XPIRe : Private Information Retrieval for Everyone

Carlos Aguilar-Melchor*, Joris Barrier†, Laurent Fousse‡, Marc-Olivier Killijian†
carlos.aguilar@polytechnique.org, {joris.barrier,marco.killijian}@laas.fr, laurent@komite.net

*XLIM, Université de Limoges, 123, av. Albert Thomas, 87060 Limoges Cedex, France
†LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex, France
‡Université de Grenoble, Laboratoire Jean-Kuntzmann, Grenoble, France

NOTE: The XPIRe library is available at https://github.com/XPIRe-team/XPIRe (GPLv3 licence), see the appendix for details.

Abstract

A single-database computationally-Private Information Retrieval (hereafter PIR) scheme is a protocol in which a user retrieves a record from a database while hiding which from the database administrators. Classic protocols require that the database server execute an algorithm over all the database content at very low speeds which impairs significantly their usability.

In [1], given certain assumptions, realistic at the time, Sion and Carbunar showed that PIR 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: PIR is of practical value. This is achieved without compromising security, using simple standard encryption schemes, and very conservative parameter choices.

In order to prove this, we provide a fast fully-usable PIR library and do a performance analysis, illustrated by use-cases, highlighting that this library is practical in a large span of situations. The PIR library allows a server to process its data at a throughput ranging from 1 Gbps on a single core of a commodity CPU to almost 10 Gbps on a high-end processor using its multi-core capabilities. After replying to a first query (or through pre-computation), there is a x3 to x5 speedup for subsequent queries (if the database content is unchanged for those queries). The performance analysis shows for example that it is possible to privately receive an HD movie from a Netflix-like database (with 35K movies) with enough throughput to watch it in real time, or to build a sniffer over a Gbit link with an obfuscated code that hides what it is sniffing.

This library is modular, allowing alternative homomorphic encryption modules to be plugged-in. We provide a slow but compact crypto module with a number theoretic Paillier encryption, and a fast crypto module with Ring-LWE based encryption (based on standard lattice-based problems and conservative parameters). The library has an auto-optimizer that chooses the best protocol parameters (recursion level, aggregation) and crypto parameters (if the crypto module implements the necessary API) for a given setting. This greatly increases the usability for non-specialists. Given the complexity of parameter settings in lattice-based homomorphic encryption and the fact that PIR adds a second layer of choices that interact with crypto parameter settings, we believe this auto-optimizer will also be useful to specialists.

I. Introduction

Homomorphic encryption has followed a curious path in the history of cryptography. Since the very beginning of public key cryptography, in the early seventies, 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 (sdcPIR or just PIR). 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.

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 single 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]).

In this paper, we focus on PIR schemes that do not need the database to be replicated, which are usually called single-database PIR schemes. User privacy in these schemes is either ensured only against computationally-bounded attackers, or relies on trusted hardware. More precisely we focus in the single-database computationally-Private Information Retrieval setting, where no trusted hardware is used. Instead, these schemes are based on cryptographic algorithms allowing to compute over encrypted data. This replaces the assumption of the presence of a tamper-proof trusted hardware by an assumption on the computational security of the cryptographic primitives used. However, this comes at a price.

A. Performance Issues in PIR

A major issue with computationally-private information retrieval schemes is that they are computationally expensive. Indeed, in order to answer a query, a database must process all of its entries. If a given 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 PIR query is therefore linear[1] on the database size. 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 [5].

In the 14th Annual Network and Distributed System Security Symposium (NDSS'07), Sion and Carbunar presented a paper on single-database PIR practicality [1]. They showed that the existing, number theory based, single-database PIR protocols were not practical and that it was always faster to send the whole database than to compute a PIR 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 PIR scheme to process the database at more than a megabit per second. Sending the whole database over most of the current Internet connexions 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.

B. Related Work

As number theoretic approaches have failed to provide efficient enough PIR schemes, some alternatives to number theory have been explored. To the best of our knowledge, the first of such schemes was proposed by Kiayias and Yung in 2001 [6], but was based on the noisy polynomial reconstruction problem which was solved in 2003 by Bleichenbacher Kiayias and Yung in [7] and by Coppersmith and Sudan in [8]. In 2006, Gasarch and Yerukhimovich published a draft [9] of a scheme based on Regev's lattice-based cryptosystem [10], in which modular multiplications were replaced by additions of vectors with many coordinates. Such a replacement was indeed a positive step forward towards practical PIR, but the underlying cryptosystem used was extremely inefficient and thus the PIR protocol was hardly usable. In 2008, Aguilar et al. [11] presented another lattice-based PIR scheme. The same authors published in [12] a GPL implementation that can process a database on a high-end processor at 200Mbits/s. A parallelized version running at 1Gbit/s on a CUDA enabled GPU was also presented. In 2010, Trostle and Parrish proposed a scheme [13] based on trapdoor groups announced at a database processing throughput for a CPU implementation slightly below the one of Aguilar et al.

These schemes represent the state-of-the-art in efficient private information retrieval. Together, these works total about eighty citations, some of them in major and recent conferences such as Financial Crypto [14], PETs [15], [16], [17], Usenix Security [18] or NDSS [19]. It is worth noting that these papers do not present new PIR schemes but rather use [11] or [13] as a building block.

[1]In fact, in [4], Lipmaa proves that using a particular representation of the database, this lower bound on computation is slightly sublinear (in $O(n/log(log(n)))$).

Despite their success, [11] and [13] share a number of issues: they are both based on non-standard problems: [11] on the decisional knapsack vector problem which is in fact an aggressive version of a Ring-LWE encryption scheme and [13] on an adhoc problem they coined trapdoor groups; the parameter setup is to be done manually, which can be a very tedious task for both non-specialists and specialists; the performance achieved are not satisfying for most real practical applications unless a powerful GPU is used.

Oblivious RAM (ORAM) protocols, which are used to access (and write on) a database privately, can handle very efficiently databases of many Terabits. It is therefore very important to note that ORAM and PIR protocols are used for different applications and cannot be exchanged one another. Indeed, in the ORAM setting the database content is encrypted data outsourced from the user. ORAM cannot be used in any efficient way to privately download elements from a public database (e.g. Netflix) which is the paradigm of PIR (see Section V).

Important note: In [20], a *communication* efficient PIR protocol, from an asymptotic perspective, is built based on a fully-homormorphic scheme. Our work focuses on usability for which the main bottleneck is the *computational* efficiency of the protocol. The scheme we use is just an additively-homomorphic building block in [20] and our contributions are related to attaining practical performance and showing that it leads to usability in real settings.

C. Contributions and Roadmap

First and foremost this paper shows that PIR is a usable primitive in a large variety of settings, with standard security assumptions and conservative parameter choices. Section III is dedicated to prove this assertion. This contradicts the main result from Sion and Carbunar [1], which was the reference on PIR usability. The analysis of Sion and Carbunar remains correct, but one of their main assumptions (that PIR 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, the NTT and Ring-LWE based private information retrieval scheme and implementation we describe in this paper offer multi-gigabit processing throughput on a commodity CPU, and an optimizer to automatically setup the system for a given setup (hardware/network/application/security). This contribution required a considerable conception and development effort, as in order to maximize performance we had to circumvent standard big number and linear-algebra libraries, and optimization for a large variety of settings required many extra remote measuring and configuration function orchestrated by an optimizer.

Third, some of the internals of our library, such as the NFLlib object, which allows fast computation with lattice-based cryptosystems, may also be seen as a contribution. We discuss the comparison with related work on these building blocks in the associated sections.

In Section II, 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 PIR called private keyword searching, in which instead of retrieving elements by their index as usually done in PIR we retrieve elements based on the keywords they are associated to. In Section III, we directly jump into the performance analysis of our library. The objective of this section is twofold: show how our library behaves on a large variety of settings and motivate the reader to dig into the details of Section IV, which presents the structure of our library and explains the performance results. The paper ends with a description of some further related work to compare PIR to other primitives in Section V and a Conclusion. The appendix presents the cryptographic system based on Ring-LWE for which most of our performance results are presented as well as a discussion regarding security (parameter choices, randomness generation, etc.). The appendix can thus be skipped by the non-specialist but is included in the paper to show that the performance obtained is by no means the result of aggressive parametrization or of using an underlying non-standard cryptosystem.

II. Basic Tools

In order to allow two different levels of reading (one for the non-specialist in cryptography, and a deeper one) we split the following subsections in two : we first introduce the most basic and important facts and then give more details and formalism.

A. Homomorphic Encryption

Basics: In this paper we are interested in additively homomorphic encryption schemes. These encryption schemes are randomized, that is for each plaintext there are many possible ciphertexts. Encryptions of the plaintext 0 can be combined with some data m through an operation we call `Absorb` and the result will remain an encryption of 0 (we say they erase the data). Encryptions of the plaintext 1 can also be combined with some data m through `Absorb` and the result will be an encryption of m (we say they absorb the data). It is also possible to combine ciphertexts with an operation we note `Sum` which obviously results in the sum of the associated plaintexts.

As for each plaintext there are many possible ciphertexts, the ciphertexts space must be larger than the plaintext space and so must be their bitsize. We note F the expansion factor of the cryptosystem (which is defined as the size in bits of the ciphertexts divided by the size in bits of the plaintexts). This factor is typically a small number $F \geq 2$. As a reference, Figure 1 presents some plaintext and ciphertext sizes for different parameters of our Ring-LWE based encryption scheme.

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: Some 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 of each coefficient (which is stored in 64bit integers). 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.

More precisely: Additively homomorphic encryption schemes are defined by four algorithms: KeyGen, to generate keys; Enc the encryption function; Dec the decryption function; Sum which takes as input a set of ciphertexts $\alpha_1, \cdots, \alpha_n$ with corresponding plaintexts $a_1, \dots, a_n$ and outputs a ciphertext α with corresponding plaintext $a_1 + \dots + a_n$; and Absorb which takes as input some data m and a ciphertext of i and outputs a ciphertext of $i * m$.

It is worth noting that to be secure (see the Appendix for a more formal definition) such a scheme *has* to be randomized. More formally, Enc is a randomized algorithm that for an input (pk, a) outputs a ciphertext from a large set following a given probability distribution. It is also worth noting that in our application of homomorphic encryption, the same entity encrypts and decrypts data and can use a secret key homomorphic encryption scheme.

Lattice based cryptography has brought to homomorphic encryption the possibility to build much more versatile schemes than the ones we use in this paper, the so called fully homomorphic encryption schemes (see [21] for the seminal result and [22] for references on this prolific field). But from a fundamental point of view it has done more than that. It has completely changed the underlying mathematical structure, and one of the consequences is that we can use tools that greatly accelerate the `Sum` and `Absorb` operations, which are fundamental to PIR protocols.

In this paper, we use a Ring-LWE based homomorphic encryption scheme: the symmetric scheme presented in [20] with some pre and post-processing to improve performance in the PIR setting. As the pre and post-processing is public and reversible, security is directly based on the ring learning with errors problem (Ring-LWE) [23], as for the unmodified scheme in [20]. The hardness of Ring-LWE is one of the major assumptions used to build lattice-based cryptosystems, and since it was presented at Eurocrypt'10, it has become probably the most standard and used one.

B. Private Information Retrieval

Basics: In this paper, we use a simple PIR protocol based on [24], described hereafter. It can be used with any additively homomorphic encryption scheme. The server hosts a database of n ℓ-bit files. The client sends a query of n ciphertexts. The i-th ciphertext will be combined to the i-th database element through the `Absorb` operation by the server. Thus, a client wanting to retrieve the i_0-th element of the database will form the query so that all the ciphertexts are different encryptions of 0 except the i_0-th which is an encryption of 1. Using the basic properties of the encryption scheme described above, when the database will do the absorb operations, all the elements will be *erased* (*i.e.* become encryptions of 0) except the i_0 which will be *absorbed* (become an encryption of the i_0-th database element). The database then calls `Sum` over the resulting ciphertexts and sends the result to the client. The client decrypts it and will obtain the i_0-th database element as desired.

If the encryption scheme ensures that the database cannot distinguish between encryptions of 0 and encryptions of 1 (which is the security definition required for the scheme as stated in the Appendix), the database cannot know which is the absorbed element and which are the erased ones and thus cannot know which element has been retrieved.

With this simple approach, query size is n times the size of a ciphetext and reply is roughly $\ell \times F$, F being the expansion factor of the encryption scheme used (more details below). To reduce query size it is possible to use this protocol recursively. We describe recursion below but it can be considered as a black-box operation which takes as parameter an integer d called *dimension* and results in a scheme in which the client only needs to send $d \times n^{1/d}$ and the reply will be of size $F^d \times \ell$. 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 3×100 ciphertexts and get the database element with an expansion factor of 8 ($d = 3$);
- ...

More details: The protocol can be formally described as follows:

Basic PIR Protocol

Setup (user):

1) Set up an instance of the encryption scheme with security parameter k.

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) Compute and return $R := Sum_{i=1}^{n} Absorb(m_i, q_i)$.

Information extraction:

1) Decrypt R and recover m_{i_0}.

It is important to note that if database files are large it is possible to process the database iteratively. For large files, as ciphertexts can only carry a limited amount of information, the database is chopped into adequately sized chunks and one reply is generated for each chunk using the above scheme. The client then has to concatenate the decrypted chunks in order to obtain the complete file.

As noted, to reduce query size it is possible to recursively use this protocol. The basic idea is that we can split the database in $\sqrt{n}$ databases (to avoid cumbersome notations we suppose that $\sqrt{n}$ is an integer). Suppose the element the client wants to retrieve is in the i-th position of the j-th sub-database. The client sends a first PIR query to retrieve an i-th element from a database with $\sqrt{n}$ elements. For each of the sub-databases, the server computes a PIR reply based on that query. Instead of sending these replies, he stores them as a new temporary database containing $\sqrt{n}$ elements (as there is one reply for each sub-database). The user is interested in only one of those replies, the one that comes from the j-th sub-database. He therefore sends a second PIR query for retrieving a j-th element from a $\sqrt{n}$-element database. The server computes the PIR reply which is sent to the user. Decrypting this reply the client gets the j-th element of the temporary database, and decrypting this again he obtains the element he wanted: the i-th element from the j-th database. Of course the client can send both queries together and this can be generalized to any level of recursion. In practice, a recursion of d levels leads to a query size in $O(dn^{1/d})$ and a reply size in $\ell \times F^d$.

In Figure 2 a database of nine elements is divided in three elements. The client wants to retrieve the eighth and sends two queries: one that allows him to retrieve the second element of each sub-database; and another one that allows him to retrieve the reply from the third database. This tree representation highlights the generality of the approach. By adding a level to this tree it is clear that a user sending three queries can retrieve an element among 27 and so on. 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 [25] 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 [25].

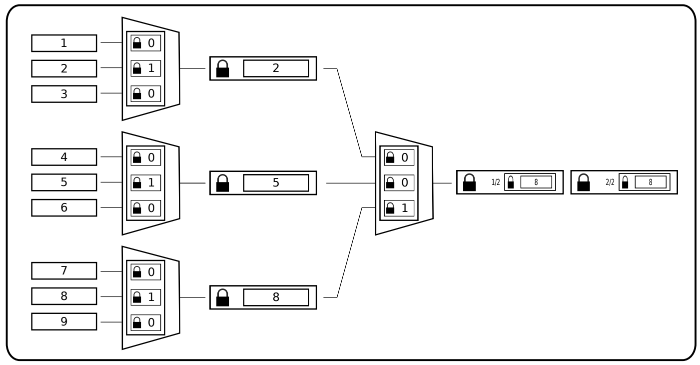

Fig. 2: Recursive usage of a PIR scheme.

C. Private Searching

Basics: The basic idea of private keyword search [26] is that the database can arrange its elements by grouping them using keywords. With this technique, users can get, using a PIR 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 (for keyword 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. More precisely, the computational cost will be the database size times the average amount of keywords a database element matches (which depends a lot on the application).

It is also possible to use this approach to filter streamed data based on private criteria [27]. The idea is to build ephemeral keyword-based databases for each message passing on the stream. 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 III 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 PIR query selecting the IP that is secretly 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 PIR query.

More details: Note that the filter always outputs a ciphertext and that it is not possible to distinguish useful outputs from encryptions of zero. It is however 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, with little overhead. These techniques are beyond the scope of this paper (see [28] for the most recent proposal on the subject).

III. Performance Analysis and Use-Cases

Our library is modular and allows several choices of underlying encryption schemes. The optimizer tests which approach (full database download with no cryptography, PIR with Paillier's cryptosystem, PIR with the lattice-based encryption scheme) is able to give the best results and advices the PIR client program to use it. **In this Section, we focus on results with our underlying lattice-based encryption scheme which generally gives the best results, and thus all the performance results are obtained forcing the optimizer's choice.** A discussion on when the optimizer will choose other alternatives is given in Section IV-D.

In this section, we analyze the performance of XPIRe using essentially two metrics: latency and user-perceived throughput. The latency measurement is the round-trip time from the moment the client starts generating the PIR query to the moment it has finished to decrypt the reply received from the database. Our library pipelines the different phases: query generation and query sending in one direction; and reply generation, reply sending, reply reception and reply decryption in the other direction. Thus, latency is supposed to follow the formula: MAX(queryGenTime, querySendTime) + MAX(replyGenTime, replySendTime, replyDecTime). Our optimizer can take this into account and select the different available parameters to minimize this value if asked to.

User-perceived throughput is the throughput (measured in bits per second) at which the user is able to get the requested element. Again, due to the pipelining between the server and the client this value is supposed to follow the formula: MIN(processingTput/n, serverUpTput, clientDownTput, clientDecTput), *n* being the number of elements in the database. Indeed, the server processes the *n* elements iteratively at a given throughput, and sends a processed chunk of information for every *n* chunks of the database processed (hence the quotient in the first argument of the MIN function).

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 IV-B. 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 PIR transaction in which they will be used.

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 private stock-market information service (dynamic data).

Experimental setting: To show that our library is usable by everyone for many applications we use commodity hardware in almost all the settings. Our PIR 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-preprocessed. 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.

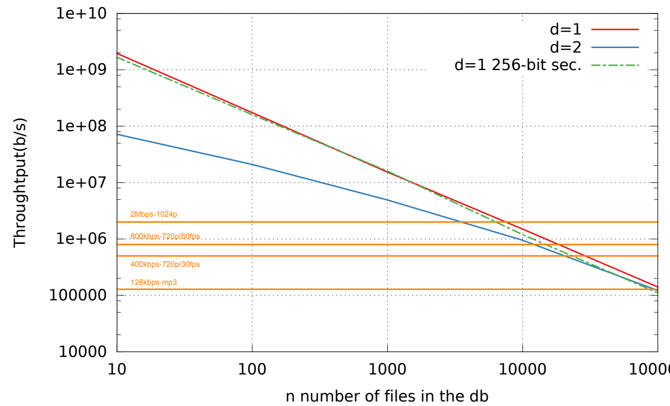

Fig. 3: User-perceived throughput of XPIRe streaming static data on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. The red (91 bits security) and green (256 bits security) lines give throughput when no recursion is done (i.e. database is processed as a one dimension array) and the blue line with one 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 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.

Security: In most of our performance results, our optimizer found that the best parameters for our Ring-LWE scheme were: polynomials of degree 2048 and a modulus of 120 bits, or polynomials of degree 1024 and a modulus of 60 bits. According to the usual assumptions presented in the Appendix, the former set of parameters is able to provide 91 bits of security, and the latter 97. As noted in the appendix, we use noise large enough to attain such security. To generate this noise, we use Salsa20/20 [29] (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 increases exponentially with the polynomial degree and computational costs increase only (almost) linearly. For example, if we use polynomials of degree 4096 (instead of 2048) with a modulus of 120 bits, 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.

A. High Throughput on Static Databases

Figure 3 shows the user-perceived throughput achieved using our library on the experimental setting laptop. 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 III-C. 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.

The red line shows performance when no recursion is done (i.e. when the database is seen as a one dimensional array of n elements and 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 polynomials of degree 2048 and a modulus of 120 bits. 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 4. 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 4). 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.

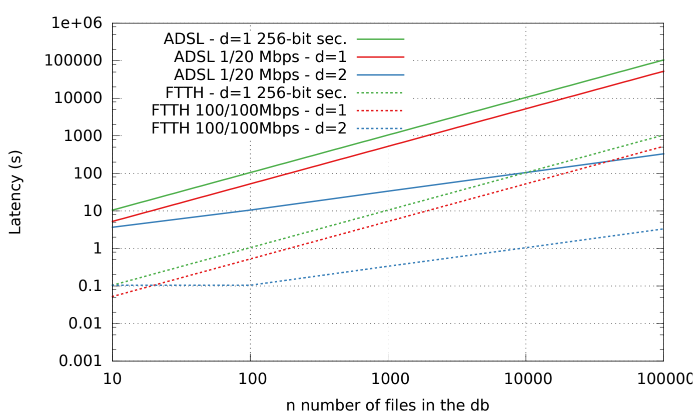

Fig. 4: Initial latency before the user starts to receive streaming data (mainly due to query generation and transmission time to the server). Filled lines are for ADSL and dotted lines for FTTH. Red, green and blue colors are associated to the same settings as in Figure 3. The lines highlight that latency 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 4 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 for a given application.

The Netflix Use-case: The Netflix movie database is composed[2] of 100.000 movies that can be streamed to individual users. These movies are stored as individual static files and can thus be pre-processed offline for performance improvement. H.265 - High Efficiency Video Coding (HEVC) is the forthcoming standard for videostreams compression[30], [31]. The attained compression levels with this codec enable to watch 720p streams at bitrates between 400Kb/s for 30fps and 800Kb/s for 60fps and 1024p at 2Mb/s. A typical bitrate for audio streams is 128Kbps for quality MP3s. These levels (128, 400Kbps, 800Kbps and 2Mb/s are represented by horizontal dashed lines on Figure 3).

Henceforth, a private Netflix-like server based on XPIRe can allow a user to privately receive a streamed movie with different tradeoffs 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.** He can get better video quality at the expense of some privacy, hiding his choice among *just* 17K movies he can get a stream at 720p-60fps and hiding it among 7K movies he can get it at 1024p-60fps.

To reach that level of privacy, the server has to dedicate one full processor per user. However, if *k-anonymity* is enough, the movie catalog could be randomly (and diversely) arranged in smaller sets of k files in order to reduce the computation necessary for the server. For example, **if movies are arranged in random anonymity sets of 100 movies, each processor can serve up to 350 users (twice that number with a higher tier processor than the one we used)**. We are conscious that the use of PIR in the Netflix scenario, while certainly good for privacy may prove to be a problem regarding both copyright management and accounting. We essentially used this use-case as an illustration of the excellent performance attained by XPIRe, at a massive scale, and not to discuss the fact that Netflix should use PIR to stream its clients, or that it would be commercially possible.

Medium Access Issues: Obtaining 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 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 (as 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 anonymity sets (or amount of users a processor can handle) would be divided by a factor seven. We consider though that in

[2] or was composed in 2009 according to the Wikipedia page for Netflix.

applications requiring very large databases and throughput, such as the Netflix-like 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 contiguous read throughput, at roughly a cost of 1000$. Note that if the server has multiple clients in parallel, the disk access cost does not grow if the threads access data synchronously, and thus scalability is not necessarily an issue.

B. High-Throughput on Dynamic Databases

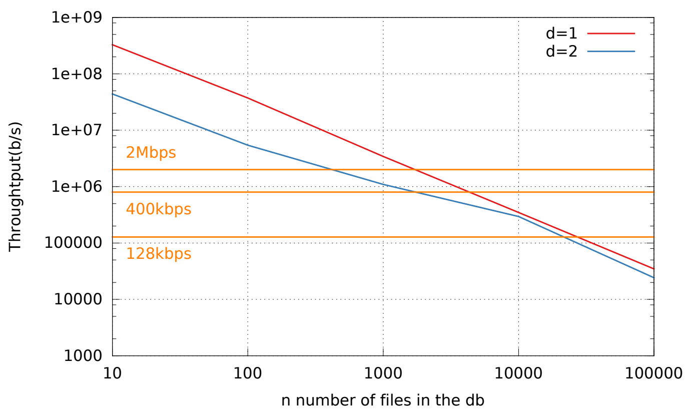

Fig. 5: User-perceived throughput of XPIRe streaming dynamic data on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. The red line gives throughput when no recursion is done (i.e. database is processed as a one dimension array) and the blue line with one level of recursion (i.e. database is processed 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 (note that scale is logarithmic).

At first sight, dynamic databases are similar to static ones apart that data is dynamic and 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 show two different settings : IPTV and a private sniffer.

The first setting is pretty simple : usual datastreams that cannot be preprocessed such as for IPTV. Figure 5 presents the same results as 3 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. As described in Section II-C, this is possible using PIR. With this approach, a PIR 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.[3] 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.

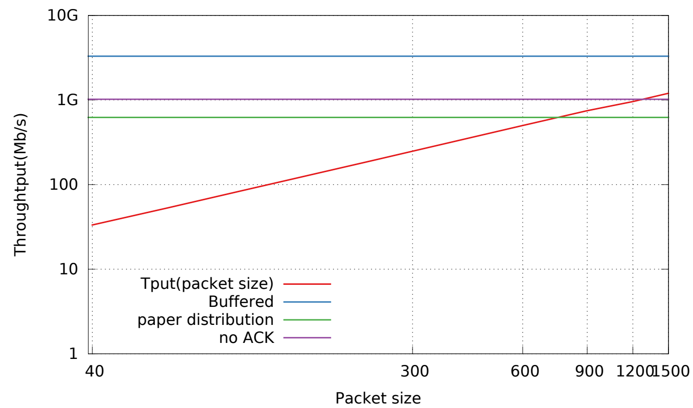

Fig. 6: Packet processing throughput for the sniffer use-case using XPIRe on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. In the red line, performance is measured for each packet size, in bytes in the x-axis, independently (i.e. measuring performance just processing 40 bytes packets, then measure performance for 80 bytes packets etc.). The green line gives the processing throughput when the traffic follows a classic bimodal distribution such as found in [32]. The purple line gives throughput for a traffic following the same pattern but for which we ignore packets smaller or equal to 60 bytes (basically ACKs). The blue line gives performance if we wait for traffic to fill buffers and only generate PIR replies when enough information has been collected to fill a ciphertext.

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 PIR reply storing the reply in the disk using the compression techniques described in Section II-C. 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 6 gives the throughput at which the sniffer is able to process the intercepted packets. As packets are much smaller than classical plaintext size, we choose 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 that plaintext size is 15Kbits. If we generate a PIR reply for each 40 byte incoming packet, most of the space available in the resulting ciphertext will be lost, but the PIR reply generation operation will not cost less (for null elements the operation is free, but for small

[3] 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.

elements the operation costs as much as for elements of the size of a plaintext). Thus, if we deal with packets of 400 bytes instead of 40, the PIR 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 classic Maximum Transmission Unit -MTU- of 1500 bytes), a packet always fits in the plaintext, the processing throughput is linear on the packet size. **If we consider a classic bimodal distribution** (40% very small packets, 40% close to MTU packets, 20% in-between packets) such as those described in [32], **the sniffer is able process a link at 600Mbps** (green line). **If we consider the sniffer is not interested in very small packets (ACKs mostly), it can process a link at slightly over 1Gbps** (purple line). **If we buffer packets** and do not generate a PIR 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 $(2048, 120)$ (or $(4096, 120)$ if 256 bits of security are desired). 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 PIR 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.

C. Low Latency on Static/Dynamic Databases

In this Section, we want to evaluate XPIRe latency, *i.e.* round trip time (RTT), in settings where data is static or dynamic. Figure 7 shows the RTT achieved using our library on the experimental setting laptop with static data and Figure 8 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 PIR such as aggregation and recursion (see Sections II-B and IV-C) are mandatory to keep RTT low. In Figure 7 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 aggregation and recursion, when beneficial, the optimizer can set the PIR 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 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 processing speed that is impacted by the need to preprocess the data in the dynamic case. One can observe the different

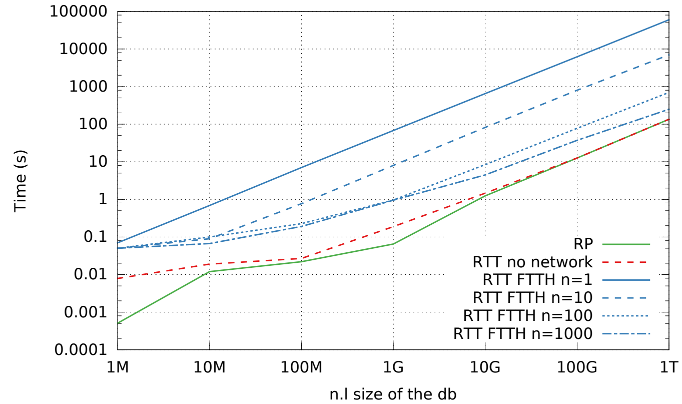

Fig. 7: Round-trip (RTT) and request processing (RP) times of XPIRe serving static data on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz on a FTTH network. When the client is local RTT (red line) matches RP (green line), specially for large databases. Each blue line gives RTT for a fixed n and varying database sizes. For large databases reply size is the limiting factor, which explains why performance is closer and 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 n grows because the optimizer uses aggregation to reach the best RTT.

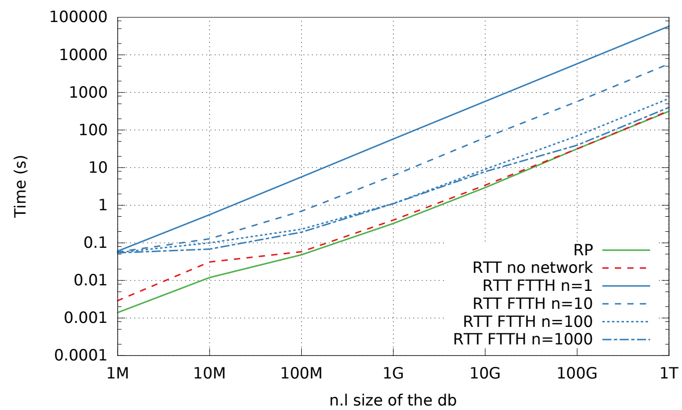

Fig. 8: Round-trip (RTT) and request processing (RP) times of XPIRe serving dynamic data on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz on a FTTH network. The comments on Figure 7 also apply. As data is not already pre-processed, request processing time is higher, but upload/download times do not 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.

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 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), and maybe some other personal choices they don't care to reveal. This first set of public parameters will allow to reduce the database size over which a second search, based on private criteria, will be done. The users can then do a PIR with keyword search (see Section II-C) to get all the profiles matching a set of private keywords. If we suppose the database has on million profiles, each of one megabit, the complete database will be of one Terabit. We must also take into account that each profile will probably 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 7 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 profles of multiple megabytes, public filtering will have to be much more efficient. But the fact that we are able to grasp having usable PIR protocols in such large social networks was unthinkable not that long ago.

NYSE Use-Case: In this last use-case, we are interested in exercizing 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 usage with this application: oriented towards throughtput or towards latency. In the first case, a client may want to register to a given set of streams of information, e.g. the ARM Holdings plc (ARMH) stream, and get served with all the information concerning this company 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 datastream concerns financial information instead of a TV stream. Refer to Section III-B for the performance in this situation.

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 8 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.

IV. Subroutine Analysis

A. General Architecture

From a logical point of view, our proposal has three different blocks: a set of homomorphic encryption schemes (Paillier, Ring-LWE-based, NTRU-based); a classic, homomorphic encryption based, PIR protocol (client, server, query generator, reply generator, reply extractor); and an optimizer which provides the best parameter settings for the PIR protocol and encryption scheme, given a strategy (smallest RTT, least resources, lower price) and a set of fixed parameters (database description, security, bandwidth, computational power). We will therefore present the library following the same structure.

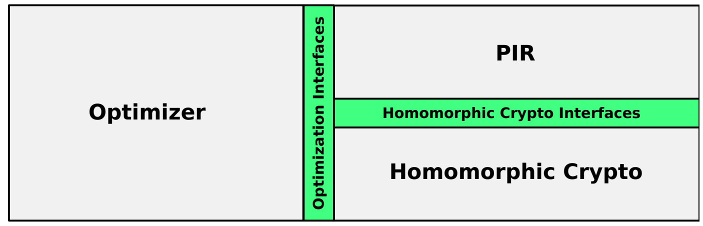

Fig. 9: Architecture of the library

Modules are pluggable and only need to implement given interfaces (in green in Fig. 9) to work together. Encryption modules can be easily added in our architecture. Replacing the PIR application is also possible but requires more work and such a possibility is beyond the scope of this paper.

B. The Homomorphic Encryption Module

On the homomorphic encryption module our main contribution is a C++ class we named NFLlib (after NTT-based Fast Lattice library). This class provides a set of tools to create and manipulate polynomials on the classic ideal lattice setting (i.e. working modulo an integer and modulo a polynomial $X^n + 1$, n being a power of two). This class uses different techniques, namely the Number Theoretic Transform (NTT) and a Chinese Remainder Theorem (CRT) representation (see the following below for a detailed description) to ensure high performance.

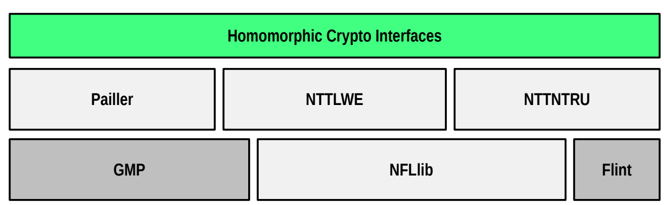

Fig. 10: Encryption module architecture

Using this object we have implemented two lattice-based encryption schemes: NTTLWE, based on a classic Ring-LWE approach [23]; and NTRU, based on the recent provably secure versions of NTRU [33]. In order to do comparisons with classical homomorphic encryption schemes we have also done a Paillier implementation based on GMP [34]. In Fig. 10 we have tried to illustrate the fact that NTTLWE and NTRU essentially rely on NFLlib for their operations and the less possible on multi-precision libraries such as GMP and FLINT [35]. Paillier's scheme, on the other hand, which is based on large integer modular exponentiation, is entirely based on GMP.

1) NFLlib: When working in an ideal lattice setting, such as the one of Ring-LWE and NTRU, we usually work with polynomials of a given degree n and coefficients are taken modulo a given integer p. Typically, n is a multiple of two above 512 and p is a product of primes of at least 30 bits. When many homomorphic operations are to be done p grows rapidly, and in the extreme case of fully-homomorphic encryption its bitsize grows to the hundreds. Additions of polynomials in these rings are done coefficient by coefficient and multiplications are polynomial products. Coefficients are reduced modulo an integer and polynomials are reduced modulo $X^n + 1$, which ensures constant representation size for the polynomials when going through such operations. Products with polynomials of such a high degree incur huge computation costs, with standard multiplication in $O(n^2)$ and even with other techniques such as Karatsuba in $O(n^{1.585})$.

In the NFLlib object we use three techniques to reduce computational cost: NTT representation of polynomials, CRT representation of integers, and pre-computation of quotients to use Shoup's fast modular multiplications.

The Homomorphic-Encryption Library of Halevi and Shoup [36] implements the encryption scheme of Brakerski, Gentry and Vaikuntanathan [37]. They provide an object they called Double-CRT which provides NTT and CRT representation of polynomials as NFLlib does. We will compare to this work at the end of this section.

a) The Number Theoretic Transform: Fast Fourier Transform (FFT) and more specifically the Number Theoretic Transform (NTT) has recently been used for obtaining quasi-linear speed in $O(n \log n)$ for polynomial multiplication [38]. In particular, in [29] NTT is used for providing a very efficient signature scheme based on lattices. The NTT transforms a vector of coefficients representing a polynomial into another vector in the same domain which represents the evaluation of the polynomial into specific locations. With polynomials represented by such vectors polynomial multiplication is done coefficient-wise, resulting in a multiplication cost in $O(n)$. Transforming a polynomial into an NTT form or back has a cost in $O(n \log n)$.

It is usually said that FFT-like multiplication has a computational cost in $O(n \log n)$ but this is because the normal multiplication of two polynomials is compared to transforming them to NTT form, multiplying them coefficient-wise, and transforming them back to the usual coefficient vector form.

Our performance highlights the fact that, in a PIR protocol, we only need to do the transforms at non critical moments: on query generation, database importation, and decryption. When generating a PIR reply, which usually is by far the most expensive operation in a PIR protocol, we only need to do the $O(n)$-cost multiplication which increases dramatically the processing speed. Query generation and reply extraction is usually done for relatively small amounts of data and database importation can often be done offline (pre-processing) and amortized by different clients.

Note that polynomial multiplication must be followed by a modular reduction by $X^n + 1$. As described in [29], before the NTT, the coefficients of a polynomial can be multiplied by the powers of ϕ, the primitive n-th root of -1, so that we get a modular reduction by $X^n + 1$ for free. In our implementation, we use Victor Shoup's algorithm for computing the NTT with David Harvey's optimizations [39] which is much more efficient than a naive NTT implementation.

b) Chinese Remainder Theorem: To maximize efficiency, the modulus used in an NFLlib instance must be a product of 60 bits primes. The idea is to use the Chinese Remainder Theorem to work on 60-bit integer tuples and thus profit from 64-bit operations while still having some spare bits for lazy modular operations. It is possible to change this setting to a product of smaller primes (e.g. $28 - 30$ bits will be better for a 32-bit architecture) just by changing some constants in our headers. To use the NTT techniques presented in the previous paragraphs we will need that a primitive n-th root of -1 exists when working modulo each of the 60-bit primes. As in [36] this will be ensured choosing primes which are congruent to 1 modulo $2n$.

The CRT states that for an integer $q = p_1 \times \cdots \times p_t$ such that the p_i are co-prime there is a bijection between elements modulo q and tuples of t integers such that additions and multiplications in the i-th coordinate are done modulo p_i. More formally, there is a ring epimorphism between $\mathbb{Z}/q\mathbb{Z}$ and $\mathbb{Z}/p_1\mathbb{Z} \times \cdots \times \mathbb{Z}/p_t\mathbb{Z}$.

In practice this means that a polynomial with coefficients reduced by a modulus $q = p_1 \times \cdots p_t$ of $t \times 60$ bits, can be transformed into t polynomials with coefficients of 60 bits (i.e. reduced modulo each of the primes p_i). If the t polynomials resulting from the CRT are in NTT representation, we will say that the initial polynomial is in NTT-CRT representation (this is called double-CRT in [36]). Combining the properties of the NTT and CRT representation we get that two polynomials in NTT-CRT representation can be added (or multiplied) by adding (or multiplying) them coordinate-wise. When the NFLlib object is used, this is how polynomials are represented and thus, using NFLlib, two polynomials of degree n and $t \times 60$ bits coefficients can be added (or multiplied) with $n \times t$ operations of the basic instruction set of a 64-bit machine. This greatly contributes to the performance results of the PIR protocol.

c) Shoup's Modular Multiplications: To multiply modulo a given p two 60-bit integers, 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 [39], David Harvey attributes to Shoup a very interesting approach to modular multiplications when the same multiplier is used many times.

The idea is to pre-compute for a given y a scaled a 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 [39].

The NFLlib object 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 schemes we built over NFLlib, only need to do a single polynomial multiplication to encrypt, and a single polynomial multiplication to decrypt, always using the same multiplier : the secret key (which is a polynomial). In the PIR protocol, the PIR 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.

d) Tools Provided: The NFLlib object has an initialization function, that for a given set of parameters initializes different variables needed for the NTT (roots of unity, its inverses, etc.), and CRT computation (pre-computed quotients of fixed integers for Shoup's modular multiplications, lifting integers, etc.). It provides some other useful functions such as: import/export functions to convert raw data into polynomials (by splitting it in sets of coefficients of given bitsize and computing the NTT-CRT representation) and back; arithmetic functions to perform additions, multiplications and fused multiplications-additions of polynomials (using Shoup algorithm to multiply the coefficients or not depending on the availability of pre-computed data); generation functions to get random[4] polynomials for a uniform distribution or for bounded coefficients following a gaussian (both are pretty useful for lattice-based cryptography); etc. The only function that is based on an external library is `poly2mpz` which transforms a polynomial in an NTT-CRT representation into a vector with the coefficients of the corresponding polynomial with arbitrary precision integers, using GMP. This function is exclusively used on decryption, and only for some parameters sets.

Parameters	$(1024, 60)$	$(2048, 120)$	$(4096, 120)$
Input size	20Kbits	100Kbits	192Kbits
Output size	128Kbits	512Kbits	1024Kbits
Expansion factor	≥ 3.2	≥ 2.56	≥ 2.65
Input throughput	4.8Gbps	5.2Gbps	5Gbps

Fig. 11: Preprocessing: deserialization (bit splitting) and NTT transform cost on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz, for different crypto parameters. Input data are supposed to be plaintext that need to be converted into polynomials in NTT form. Input sizes are the maximum plaintext sizes given in Figure 1. NTT results in an expansion factor if transformed polynomials have small coefficients (e.g. in the first setting 20-bit coefficients are transformed into 60-bit values stored in 64-bit integers), hence output sizes and expansion factor values. Performance is measured as the throughput at which the input data is processed. This value follows closely the inverse of the expansion factor as a smaller expansion factor results in processing more data for the same transform cost.

Figure 11 gives performance results for importation (deserialization and NTT transform). Importing data into NTT polynomials corresponds to the many times aforementioned pre-processing. The data splitting operation is pretty fast, and the main performance bottleneck is computing the NTT in our polynomial ring. Tests correspond to the same laptop as in Section III.

e) Comparison with [36]: The Double-CRT object proposed in [36] is different from NFLlib in two ways. First, it is more general as it can handle arbitrary polynomial degrees (and not just powers of two) and moduli which are products of primes with any size below 60 bits. It can handle dynamic changes in representation by adding more primes to the moduli or by removing them through an operation called *modulus reduction* which is very important for fully-homomorphic encryption. It also provides some more functions for polynomial arithmetic such as divisions, automorphisms, etc.

On the other hand, in non-fully homomorphic applications the Double-CRT is a little bit of an overkill. In order to use it, a fully homomorphic encryption environment must be created which ties the object to this setting. NFLlib only requires two integers defining a polynomial degree and modulus bitsize to be instantiated. Performance for polynomial multiplications and additions is roughly as good as with NFLlib for a similar setting, as Fig. 12 shows, except if we use quotient pre-computation in which case NFLlib is almost an order of magnitude faster. Performance has been tested on the same reference laptop as the one used in Section III (MSI GT60), but only using a single core (as Double-CRT is based on NTL which is not reentrant).

Class name	Multiplication	Addition
`Double-CRT`	3.4e-05	5.2e-06
`NFLlib-noShoup`	2.9e-05	2.3e-06
`NFLlib`	2.6e-06	2.3e-06

Fig. 12: Multiplication and addition times in seconds for polynomials of degree 1024 and 60-bit coefficients, *on a single-core* of a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. We replaced usual moduli in Double-CRT by a product of two 30-bit integers for the comparison to be fair. By default Double-CRT would split the 60-bit modulus in much smaller values resulting in much worse performance in this case.

Note that in our PIR protocol, we use Fused Multiply and Add (FMA) operations, which costs exactly the same as multiplications due to hardware separation. Using the multi-threaded capabilities of the reference laptop, and the reentrancy of our library, we are able to do a FMA of $(1024, 60)$ polynomials in 0.8e-06 seconds.

Finally, memory usage is much lower with NFLlib, 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 NFLlib 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.

NFLlib and the schemes we developed based on it will therefore 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 or moduli flexibility should use Double-CRT.

2) NTT-based Ring-LWE Encryption: Our scheme is basically the symmetric homomorphic encryption scheme of [20], which is described in the Appendix. The homomorphic encryption scheme resulting from the modifications we propose is, from a security point of view, equivalent to the scheme described in the appendix as all the modifications are public and reversible.

The basic idea is that the polynomials that usually describe the inputs (secret key, randomness, messages) are preprocessed by computing their NTT. After decryption an inverse NTT is performed to retrieve the message. With such a transformation, encryption and decryption can be done by coordinate-wise multiplication and additions which leads to very high performance results.

[4]As already stated, to generate randomness we use the pseudo-random number generator based on Salsa20/20 described in [29].

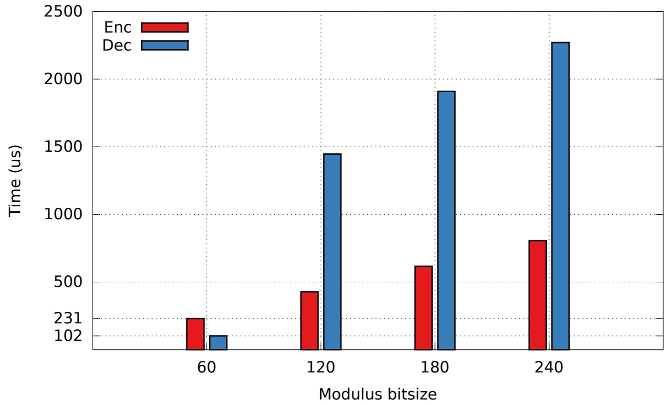

Fig. 13: Encryption and decryption times for polynomial degree 4096 and varying modulus size, on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz. Note that encryption costs increase linearly in the modulus size but also the size of the associated ciphertexts and plaintexts. The large jump in decryption costs comes from the usage of GMP for moduli strictly above 60 bits.

Describing how each algorithm is transformed by the usage of the NTT 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 compute a NTT. Indeed the NTT and inverse NTT 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 applying the NTT 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, or a constant). It is therefore always possible to use Shoup's rapid modular multiplications.

Having these two ideas in mind it is easy to see that encryption requires only the computation of a single NTT and some basic operations (negligible compared to the NTT). 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 in the scheme described in the Appendix, we need to get the value of each coordinate in non CRT representation. 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 13 shows this evolution.

This is the only point in which we use GMP on the NTTLWE object (by using the `poly2mpz` function of NFLlib). In practice this results in a very significant performance drop. Thus, even if higher moduli give better results for expansion factor or PIR reply generation throughput, the optimizer will almost never choose parameters with a moduli beyond 120 bits as decryption costs quickly become the main bottleneck for performance.

Note however that for a modulus of 60 bits, performance is surprisingly high. We are able to generate a query at 2.2Gbits/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 2.5Gbits/s, but decryption suffers from the CRT lifting and an incoming reply can "only" be decrypted at 710Mbits/s.

3) NTT-based NTRU Encryption: We used a similar approach to develop an NTT-based NTRU encryption scheme. However, even with a moduli of 60 bits, the decryption step requires a double precision floating point division for each coordinate of a ciphertext. The decryption performance is so low when compared with NTTLWE that NTTNTRU is never chosen by the optimizer as a suitable replacement of NTTLWE. For these reasons we will not provide specific details about this part of the library.

C. The PIR Module

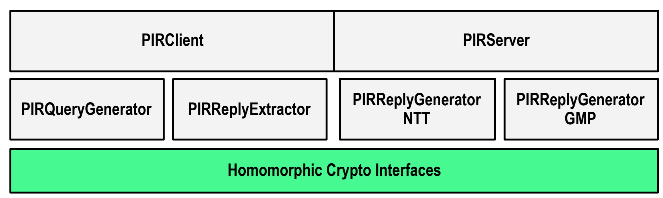

Fig. 14: PIR architecture

To be precise, the PIR module as described in the general architecture is composed of a sub-module which contains all the PIR objects and functions (query generator, reply generator, reply extractor); and a wrapper sub-module which provides the classical PIR on a database application (client, server). This wrapper module also implements most of the interface to optimize this application through functions allowing to measure network and PIR processing performance.

Our main contribution in this module is the generality and flexibility it provides. It can handle PIR techniques such as recursion and aggregation (more on this below) and can be used with any homomorphic encryption scheme implementing a given interface.

a) Recursion and Aggregation: The PIR protocol used is basically the protocol of Kushilevitz and Ostrovsky. By itself the protocol is not a contribution of this paper (but to the best of the authors knowledge this is the first time it is fully implemented with all its generality) so we won't describe it more in detail than what we did in Section II-B. As noted in that Section, in order to download the i-th element of a database with n ℓ-bit long elements, a client using d levels of recursion sends a query composed of $d \times n^{1/d}$ ciphertexts and receives a database reply of size $F^d \times \ell$, F being the expansion factor of the encryption scheme used.

If a database has many small elements, it may be interesting to aggregate them. The PIR Client and Server are able to do element aggregation. With it, for any integer α, the n elements of ℓ bits are seen as n/α elements of up to $\alpha \times \ell$ bits (with some rounding and padding if n/α is not an integer). One natural case in which aggregation will be sued is if absorption size is larger than ℓ. For example if each ciphertext can absorb 32kbits of information but database elements are only 1kbit long aggregating them in groups of 32 elements will reduce query size without increasing reply size.

b) The Homomorphic Encryption Interface: The PIR module has been built so that any homomorphic encryption scheme implementing a given set of generic interfaces can be used with it. Of course, the encryption scheme must implement basic encryption operations such as encrypt and decrypt and basic homomorphic operations such as `Absorb` and `Sum`, but there are other less evident functions that need to be implemented. For example, the cryptosystem must be able to propose a set of instantiations for a given security level. It must also be able to do self-tests for client and server performance, and provide informations such as plaintext, and cihertext, and key sizes, etc. All these functions are documented in the library.

D. Optimizer

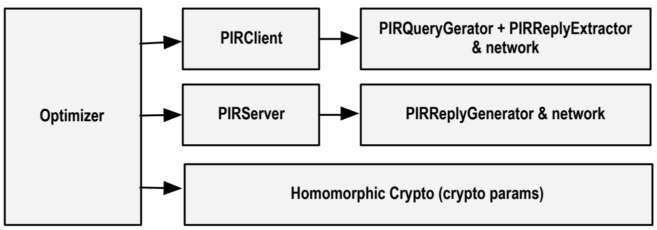

Fig. 15: Optimization architecture

The optimizer module provides the best parameters given a target function: round trip time (since the query starts being generated until the reply is completely decrypted), total resources (total cpu time for the client and server plus total network time if bandwidth is fully used), or cost (based on a cloud service dollar cost per second of CPU and Gigabyte of network usage of a quad-core server). By default these function are optimized in a *reasonable* mode in which the function to minimize is added to a scaled version of the total resources so that the amount of resources used does not grow to much for a small improvement on the unmodified target function. It is possible to change this behavior at compilation to strictly optimize unmodified target functions. Adding or modifying target functions is also very easy.

c) Modus Operandi: The optimizer can be called directly. However, the PIR client has some nice tools that are skipped if we use it that way. If a user only wants to obtain the optimizer results he can call the PIR client in dry-run mode. The PIR client, whether it is on dry-run mode or not, parses the options from the command line and configuration files, does network performance tests and gives a set of fixed variables to the optimizer. The optimizer searches the best parameters considering those fixed variables as constraints.

Among these variables there is a database description (number of elements and size), bandwidth values, minimum security bits, the target function and constraints on the different parameters. Among those constraints there can be maximums, minimums, or fixed values for parameters such as aggregation and recursion, and a given encryption scheme or even a given set of parameters for an encryption scheme. Given these constraints, the optimizer defines a space of possibilities to explore and does a full search estimating the target function values to find the best option. The only exception to full search is aggregation, for which there is a dichotomy until the number of possibilities is small enough. The reason why we do a dichotomy for aggregation is that its effects are pretty easy to evaluate when the space of possibilities is big: query related costs

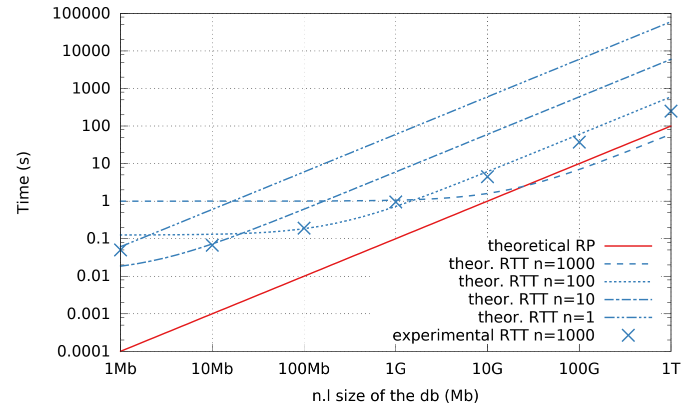

Fig. 16: Theoretical and experimental Round-trip times (RTT) and theoretical request processing times (RP) of XPIRe serving static data on a MSI GT60 laptop with a Core i7-3630QM 2.67GHz on a FTTH network. The experimental results (crosses) show that the optimizer defines the aggregation parameter to lower n (the number of database elements) when needed so that the performance result always matches the best theoretical line. Results are optimal for small databases and reasonable but not optimal for large databases.

go down and reply related costs go up. For other parameters this is not true and thus we can hardly do convexity assumptions. Even with the full space search for most of the parameters, the optimizer runs in a few milliseconds and gives very good results.

To illustrate the optimizer work, Figure 16 shows both the theoretical and experimental round trip time for various database sizes and n values. The red line represents the theoretical request processing necessary to handle the database size, the blue lines represent the theoretical RTT when it is bounded by networking issues (uploading requests or downloading replies). These theoretical lines represent what would be the RTT for various n when no aggregation nor recursion is used. For example, for $n = 1000$ in 1Mb to 1Gb databases, uploading the request would take 1 second. The blue crosses show the experimental results obtained with the help of the optimizer. As one can see, in practice, the optimizer chooses the PIR parameters in order to match (almost all the time) the best RTT attainable. For example, in the case of 10Mb database, it used aggregation to match the theoretical $n = 10$ performance. The differences observed in some cases can be explained by practical concerns, *e.g.* the 0.7 second RTT for 1Mb database is due to delays introduced by the networking layers or the difference with the optimal RTT (near RP time) for large databases such as 100Gb and 1Tb are due to the fact that the optimizer chose recursion instead of aggregation but underestimated its cost. Generally speaking, we can say that the choices of the optimizer are not always the perfect ones but on the other hand are always reasonable.

d) Other cryptosystems: As noted before, in almost all situations the Ring-LWE based PIR is chosen by the optimizer, as it gives the best results. In some extreme cases however, the optimizer chooses to do a Paillier based PIR or a trivial (full-database download) PIR. The Paillier based PIR will be chosen for extremely small bandwidths in which case the PIR 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 15Gbps for static pre-processed databases, and 3Gbps for dynamic databases.

Other 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 PIR 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 are pretty scarce.

V. Further (and Farther) Related Work

A field of research closely related to single-database PIR is the one of Oblivious Transfer [40]. Oblivious Transfer schemes are single-database SPIR schemes except that, instead of aiming to limit the communication cost, they aim to limit the computational cost for the user and the database. In such schemes an encrypted version of the whole database is usually sent to the user. Oblivious Transfer is a fundamental cryptographic primitive mostly used in theoretical proofs, the most interesting application being probably that it is complete for secure multi-party computation [41]. As Oblivious Transfer schemes are mainly theoretical tools and answer to different challenges from single-database PIR schemes, we do not consider them in this paper.

The first single-database PIR scheme was presented in 1997 by Kushilevitz and Ostrovsky [42]. In this scheme, user privacy is related to the intractability of a number theory problem, instead of based on the assumption that different replicas exist and do not collude against their users. Since then, improved number theory based schemes have been proposed by different authors [24], [43], [44], [45], [25], [46]. They are called computationally-private information retrieval (cPIR for short) schemes, as opposed to information-theoretic PIR schemes. As this is the only family of schemes we consider in this paper, we will refer to them as simply PIR protocols, without specifying that we refer to the single-database computationally-Private Information Retrieval setting.

In [3], Gasarch presents a survey of general PIR schemes, including replicated database schemes, primitives implying and implied by the existence of PIR schemes, theoretical bounds and many other subjects. In 2007, Ostrovsky and Skeith III [47] published an eprint transcription of an invited talk to the 10th International Conference on Theory and Practice of Public-Key Cryptography (PKC 2007), which also surveys PIR schemes. They present how this field of research is connected to other primitives like Oblivious Transfer [48] or collision-resistant hashing [49], and provide a presentation of the different theoretical approaches used to obtain PIR schemes. For more details on these subjects, the readers are referred to these papers.

An important related research domain is the one of Oblivious RAM (or ORAM). ORAM schemes allow a client to conceal its access patterns to a remote storage by shuffling and re-encrypting data. Recent ORAM protocols (e.g. see [50]) are extremely efficient and allow read and write operations over large databases with small overhead, which has many applications, specially on the era of cloud computing. However, this setting is pretty different from the one of PIR as the ORAM data belongs to the client and is accessible only to him. Thus ORAM protocols do not allow random clients to access a public database hiding their access patterns.

It has been known for a while that it is possible to transform an ORAM protocol into a PIR protocol using a trusted hardware module (e.g. see [51]). Indeed, an ORAM client inside the module can build an encrypted version of a database in ORAM and retrieve elements for different clients. In such a setting the PIR clients only need to build and encrypted tunnel with the module (after proper authentication giving assurance to the users that they are communicating with a trusted hardware module), and tell the module which element they want to retrieve. 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.

VI. Conclusion

Lattice based cryptography has done a lot of noise with its breakthroughs on worst-case to average-case reductions and with 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 changed completely the attainable performance in a non-asymptotic sense. PIR has often been considered as a protocol that would never be practical [1]. Lattice-based cryptography brings a real overhaul on this, as PIR becomes feasible even for people that don't own 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. This 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 increase by a factor one hundred the times we presented in our experiments. These results are 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.

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] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of STOC'09*, pp. 169–178, ACM Press, 2009.

[22] H. Lipmaa, "Fully homomorphic encryption reference list."

[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] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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/.

[33] D. Stehlé and R. Steinfeld, "Making ntruencrypt and ntrusign as secure as standard worst-case problems over ideal lattices," *IACR Cryptology ePrint Archive*, vol. 2013, p. 4, 2013.

[34] T. Granlund, "GMP: The GNU Multiple Precision Arithmetic library." https://gmplib.org/, 1991–2014.

[35] W. Hart, F. Johansson, and S. Pancratz, "FLINT: Fast Library for Number Theory," 2014. Version 2.4.1, http://flintlib.org.

[36] S. Halevi and V. Shoup, "Design and implementation of a homomorphic-encryption library," 2013.

[37] 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.

[38] N. Gttert, 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.

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

[40] 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.

[41] J. Kilian, "Founding Cryptography on Oblivious Transfer," in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, (Chicago, Illinois), pp. 20–31, 2–4 May 1988.

[42] 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.

[43] E. Mann, "Private access to distributed information, Technion Master's Thesis, israel," 2004.

[44] C. Cachin, S. Micali, and M. Stadler, "Computationally Private Information Retrieval with Polylogarithmic Communication," in *18th Annual Eurocrypt Conference (EUROCRYPT'99), Prague, Czech Republic*, vol. 1592 of *Lecture Notes in Computer Science*, pp. 402–414, Springer, 1999.

[45] Y.-C. Chang, "Single Database Private Information Retrieval with Logarithmic Communication," in *Information Security and Privacy: 9th Australasian Conference (ACISP'04), Sydney, Australia*, vol. 3108 of *Lecture Notes in Computer Science*, pp. 50–61, Springer, 2004.

[46] C. Gentry and Z. Ramzan, "Single-Database Private Information Retrieval with Constant Communication Rate," in *32nd Annual International Colloquium on Automata, Languages and Programming (ICALP'05), Lisbon, Portugal*, vol. 3580 of *Lecture Notes in Computer Science*, pp. 803–815, Springer, 2005.

[47] R. Ostrovsky and W. E. Skeith III, "A Survey of Single Database PIR: Techniques and Applications." Cryptology ePrint Archive, Report 2007/059, 2007. http://eprint.iacr.org/.

[48] M. Rabin, "How to exchange secrets by oblivious transfer," Tech. Rep. TR-81, Harvard Aiken Computation Laboratory, 1981.

[49] Y. Ishai, E. Kushilevitz, and R. Ostrovsky, "Sufficient Conditions for Collision-Resistant Hashing," in *Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Pro-*

ceedings, vol. 3378 of *Lecture Notes in Computer Science*, pp. 445–456, Springer, 2005.
[50] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), pp. 299–310, ACM, 2013.
[51] S. W. Smith and D. Safford, "Practical server privacy with secure coprocessors," *IBM Systems Journal*, vol. 40, no. 3, pp. 683–695, 2001.
[52] Z. Brakerski, C. Gentry, and S. Halevi, "Fully homomorphic encryption without bootstrapping," in *ITCS 2012 (to appear)*, Available at http://eprint.iacr.org/2011/277, 2012.
[53] D. Pointcheval, "Le chiffrement asymétrique et la sécurité prouvée," *Habilitation à diriger des recherches, Université Paris VII*, 2002.
[54] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of Computer and System Sciences*, vol. 28, no. 2, pp. 270–299, 1984.
[55] 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.

Appendix A
Our Ring-LWE Encryption Scheme

This is basically the symmetric homomorphic encryption scheme of [20] with a composite modulus. Security reduction is preserved using the same arguments as in [52]. 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. [53]). Based on the analysis of [20], our scheme ensures indistinguishability if the standard lattice problem Ring-LWE is hard. This property offers strong guaranties on ciphertext secrecy as proved by Goldwasser and Micali [54]. When used for a PIR 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.

Notations: $\mathbb{Z}_q$ denotes the set of relative integers modulo q (and not q-adic integers). 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 such that after each operation they are reduced by division modulo $X^n + 1$. 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.

ParamGen$(1^k, h_a)$**:**
Input: A security parameter k; A maximum number of additions h_a
Output: A modulus q; A degree n for a quotient polynomial; A distribution χ

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 : $s \leftarrow R_q$

Encrypt(s, m)**:**
Input: A secret key s in the polynomial ring R_q; A message m in the polynomial ring R_q with coefficients in $[0..t[$
Output: A ciphertext $(a, b) \in R_q^2$

1) $a \leftarrow R_q$
2) $e \leftarrow \chi$
3) $e' = e \otimes t_v + m$ where $t_v \in R_q$ has all its coefficients set to t
4) $b = (a * s) + e'$
5) Output: (a, b)

Decrypt$(s, (a, b))$**:**
Input: A secret key $s \in R_q$; A ciphertext $(a, b) \in R_q^2$
Output: A plaintext $m \in \mathbb{Z}_t^n$

1) $e = b - (a * s)$
2) Output: $m = e \mod t$

Add$((a_1, b_1), (a_2, b_2))$**:**
Input: Two ciphertexts, encryptions of m_1 and m_2
Output: A ciphertext that decrypts to $m_1 + m_2 \bmod t$

1) Output: $(a_1 + a_2, b_1 + b_2)$

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

1) Output: $(p * a, p * b)$

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 each prime is congruent to 1 modulo $2n$ (in order to be able to use the NTT). This function generates parameters following the same approach as in [55], with some more conservative choices. The noise distribution is a discrete gaussian of parameter $s = k$, the security parameter. For any value of k greater or equal to 80, we therefore use a noise level well above the bound needed by the security reductions $s > 2\sqrt{n}$ for all values of n. As usual we truncate this distribution taking into account the security to be below a statistical distance of 2^{-k}.

In order to obtain k bits of security we consider an attacker that wants to obtain an advantage $\varepsilon = 2^{-k/2}$ with $\tau = 2^{-k/2}$ computing cycles. As noted in [55] this is the best strategy for an attacker in our setting. As a query can be composed of many ciphertexts, we suppose that the attacker is able to build an attack with an optimal amount of ciphertexts. For a couple of parameters (n, q) we start by setting k to a small value, iteratively increase it, and test if it fulfills the conditions of [55]. More precisely, for each test we follow the approach of [55] to fix the size of the aimed vector size for LLL using the advantage $\beta = (q/s)\sqrt{(}\ln(1/\varepsilon)/\pi)$. We then define the best root Hermite factor δ the attacker is able to use in LLL using $\tau = 2^{-k/2}$ cycles which is given by $\log(\tau) = 1.8/\log(\delta) - 80$ (note that we consider cycles and not seconds, which changes the constant in this formula with respect to [55]). We then compute the shortest vector we can get with LLL given δ which is $2^{2\sqrt{n \log q \log \delta}}$ and see if it is below the aimed vector size. If so, the test is failed. If not, we increase k and do another test.

Appendix B
Using the library

This section is a reshaped version of the file README.md at the root of the XPIRe project: https://github.com/XPIRe-team/XPIRe. It is included in the paper:

- to help the reader that wants to experiment with the library,
- to give some ideas of how installation/usage works for the reader that does not want or cannot experiment with it.

A. Installation

Requirements: g++≥4.8, gcc≥4.8, 64-bit Linux OS (Mac OSX should be supported soon).

Get a copy of the project with:
- git clone git@github.com:XPIRe-team/XPIRe.git
- or by downloading from https://github.com/XPIRe-team/XPIRe/archive/master.zip

Then execute the following commands to compile everything (boost, gmp, mpfr, create essential files, build client and server):

```
$ cd xpire
$ make
```

When this is done you can check that the server/client work correctly with the following commands:

```
$ cd server
$ ./check-correctness
```

The first test should be pretty long (to build initial performance caches) and then a set of tests should display CORRECT or "Skipping test...". If you get INCORRECT tests then something went wrong ...

B. Usage

XPIRe is composed of a server and a client. Both must be started on their respective directories. Thus to start the server execute:

```
$ cd server
$ ./build/PIRServer
```

And to start the client execute (on a different terminal):

```
$ cd client
$ ./build/PIRClient
```

By default the client tries to reach a local server but a given IP address and port can be specified, use –help to get help on the different options for distant connections.

If run without options the PIR server will look for files in a directory db inside the server directory and consider each file is a database element. The client will present a catalog of the files and ask the user to choose a file. When this is done the client will run an optimizer to decide which are the best cryptographic and PIR parameters to retrieve the file. Then he will send an encrypted PIR Query (i.e. a query that the server will mix with the database without understanding which element it allows to retrieve) to the server. The server then computes an encrypted PIR reply and sends it to the client. Finally, the client will decrypt this reply and store the resulting file in the reception directory inside the client directory.

C. Available options for the server (PIRServer command)

`-h,--help`
Print a help message with the different options.

`-z, --driven arg`
Server-driven mode. This mode is to be used when multiple clients will connect to the server with the same cryptographic and PIR parameters. This allows the server to import the database into RAM and to perform precomputations over the database for the first client which *significantly increases the performance for the following clients if LWE-based cryptography is used*. The first client will ask for a given configuration (depending on its optimizer and on the command-line constraints given to the client). After this configuration client, the server will tell the following clients that he is in server-driven mode and that the configuration is imposed. The configuration given by the first client is stored in file arg or in exp/PIRParams.cfg if arg is not specified for further usage (see -L option).

`-L, --load_file arg`
Load cryptographic and PIR parameters from arg file. Currently unavailable (see issues).

`-s, --split_file arg (=1)`
Only use first file in db directory and split it in arg database elements. This allows to have a large database with many fixed size elements (e.g. bits, bytes, 24-bit depth points) into a single file which is much more efficient from a file-system point of view than having many small files. Building databases from a single file with more complex approaches (e.g. csv, or sqlite files) would be a great feature to add to XPIRe.

`-p, --port arg (=1234)`
Port used by the server to listen to incoming connections, by default 1234.

`--db-generator`
Generate a fake database with random elements instead of reading it from a directory. This is useful for performance tests. It allows to deal with arbitrary databases without having to build them on the file-system and to evaluate performance costs without considering disk access limitations.

`-n, --db-generator-files arg (=10)`
Number of files for the virtual database provided by the DB generator.

`-l [ --db-generator-filesize ] arg (=12800000)`
Filesize in bytes for the files in the virtual database provided by the DB generator.

`--no-pipeline` No pipeline mode. In this mode the server executes each task separately (getting the PIR Query, computing the reply, sending it). Only useful to measure the performance of each step separately.

D. Available options for the client (PIRClient command)

`-h, --help`
Display a help message.

`-i, --serverip arg (=127.0.0.1)`
Define the IP address at which the client will try to contact the PIRServer.

`-p [ --port ] arg (=1234)`
Define the port at which the client will try to contact the PIRServer.

`-c, --autochoice`
Don't display the catalog of database elements and automatically choose the first element without waiting for user input.

`--dry-run`
Enable dry-run mode. In this mode the client does not send a PIR Query. It runs the optimizer taking into account the command-line options and outputs the best parameters for

each cryptosystem (currently NoCryptography, Paillier and LWE) with details on the costs evaluated for each phase (query generation, query sending, reply generation, reply sending, reply decryption). If a server is available it interacts with it to set the parameters: client-server throughput and server-client throughput. It also requests from the server the performance cache to evaluate how fast the server can process the database for each possible set of cryptographic parameters. If no server is available it uses default performance measures. The other parameters are set for the default example: a thousand mp3 files over ADSL, aggregation disabled and security k=80. Each of these parameters can be overridden on the command line.

`--verbose-optim`
Ask the optimizer to be more verbose on the intermediate choices and evaluations (as much output as in the dry-run mode).

`--dont-write`
Don't write the result to a file. For testing purposes, it still will process the reply (decryption of the whole answer).

`-f, --file arg`
Use a config file to test different optimizations in dry-run mode (see exp/sample.conf). Must be used with the –dry-run option or it is ignored.

E. Available options for the optimizer (through PIRClient command)

`-n, --file-nbr arg`
Used in dry-run mode only: Override the default number of database elements.

`-l, --file-size ] arg`
Used in dry-run mode only: Override the default database element size (in bits).

`-u, --upload arg`
Force client upload speed in bits/s (bandwidth test will be skipped). This is valid in dry-run or normal mode (e.g. if a user does not want to use more than a given amount of his bandwidth).

`-d, --download arg`
Force client download speed in bits/s (bandwidth test will be skipped). This is valid in dry-run or normal mode.

`-r, --crypto-params arg`
Limit with a regular expression arg to a subset of the possible cryptographic parameters. Parameters are dependent on each cryptographic system:
** NoCryptography if a trivial full database download is to be done after which PIRClient stores only the element the user is interested in.
** Paillier:A:B:C if Paillier's cryptosystem is to be used with A security bits, a plaintext modulus of B bits and a ciphertext modulus of C bits.
** LWE:A:B:C if LWE is to be used with A security bits, polynomials of degree B and polynomial coefficients of C bits.
For example it is possible to force just the cryptosystem with NoCryptography.* or LWE.*, or ask for a specific parameter set like Paillier:80:1024:2048. Specifying the security with this option is tricky as it must match exactly so better use -k for this purpose.

`-k, --security arg (=80)`
Minimum security bits required for a set of cryptographic parameters to be considered by the optimizer.

`--dmin arg (=1)`
Min dimension value considered by the optimizer. Dimension is also called recursion in the literature. It is done trivially (see the scientific paper) and thus for dimension d query size is proportional to $d \times n^{1/d}$ and reply size is exponential in d. For databases with many small elements a d¿1 can give the best results, but only in exceptional situations having $d > 4$ is interesting.

`--dmax arg (=4)`
Max dimension value considered by the optimizer.

`-a, --alphaMax arg (=0)`
Max aggregation value to test (1 = no aggregation, 0 = no limit). It is sometimes interesting to aggregate a database with many small elements into a database with fewer but larger aggregated elements (e.g. if database elements are one bit long). This value forces the optimizer to respect a maximum value for aggregation, 1 meaning that elements cannot be aggregated.

`-x, --fitness arg (=1)`
Set fitness method to:
0=SUM Sum of the times on each task
1=MAX Max of server times + Max of client times
2=CLOUD Dollars in a cloud model (see source code)
This sets the target function of the optimizer. When studying the different parameters the optimizer will choose the one that minimizes this function. 0 corresponds to minimizing the resources spent, 1 to minimizing the round-trip time (given that server operations have are pipelined and client operations are also, independently, pipelined), 2 corresponds to minimizing the cost by associating CPU cycles and bits transmitted to money using a cloud computing model.