSION

In this paper, we have proposed a response-efficient single-server PIR scheme. Our protocol has a response overhead of just 4.2x in comparison to an insecure baseline. Additionally, the computation cost of our scheme is comparable to SealPIR, which makes it the most efficient single-server PIR scheme. We further improve the Stateful PIR scheme of [47] by introducing a novel offline phase that

[2] Ali et al.'s scheme works best when the record 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.

drastically reduces the response overhead of the overall protocol. We further integrated OnionPIR in the Stateful PIR framework to show around 7x improvement in total computation time.

Future Directions. In OnionPIR we have mainly focused on improving the response size. However, like prior homomorphic encryption-based schemes, the computational cost of vanilla OnionPIR protocol is still quite high. The recent stateful PIR scheme has significantly improved the computational cost however, these schemes are not useful for the applications where the database is updated frequently and the client has limited storage, such as mobile devices. Therefore, it is still desirable to reduce the computational overhead of vanilla single server PIR schemes. One potential direction that could improve the computation time is to improve protocol implementation. In our experiments, we noticed that more than 80 percent of the server compute time is due to number-theoretic transformation (NTT) based polynomial-polynomial multiplications. In our implementation of OnionPIR, we have used the NFLlib library that has implemented NTT using AVX2 specialization. This results in a factor of four improvements for each polynomial multiplication. Recent research efforts have demonstrated that using specialized hardware such as GPU and FPGA could significantly reduce the computational overhead of polynomial multiplications [49]. An interesting future direction is to improve the computation time of PIR schemes by integrating them with specialized hardware.

For the applications in which the database remains static or updates less frequently, a stateful PIR framework is a promising option. Although in this paper we have removed the requirement of downloading the entire database in a stateful PIR framework, our proposed PBSR construction still requires considerable server computation and client state. Hence, it is interesting to explore further improvements to PBSR.

REFERENCES

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

[2] 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, ,2016*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.).

[3] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard.* Technical Report. HomomorphicEncryption.org, Toronto, Canada.

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

[5] 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, SP*.

[6] 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*.

[7] Omer Barkol, Yuval Ishai, and Enav Weinreb. 2010. On Locally Decodable Codes, Self-Correctable Codes, and t-Private PIR. *Algorithmica* (2010).

[8] Richard Beigel, Lance Fortnow, and William I. Gasarch. 2006. A tight lower bound for restricted pir protocols. *Comput. Complex.* (2006).

[9] 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 2012*.

[10] Amos Beimel, Yuval Ishai, and Tal Malkin. 2004. Reducing the Servers' Computation in Private Information Retrieval: PIR with Preprocessing. (2004).

[11] 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 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings (Lecture Notes in Computer Science)*.

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

[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 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II.*

[14] Christian Cachin, Silvio Micali, and Markus Stadler. 1999. Computationally Private Information Retrieval with Polylogarithmic Communication. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, 1999.*

[15] Ran Canetti, Justin Holmgren, and Silas Richelson. 2017. Towards Doubly Efficient Private Information Retrieval. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II.*

[16] Yan-Cheng Chang. 2004. Single Database Private Information Retrieval with Logarithmic Communication. In *Information Security and Privacy: 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings (Lecture Notes in Computer Science).*

[17] Hao Chen, Ilaria Chillotti, and Ling Ren. 2019. Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE. ACM.

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

[19] 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, 1997.*

[20] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private Information Retrieval. In *36th Annual Symposium on Foundations of Computer Science, Wisconsin, USA, 23-25 October 1995.*

[21] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. (1998).

[22] 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, May 10-14, 2020.*

[23] 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 2001 (Lecture Notes in Computer Science).*

[24] Yun Deng and Tony T. Lee. 2006. Crosstalk-free Conjugate Networks for Optical Multicast Switching. *CoRR* abs/cs/0610040 (2006).

[25] Klim Efremenko. 2009. 3-query locally decodable codes of subexponential length. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009.*

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

[27] 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).*

[28] Craig Gentry and Zulfikar Ramzan. 2005. Single-Database Private Information Retrieval with Constant Communication Rate. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005 (Lecture Notes in Computer Science).*

[29] 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, 2013.*

[30] 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.*

[31] 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.*

[32] 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, 1987, New York, New York, USA*, Alfred V. Aho (Ed.).

[33] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* (1996).

[34] 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.*

[35] 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.*

[36] 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, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II.*

[37] Ryan Henry. 2016. Polynomial Batch Codes for Efficient IT-PIR. *Proc. Priv. Enhancing Technol.* (2016).

[38] 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, 2004.*

[39] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2006. Cryptography from Anonymity. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 2006.*

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

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

[42] 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 '97.*

[43] Helger Lipmaa. 2005. An Oblivious Transfer Protocol with Log-Squared Communication. In *Information Security, 8th International Conference, ISC 2005 (Lecture Notes in Computer Science).*

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

[45] 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.*

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

[47] 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.

[48] Quarkslab. [n.d.]. quarkslab/NFLlib. https://github.com/quarkslab/NFLlib

[49] 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*, James R. Larus and Karin Strauss (Eds.).

[50] 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.

[51] 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, Proceedings*, Kazuo Ohta and Dingyi Pei (Eds.).

[52] 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 2005.*

[53] Sergey Yekhanin. 2007. Towards 3-query locally decodable codes of subexponential length. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, 2007.*