Parallelizable and Updatable Private Information Retrieval

Boyan Litchev
MIT PRIMES

Simon Langowski (Mentor)
MIT

January 16, 2024

In traditional fully homomorphic encryption (FHE), number-theoretic transforms (NTTs) are utilized to speed up the process of multiplication. After multiplication, the ciphertext noise increases multiplicatively, meaning that few multiplications can be applied successively. To reduce this noise, schemes [8, 6, 7] apply modulus and key-switching after multiplication. However, these operations cannot be applied to the NTT forms of ciphertexts, so ciphertexts have to be converted out of NTT form, using a significant amount of processing time and preventing parallelization. In the setting of private information retrieval (PIR), small ciphertext values, low multiplicative depth, and the usage of fresh ciphertexts in multiplications mitigate noise even without key and modulus-switching. We explore the efficiency of removing key and modulus-switching from the computation process for PIR, eliminating the need for intermediate number-theoretic transforms. This also aids in updating the result of a query when the database is modified.

1 Introduction

Private Information Retrieval (PIR) is a protocol that allows users to submit a query to a database to retrieve a database item, without revealing which item was retrieved. Recently, it has been used for applications such as metadata-private communication [3, 5] and anonymous streaming [9].

There are two main variants of PIR: Information-theoretic PIR (IT-PIR), which assumes multiple non-colluding servers, and computational PIR (CPIR) [4, 11, 2, 13, 10, 14], which makes no such assumptions, but is slower as a result. Additionally, some schemes use a single query and answer between the client and database, while others have multiple rounds of communication. In this work, we focus on single-round CPIR.

Single-round CPIR is composed of three procedures: a query generation procedure that generates a query that encodes but doesn't reveal the desired index, an answer procedure that uses the query and the database to generate a response, and a decrypt procedure that the client can use to extract the database element from the response.

In order to generate a response using the query and database, CPIR utilizes homomorphic multiplications and additions, which let it operate on the encrypted values. To perform multiplications quickly, number-theoretic transforms (NTTs) are utilized to convert ciphertexts into their point forms. Under current homomorphic encryption schemes [6, 8, 7], ciphertexts are noisy encodings of plaintext values, and operations increase the noise present in a ciphertext. After too much noise growth, ciphertexts are no longer decryptable, so noise growth is controlled through steps called modulus and key switching.

However, these steps require switching out of the NTT form of a ciphertext, which uses a significant amount of computer time. So, we introduce PrimesPIR, a PIR protocol that does not use modulus and key switching, but rather uses the structure of the PIR answer procedure to minimize noise growth. Because of this, it is able to keep ciphertexts in NTT form throughout the computation process, reducing computation time.

In addition, keeping ciphertexts in NTT form allows us to parallelize our scheme along the length of the polynomials used in homomorphic encryption, allowing for a GPU implementation of our protocol.

We also design PrimesPIR to support two additional operations: a sparse answer procedure that generates a response for a sparse database, and an update procedure that takes in a query, database, database modification, and previous response and generates a response to the modified database.

This feature allows PIR to be effectively used for services such as anonymous email retrieval, since the queries to users' mailboxes can be updated when new messages are sent, and empty mailboxes can be created without significantly impacting response time.

So, in summary PrimesPIR aims to:

- Speed up PIR answering time by removing conversions out of NTT form
- Allow for parallelization across multiple cores
- Support fast updates to responses when the database is modified.

2 Background

We now more formally introduce PIR, and the homomorphic operations that make current schemes possible.

2.1 Homomorphic Encryption

Though there are several FHE schemes [6, 7, 8], in this work we will focus on BGV [6], which is founded on the hardness of the Ring Learning with Errors problem.

Notation. We will use elements of the polynomial ring $R = \mathbb{Z}[X]/[X^n + 1]$. Since elements of this ring have n coefficients, we say that n is the polynomial (poly) length. We use R_q to denote R/qR, for some integer q which is coprime to n. In practice, n is a power of 2 so that number theoretic transforms can be computed efficiently, as will be discussed later.

Ring Learning With Errors (RLWE) is a generalization of the learning with errors problem that states that for a uniformly random secret key $s \leftarrow R_q$, errors e_i drawn from a distribution E, and $a_i \leftarrow R_q$, the pairs $(as + e, -a)$ are indistinguishable from pairs of uniformly random values, as long as certain parameter constraints are satisfied.

[12] provides specific estimates of the security levels achieved by various parameter combinations. Generally, smaller values of q provide more security. Additionally, larger polynomial lengths have larger maximum values of q to maintain the same level of security. For example, a polynomial length of 2^{11} requires q to have at most 53 bits, while a polynomial length of 2^{12} can have q have up to 107 bits in order to maintain 128 bits of security.

Ciphertext Representation. To encrypt the message m, we can generate the two-term BGV ciphertext $(as + te + m, -a)$ for a plaintext modulus t which is relatively prime to q. The security of sending such ciphertexts follows from RLWE: $(as + e, -s)$ is indistinguishable from random, therefore $(ast + te, -at)$ is indistinguishable from random. As a is uniformly sampled from R_q, so is at, and therefore $(as + te, -a)$ is also indistinguishable from random pairs under the RLWE assumption. For a message m, this means the BGV ciphertext $(as + te + m, -a)$ is also indistinguishable from random pairs of values, and is therefore secure.

All freshly encrypted BGV ciphertexts have two terms, but after multiplications it is possible for the number of terms to grow. Generally, to decrypt a BGV ciphertext $c = (f_0, f_1, \ldots)$, a client computes $f_0 s^0 + f_1 s^1 + f_2 s^2 + \ldots$. For the two-term ciphertext $(as+te+m, -a)$, this means that the ciphertext decrypts to $as+te+m-as = te+m$. Note that the error is multiplied by t, meaning that the value of $m \bmod t$ is preserved as long as $te < q$. This has two important effects: Firstly, plaintexts in BGV are transmitted modulo t, as values larger than that would be affected by the random error. So, t is called the plaintext modulus. Secondly, it is important that the error of a message does not grow to be larger than q, since then $te \bmod q$ would interfere with the message bits. This means that having larger values of q, and therefore larger values of n, can help prevent error overflow.

Note that to encrypt a BGV plaintext, we just generate the element (m). Additionally, for ciphertexts with more than two terms multiplication is defined to preserve this decryption property; computing $f_0 s^0 + f_1 s^1 + f_2 s^2 + \ldots$ will result in $m + te$ for some error e.

Addition. Homomorphic additions allow us to compute the sum of two encrypted values. To add two ciphertexts $c_0 = (f_0, f_1, \ldots)$ and $c_1 = (g_0, g_1, \ldots)$ we add them term-wise to get $(f_0 + g_0, f_1 + g_1, \ldots)$. Note that this decrypts to:

$$(f_0+g_0)s^0+(f_1+g_1)s^1+\ldots = (f_0s^0+f_1s^1+\ldots)+(g_0s^0+g_1s^1+\ldots) = \text{Dec}(c_0)+\text{Dec}(c_1)$$

Since the decryptions of the c_0 and c_1 have noise, and are added, noise grows additively after addition.

Multiplication. After homomorphic multiplication, we want the decryptions of the two ciphertexts to have been multiplied. Since $\text{Dec}(c_0) = f_0s^0 + f_1s^1 + \ldots + f_as^a$ and $\text{Dec}(c_1) = g_0s^0 + g_1s^1 + \ldots + g_bs^b$,

$$\text{Dec}(c_0) \cdot \text{Dec}(c_1) = f_0g_0s^0 + (f_1g_0 + f_0g_1)s^1 + \ldots + f_ag_bs^{a+b}$$

$$\Rightarrow c_0 \cdot c_1 = (f_0g_0, f_1g_0 + f_0g_1, \ldots, f_ag_b)$$

Notably, the size of the ciphertext increases after multiplication. Additionally, if $\text{Dec}(c_0) = te_0 + m_0$ and $\text{Dec}(c_1) = te_1 + m_1$, this means that $\text{Dec}(c_0 \cdot c_1) = e_0e_1t^2 + e_0m_1 + e_1m_0 + m_0m_1$ So, though the message has been multiplied, so has the error. Additionally, the error has been further multiplied by an additional factor of t. In order to keep track of the error (or noise) in various ciphertexts, we introduce the following two terms:

Definition 1 (Fresh Ciphertext)**.** A fresh ciphertext is a ciphertext which is a direct encryption of a message, with no operations performed on it. Its error is indistinguishable from a random element of E. A fresh ciphertext has a noise level of 1.

Definition 2 (Noise Level)**.** The Noise Level of a ciphertext is the bits of noise of that ciphertext divided by the bits of noise in a fresh ciphertext. In practice, this means that the Noise Level of $c_0 + c_1$ is roughly the maximum of their individual Noise Levels, while the Noise Level of c_0c_1 is the sum of the noise terms of c_0 and c_1.

We now look at several commonly used optimizations for homomorphic multiplication.

CRT Form. In order to do homomorphic multiplications, we have to multiply two elements of R_q, which involves multiplying many elements of $\mathbb{Z}_q$, as all of the coefficients of each polynomial are in $\mathbb{Z}_q$. Since q typically has far more than 64 bits, this would require multiple int-by-int multiplications. To avoid doing that, we have $q = q_1q_2q_3\ldots$, where all q_i are pairwise relatively prime and can fit within a single integer. So, $a \leftarrow \mathbb{Z}_q$ is stored as $a_1 \bmod q_1, a_2 \bmod q_2, \ldots$, which is a's CRT form. Multiplications (and additions) are then done modulo all q_i, which takes fewer overall multiplications. By the Chinese remainder theorem, this produces a set of modular relations with the correct (and unique) value for $ab \bmod q$.

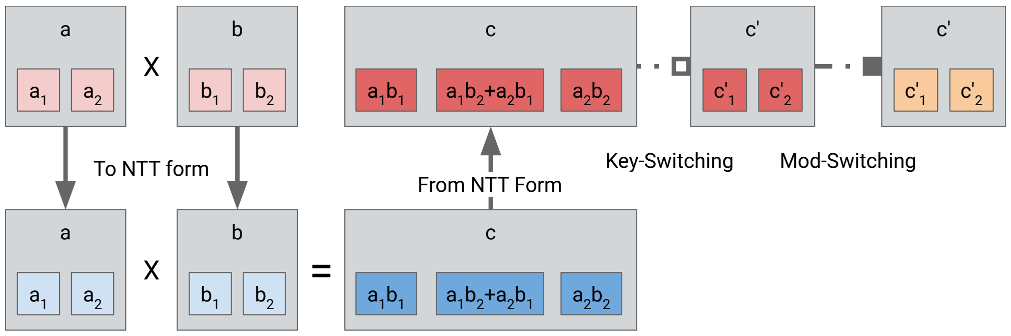

Figure 1: An FHE multiplication optimized using conversions to NTT form, key-switching, and modulus-switching. Grey boxes represent ciphertexts, and their sub-boxes represent elements of the ring $\mathbb{Z}[X]/[X^n + 1]$.

NTT Form. Coefficient-wise multiplication of two polynomials in R_q would take $O(n^2)$ multiplications, which is inefficient. Instead, the polynomials are evaluated at n points, and then multiplied at each of the n points, which takes $O(n)$ operations. Since polynomials in R_q have n coefficients, the original coefficients can be recovered from the n points.

However, in order for the conversion to and from point form to be computable in $n\log(n)$ time, the polynomial has to be evaluated on the $2n$-th roots of unity, and $2n$ must be a power of 2. Transforming such a polynomial to point-form is referred to as a Number-Theoretic Transform (NTT). For these roots to exist, all q_i are chosen to be primes that are one more than a multiple of $2n$.

By converting the elements of R_q to CRT and NTT forms, we can speed up homomorphic multiplications. However, after multiplications ciphertexts both have an increased size and increased noise. To prevent this, modulus and key switching are typically used after multiplications.

Modulus Switching reduces the noise of the product by treating the polynomial coefficients in $\mathbb{Z}_q$ as elements of $\mathbb{Z}$, and then doing integer division [1] on the coefficients. Though division is possible in NTT form, integer division is not, since rounding the point-form of a polynomial doesn't correspond to rounding the coefficients of a polynomial. So, elements are typically conveted out of NTT form for modulus switching to occur.

Key Switching. On a high level, key switching encrypts s^2 under s, and then uses that to convert the s^2 coefficient of the ciphertext into a term that is linear in s. This allows the degree of the ciphertext to be reduced back to 1 after multiplications. However, in order for this to happen without excessive noise growth, key-switching does a binary decomposition of the coefficients of the polynomial. This cannot happen when the polynomial is in point-form, so key-switching requires switching out of NTT form.

[1]Some additional steps are taken to preserve the correctness of the message

2.2 Private Information Retrieval

Private information retrieval (PIR) aims to retrieve a database item without revealing which item was retrieved. Trivially, this could be done by sending the entire database to a user, thus transmitting the desired item. However, this would result in a prohibitively large network overhead, since the response size would be the size of the database.

So, PIR aims to decrease network costs by compressing the d-element database into a ciphertext that contains the information of the desired database index. Traditionally, this is done using three operations – the client can generate a query or decode a response, and the server must have a procedure to "answer" a query by generating a response that can be decrypted by the client. Formally, there are three procedures such that:

$$\begin{aligned}\mathrm{Query}(\mathrm{idx}) &= \mathrm{query}\\ \mathrm{Answer}(\mathrm{query}, \mathrm{db}) &= \mathrm{response}\\ \mathrm{Decode}(\mathrm{response}) &= \mathrm{database}[\mathrm{index}]\end{aligned}$$

In practice, the answer procedure is the most expensive: in order to hide which database element was retrieved, all database elements must be involved in the computation process–if any element was not involved, it would leak information about the retrieved index. Because of this, we will focus on how the server compresses the database down to a smaller ciphertext.

A linear PIR protocol accomplishes the goal of compression using a one-hot query $Q = (c_0, c_1, \ldots, c_i, \ldots, c_{d-1})$, where c_i encodes a 1 for the desired database index and 0 for all other indices. The database $D = (p_0, p_1, \ldots, p_{d-1})$ is then multiplied by the query to produce the result $R = \sum_{i=0}^{d-1} c_i p_i$. For all non-desired elements, $c_i p_i = 0$ since $c_i = 0$, and for the desired element $c_i p_i = p_i$ as $c_i = 1$, so this sum generates a ciphertext encoding just the desired element.

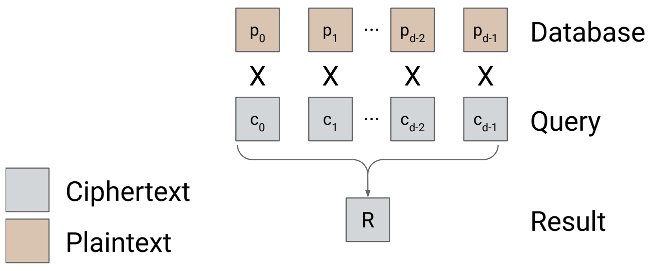

Figure 2: A linear PIR scheme. Note that homomorphic encryption is used so that operations can be performed using the query values.

However, linear PIR uses a query as large as the database, which has prohibitive network costs. SealPIR [4] solves this issue with a query-unpacking operation that allows all of the P coefficients of a query ciphertext polynomial to be used to store 1s or 0s during transmission. So, only $\frac{d}{P}$ ciphertexts are needed to transmit the query.

However, the query-unpacking process is slow, and adds $O(d)$ operations to the answer procedure, significantly slowing down the answer computation time.

Two-dimensional databases can reduce the query size while maintaining a relatively fast answer procedure, as shown in Figure 3. Formally, if we have a database $D = (p_{0,0}, p_{0,1}, \ldots, p_{0,m-1}, p_{1,0}, p_{1,1}, \ldots, p_{1,m-1}, \ldots p_{n-1,m-1}$ with $mn = d$, then we can have query $Q = (c_{0,0}, c_{0,1}, \ldots c_{0,m-1}, c_{1,0}, c_{1,1}, \ldots c_{1,n-1})$, where both $(c_{0,0}, \ldots c_{0,m-1})$ and $(c_{1,0}, \ldots c_{1,n-1})$ are one-hot vectors. Then, the result $R = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} c_{0,i} c_{1,j} p_{i,j}$ is computed, and it encodes the desired element, since all other elements are multiplied by zero.

Note that in practice this is computed as $R = \sum_{i=0}^{n-1} c_{0,i} \left(\sum_{j=0}^{m-1} c_{1,j} p_{i,j} \right)$, meaning $d + n$ multiplications (and not $2d$) are now required to compute the result. In addition, the two-dimensional structure means that in an m by n database, only $m + n$ expanded query ciphertexts are needed instead of mn.

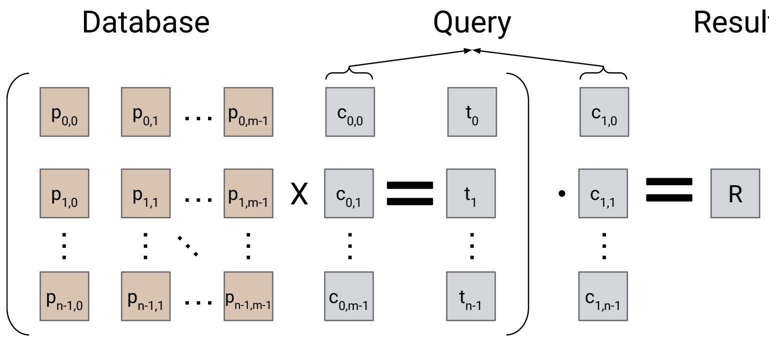

Figure 3: Reformatting the database into a rectangular matrix results in a smaller query at the cost of slightly more multiplications.

Further increasing the number of dimensions of the database eventually results in a procedure we call "**folding**", where each dimension has size two, as shown in Figure 4. Since the query for each dimension is a one-hot vector, one of the two ciphertexts will always be an encoding of 0, and the other will always encode 1. This means one of the ciphertexts is always 1 minus the other, so in practice only one ciphertext is sent. Therefore, the query length for a folding PIR protocol is only $\log_2(d)$, and from a computational perspective $2d$ multiplications are required.

Such a scheme is partially used by Spiral [11], which has one large dimension and then many small dimensions of size two. It also improves on PIR by composing two FHE schemes to yield smaller noise after multiplications. However, we do not consider such optimizations.

3 Updatability

In addition to the operations Query(idx), Answer(query,db), and Decode(response), we introduce an *update* operation Update(response,query,index,new,db) that can modify a

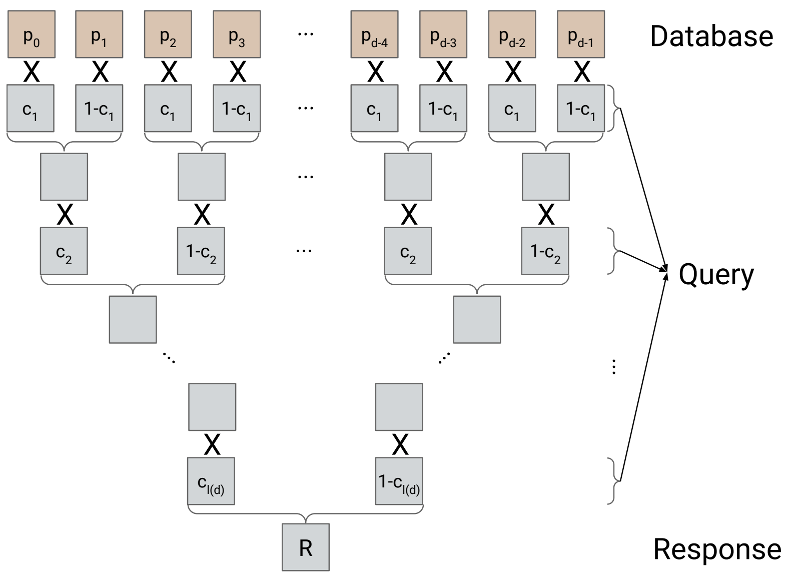

Figure 4: “Folding”: each ciphertext folds the database in half

PIR response in response to changing the database at a given index to a new value. Specifically a valid update procedure must satisty:

$$\text{Decode(Update(Answer[query,db],idx}_2\text{,query,new,db))} = \begin{cases} \text{new} & \text{if idx}_2 = \text{idx}_1 \\ \text{db[idx}_1\text{]} & \text{if idx}_2 \neq \text{idx}_1 \end{cases}$$

where

$$\text{query} = \text{Query(idx}_1\text{)}$$

In addition, we require that an Update procedure be infinitely repeatable; one should be able to perform an arbitrary number of updates.

Note that any update procedure implicitly generates a protocol for evaluating a query to a sparse database, since we can start with a database with every element equal to zero, and then update that database into one with our desired elements.

3.1 Linear PIR (SealPIR with d=1)

Firstly, we define an update procedure for a linear PIR protocol. If our answer procedure computes the result $R = \sum_{i=0}^{d-1} c_i p_i$, then a valid update procedure is:

$$\text{Update(response,query,idx,new,db)} = \text{response} + \text{(new-db[idx])} \cdot \text{query[idx]}$$

as shown in Figure 5.

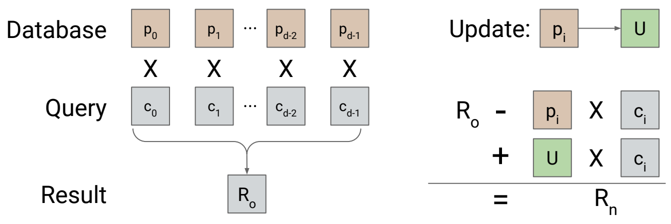

Figure 5: Updating a linear PIR protocol.

The result of this update is equivalent to the result that would have been produced by running the answer procedure on the updated database, so the correctness of the decoding of the updated response is guaranteed by the correctness of the original answer procedure.

Notably, this procedure is fast: an update can be performed in $O(1)$ time.

3.2 Higher-dimensional PIR

Now, we consider a two-dimensional database. If our answer procedure computes the result $R = \sum_{i=0}^{n-1} c_{0,i} \left(\sum_{j=0}^{m-1} c_{1,j} p_{i,j}\right)$, then a naive update procedure would update the element $p_{i,j}$ to U by computing the answer $R' = R + (U - p_{i,j})c_{0,i}c_{1,j}$.

However, this procedure causes long-term noise growth, meaning it is not infinitely repeatable, and therefore it does not constitute an update procedure:

Because key- and mod-switching alter the noise in a ciphertext, the noise contained in $c_{0,i} \left(\sum_{j=0}^{m-1} c_{1,j} p_{i,j}\right)$ is not the same as the noise in $\left(\sum_{j=0}^{m-1} c_{0,i} c_{1,j} p_{i,j}\right)$. Note that a naive update procedure can update the answer procedure $R = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} c_{0,i} c_{1,j} p_{i,j}$. However, the noise of the optimized version of this procedure differs, and each time an update is performed the noise increases, resulting in a long-term noise growth that eventually prevents decryption of the result.

To actually update the answer procedure which computes $R = \sum_{i=0}^{n-1} c_{0,i} \left(\sum_{j=0}^{m-1} c_{1,j} p_{i,j}\right)$, one has to compute $R' = R - c_{0,i} \left(\sum_{k=0}^{m-1} c_{1,k} p_{i,k}\right) + c_{0,i} \left(\left(\sum_{k=0}^{m-1} c_{1,j} p_{i,k}\right) - c_{1,j} p_{i,j} + U\right)$, where $p_{i,j}$ has been updated to the element U.

Instead of running in $O(1)$ time as with a linear scheme, this runs in $O(\frac{d}{n})$ time, where n is the size of the first dimension. Practically, this means protocols such as Spiral [11] and SealPIR [4] with a dimension of at least two cannot be updated rapidly.

We now turn to resolving this difficulty through PrimesPIR.

4 PrimesPIR

In current PIR schemes [4, 11], all multiplications are followed by modulus and key-switching steps to reduce the ciphertext size, as shown in Figure 1. Due to the rounding involved in modulus and key-switching, ciphertexts need to be switched out of NTT form after each multiplication, reducing the speed of these schemes.

In PrimesPIR, we implement a folding protocol that avoids the mod- and key-switching steps of FHE multiplicaitons, resulting in the multiplications shown in Figure 6. Because key- and modulus-switching are skipped, there is no need to convert ciphertexts out of NTT form, so expensive NTT conversions are removed from the NTT process.

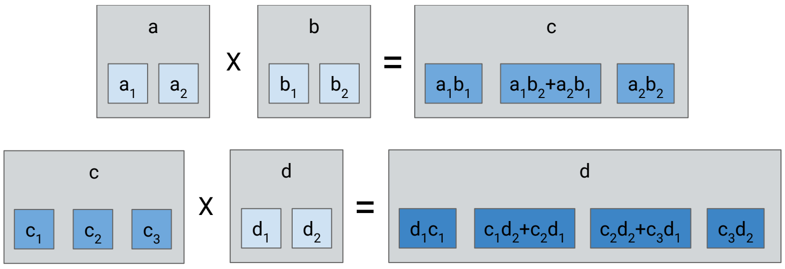

Figure 6: Multiplying ciphertexts without key- and modulus- switching means they can be left in NTT form, at the cost of an increased ciphertext size.

After each multiplication that does not use key-switching, the ciphertext size grows, so subsequent multiplications require more operations. However, since the database is 'folded' in half after each multiplication, there are fewer of these larger multiplications, meaning that in total only $8d$ polynomial multiplications are required, which is twice as much as the $4d$ required by a folding scheme with key- and mod-switching (folding requires $2d$ ciphertext multiplications). Each of these $8d$ multiplications is faster than those in a traditional PIR scheme due to the lack of an NTT transform. In addition, these $8d$ multiplications allow for better use of computer architecture through parallelization.

For simplicity, we do not use the query-packing approach of Spiral and SealPIR. Since we have so few query ciphertexts (only $\log(d)$, where d is the database length), we estimate that adding query-packing would reduce query sizes by a factor of $\log(d)$ without significantly increasing speed.

Parallelization. Because all of our multiplications are done in NTT form, our scheme can be run independently on each of the P different point evaluations of the polynomials involved in the computation process. This means that our scheme can easily be implemented on a large number of threads, or even on a GPU.

Update Procedure. Before, we noted that as a result of key- and modulus switching, PIR protocols that structure their databases as higher-dimensional objects cannot be updated. Since PrimesPIR does not use key- and modulus-switching, we are able to efficiently update our protocol in $\log_2(d)$ time, since there are $\log_2(d)$ dimensions in our folding protocol.

5 Evaluation

When implementing this scheme, we only focused on speeding up server answer times, and not on reducing client-side query generation, decryption, or network costs. So, we used lattigo [1] for all client-side operations.

In addition, we implemented the answer procedure for our scheme in C++, with a second version using CUDA C++.

In Figure 7, we display the time it took to run the answer procedure, query size, and response size for SealPIR, the C version of Spiral, and PrimesPIR for various database sizes and 32KB elements.

Note that of the three schemes, PrimesPIR can be updated in $O(\log_2(d))$ time, SealPIR can be updated in $O(1)$ time when a one-dimensional database structure is used, and Spiral requires $O(\sqrt{d})$ time for an update, since Spiral uses key and modulus switching on a high-dimensional database.

Of the four variants of Spiral, we compare to SpiralStream since it also doesn't pack queries into a single ciphertext.

Number of Items	SealPIR (d=1)*			SpiralStream			Our Scheme		
	Query Size (MB)	Response Size (MB)	Answer Time (s)	Query Size (MB)	Response Size (MB)	Answer Time (s)	Query Size (MB)	Response Size (MB)	Answer Time (s)
2^{12}	*	*	*	3.9	.074	0.35	18.9	10.2	8.01
2^{11}	.089	.185	9.25	2	.074	0.23	17.3	9.4	3.96
2^{10}	.089	.185	4.69	2	.074	0.16	15.7	8.7	1.82
2^{9}	.089	.185	2.36	2	.074	0.16	14.2	7.9	0.89
2^{8}	.089	.185	1.21	2	.074	0.16	12.6	7.1	0.44
2^{7}	.089	.185	0.67	2	.074	0.16	11	6.3	0.24

Figure 7: Benchmarks for Spiral, Seal, and our scheme. All trials were run on my laptop (6-core AMD Ryzen 5 5500U, 20GB RAM, 2.10 GHz) using a single core. Each item stored 32KB of data. *Data for Seal for was gathered by retrieving four 8KB items. Using Seal with $d = 2$ results in an answer time that is $\tilde{4}$x faster, but significantly impedes updatability.

Unfortunately, PrimesPIR's network costs are much larger than current schemes. Query costs would continue to be larger than Seal's even with query packing, and our response size is significantly larger than Seal or Spiral's. This is because after each multiplication, the ciphertext size increases and cannot be decreased without key-switching. In order to reduce the response size, we would have to either reduce the noise growth (letting us use a smaller ratio of $\frac{q}{t}$, thereby reducing ciphertext size), or reduce the multiplicative depth (which would also reduce noise growth).

Additionally, though it is not reflected in these results, a downside of our scheme is that it requires large parameters due to larger-than-expected noise growth. So, unlike Spiral, our minimum ciphertext size is quite large, meaning our scheme would struggle to support PIR applications with small database and response sizes, since we would still have to support that ciphertext size.

However, our answer computation time is very promising. On a single core, Our scheme is already $\tilde{2}$x faster than SealPIR, and is only an order of magnitude slower than Spiral without composing multiple FHE schemes for smaller noise growth.

Number of Items	1 thread	12 threads		64 threads		GPU	
	Time (ms)	Time (ms)	Speedup	Time (ms)	Speedup	Time (ms)	Speedup
2^8	1500	144	11x	69	22x	51	30x
2^7	750	71	11x	36	21x	9	83x

Figure 8: Answer times, and the speedup over a single thread achieved by our protocol. Trials were run on a different machine than those in Figure 7–these trials used a NVIDIA GeForce RTX 3090, had a 2.2GHz, 32-Core processor, and 251 GB of RAM.

With more cores, and even a GPU, our scheme shows considerable speed increases, as demonstrated in Figure 8. We are currently working on expanding our GPU implemtation to work for larger database sizes.

6 Conclusion

PrimesPIR provides a method of reducing the expensive ansewr time of private information retrieval, while also being compatible with sparse databases and use cases involving updates to the database. However, it creates large network costs that may be prohibitive in practical settings.

In the future, we would like to implement further optimizations using the idea of avoiding the key- and modulus-switching steps of private information retrieval:

- Gathering timing data for the update procedure and sparse database protocol
- Extending our GPU implementation to work on larger databases

- Further optimizations to the GPU version of our scheme
- Reducing the network costs of our protocol using strategies explored by [4].
- More rigorous analysis of the noise growth achieved by update procedures.

7 Acknowledgments

I would like to thank Simon Langowski for his guidance and support of this project, and MIT PRIMES for making this project possible.

References

[1] Lattigo v4. Online: `https://github.com/tuneinsight/lattigo`, August 2022. EPFL-LDS, Tune Insight SA.

[2] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private Information Retrieval for Everyone. *Proceedings on Privacy Enhancing Technologies*, avril 2016:155–174, April 2016.

[3] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 313–329. USENIX Association, July 2021.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. Cryptology ePrint Archive, Paper 2017/1142, 2017. `https://eprint.iacr.org/2017/1142`.

[5] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, Savannah, GA, November 2016. USENIX Association.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Paper 2011/277, 2011. `https://eprint.iacr.org/2011/277`.

[7] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Paper 2016/421, 2016. `https://eprint.iacr.org/2016/421`.

[8] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. `https://eprint.iacr.org/2012/144`.

[9] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 91–107, Santa Clara, CA, March 2016. USENIX Association.

[10] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. Incremental Offline/Online PIR. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1741–1758, Boston, MA, August 2022. USENIX Association.

[11] Samir Jordan Menon and David J. Wu. Spiral: Fast, high-rate single-server pir via fhe composition. Cryptology ePrint Archive, Paper 2022/368, 2022. `https://eprint.iacr.org/2022/368`.

[12] Johannes Mono, Chiara Marcolla, Georg Land, Tim Güneysu, and Najwa Aaraj. Finding and evaluating parameters for bgv. Cryptology ePrint Archive, Paper 2022/706, 2022. `https://eprint.iacr.org/2022/706`.

[13] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. Cryptology ePrint Archive, Paper 2021/1081, 2021. `https://eprint.iacr.org/2021/1081`.

[14] Sarvar Patel, Joon Young Seo, and Kevin Yeo. Don't be dense: Efficient keyword pir for sparse databases. Cryptology ePrint Archive, Paper 2023/466, 2023. `https://eprint.iacr.org/2023/466`.