Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian

XPIR : Private Information Retrieval for Everyone

Abstract: A Private Information Retrieval (PIR) scheme is a protocol in which a user retrieves a record from a database while hiding which from the database administrators. PIR can be achieved using mutually-distrustful replicated databases, trusted hardware, or cryptography. In this paper we focus on the later setting which is known as single-database computationally-Private Information Retrieval (cPIR). Classic cPIR protocols require that the database server executes an algorithm over all the database content at very low speeds which impairs their usage. In [1], given certain assumptions, realistic at the time, Sion and Carbunar showed that cPIR schemes were not practical and most likely would never be. To this day, this conclusion is widely accepted by researchers and practitioners. Using the paradigm shift introduced by lattice-based cryptography, we show that the conclusion of Sion and Carbunar is not valid anymore: cPIR is of practical value. This is achieved without compromising security, using standard crytosystems, and conservative parameter choices.

Keywords: cPIR, Lattice-Based Cryptography

DOI 10.1515/popets-2016-0010
Received 2015-08-31; revised 2015-11-19; accepted 2015-12-02.

NOTE: XPIR is free (GPLv3) software and available at https://github.com/XPIR-team/XPIR and the evolution of the underlying fast-lattice library at https://github.com/quarkslab/NFLlib

1 Introduction

Homomorphic encryption has followed a curious path in the history of cryptography. Since the very beginning of public key cryptography, it has been presented as a holy grail able to provide the most incredible and powerful applications. Yet, even with the recent breakthroughs due to lattice based cryptography, homomorphic encryption is almost never used in practice.

Among the potential applications of homomorphic encryption, one of the oldest and most emblematic is single-database computationally-Private Information Retrieval. With such a protocol, a user can retrieve a record out of n from a database, without having to reveal which one to the database administrators (security being derived from computational hardness assumptions). A trivial way to obtain such privacy is to simply download the whole database and to dismiss the elements the client is not interested in.

Private Information Retrieval (PIR) schemes aim to provide the same confidentiality to the user (with regard to the choice of the requested element) that downloading the entire database does, with sub-linear communication cost. PIR was introduced by Chor, Goldreich, Kushilevitz, and Sudan in 1995 [2]. They proposed a set of schemes to implement PIR through replicated databases that provide users with information-theoretic security, so long as some of the database replicas do not collude against the users.

Note, however, that PIR schemes do not ensure database confidentiality: a user can retrieve more than a database element using a PIR scheme without the database being aware of it. A PIR scheme ensuring that users retrieve a single database element with each query is called a Symmetric PIR (or SPIR) scheme. Generic transformations exist from PIR to SPIR but this is beyond the scope of this paper (see [3]).

Performance in initial PIR schemes was evaluated when retrieving single-bit elements from a database. All recent protocols [4–6] are evaluated when retrieving multi-bit elements from a database. This setting was initially called Private Block Retrieval [2] but nowadays both terms are used interchangeably. All-or-nothing--disclosure-ofsecrets (ANDOS) protocols (e.g. [7]) are simply multi-bit SPIR protocols.

In this paper, we focus on PIR schemes that do not need the database to be replicated, and whose security is based on the computational security of a cryptographic algorithm, which are usually called single-database computationally-Private Information Retrieval (cPIR) schemes. This replaces the assumption of having replicas which do not collude by a computational security assumption. However, this comes at a price.

Carlos Aguilar-Melchor: Univ de Toulouse, IRIT, France, E-mail: carlos.aguilar@enseeiht.fr
Joris Barrier, Marc-Olivier Killijian: CNRS, Univ de Toulouse, LAAS, France
Laurent Fousse: Univ de Grenoble, LJK, France

1.1 Performance Issues in cPIR

A major issue with computationally-private information retrieval schemes is that they are computationally expensive. In order to answer a query, a database must process all of its entries. If a protocol does not process some entries, the database will learn that the user is not interested in them. This would reveal to the database partial information on which entry the user is interested in, and therefore, it is not as private as downloading the whole database and retrieving locally the desired entry. The computational cost for a server replying to a cPIR query is therefore linear on the database size. More precisely, in [8], Lipmaa proves that using a particular representation, this lower bound is slightly sub-linear (in $O(n/log(log(n)))$. Moreover, most of the schemes have a very large cost per bit in the database, a multiplication over a large modulus. This restricts both the database size and the throughput shared by the users and thus, limits their usage for many databases as well as for other applications such as private keyword search [9].

In NDSS'07, Sion and Carbunar presented a paper on cPIR practicality [1]. They showed that the existing, number theory based, cPIR protocols were not practical and that it was always faster to send the whole database than to compute a cPIR reply. Indeed, basing the security of the underlying number theoretic encryption schemes on the hardness to factor a 1024 bit RSA modulus, one could not expect a cPIR scheme to process the database at more than a megabit per second. Sending the whole database over most of the current Internet connections is at least an order of magnitude faster (and generally two orders of magnitude in local area networks). They also argued that this performance gap would continue as long as usual laws on computational power and bandwidth evolution do.

Focused issue. As in [1], we tackle the issue of practical usage of cPIR. The main performance metric we use is the time needed for a client to retrieve an element privately, supposing one or more clients are querying a server with a commodity-CPU and can exchange data with the server at various speeds (xDSL, FTTH, etc.). We consider that the client is ready to pay a significant overhead for privacy, and compare the time needed using different approaches (trivial full database download, number-theory based cPIR, lattice-based cPIR).

1.2 Related Work

As number theoretic approaches failed to provide efficient cPIR schemes, some alternatives were explored [4–6], but all of them were based on non-standard problems and have been broken [10–14].

The schemes of Aguilar et al. [5] and Trostle and Parrish [6] represent the state-of-the-art in efficient private information retrieval, allowing to reach processing speeds of hundreds of megabits per second on high-end CPUs and up to one gigabit per second on GPUs [15]. These works total about eighty citations, and have been used as a fundamental building block (or as a benchmark) in major and recent venues such as Usenix Security [16] (2011), NDSS [17, 18] (2013, 2014), and PETs [19–22] (2010, 2012, 2014). This paper presents a potential replacement for them with some additional features: security is based on a standard problem, Ring-LWE [23], with conservative parameters choices; multi-gigabit per second processing throughput on an average CPU; and an auto-optimizer to simplify its usage by non specialists.

A noteworthy exception to this list of schemes is the cPIR scheme of Gasarch and Yerukhimovich [24] which relies on a lattice-based standard encryption scheme [25]. However, this underlying encryption scheme has an extremely large expansion factor (large ciphertexts encoding only a few bits) that compromises the efficiency of the cPIR scheme.

Alternatives to cPIR. Oblivious RAM (ORAM) protocols, which are used to access (and write on) a database privately, can handle efficiently databases of many Terabits. However, ORAM and PIR protocols are used for different applications and cannot be exchanged. Indeed, in the ORAM setting the database content is encrypted data outsourced from the user. ORAM cannot be used directly to privately download elements from a public database which is the paradigm of PIR, for example accessing a movie database *à la* Netflix. In ORAM the data in the server is encrypted and user must know this key to access the data *and to hide their access patterns*. With PIR, no shared secret is needed, i.e. each user has an independent key whose objective is just to hide his own choice. ORAM is thus able to deal with larger databases but can only be used in personal or group-with-a-shared-key settings (e.g. for an encrypted and shared filesystem in the cloud).

It is possible to transform an ORAM protocol into a PIR protocol using a hardware module (e.g. see [26]). When using a trusted hardware module is an acceptable constraint, such protocols allow clients to send expressive queries (interpretable by the module) to define the elements to be retrieved and have very low overhead.

Instead of using a trusted hardware module, one can build efficient PIR protocols using replicated databases as shown by Chor, Goldreich, Kushilevitz, and Sudan [27] and by Olumofin and Goldberg [28]. If replicating a database ont not colluding servers is an acceptable constraint, such protocols allow to retrieve data privately with very small computational and communication overhead.

Interesting recent work by Devet and Goldberg [22] uses replicated-database PIR and cPIR jointly to achieve high performance results without compromising security when databases collude. This paper requires the database to be replicated, and therefore is very different from our setting. The proposed protocol uses Aguilar et al.'s cPIR scheme [5] as a building block, replacing it with our protocol would result in a performance boost and provide a secure instance of their construction.

Works considering only computational or communication costs. In the Oblivious Transfer setting [29], the objective is to limit the computational cost for the user and the database without considering communication efficiency. The whole database is sent encrypted to the client together with some extra information, with the added benefit that the server is guaranteed that the client can retrieve information about one element only per query.

Some cPIR protocols focus only on communication efficiency without considering computational costs. In [30], a *communication* efficient cPIR protocol, from an asymptotic perspective, is built based on a fully-homomorphic scheme. The underlying encryption scheme we use is just an additively-homomorphic building block in [30] but our objective is to allow users to retrieve elements faster than the trivial solution of downloading the entire database in realistic settings. This implies taking into account computational and communication constraints.

In [31], an implementation of a fully-homomorphic encryption based scheme is given. The contribution of this scheme is on the communication overhead, which they show to be very small in some settings (when multiple database elements are retrieved). Computational costs are considered but this paper does not give a contribution in this sense that the database is at best processed at 20Kbits/s which is below the processing throughput of classic, number-theory based, cPIR schemes. As already noted, sending the whole database can be done at a higher throughput in most settings allowing to retrieve an element privately much faster.

Finally, in [32] Kiayias et al. present a new cPIR scheme based on a number-theoretic homomorphic encryption scheme. The communication performance is extremely interesting, but again computational performance is still an issue. Adapting such work to lattice-based encryption schemes would be of great practical and theoretical interest.

1.3 Contributions and Roadmap

First and foremost this paper shows that cPIR is a usable primitive in a large variety of settings, with standard security assumptions and conservative parameter choices. Section 4 is dedicated to proving this assertion. This contradicts the main result from Sion and Carbunar [1], which was the reference on cPIR usability. The analysis of Sion and Carbunar remains correct, but one of their main assumptions (that cPIR would be based on number-theoretic schemes) does not need to be true any more, thanks to the arrival of lattice-based homomorphic encryption schemes.

Second, we provide a highly efficient and usable Ring-LWE cPIR implementation. From a fundamental point of view our contribution is to show that Ring-LWE operations required for reply generation benefit extensively from pre-processing, using adequately known techniques (*Fast Fourier Transform*-like representation) and introducing new ones (Newton quotient pre-computation). From a practical point of view the implementation we provide offers multi-gigabit processing throughput on a commodity CPU, and an optimizer to automatically tune the system (hardware/network/application/security).

In Section 2, we present the basic tools a reader should be comfortable with in order to understand the rest of the paper: homomorphic encryption, which allows to compute over encrypted data; the objectives and the classical approaches to obtain private information retrieval protocols; and a special setting of cPIR called private keyword searching, in which instead of retrieving elements by their index as usually done in cPIR we retrieve elements based on the keywords they are associated to. In Section 3, we first present an overview of our protocol and describe the optimization process, and then we present our fast Ring-LWE based cPIR, the proposed algorithmic optimizations and performance results for the basic operations: query generation (encryption), database importation, reply generation, and reply extraction (decryption). In Section 4 we present a more high level performance analysis of our library. The objective of this section is twofold: show how our library behaves on a large variety of settings and prove that cPIR is better than trivial PIR in most cases, contradicting the main result from Sion and Carbunar [1]. The main part of the paper ends with a Conclusion.

2 Basic Tools

2.1 Homomorphic Encryption

Additively homomorphic encryption schemes are defined by four algorithms: KeyGen, to generate keys; Enc the encryption function; Dec the decryption function; Add which takes as input two ciphertexts α_1, α_2 with corresponding plaintexts $db_1, \cdots, db_n$ and outputs a ciphertext α with corresponding plaintext $db_1 + \ldots + db_n$; and Absorb which takes as input some data db and a ciphertext of m and outputs a ciphertext of $m * db$. If $m = 0$, db is erased, and if $m = 1$ it is absorbed.

From the security point-of-view, one must achieve indistinguishability against chosen-plaintext attacks which corresponds to the highest security an homomorphic encryption scheme can achieve (see e.g. [33]). This property offers strong guaranties on ciphertext secrecy as proved by Goldwasser and Micali [34].

We use the Ring-LWE based homomorphic encryption scheme presented in [30]. We describe it below, in order to justify some of our performance results. As in [30] we present the symmetric version and then explain how an asymmetric scheme can be derived.

Notations: $\mathbb{Z}_q$ denotes the set of relative integers modulo q. If S is a set $x \leftarrow S$ represents a uniform sample from S, for a distribution χ, $x \leftarrow \chi$ represents a sample following that distribution. $R_q = \mathbb{Z}_q[X]/ < X^N + 1 >$ represents the polynomials with coefficients over $\mathbb{Z}_q$ such that after each operation they are reduced by division modulo $X^N + 1$. Note that we use uppercase N for the polynomial degree (which is an unusual notation) to distinguish the polynomial degree from the number of elements in the database we will be dealing with in the cPIR protocol. Unless specified otherwise, all scalar operations are mod q. For two polynomials $a, b \in R_q$, $a+b$ is the polynomial obtained by adding their coefficients, $a*b$ is the usual polynomial multiplication reduced modulo $X^N + 1$, and $a \otimes b$ is the polynomial obtained by multiplying their coefficients coordinate-wise.

SKE.ParamGen takes as an input a security parameter k and a maximum number of additions h_a and outputs a set of parameters. For performance reasons we force among the outputs of this function $N \in \{1024, 2048, 4096\}$ and q to be a multiple of 60-bit or 30-bit primes such that each prime is congruent to 1 modulo $2N$ in order to be able to use the NTT (see Section 3.2.2). This function generates parameters following the approach of [35].

A symmetric Ring-LWE encryption scheme

```
SKE.ParamGen($1^k, h_a$):
Input: A security parameter $k$; A number of additions $h_a$
Output: A modulus $q$; A polynomial degree $n$; A distribution $\chi$

SKE.KeyGen($q, N$):
Input: A modulus $q$; A polynomial degree $N$
Output: A polynomial in $R_q = \mathbb{Z}_q[X]/ < X^N + 1 >$
1.  Output: $s \leftarrow \chi$

SKE.Encrypt($s, db$):
Input: A secret key $s$ in the polynomial ring $R_q$; A message $db$ in the polynomial ring $R_q$ with coefficients in $[0..t[$
Output: A ciphertext $(a, b) \in R_q^2$
1.  $a \leftarrow R_q$, $e \leftarrow \chi$
2.  $e' = e \otimes t_v + db$ where $t_v \in R_q$ has its coefficients set to $t$
3.  $b = (a * s) + e'$
4.  Output: $(a, b)$

SKE.Decrypt($s, (a, b)$):
Input: A secret key $s \in R_q$; A ciphertext $(a, b) \in R_q^2$
Output: A plaintext $db \in \mathbb{Z}_t^n$
1.  $e = b - (a * s)$
2.  Output: $db = e \mod t$

SKE.Add($(a_1, b_1), (a_2, b_2)$):
Input: Two ciphertexts, encryptions of $db_1$ and $db_2$
Output: A ciphertext that decrypts to $db_1 + db_2 \mod t$
1.  Output: $(a_1 + a_2, b_1 + b_2)$

SKE.Absorb($db, (a, b)$):
Input: A polynomial $m \in R_q$ with coefficients in $\{0..t-1\}$; A ciphertext $(a, b) \in R_q^2$, encryption of a constant polynomial $m$
Output: A ciphertext which decrypts to $db * m$
1.  Output: $(db * a, db * b)$
```

This scheme is exactly the public-key encryption scheme presented in [28]. It ensures indistinguishability against chosen plaintext attacks if the standard lattice problem Ring-LWE is hard (Theorem 3 and Lemma 4 of [28]). The hardness of Ring-LWE is one of the major assumptions used to build lattice-based cryptosystems, and since its presentation at Eurocrypt'10, it has become probably the most standard and used one.

As the scheme is randomized, ciphertexts must be larger than plaintexts. We note $F > 1$ the associated expansion factor (ciphertext bitsize divided by plaintext bitsize). As a reference, Figure 1 below presents some

Our Ring-LWE public key encryption scheme

```
$\mathbf{ParamGen}(1^k, h_a) = \mathbf{ParamGen}(1^k, h_a)$
$\mathbf{SKE.Decrypt}(sk, (a, b)) = \mathbf{SKE.Decrypt}(sk, (a, b))$
$\mathbf{Add}((a_1, b_1), (a_2, b_2)) = \mathbf{SKE.Add}((a_1, b_1), (a_2, b_2))$
$\mathbf{Absorb}(db, (a, b)) = \mathbf{SKE.Absorb}(db, (a, b))$

$\mathbf{KeyGen}(q, N)$:
Input: A modulus $q$; A polynomial degree $N$
Output: Three polynomials in $R_q = \mathbb{Z}_q[X]/ < X^N + 1 >$
1.  $sk \leftarrow$ SKE.KeyGen$(q, N)$
2.  $pk = (pk1, pk2) \leftarrow$ SKE.Encrypt$(s, 0)$
3.  Output: $((pk1, pk2), sk)$

$\mathbf{Encrypt}(pk = (pk1, pk2), m)$:
Input: A public key $pk = (pk1, pk2)$ with $pk1, pk2$ in the polynomial ring $R_q$; A message $db$ in the polynomial ring $R_q$ with coefficients in $[0..t[$
Output: A ciphertext $(a, b) \in R_q^2$
1.  Define $\chi'$ as $\chi$ with a variance multiplied by $N \log N$
2.  $u, e \leftarrow \chi$, $e' \leftarrow \chi'$
3.  Define $t_v \in R_q$ with all its coefficients set to $t$
4.  $a = pk1 * u + e \otimes t_v$
5.  $b = pk2 * u + e' \otimes t_v + db$
6.  Output: $(a, b)$
```

plaintext and ciphertext sizes for different parameters of our Ring-LWE based encryption scheme.

2.2 Private Information Retrieval

The basic security model for cPIR protocols is to ensure an indistinguishability (i.e. semantic security) property.

Definition 1 (cPIR Protocol). *A (Single-Database) Computationally-Private Information Retrieval Protocol is a probabilistic polynomial-time (PPT) algorithm Γ, that for input k outputs the description of a polynomial-time randomized algorithm $Q(\cdot, \cdot)$ (the query generator), and two polynomial-time deterministic algorithms, $R(\cdot, \cdot)$ (the reply generator), and $X(\cdot, \cdot)$ (the reply decoder), with the two following properties:*

(Correctness) For any number of elements n, index i, and database contents $\{db_1, ..., db_n\}$, and any $(q, s) \leftarrow Q(i, n)$, $X(R(q, \{db_1, ..., db_n\}), s)$ outputs db_i.

(Privacy) For any positive integers $n > i_1 > i_2$, any polynomial-time probabilistic attacker A, there exists a negligible function (i.e. asymptotically smaller than any inverse polynomial) in k, ϵ such that,

$$Pr(A(\alpha, i_1, i_2, \mathcal{P}_{pir}) = i | Q \leftarrow \Gamma[k]; (\alpha, s) \leftarrow Q(i, n)) < 1/2 + \epsilon(k)$$

i being randomly chosen among $\{i_1, i_2\}$, and $\mathcal{P}_{pir}$ being the set of all the public information about the PIR protocol.

Parameters	Max Sec	Plaintext	Ciphertext	F
(1024,60)	97	$\leq$ 20Kbits	128Kbits	≥ 6.4
(2048,120)	91	$\leq$ 100Kbits	512Kbits	≥ 5.12
(4096,120)	335	$\leq$ 192Kbits	1Mbit	≥ 5.3

Fig. 1. Parameter sets for our Ring-LWE encryption scheme. Ciphertexts are made of two polynomials. The first parameter defines the number of coefficients per polynomial and the second the number of bits per coefficient (stored in 64bit registers). From these values, ciphertext sizes can be easily deduced. Maximum theoretical security is only attained if enough noise is included in the ciphertexts and the noise generator matches this security. Plaintext size is slowly (logarithmically) reduced if we want to do a lot of Sum operations. Similarly, the expansion factor stays very close to its optimum.

In this paper, we use a simple cPIR protocol whose basic idea is the one of [27], using the performance improvements on multi-bit database elements of [7] (but without taking into account the zero-knowledge proof ensuring symmetric privacy that is also proposed in [7]), described hereafter. It can be used with any additively homomorphic encryption scheme. The protocol can be formally described as follows:

Security is defined with an indistinguishability game: no attacker can in practice distinguish queries for any two database elements. When used for a cPIR protocol, an encryption scheme with indistinguishability against plaintext attacks ensures that two queries for two different elements of a database are indistinguishable, using a standard hybrid argument (see [7] for a formal definition and proof).

With this simple approach, query size is n times the size of a ciphertext and reply is roughly $\ell \times F$, F being the expansion factor of the encryption scheme used. To reduce query size it is possible to aggregate them by groups of size α and obtain a database with $\lceil n/\alpha \rceil$ elements of size $\ell \times \alpha$. It is also possible to use this protocol recursively. Due to space restrictions the recursive version of this algorithm is presented in the Appendix. In practice, recursion takes as parameter an integer d called *dimension* and results in a scheme in which the client only needs to send $d \times n^{1/d}$ query elements (ciphertexts) and the reply will be of size (roughly) $F^d \times \ell$. For example if $F = 2$ and we have a database with one million elements, it is possible to: send a query of 10^6 ciphertexts and get the database element with an expansion factor of 2 ($d = 1$, no recursion); send a query of 2×1000 ciphertexts and get the database element with an expansion factor of 4 ($d = 2$); send a query of

```
Basic cPIR Protocol
Setup (user):
1.  Set up an instance of the cryptosystem with $k$ security bits

Query Generation to retrieve element $i_0$:
1.  For $i$ from 1 to $n$ generate the $i$-th query element $q_i$ as
      - A random encryption of zero if $i \neq i_0$
      - A random encryption of one if $i = i_0$
2.  Send the ordered set $\{q_1, \cdots, q_n\}$ to the database

Reply Generation:
1.  Note $db_i$ the database elements, $\ell$ the bit size of the $db_i$ and
    $\ell_0$, the bits that can be absorbed in a ciphertext
2.  For $i$ from 1 to $n$
      - Split $db_i$ in chunks of $\ell_0$ bits noted $db_{i,j}$ for $j \in$
      $[1..ceil(\ell/\ell_0)]$
3.  For $j$ from 1 to $ceil(\ell/\ell_0)$
      - Compute $R_j := Sum_{i=1}^{n} Absorb(db_{i,j}, q_i)$
4.  Return $R = (R_1, \ldots, R_{ceil(\ell/\ell_0)})$

Reply extraction:
1.  Decrypt the coordinates of the reply vector $R$ and recover
    $db_{i_0}$ as the concatenation of the decrypted chunks
```

3 × 100 ciphertexts and get the database element with an expansion factor of 8 ($d = 3$); etc.

The reader is referred to [7] for a more elaborated description of these techniques and a justification of correctness. It is possible to make different choices on how the database is split and to change the cryptographic parameters used on each level to improve the performance of recursion. For a complete description, generalization and optimization of this process, the reader is referred to [36] which proposes many interesting variants. In our library, we have decided to stick to the basic approach for recursion although it would be interesting to develop other optimizations, such as those proposed in [32, 36].

2.3 Private Searching

The basic idea of private keyword search [37] is that the database can arrange its elements by grouping them using keywords. With this technique, users can get, using a cPIR protocol, all the database elements that match a given keyword. In this case, the query size is proportional to the amount of possible keywords D (Dictionary size) and the computational cost for the server may change as a database entry that is associated to multiple keywords will be copied once in front of each keyword. Thus, the computational cost will be the database size times the average amount of keywords a database element matches.

It is also possible to use this to filter streamed data based on private criteria [38]. The idea is to build ephemeral keyword-based databases for each message passing. These databases have null elements everywhere except in front of the keywords that the passing message matches. The computational cost to process a packet is therefore its size times the number of keywords it matches (null elements cost nothing to process). With such an approach it is possible to build a filter that outputs for every passing message an encryption of zero when the message does not match the keyword and an encryption of the message when it does.

We use this approach to build a sniffer over a gigabit link in Section 4 that is only interested in messages corresponding to a given IP address. In this sniffer, the streamed messages are the packets on the network, the keywords are the set of IP addresses used in a local area network, and a packet matches the IP-keyword corresponding to its source and destination address. The sniffer's code includes a cPIR query selecting the IP that is being observed and thus even analyzing the code of the sniffer it is not possible to learn which is the IP address as the chosen keyword is hidden in the cPIR query.
Important note: After processing an input, the filter always outputs a ciphertext and it is not possible to distinguish useful outputs from encryptions of zero. If we store all these results we will do not better than a trivial PIR based equivalent (which would store unencrypted all the input data). The main interest of using cPIR, is that it is possible to compress the output so that encryptions of zero are packed and useful outputs preserved, even if it is not known which of the outputs are useful. These techniques are beyond the scope of this paper (see [39] for the most recent proposal on the subject), but it is possible to have efficient filters which have only a small overhead with respect to an unencrypted filter that would store only the keyword-matching data.

3 Proposed protocol

3.1 Overview and auto-optimization

An overview of the proposed protocol is described below. We say we are in server-driven mode if the server enforces a given set of PIR parameters (aggregation and recursion depth) and encryption parameters. In this case, only steps 1 (conditional jump on server-driven mode), 4 (choice of the element) and 5 (retrieval) of the protocol are executed. By opposition, if we are in client-

```
PIR client-server protocol (overview)

Input: Recursion range $(d_1, d_2)$, Aggregation range $(\alpha_1, \alpha_2)$, encryption parameters list $EncParams$, upload/download usable bandwidth $(U, D)$, target optimization function $f_{\text{target}}$, index of the element to retrieve $i$ (pot. undefined), boolean $serverDriven$
Output: Chosen database element
1. If serverDriven is true
      - Server: Send mandatory parameters to the client
      - Client: Check if the parameters give enough security
      - Jump to step 4
2. If performance results do not exist for all parameters in $EncParams$
      - Client and Server: Run the Performance cache algorithm
   → Performance results
3. If multiple PIR and encryption parameters are possible given the input constraints
      - Client and Server: Run the Optimization algorithm
   Else use the only possible choice
   → Optimal parameters for this setting given the constraints
4. If $i$ is undefined
      - Client and Server: Run the Choice algorithm
   → Chosen index $i$
5. If the chosen encryption algorithm is no cryptography
      - Client: Download database, keep element of index $i$
   Else
      - Client: Run the Query gen. algorithm and send result
      - Server: Run the Reply gen. algorithm and send result
      - Client: Run the Reply extraction algorithm and return result
   → Chosen database element
```

driven mode, the client will make an optimization to determine the best possible parameters given the setting and input constraints (steps 2-3), and use them (steps 4-5). By default we suppose that optimal parameters have already been found and we are in server-driven mode.

The algorithms run in this protocol have self explanatory names but for completeness they are informally described in the appendix.

In XPIR there is a set of predefined encryption schemes: no cryptography (trivial PIR with a full database download), Paillier [40], and Ring-LWE (see Section 2.1). Each of the encryption schemes has a predefined list of possible parameters for security varying from 80 to 256 bits (following NIST recommendations for factoring-based cryptography for Paillier and [35] for lattice-based schemes).

By default, the target function for optimization is the round-trip time of the retrieval which, using self-explanatory variable names, is given by the function $max(queryGenerationTime, querySendingTime) + max(replyGenerationTime, replySendingTime, replyDecryptionTime)$. This is due to the fact that query generation and sending are pipelined, then the server waits until it has the complete query and reply generation, sending and decryption are pipelined. We have pre-defined some other target functions such as minimum ressources (which takes the sum of all the values) and a weighted equivalent called cloud cost (which gives a dollar value to each CPU millisecond and a to each bit transmitted and gives the cost of the operation). The target function can be chosen on the command-line.

3.2 Fast Ring-LWE based cPIR

Our contribution on performance is focused on two points: an efficient NTT-CRT (see Section 3.2.1) representation and associated transforms; and the usage of Newton quotients for query elements (see Section 3.2.2).

The idea is that most of the computational costs in query generation, reply generation, and reply extraction come from polynomial multiplications. For example, for reply generation, computational costs are overwhelmingly concentrated on the absorption phase of the basic cPIR protocol described in Section 2.2. With our Ring-LWE scheme this phase can be written as follows.

```
For $j$ from 1 to $ceil(\ell/\ell_0)$
   Compute $R_j = (\sum_{i=1}^{n} db_{i,j} * q_{i,1}, \sum_{i=1}^{n} db_{i,j} * q_{i,2})$
```

noting $q_{i,1}, q_{i,2}$ the two polynomials forming the query elements and all sums and products being in R_q (polynomials reduced modulo $X^n + 1$ and coefficients modulo q).

In NTT-CRT representation, the computational cost of multiplying two polynomials passes from $O(n^2 \times \log^2 q)$ to $O(n \times \log q)$. Such a representation is not new and our contribution is on performance as the time required to get the NTT-CRT representation is divided by a factor 10 and the time required to compute the polynomial products once in that representation is divided by a factor 2 to 3.

We are not aware of the usage of pre-computed Newton coefficients in lattice-based cryptography. The idea is that when multiplying two polynomials, the associated scalar products must be reduced $\bmod q$ which increases the computational cost of these basic operations considerably. In the reply generation algorithm the products are always of the form “some data transformed into a polynomial” times “a query element”. Thus, query elements are used in many multiplications. Pre-computed Newton quotients for re-used multiplicands (such as the query scalars) allows to replace the usual "multiply and divide by the modulus" by a specific modular multiplication algorithm with two integer multiplications and a conditional subtraction (eliminating thus the costly division).

3.2.1 The NTT-CRT representation and transforms

In XPIR we use a mixed NTT-CRT representation to reduce computational costs: Number-Theoretic Transform (NTT) for polynomials [41] and Chinese Remainder Theorem (CRT) for integers. We call the part of XPIR allowing to apply the transforms and compute efficiently on this representation NTTTools. The homomorphic encryption library of Halevi and Shoup [42] implements the encryption scheme of Brakerski, Gentry and Vaikuntanathan [43]. They provide an object they called Double-CRT which provides NTT and CRT representation of polynomials as NTTTools does. We will compare to this work in this section.

Using the NTT and the CRT to accelerate polynomial multiplications is not new and will not be described in detail in this paper, we will just focus on the impact of their usage. The reader is for example referred to [42]. Using an NTT representation allows to compute polynomial multiplications with a linear cost in N instead of quadratic for the trivial algorithm. Transforming a polynomial into NTT form and back can be done in quasi-linear speed (in $O(N \log N)$). The CRT representation ensures that the multiplication cost is also linear in $\log p$, instead of quadratic for a trivial algorithm. Transforming an integer into CRT representation and back has a quadratic cost in $\log p$.

Figure 2 illustrates pre-processing performance, which corresponds to importing data into NTT-CRT form polynomials, and processing, which correspond to fused multiply and add (FMA) operations. The data splitting and CRT (if done) operations are pretty fast, and the main performance bottleneck in pre-processing is computing the NTT in our polynomial ring. Tests correspond to the same laptop that will be used in Section 4 using all of its cores with multi-threading.

Implications on cPIR performance These values have an impact in cPIR reply generation. Pre-processing corresponds to database element importation and processing to reply generation. When launching a server, database elements can be imported into RAM in NTT-CRT form at roughly 5Gbits/s. After importation, the database is processed during the reply generation phase at roughly 20Gbits/s. If data is quickly obsolete (e.g. IPTV streams) the main bottleneck is getting the data into NTT-CRT form and processing is limited by the importation phase to roughly 5 Gbits/s. For comparison purposes, using a Paillier based cPIR the same computer is able to process the database at 1Mbit/s (for a modulus of 2048 bits giving 112 bits of security). Again, for comparison purposes, in trivial PIR (i.e. full database download) processing a bit corresponds to sending it to the client and thus the database is processed at the available download throughput (e.g. a 100Mbit/s FTTH line). Thus trivial PIR will generally be faster than Paillier based cPIR but slower than our Ring-LWE implementation.

Parameters	$(1024, 60)$	$(2048, 120)$	$(4096, 120)$
Input size (per poly)	**20Kbits**	**100Kbits**	**192Kbits**
Pre-processing (per poly)	4.2us	19us	38us
Pre-processing (PIR tput)	4.8Gbps	5.2Gbps	5Gbps
Processing (per poly)	0.57us	2.3us	4.8us
Processing (PIR tput)	18Gbits/s	22Gbits/s	20Gbits/s

Fig. 2. PIR pre-processing and processing time and throughput on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz, for different crypto parameters. Input sizes are the maximum plaintext sizes given in Figure 1. Pre-processing of a polynomial corresponds to NTT and CRT transforms, the main operation during database importation. Inverse transforms give similar results. Processing corresponds to a fused multiply and add (FMA), the main operation during reply generation. This operation's throughput will vary a lot depending on memory saturation: in this setting, if all operands and result change on each operation, processing time is multiplied by three with respect to the given values. Here we used the same memory transfers as in our PIR scheme: for a given thread only one operand varies most of the time. Throughput is given with respect to input data: in pre-processing for each polynomial (Input size) bits are treated; in processing two polynomials must be processed to deal with (Input size) bits.

Comparison with [42] The Double-CRT object proposed in [42] is much more elaborated than NTTTools, has many supplemental functions needed for fully-homomorphic encryption and is more flexible as polynomial degrees. On the other side, the simplicity of our setting allows some interesting choices. First we use Harvey's NTT algorithm [44] which is very fast but requires polynomials degrees to be powers of two. The primes potentially forming the moduli are defined statically enabling various compile-time optimizations. And Double-CRT is built over NTL which in turn is built over GMP, whereas we have built our library without using any external library which results in a big performance improvement.

Parameters	$(\mathbf{1024, 60\Vert 44})$	$(\mathbf{2048, 120\Vert 132})$
Pre-processing (Double-CRT)	178us	1100us
Pre-processing (NTTTools)	16us	78us
Processing (Double-CRT)	5us	27us
Processing (NTTTools)	2.3us	9.6us

Fig. 3. Pre-processing (NTT and CRT) and processing (multiply and add) times with Double-CRT and NTTTools. Modulus size must be a multiple of 44 in Double-CRT (this allows them to do double precision floating point operations for modular reductions). We chose moduli sizes to be the closest possible. **Tests are on a single-core** (as Double-CRT gave a segmentation fault with openmp) of a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. Pre-processing is much faster with NTTTools (x10), mainly due to Harvey's NTT algorithm (which is usable as we restricted ourselves to powers of two for polynomial degrees). In processing the gap is smaller (between x2 and x3) but NTTTools still performs better.

HElib supposes that the user defines the homomorphic computations he needs to do and then a routine defines a complete FHE context for him. In particular the user cannot choose to just use one or two primes, so we had to tweak the code to do comparable tests.

Performance for polynomial multiplications is noticeably improved (between x2 and x3) with NTTTools, as Figure 3 shows. The gap is larger for pre-computation (x10). The reason for this is our choice to restrict polynomial degrees to powers of two, which opens up the usage of nice algorithms such as the one in [44].

Finally, memory usage is much lower with NTTTools, which is not surprising given that we are in a simpler setting. For polynomials of degree 1024 and 60-bit coefficients, the memory footprint in NTTTools is of 8 Kbytes by default and twice that with pre-computed quotients. Using Double-CRT it is harder to evaluate the footprint as some data (such as the FHE context) is shared, but for large amounts of Double-CRT objects memory usage increases linearly at 40Kbytes per object.

NTTTools and the schemes we developed over it will thus be an interesting replacement of Double-CRT for those looking for fast basic polynomial computation on the ideal setting or simple homomorphic operations. Those looking for more advanced operations should use Double-CRT.

3.2.2 Pre-computing Newton Coefficients

The basic scalar operations in XPIR are done modulo a 60-bit integer. To multiply two 60-bit integers modulo a given p, one option is to use a 64×64 to 128-bit multiplication (which is not a basic operation in usual 64-bit instruction but has a reasonable cost) and then retrieve the remainder of the division by p. As the product is in a 128 bit variable, this integer division is pretty costly. In [44], David Harvey attributes to Shoup a very interesting approach to modular multiplications when the same multiplier is used many times.

Parameters	$(\mathbf{1024, 60\Vert 44})$	$(\mathbf{2048, 120\Vert 132})$
`Double-CRT`	3.4us	20.2us
`NTTTools-noQuotients`	29us	115us
`NTTTools`	1.8us	7.3us

Fig. 4. Multiplication times in seconds for different parameters (see Figure 3), **on a single-core** of a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. Note that Double-CRT has much better performance without quotient pre-computation. This is due to the choice of 44-bit moduli which allows floating point and rounding based modular multiplications. With quotient pre-computation we have x2 to x3 better performance with NTTTools. Without quotient pre-computation performance would very bad as we cannot use floating point operations as DoubleCRT can do for 44 bit moduli, but in our protocol we always manage to have such quotients.

The idea is to pre-compute Newton quotients for specific operands that are used many times. Thus, we compute for a given y a scaled approximation to y/p. This is done in our setting by putting y in a 128 bit variable, multiplying it by 2^{64} (with the shift operator <<) and doing a costly integer division by p. This will give us y', the first 64 bits of y/p (multiplied by 2^{64}). The idea is that the costly operation from the multiplication (the integer division) is pre-computed once and then, when we need to do a multiplication $xy \mod p$ we will use a special algorithm taking as input x, y, y' which gives us the result at a lesser cost. The algorithm is pretty simple.

1. $q = xy'/2^{64}$
2. $r = xy - qp \mod 2^{64}$
3. $if r > p : r = r - p$

This algorithm requires just two integer multiplications a shift and a conditional subtraction which is extremely fast when compared to the usual integer division required. Of course if y is used only once there is no gain as one more integer multiplication than in the trivial algorithm is done and an integer division is done during the pre-computation. However if y is used in many multiplications the speedup is considerable. Correctness is proven in [44].

NTTTools provides functions to pre-compute the data needed for this algorithm for a polynomial. When such data is available, polynomial multiplications are done in $2n$ (normal) integer multiplications instead of n modular multiplications. Of course the performance of an application depends on how often the same operands are used. The encryption scheme we built over NTTTools, only needs to do two polynomial multiplications to encrypt, and a single polynomial multiplication to decrypt, always using the same multipliers: the secret and public keys. In the cPIR protocol, the reply is generated by constantly multiplying database element chunks with query elements. In most settings, each query element is multiplied many times by different chunks. Both the secret key and the queries use the pre-computation mechanism. In practice, there is not a single multiplication in our code which does not pass through this process and in almost every case pre-computation is amortized tens or hundreds of times.

3.2.3 Encryption and Decryption Performance

Note that the homomorphic encryption scheme resulting from the modifications we propose in this section is, from a security point of view, equivalent to the scheme described in Section 2.1 as all the modifications are public and reversible for attackers.

The basic idea is that the polynomials that usually describe the inputs (secret key, randomness, messages) are pre-processed by transforming them into NTT-CRT representation. With such a transformation, encryption and decryption operations can be done by coordinate-wise multiplication and additions which leads to very high performance results.

Describing how each algorithm is transformed by the usage of the NTT-CRT representation is of little interest and pretty straightforward. There are only two important points. The first is that each time there is an uniform polynomial in the encryption scheme algorithms we do not need to do change the representation. Indeed the NTT, CRT and inverse NTT, inverse CRT are one-to-one functions that map a finite space to itself and thus are permutations of their domain. Thus taking a uniform element and changing the representation to NTT-CRT is exactly the same as just taking a uniform element. The second is that each time there is a product to compute, one of the two terms is long-lived (the secret key, the public key, or a constant). It is therefore always possible to use rapid modular multiplications using pre-computed Newton quotients.

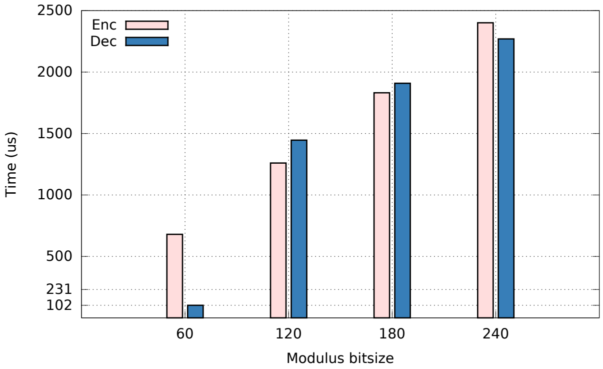

Fig. 5. Encryption and decryption times for polynomial degree 4096 and varying modulus size, on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. Note that encryption costs increase linearly in the modulus size but also the size of the associated ciphertexts and plaintexts. The large jump in decryption costs comes from the usage of GMP for moduli strictly above 60 bits.

Having these two ideas in mind it is easy to see that encryption requires only the computation of three NTT-CRT transformations and some basic operations. This is specially true as all the arithmetic operations we do are coordinate-wise and use a CRT representation allowing to handle numbers through the basic instruction set. This is not true for decryption. At first sight, the most costly operation in decryption will be the inverse NTT. It is, if we use a modulus of 60 bits, but not for larger moduli. Indeed, it is important to notice that all the arithmetic operations use the basic instruction set *except* the separation of the noise and the message in the decryption function. If we are using more than one modulus, in order to separate the noise and the message, we need to get the value of each coordinate in non CRT representation (in CRT representation there is no simple euclidean division). This is done by multiplying the elements of the CRT tuple by what we call *lifting coefficients*. This operation is done without modulus reduction and requires a few multiplications of $\log_2 q$ bits elements. For this operation we need to use a multiprecision library. In practice the decryption cost is multiplied by a factor 10 as soon as we start using such a library. Figure 5 shows this evolution.

This is the only point in which we use GMP on the NTTLWE object (by using the `poly2mpz` function of NTTTools). In practice this results in a very significant performance drop. Note however that for a modulus of 60 bits, performance is surprisingly high. We are able to generate a query at 700Mbits/s and decrypt an incoming reply at 5Gbits/s. This is quite independent of the polynomial degree as the costs of encryption and decryption increase linearly in it but ciphertext and plaintext size too. In practice, a laptop can send queries and receive and decrypt at max available bandwidths in all settings, using a single core. With a modulus of 120 bits, encryption scales well as it is possible to generate a query at 850Mbits/s, but decryption suffers from the CRT lifting and the reply can "only" be decrypted at 710Mbits/s.

In practice, decryption is only the bottleneck for very large moduli (e.g. 480 bits) or if we are on special settings such as connected through a Gigabit line. Small moduli (at most 120 bits) are however generally chosen by the optimizer because of query size. Indeed, for moduli beyond 120 bits the increase in query size (that must be sent on a limited bandwidth line), adds more time to the round-trip time (or total ressources spent) than what is gained in reply expansion factor or cPIR reply generation throughput.

4 Performance and Use-Cases

In this section, we analyze the performance of XPIR using two metrics: latency and user-perceived throughput. The latency measurement is the round-trip time from the moment the client starts generating the cPIR query to the moment it has finished to decrypt the reply. User-perceived throughput is the throughput (measured in bits per second) at which the user is able to get the requested element after decryption.

We will consider two types of settings for our databases: static databases in which pre-processing of the database elements can be done; and dynamic databases whose contents are ephemeral (TV Streams, sensor data, etc.) and which cannot be pre-processed ahead of time. Pre-processing is independently executed for each element at speeds that vary from 5Gbps (for a high-end laptop) to 10Gbps (for a high end server) as shown in Section 3.2.1. A database is thus considered static if the life-time of an element is well larger than its conversion time (e.g. 1-2 seconds for a 10Gbit movie) and the elements are known early enough with respect to the first cPIR transaction in which they will be used.
Use-cases To illustrate the versatility of our library, we highlight performance values with four use-cases combining dynamic/static settings and throughput/latency goals. For high throughput applications we use a Netflix-like server (relatively static data) and a sniffer that obfuscates what he is interested in (dynamic data). For low latency applications we use a Match.com-like online dating database server (relatively static data) and a stock-market information service (dynamic data). **Note that finding the best application for a fast cPIR protocol is beyond the scope of this paper.** The presented use-cases are chosen because they give stressful settings for cPIR, not because cPIR is the only or the best choice to solve privacy issues in these settings. Our goal is to show that cPIR is better than trivial PIR and that using it is *feasible* in huge databases with strong constraints on client obtained throughput.

4.1 Experimental setting

To show that our library is usable by everyone for many applications we use commodity hardware in almost all the settings. Our cPIR Server runs on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz (mobile), and 8GB of DDR3 RAM. As our library is able to process database content very fast, the data storage medium considerably influences performance, specially if this data is pre-processed. In our evaluation, we use two media: RAM (100Gbit/s access), or an OCZ Vertex 460 SSD (4Gbit/s access). The contiguous read speed of our SSD is sufficient to feed the server in all of the dynamic data settings. If data is static, we are able to process it quite faster than what a usual SSD disk can offer. If the database is in RAM this is of course not an issue, but in some applications such as the Netflix-like server, the database is huge and does not fit in RAM. We discuss this issue in the associated Section. FTTH and ADSL lines were simulated by introducing appropriated waiting timers in the client and server which are in fact connected by a gigabit line.
Security In most of our performance results, the optimizer found that the best parameters for the Ring-LWE scheme were $(2048, 120)$ or $(1024, 60)$. According to the parameter generation presented in Section 2.1, the former set of parameters is able to provide 91 bits of security, and the latter 97. To generate randomness for our scheme, we use Salsa20/20 [45] (Salsa20/20 is able to provide up to 256 bits of security), and thus even if a set of parameters for Ring-LWE is able to provide theoretically more security, 256 is thus an upper bound (this is the standard maximum security usually considered).

Security scales extremely well in lattice-based cryptography. For a constant moduli, security (in attacker operations) increases exponentially with the polynomial degree and computational costs increase only (almost)

linearly. For example, if we use parameters $(4096, 120)$ (instead of $(2048, 120)$), the theoretical security can go up to 335 bits. Again, in our implementation security is bounded to 256 bits. In such a high security setting, query generation, reply pre-computation, reply generation, and reply decryption have a cost that is just increased by a factor 2 (more precisely 2.18 for pre-computation and 2 for the rest). With such parameters, each ciphertext can contain more data (almost twice), and thus the security increase comes at very little cost. We will present the costs with the high security (4096, 120) parameter set in the first figure, and then let the optimizer choose the best parameters, with a minimum security set to 91 bits to be able to use the (2048, 120) parameters which are a good compromise between ciphertext size, reply generation throughput and security.
Paillier The results presented in this section correspond to cPIR with Ring-LWE and trivial PIR (to show the interest of cPIR over full database download). Using Paillier always gave worse performance, and therefore does not appear on the different figures.

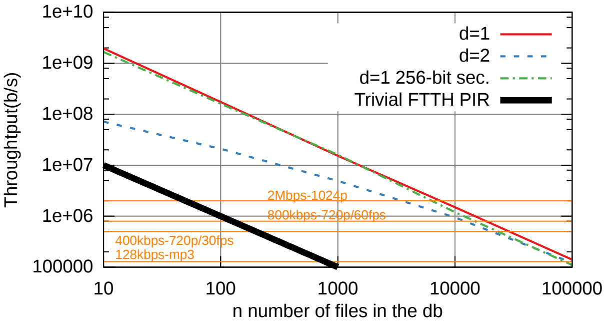

Fig. 6. User-perceived throughput of XPIR streaming static data on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. Trivial PIR over a 100Mbits/s FTTH line (thick black line) is between ten and two hundred times slower than cPIR. The red filled (91 bits security) and green dash-and-dotted (256 bits security) lines give throughput when no recursion is done (i.e. database is processed as a one dimension array) and the blue dashed line with one level of recursion (i.e. database is processed as a two dimension array). The horizontal lines correspond to the needed throughput to see a movie in 1024p (2Mbps), 720p 60Hz (800Kbps) and 720p 30Hz (400Kbps), or to listen to a 128Kbps audio file. Performance on a server with a better processor (e.g. ten-core Xeon E7-4870) roughly doubles and caps at that level as RAM bandwidth is saturated.

4.2 High Throughput on Static Databases

High-throughput applications (i.e. applications requiring a high user-perceived reception throughput) only make sense if the database elements are big enough, if they are very small and quickly sent we consider the essential issue is latency which will be studied in Section 4.4. We therefore consider here only databases with files going from 10Mbit and up. Our experimental results showed user-perceived throughput is independent of file sizes when they were in that range, henceforth the lines in this Figure are valid for any file size greater or equal to 10Mbit.

Figure 6 shows the user-perceived throughput achieved using our library on the experimental setting laptop. The red line shows performance when no recursion is done (i.e. when query size is proportional to n). This line was obtained using the best parameters for throughput (which were given by the optimizer): no recursion, no aggregation, and Ring-LWE cryptography with parameters $(2048, 120)$. With these parameters, ciphertext size (and thus query element size) is 500Kbits and the expansion factor of encryption is $F \simeq 5$. Therefore, in order to get an element at a user-perceived throughput of 2Mbits/s actually 10Mbits/s of bandwidth will be used. This setting is the most favorable from a throughput point of view, but query size can be a problem when the number of elements n grows, as we will see in Figure 7. Note that this line is pretty close to the straight line defined by $15/n$ Gbps (more precisely values slowly drift from $19/n$ Gbps to $14/n$ Gbps for large n values).

The green line shows the same results as the red line in a higher security setting (256 bits security). As noted previously this has almost no impact on processing but doubles the size of each ciphertext and query size (as we will see in Figure 7). Note that the scale is logarithmic, and thus even if the difference with the red line is very small, in this setting performance is roughly 10% worse.

The blue line shows performance with one level of recursion (i.e. when the database is seen as a two-dimensional $\sqrt{n} \times \sqrt{n}$ array and query size is proportional to $2\sqrt{n}$). Recursion results in a significant computational overhead for small databases as the database is processed a first time resulting in an intermediate database of size $F\sqrt{n}$, that we have to process again before getting the final reply. In our implementation the cost of processing this database is roughly ten times the usual cost. If $\sqrt{n} >> 10F$ computation over this intermediate database is negligible as it is small enough with respect to the initial database. Indeed, the Figure shows that the overhead of a level of recursion fades out as n grows.

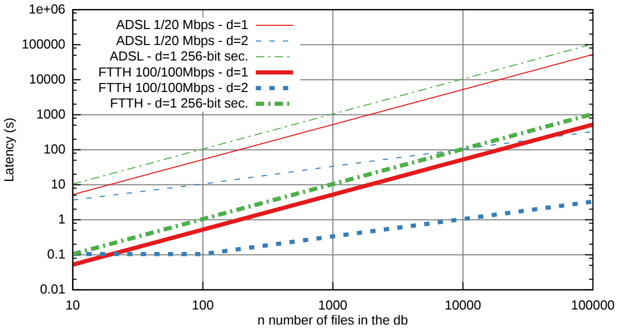

Fig. 7. Initial latency before the user starts to receive streaming data (mainly due to query generation and transmission time to the server). There is no initial latency for trivial PIR. Thin lines are for ADSL and thick lines for FTTH. Colors and line styles are associated to the same settings as in Figure 6. The results highlight that latency grows linearly in n in dimension 1 and in $\sqrt{n}$ in dimension 2, and that the main bottleneck is the available upload bandwidth.

Initial latency: Even if obtaining the best user-perceived throughput is the goal of an application, an important parameter is how much the user will have to wait until he starts receiving the requested stream. Figure 7 highlights the benefit of using a level of recursion for databases with many elements. This is specially true when $n \geq 1000$ as we have seen that this implies almost no computational overhead in this case. On a FTTH line, latency will be below ten seconds (if we use d=2 for $n \geq 1000$). An ADSL line has limited upload bandwidth, henceforth latency ranges from 5 to 500 seconds. Therefore, in such a case, one level of recursion should definitely be used, even if it implies a significant overhead for the reply generation. The strange behaviour of the FTTH lines for a small number of elements comes from the fact that we use TCP sockets to transmit the queries and for very small time values, buffering and windowing gets in the way. It is possible to tune the low level sockets or to use UDP to have a more linear behaviour if needed.

The Netflix Use-case The Netflix movie database is composed (more precisely was composed in 2009 according to the Wikipedia page for Netflix) of 100.000 movies that are stored as static files and can thus be pre-processed for performance improvement. H.265 - High Efficiency Video Coding (HEVC) is the forthcoming standard for video-streams compression[46, 47]. The attained compression levels with this codec enable to watch 720p streams at bit-rates between 400Kb/s for 30fps and 800Kb/s for 60fps and 1024p at 2Mb/s. A typical bit-rate for audio streams is 128Kbps for quality MP3s. These levels (128, 400Kbps, 800Kbps and 2Mb/s are represented by horizontal lines on Figure 6). Please note that we use this use-case as a simple example. For the reader interested in an in-depth study of a private Netflix-like application, please refer to [48].

Given the results of Figure 6, a Netflix-like server based on XPIR allows a user to privately receive a streamed movie with different trade-offs between privacy and quality. **If the user is willing to receive a 720p-30fps video stream he can hide his choice among 35K movies from the server.** Of course a computational trade-off is also possible, with 8K movies and 720p-30fps eight percent of the server's CPU is used and thus it is possible to handle 12 users per processor. *Medium Access Issues:* Obtaining experimental results with databases of up to 10 Gbits was simple as they fit in RAM. To obtain performance results with the largest databases, we processed them in large chunks that did fit in our RAM *removing the disk transfer times* for each chunk. If we use our SSD disk to access the data and take into account the transfer times, disk access is the bottleneck and thus we obtain as performance result a straight line at $2/n$ Gbps (our disk allows 4 Gbps access and pre-computed data is twice larger than the initial data). In the use-case described, this would mean that the maximum amount of movies among which the choice is hidden would be reduced by a factor seven for a given resolution. We consider though that in applications requiring large databases and throughput, such as the Netflix use-case, the provider has high performance disks. In order to match the computational performance of our library it is possible to use for example two OCZ Vertex RevoDrive PCIe SSD in RAID 0 which delivers 30Gbps read throughput, at roughly a cost of 1000$.
Multiple Users: **Note that if data is accessed synchronously for concurrent users, disk access costs do not increase, so scalability is not an issue.** Thus, if for example if 12 users hide their choice among 8K movies with quality 720p-30fps the server can access the database at just eight percent of its processing speed $15 * 8/100 = 1.2$Gbits/s so a commodity hard disk is enough. For an excellent analysis of a private Netflix use-case, the related CPU and I/O performance trade-offs, see [48]. We also further discusses this in the conclusion.

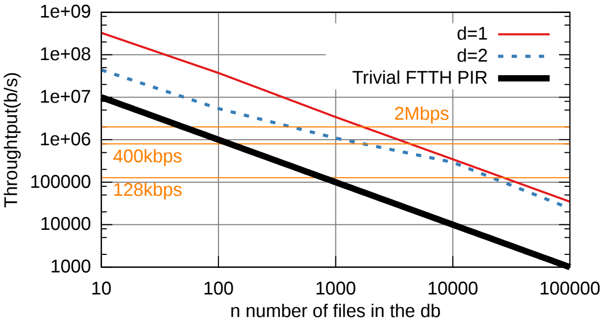

Fig. 8. User-perceived throughput of XPIR streaming dynamic data on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. Trivial PIR over a 100Mbits/s FTTH connection (thick black line) is between five and fifty times slower than cPIR. The red line gives throughput when no recursion is done (i.e. database is processed as a one dimension array) and the blue line with one level of recursion (i.e. database is processed as a two dimension array). The horizontal lines correspond to the needed throughput to see a movie in 1024p (2Mbps), 720p-60fps (800Kbps) and 720p-30fps (400Kbps), or to listen to a 128Kbps audio file. Performance on a server with a better processor (e.g. ten-core Xeon E7-4870) can be two to three times higher.

4.3 High Throughput on Dynamic Data

At first sight, dynamic databases are similar to static ones apart that data cannot be pre-processed offline, such as it is the case with IPTV for example. However, they can have a large span of shapes and contents and are not always a simple extension of static databases to "infinite size" files. An exhaustive analysis of dynamic databases is beyond the scope of this paper, but we show two different settings : IPTV and a private sniffer.

The first setting is pretty simple : usual data streams that cannot be pre-processed such as for IPTV. Figure 8 presents the same results as 6 but with dynamic data. As one can see, the user-perceived throughput is roughly divided by six. For an IPTV like application, a single processor can handle one hundred 720p-30fps streams for 50 simultaneous clients (*e.g.* classical TV), or five thousand such streams for a single client (*e.g.* a large set of distant IP web cameras). The second setting is more tricky, as the dynamic data elements are most of the time null, and the non nulls can be very small. We describe this setting in our second use-case.

The Private Sniffer Use-Case In this use-case we suppose someone creates a sniffer that stores all the packets that have a given source IP address, but wants to ensure that nobody that would find the sniffer and analyze its code could learn which IP the sniffer is interested in. Of course, it is possible to store every message (as with trivial PIR) but, as described in Section 2.3, using cPIR the storage is much more compact.

With this approach, a cPIR query is generated and each query element is associated to a given source IP. The first question we can ask is: how large can be the IP range? Suppose we use either $(1024, 60)$ parameters or $(2048, 120)$ parameters with Ring-LWE encryption. Each query element is 128Kbit long in the former case and 512Kbit in the latter. If we aim to cover a class B network range (65535 addresses) the query size will be 1Gbyte in the former case and 4Gbytes in the latter.[1] It is important to note that this query size is not something that must be sent regularly, for most sniffers it will define how it behaves (it is an encrypted part of the sniffing program) and used to store large amounts of results in a local hard drive before being retrieved (of course results can also be sent through the network). This size does not affect performance either, as our results on processing throughput have proven to be independent of how many elements the query has, as long is it fits in RAM, which we assumed to be true.

For every packet the sniffer intercepts, he builds a database such that each query element is associated to a null element, except the query element corresponding to the source IP of the intercepted packet which is associated to the packet. Then the sniffer generates a cPIR reply storing the reply in the disk using the compression techniques described in Section 2.3. The dynamic database is thus pretty special as it is almost null and the element to process will be often much smaller (between 320 bits and 12Kbits) than what can be absorbed in a ciphertext (roughly 20Kbits for the smaller parameters and 90Kbits for the larger ones). A trivial implementation will thus not use all the power our library can provide in other settings.

The red line in Figure 9 gives the throughput at which the sniffer is able to process the intercepted packets. As packets are much smaller than classical plaintext size, we chose the smallest cryptographic parameters possible, *i.e.* $(1024, 60)$. We consider that, after absorption, the ciphertext can undergo up to one thousand sums (for operations such as insertion on a bloom filter, etc.). Given the internal structure of our cryptosystem, this implies a plaintext size of 15Kbits. If we generate

[1] In fact, the IP range can be arbitrarily large if we associate multiple IPs, or a hash of the IP to each query element. In that case we will obtain packets from different IP sources and the size of the query will determine the efficiency of the filtering done.

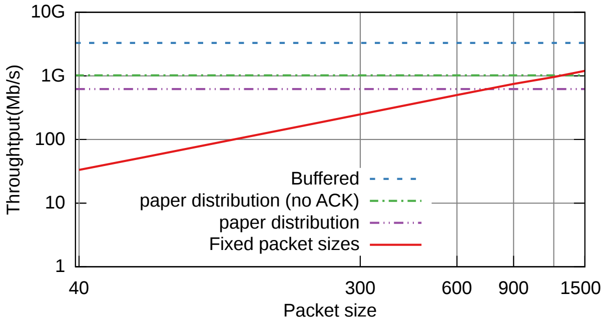

Fig. 9. Packet processing throughput for the sniffer use-case using XPIR on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. Trivial PIR performance does not make sense in this setting. The red line, measures performance each packet size, in bytes in the x-axis, independently (i.e. measuring performance just processing 40 bytes packets, then measure performance for 80 bytes packets etc.). The green line gives the processing throughput when the traffic follows a classic bi-modal distribution such as found in [49]. The purple line gives throughput for a traffic for which we ignore packets of size below 60 bytes (basically ACKs). The blue line gives performance if we wait for traffic to fill buffers and only generate cPIR replies when enough information has been collected to fill a ciphertext.

a cPIR reply for each 40 bytes incoming packet, most of the space available in the resulting ciphertext will be lost, but the cPIR reply generation operation will not cost less (for null elements the operation is free, but for small elements the operation costs as much as for a complete plaintext). Thus, if we deal with packets of 400 bytes instead of 40, the cPIR reply generation costs the same, but we process ten times more information. As even for the largest sizes (we consider usual packet sizes, up to the standard MTU (aximum Transmission Unit), i.e. 1500 bytes), a packet always fits a plaintext, the processing throughput is linear on the packet size.

If we consider a classic bi-modal distribution (40% very small packets, 40% close to MTU packets, 20% in-between packets) such as those described in [49], **the sniffer is able process a link at 600Mbps** (purple line). **If we consider the sniffer is not interested in very small packets (ACKs mostly), it can process a link at slightly over 1Gbps** (green line). **If we buffer packets** and do not generate a cPIR reply until we have enough data from a given source IP address to fill a plaintext, we can do much better. In this case we can choose parameters giving better processing speeds such as $(2048, 120)$. In such a setting **we can process a link at roughly 3Gbps** (blue line), for parameters $(2048, 120)$ if we buffer 90Kbits of data for a given IP source before generating a cPIR reply (using the higher security parameters we get almost the same performance but with a query twice larger).

Of course, implementing a complete private searching prototype would imply looking into other concerns, such as making sure that other aspects (packet interception, compression function such as Bloom filters on the output, etc.) are able to cope with this throughput, but this is beyond the scope of this paper.

4.4 Latency on Static/Dynamic Databases

In this Section, we want to evaluate XPIR latency, *i.e.* round trip time (RTT), in settings where data is static or dynamic. Figure 10 shows the RTT achieved with static data and Figure 11 with dynamic data. The x-axis represents the size of the database ranging from 1Mb to 1Tb. The green line shows the request processing time (RP), the red line shows the RTT with no network (i.e. the client on the same machine as the server), and the various blue lines represent the RTT with a FTTH network for different values for n. While, when considering throughput, the request processing and data importation were the most striving parameters, when looking at RTT, performance results of a balance between reply processing time and upload/download times.

It is very important to note that usual techniques in cPIR such as aggregation and recursion (see Section 2.2) are mandatory to keep RTT low. In Figure 10 we used parameters $(1024, 60)$ for the Ring-LWE cryptosystem and thus query element size is 128Kb and $F \simeq 6$. For $n = 10000$ and $l = 1Mb$, if no aggregation and no recursion is used, sending the query $(10000 * 128Kb)$ over the FTTH link takes 12.8 seconds and sending the reply (6*1Mb) takes 0.06 seconds while generating the query (at 2.2 Gbps) takes 0.05 second, processing it (at 10 Gbps) takes 0.1 second and decrypting the reply (at 5.6Gbps) takes about 1ms. Using recursion divides query sending time by a factor 50 and has little impact on the other times so it is clearly beneficial.

Using aggregation and recursion, when beneficial, the optimizer can set the cPIR parameters in order to transform the shape of a database with a high n value into a database with a smaller n. This is why, on both Figures, the higher is n the lower is the RTT. Indeed, the shape of the database is transformed in order to lower this parameter if a smaller n is more favorable. As one can observe, the high n lines tend to approach the RTT limit which is the RP line. The only difference between static and dynamic databases lies in the request process-

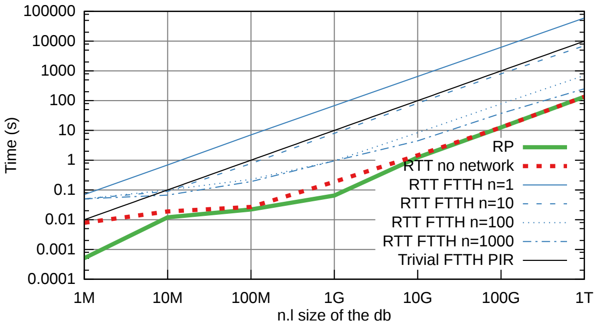

Fig. 10. Round-trip time (RTT) and request processing (RP) times of XPIR serving static data on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz on a FTTH network (database sizes are in bits). Trivial PIR (from top to bottom the second filled line) is faster than cPIR for databases under ten elements, which is natural as cPIR has a reply expansion factor around five. For databases with more elements, cPIR can be up to fifty times faster. When the client is local, RTT (red thick dashed line) matches RP (green thick filled line), specially for large databases. Each thin blue line gives RTT for a fixed n and varying database sizes. For large databases reply size is the limiting factor, which explains why performance is closer to ideal RTT as n grows (when n grows for a fixed database size ℓ shrinks). For small databases, query size is the limiting factor. RTT does not grow as fast as n because the optimizer uses aggregation.

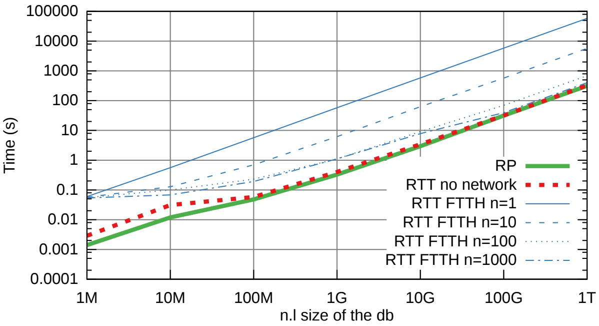

Fig. 11. Round-trip (RTT) and request processing (RP) times of XPIR serving dynamic data on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz on a FTTH network (database sizes are in bits). Trivial PIR has been masked for clarity as the same remarks as most comments on Figure 10 also apply. As data is not already pre-processed, request processing time is higher, but upload/download times do not change. This explains why blue lines are almost identical except for the fact that the gap to reach ideal RTT is smaller. In practice this implies that RTT is not affected much by pre-processing except for very large databases.

ing speed that is impacted by the need to pre-process the data in the dynamic case. One can observe the different values of request processing (red dashed lines on both Figures). Henceforth, with dynamic databases, the high n lines will tend towards the RP line later, *i.e.* with larger databases. This implies that in most networked situations RTT will be similar for static and dynamic databases, except for the largest ones.

Match.com Use-Case In this use-case we consider that an online dating database server wants to provide a paying private keyword search mechanism to its clients. When using this system, users can define some public criteria, such as the city in which they would like to meet people (which is anyways probably revealed by their IP). This set of public parameters will reduce the database size over which a second search, based on private criteria, will be done. The users can then do a cPIR-based keyword search (see Section 2.3) to get the profiles matching a set of private keywords. If we suppose the database has one million profiles, each of one megabit, the complete database will be of one Terabit. We must also take into account that each profile may match a set of keywords and that reply generation costs are multiplied by the average number of matching keywords in a private keyword search. If we suppose that the average profile has five keywords, using the RTT given in Figure 10 a user would have to wait for ten minutes before having a reply which is probably too much for a web experience. Using the public keyword pre-filtering we described we can hope to divide the size of the database by a factor 10 to 100 (if users are distributed in various cities and public keywords are specific enough) which would lower the waiting time to 6-60 seconds, a much more reasonable time for a search. Of course if we consider Match.com 5 Millions users (according to Wikipedia's page which cites 2014 sources) and profiles of multiple megabytes, public filtering will have to be much more efficient. But the fact that we are able to grasp having usable cPIR protocols in such large social networks was unthinkable not that long ago.

NYSE Use-Case In this last use-case, we are interested in using XPIR on dynamic streams with the lowest latency possible. The New-York Stock Exchange (NYSE) Secure Financial Transaction Infrastructure (SFTI) high-end service serves 5-10Gbps of data concerning various worldwide stock markets. The Bloomberg "snooping" scandal is a good illustration of why one would want to keep private the financial information one is interested in. One can see two different type of usages with this application: oriented towards throughput or towards latency. In the first case, a client may want to register to a given set of streams of information, and get served with all the information concerning the associated companies coming from stock markets, analysts, etc. with a constant stream of up to date information. In such a case, the application is very similar to an IPTV service where the data-stream concerns financial information instead of a TV stream. Refer to Section 4.3 for performance results.

In the second case, a client wants to retrieve as fast as possible the last bunch of information concerning a company. In this case, the stock market service can be seen as collecting data generated by remote sensors and giving access to this dynamic data to its clients on a per request basis. The most striving question is thus how long does it take for the client to retrieve the information on a given company, in other words, how fresh is the data? For example, suppose a user wants to grab some information from the last 100ms (we cannot expect to get much more recent data given the underlying network RTTs). In the SFTI 5Gbit stream the amount of data corresponding to 100ms should be 500Mbits. As such data is composed of many elements we can expect that latency will be close to the optimal line in Figure 11 and thus the user should get the information in roughly 100ms, which is a reasonable waiting time for information that already is old of 100ms.

4.5 Other cryptosystems

As noted before, in almost all situations the Ring-LWE based cPIR is chosen by the optimizer, as it gives the best results. In some extreme cases however, the optimizer chooses to do a Paillier based cPIR or a trivial (full-database download) PIR. The Paillier based cPIR will be chosen for extremely small bandwidths in which case the cPIR reply generation throughput is not important as most of time is spent sending the reply and reply expansion factor is the most important parameter. On the opposite side, trivial PIR will be of course the natural choice when available bandwidth is higher than our database processing throughput. The limit should therefore be not very far of 20Gbps for static pre-processed databases, and 5Gbps for dynamic databases. Other extreme settings in which trivial PIR will be the natural choice exist. An example is for database with two to four elements. In this case a cPIR reply with our Ring-LWE scheme will be larger than the database itself due to our encryption scheme's expansion factor. Another example is for very small databases in which query size may be larger than database size. For example, using an ADSL connection (1Mbps upload / 20Mbps download) on a 10Mbit database with ten elements, sending a Ring-LWE query will take at least 1 second, whereas the full database download only needs half a second (note that using aggregation to reduce query size does not solve the issue). Of course, such settings may in some situation correspond to real life situations, but only scarcely.

5 Conclusion

Lattice based cryptography brought groundbreaking advances on worst-case to average-case reductions and on fully-homomorphic encryption. However it has been for a long time seen as impractical, despite its excellent asymptotic results. This field of research has matured a lot. The arrival of the ideal lattice setting, and the development of many performance tweaks has completely changed attainable performances in a non-asymptotic sense. cPIR has often been considered as an impratictal protocol [1] but lattice-based cryptography brings a real overhaul on this, as cPIR becomes feasible even without a high-end server. We have shown that our protocol can be used to process a wide range of databases in a few seconds, even for 100Gb databases.

These experiments would have taken thousands of seconds with a number theory cryptosystem as Paillier, which would have processed the database at 1Mbit/s. Sending the database, even over a 100Mbit/s link would have increased by a factor one hundred the times we presented in our experiments. The results presented in this paper were obtained on a commodity laptop, using a high end server in a multi-core setting can only increase this difference further. However this is not our purpose, what we wanted to highlight is that lattice-based cryptography has transformed the utterly impractical into something feasible by everyone. As we want to show that it is feasible by everyone, we have included the auto-optimize tools that will allow anybody to use our library without being an expert on cryptography. We are eager to hear from these people's experiences.

Finally it is important to note that even if we have shown that cPIR can be used with very large databases, in practice, as databases grow, cPIR can be used as a building block combined with other techniques to improve scalability. A nice example of that is Popcorn [48] which cleverly combines cPIR and ITPIR to provide a much more efficient Netflix-like private streaming service than the simple example we used. Another example is the communication and computationally efficient ORAM protocols that use cPIR as a building block. We also hope that XPIR will increase the span of applications that use cPIR as an (efficient ☺) subroutine.

References

[1] R. Sion and B. Carbunar, "On the Computational Practicality of Private Information Retrieval," in *14th ISOC Network and Distributed Systems Security Symposium (NDSS'07), San Diego, CA, USA*, 2007.

[2] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private Information Retrieval," in *46th IEEE Symposium on Foundations of Computer Science (FOCS'95), Pittsburgh, PA, USA*, pp. 41–50, IEEE Computer Society Press, 1995.

[3] W. Gasarch, "A Survey on Private Information Retrieval," *Bulletin of the European Association for Theoretical Computer Science*, vol. 82, pp. 72–107, Feb. 2004. Columns: Computational Complexity.

[4] A. Kiayias and M. Yung, "Secure Games with Polynomial Expressions," in *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2001.

[5] C. Aguilar Melchor and P. Gaborit, "A Fast Private Information Retrieval Protocol," in *The 2008 IEEE International Symposium on Information Theory (ISIT'08), Toronto, Ontario, Canada*, pp. 1848–1852, IEEE Computer Society Press, 2008.

[6] J. T. Trostle and A. Parrish, "Efficient computationally private information retrieval from anonymity or trapdoor groups," in *ISC* (M. Burmester, G. Tsudik, S. S. Magliveras, and I. Ilic, eds.), vol. 6531 of *Lecture Notes in Computer Science*, pp. 114–128, Springer, 2010.

[7] J. P. Stern, "A New Efficient All-Or-Nothing Disclosure of Secrets Protocol.," in *13th Annual International Conference on the Theory and Application of Cryptology & Information Security (ASIACRYPT'98), Beijing, China*, vol. 1514 of *Lecture Notes in Computer Science*, pp. 357–371, Springer, 1998.

[8] H. Lipmaa, "First cpir protocol with data-dependent computation," in *Proceedings of the 12th International Conference on Information Security and Cryptology*, ICISC'09, (Berlin, Heidelberg), pp. 193–210, Springer-Verlag, 2010.

[9] R. Ostrovsky and W. E. Skeith III, "Private Searching on Streaming Data," in *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, vol. 3621 of *Lecture Notes in Computer Science*, pp. 223–240, Springer, 2005.

[10] D. Bleichenbacher, A. Kiayias, and M. Yung, "Decoding of Interleaved Reed Solomon Codes over Noisy Data," in *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings* (J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, eds.), vol. 2719 of *Lecture Notes in Computer Science*, pp. 97–108, Springer, 2003.

[11] D. Coppersmith and M. Sudan, "Reconstructing curves in three (and higher) dimensional space from noisy data," in *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, STOC'2003 (San Diego, California, USA, June 9-11, 2003)*, (New York), pp. 136–142, ACM Press, 2003.

[12] S. Arora and R. Ge, "New algorithms for learning in presence of errors," in *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, pp. 403–415, Springer, 2011.

[13] J. Bi, M. Liu, and X. Wangi, "Cryptanalysis of a homomorphic encryption scheme from isit 2008," in *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*, pp. 2152–2156, 2012.

[14] T. Lepoint and M. Tibouchi, "Cryptanalysis of a (somewhat) additively homomorphic encryption scheme used in pir," in *WAHC'15 - 3rd Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, 2015.

[15] C. Aguilar Melchor, B. Crespin, P. Gaborit, V. Jolivet, and P. Rousseau, "High-speed Private Information Retrieval Computation on GPU," in *Second International Conference on Emerging Security Information, Systems and Technologies (SECURWARE'08), Cap Esterel, France*, pp. 263–272, IEEE Computer Society Press, 2008.

[16] P. Mittal, F. G. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, "Pir-tor: Scalable anonymous communication using private information retrieval.," in *USENIX Security Symposium*, 2011.

[17] R. Henry, Y. Huang, and I. Goldberg, "One (block) size fits all: Pir and spir with variable-length records via multi-block queries," *Proceedings of NDSS*, 2013.

[18] T. Mayberry, E.-O. Blass, and A. H. Chan, "Efficient private file retrieval by combining ORAM and PIR," in *Proceedings of Annual Network & Distributed System Security Symposium*, pp. 1–11, Citeseer, 2014.

[19] E.-O. Blass, R. Di Pietro, R. Molva, and M. Önen, "Prism – privacy-preserving search in mapreduce," in *Privacy Enhancing Technologies* (S. Fischer-Hübner and M. Wright, eds.), vol. 7384 of *Lecture Notes in Computer Science*, pp. 180–200, Springer Berlin Heidelberg, 2012.

[20] F. Olumofin, P. Tysowski, I. Goldberg, and U. Hengartner, "Achieving efficient query privacy for location based services," in *Privacy Enhancing Technologies* (M. Atallah and N. Hopper, eds.), vol. 6205 of *Lecture Notes in Computer Science*, pp. 93–110, Springer Berlin Heidelberg, 2010.

[21] F. Olumofin and I. Goldberg, "Privacy-preserving queries over relational databases," in *Privacy Enhancing Technologies* (M. Atallah and N. Hopper, eds.), vol. 6205 of *Lecture Notes in Computer Science*, pp. 75–92, Springer Berlin Heidelberg, 2010.

[22] C. Devet and I. Goldberg, "The best of both worlds: Combining information-theoretic and computational pir for communication efficiency," in *Privacy Enhancing Technologies*, pp. 63–82, Springer, 2014.

[23] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *EUROCRYPT'2010*, vol. 6110 of *Lecture Notes in Computer Science*, pp. 1–23, Springer, 2010.

[24] W. Gasarch and A. Yerukhimovich, "Computational inexpensive PIR," 2006. Draft available online at http://www.cs.umd.edu/~arkady/pir/pirComp.pdf.

[25] O. Regev, "New lattice based cryptographic constructions," *Journal of the ACM*, vol. 51, no. 6, pp. 899–942, 2004.

[26] S. W. Smith and D. Safford, "Practical server privacy with secure coprocessors," *IBM Systems Journal*, vol. 40, no. 3, pp. 683–695, 2001.

[27] E. Kushilevitz and R. Ostrovsky, “Replication is not needed: Single database, computationally-private information retrieval (extended abstract),” in *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 364–373, 1997.

[28] F. Olumofin and I. Goldberg, “Revisiting the computational practicality of private information retrieval,” in *Financial Cryptography and Data Security* (G. Danezis, ed.), vol. 7035 of *Lecture Notes in Computer Science*, pp. 158–172, Springer Berlin Heidelberg, 2012.

[29] Gilles Brassard and Claude Crépeau and Jean-Marc Robert, “All-or-Nothing Disclosure of Secrets,” in *CRYPTO* (A. M. Odlyzko, ed.), vol. 263 of *Lecture Notes in Computer Science*, pp. 234–238, Springer, 1986.

[30] Z. Brakerski and V. Vaikuntanathan, “Fully homomorphic encryption from ring-lwe and security for key dependent messages,” in *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference*, vol. 6841, p. 501, 2011.

[31] Y. Doröz, B. Sunar, and G. Hammouri, “Bandwidth efficient pir from ntru,” in *2nd Workshop on Applied Homomorphic Cryptography and Encrypted Computing - WAHC’14*, pp. 195–207, Springer, 2014.

[32] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang, “Optimal rate private information retrieval from homomorphic encryption,” *PoPETs*, vol. 2015, no. 2, pp. 222–243, 2015.

[33] D. Pointcheval, “Le chiffrement asymétrique et la sécurité prouvée,” *Habilitation à diriger des recherches, Université Paris VII*, 2002.

[34] S. Goldwasser and S. Micali, “Probabilistic encryption,” *Journal of Computer and System Sciences*, vol. 28, no. 2, pp. 270–299, 1984.

[35] R. Lindner and C. Peikert, “Better key sizes (and attacks) for lwe-based encryption,” in *CT-RSA* (A. Kiayias, ed.), vol. 6558 of *Lecture Notes in Computer Science*, pp. 319–339, Springer, 2011.

[36] H. Lipmaa, “An oblivious transfer protocol with log-squared communication,” in *8th Information Security Conference (ISC’05), Singapore*, vol. 3650 of *Lecture Notes in Computer Science*, pp. 314–328, Springer, 2005.

[37] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold, “Keyword Search and Oblivious Pseudorandom Functions,” vol. 3378 of *Lecture Notes in Computer Science*, pp. 303–324, Springer, 2005.

[38] R. Ostrovsky and W. E. Skeith III, “Private searching on streaming data,” *J. Cryptology*, vol. 20, no. 4, pp. 397–430, 2007.

[39] M. Finiasz and K. Ramchandran, “Private Stream Search at the same communication cost as a regular search: Role of LDPC codes,” in *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*, pp. 2556–2560, 2012.

[40] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *18th Annual Eurocrypt Conference (EUROCRYPT’99), Prague, Czech Republic*, vol. 1592 of *Lecture Notes in Computer Science*, pp. 223–238, Springer, 1999.

[41] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss, “On the design of hardware building blocks for modern lattice-based encryption schemes,” in *Cryptographic Hardware and Embedded Systems – CHES 2012* (E. Prouff and P. Schaumont, eds.), vol. 7428 of *Lecture Notes in Computer Science*, pp. 512–529, Springer Berlin Heidelberg, 2012.

[42] S. Halevi and V. Shoup, “Design and implementation of a homomorphic-encryption library,” 2013.

[43] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(leveled) fully homomorphic encryption without bootstrapping,” in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS ’12, (New York, NY, USA), pp. 309–325, ACM, 2012.

[44] D. Harvey, “Faster arithmetic for number-theoretic transforms,” *J. Symb. Comput.*, vol. 60, pp. 113–119, 2014.

[45] T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe, “Software speed records for lattice-based signatures,” in *Post-Quantum Cryptography* (P. Gaborit, ed.), vol. 7932 of *Lecture Notes in Computer Science*, pp. 67–82, Springer-Verlag Berlin Heidelberg, 2013. Document ID: d67aa537a6de60813845a45505c313, http://cryptojedi.org/papers/#lattisigns.

[46] ISO/IEC, “High efficiency coding and media delivery in heterogeneous environments – part 2: High efficiency video coding,” Tech. Rep. ISO/IEC 23008-2:2013, International Standards Organization Publication, 2013.

[47] J. Ohm, G. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, “Comparison of the coding efficiency of video coding standards;including high efficiency video coding (hevc),” *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, pp. 1669–1684, Dec 2012.

[48] T. Gupta, N. Crooks, S. Setty, L. Alvisi, and M. Walfish, “Scalable and private media consumption with popcorn.” Cryptology ePrint Archive, Report 2015/489, 2015. http://eprint.iacr.org/.

[49] R. Sinha, C. Papadopoulos, and J. Heidemann, “Internet packet size distributions: Some observations,” Tech. Rep. ISI-TR-2007-643, USC/Information Sciences Institute, May 2007. Orignally released October 2005 as web page http://netweb.usc.edu/~rsinha/pkt-sizes/.

A Algorithms

Algorithms for the PIR client-server protocol

Performance cache generation (Client and Server):
Input: Set of parameters for encryption schemes
Output: Set of performance results for these parameters

```
1. Client: For each set of parameters of each encryption scheme
     - Evaluate encryption and decryption throughput
     - Store the resulting values for this set of parameters
2. Server: For each set of parameters of each encryption scheme
     - Evaluate precomputation and reply generation throughput
     - Store the resulting values for this set of parameters
```

Optimization (Client and Server):
Input: Recursion range (d_1, d_2), Aggregation range (α_1, α_2), potential encryption parameters list *EncParams*, upload/download usable bandwidth (U, D), target optimization function f_{target}
Output: Best crypt and PIR parameters taking f_{target} as a measure

```
1. Server: Send optimization information
     - The database shape ($n$ and $\ell$)
     - The server performance cache

2. Client: If $U$ or $D$ are null do a bandwidth test to redefine them
3. Client: Optimize
     For every encryption scheme parameters in EncParams
       For every dimension $d$ between $d_1$ and $d_2$
         For every aggregation value $\alpha$ between $\alpha_1$ and $\alpha_2$
           - Estimate queryGenerationTime with the performance cache
           - Estimate querySendingTime with the upload bandwidth
           - Estimate replyGenerationTime with the performance cache
           - Estimate replySendingTime with the download bandwidth
           - Estimate replyDecryptionTime with the performance cache
           - Give a performance measure applying $f_{\text{target}}$ to these values
4. Client: Output the parameters with the best performance measure
```

Choice (Client and Server):
Input: Database
Output: Index of the element chosen by the user

```
1. Server: For each file send a description and an associated index or a global description of the set of files
2. Client: Present the catalog to the user, and return the chosen index
```

Query generation (Client):
Input: PIR parameters (n, ℓ, α, d), crypto parameters *Enc.Params*, chosen index i
Output: Query

```
1. Redefine $n = ceil(n/\alpha)$ and note $n_1 = \cdots = n_d = ceil(n^{1/d})$
2. Define $(i_1, \ldots, i_d)$ the decomposition in base $ceil(n^{1/d})$ of $i$
3. For $j$ in $[1..d]$ generate a query $Q_j$ with the Basic cPIR protocol for retrieving an element of index $i_j$ in a database of $n_j$ elements using an encryption scheme based on Enc.Params
4. Return $Q = (Q_1, \ldots, Q_d)$
```

Reply generation (Server):
Input: PIR parameters (n, ℓ, α, d), crypto parameters *Enc.Params*, query $(Q_1, \ldots, Q_d)$, database elements $(db_1, \ldots, db_n)$
Output: PIR reply

```
1. Redefine $n = ceil(n/\alpha)$ and note $n_1 = \cdots = n_d = ceil(n^{1/d})$
2. For $j \in [1..n]$ redefine $b_j$ as the aggregation of $db_{j\alpha+k}$ for $k \in [1..\alpha]$
3. For $j \in [1..n]$
     - Note $(j_1, \ldots, j_d)$ the decomposition of $j$ in base $ceil(n^{1/d})$
     - Define $db_{(j_1,\ldots,j_d)} = db_j$
4. For $j \in [1..d]$
     For each tuple $(i_{j+1}, \ldots, i_d)$ in $[1..n_{j+1}] \times \cdots \times [1..n_d]$
       - Compute using the cPIR basic algorithm a PIR reply using
           $(db_{(1,i_{j+1},\ldots,i_d)}, \ldots, db_{(n_j,i_{j+1},\ldots,i_d)})$ as a database and $Q_j$
       - Define with this PIR reply $db_{(i_{j+1},\ldots,i_d)}$ if $j < d$ or $RES$ if $j = d$
5. Return $RES$
```

Reply extraction (Client):
Input: PIR parameters (n, ℓ, α, d), crypto parameters *Enc.Params*, chosen index i, PIR reply
Output: Element of index i in the database

```
1. Decrypt the $d$ encryption layers of the reply to get $\alpha$ elements
2. Return the element corresponding to index $i$ and drop the others
```

Remark (convexity): If $\alpha_1 = 1$, $\alpha_2 = n$ and n is large (say 10^9) the optimizer may have to do a lot of tests before reaching the best result. To lower the amount of tests we used a convexity assumption to do a dichotomy when looking for the best α. When α grows, query size is reduced and reply size increased. Using this monotony it is reasonable to assume that the target functions we use are close to convex. In practice the optimizer always returned very reasonable results and was able to run in a few milliseconds for any database.