Efficient Private Information Retrieval

Konstantinos Nikolopoulos

The Graduate Center, City University of New York

How does access to this work benefit you? Let us know!

Follow this and additional works at: https://academicworks.cuny.edu/gc_etds

Part of the Digital Communications and Networking Commons

Recommended Citation

https://academicworks.cuny.edu/gc_etds/3158

This Dissertation is brought to you by CUNY Academic Works. It has been accepted for inclusion in All Dissertations, Theses, and Capstone Projects by an authorized administrator of CUNY Academic Works. For more information, please contact deposit@gc.cuny.edu.
Efficient Private Information Retrieval

by

Konstantinos F. Nikolopoulos

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2019
This manuscript has been read and accepted by the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Professors:

**Professor Spiridon Bakiras**

Date: 
Chair of Examining Committee

**Professor Robert Haralick**

Date: 
Executive Officer

Supervisory Committee:

**Spiridon Bakiras**
**Abdullah Uz Tansel**
**Sven Dietrich**
**Efstathios Zachos**

Supervisory Committee

**THE CITY UNIVERSITY OF NEW YORK**
Abstract

Efficient Private Information Retrieval

by

Konstantinos F. Nikolopoulos

Adviser: Professor Spiridon Bakiras

A vast amount of today’s Internet users’ on line activities consists of queries to various types of databases. From traditional search engines to modern cloud based services, a person’s everyday queries over a period of time on various data sources, will leave a trail visible to the query processor, which can reveal significant and possibly sensitive information about her. Private Information Retrieval (PIR) algorithms can be leveraged for providing perfect privacy to users’ queries, though at a restrictive computational cost. In this work, we consider today’s highly distributed computing environments, as well as certain secure-hardware devices, for optimizing existing PIR solutions. In particular, we initially employ available secure-hardware in a novel approach with the goal of providing faster and constant private query responses, by sacrificing some degree of privacy. Further on, we utilize the widely used Message Passing Interface (MPI) protocol for designing a library which can be used in third party software for performing private queries.
Acknowledgments

I would like to thank my advisor, Spiridon Bakiras, for his excellent guidance and support in this work.

My thesis committee members, for their valuable input and ideas.

Especially my family, for all the love, encouragement and support through the years.
## CONTENTS

2.1.4 Database Updates ........................................... 16

2.2 Private Page Retrieval Algorithm ................................ 18
   2.2.1 Algorithm .................................................. 18

2.3 Secure Hardware Deployment ...................................... 21

2.4 Conclusions ....................................................... 27

3  A library for efficient computational PIR .......................... 28
   3.1 Preliminaries .................................................... 30
      3.1.1 Message Passing Interface ................................ 30
   3.2 Design .......................................................... 32
      3.2.1 Database Partitioning ..................................... 33
      3.2.2 Query Response ........................................... 35
   3.3 Implementation .................................................. 35
      3.3.1 General Layout ............................................ 35
      3.3.2 Public Parameters ......................................... 37
      3.3.3 Datatypes .................................................. 38
      3.3.4 Database Initialization ................................... 40
      3.3.5 Data Storage .............................................. 42
      3.3.6 Query Computation ........................................ 42
      3.3.7 Error handling ............................................. 43

4  A private key-value store ........................................... 44
   4.1 Design .......................................................... 45
      4.1.1 Page size .................................................. 46
      4.1.2 Hashtable buckets ......................................... 47
      4.1.3 Queries .................................................... 48
List of Tables

2.1 Summary of symbols .................................................. 13
2.2 System specifications .................................................. 21
3.1 Public Parameters for the Gentry-Ramzan scheme ............... 38
List of Figures

1.1 4x4 transformed database ........................................ 9

2.1 System architecture .................................................. 12
2.2 Data structures at the secure hardware ............................ 18
2.3 The page retrieval algorithm .......................................... 20
2.4 Page retrieval costs for 1KB pages (c = 2) ....................... 22
2.5 Page retrieval costs for 10KB pages (c = 2) .................... 24
2.6 Response time as a function of $c = 1 + \varepsilon \ (B = 1KB)$ ........ 25
2.7 Page retrieval costs for 1TB database (c = 2) .................. 26

3.1 PPIR overview .......................................................... 33
3.2 Database split for parallelized Gentry-Ramzan scheme (page size $k = 4$) ... 34

4.1 Bucket Layout in the hash table .................................... 48
4.2 A private key-value store ............................................. 49
Chapter 1

Introduction

Internet users are becoming increasingly wary of the potential privacy risks associated with their everyday online activities. Web search engines, for example, maintain detailed logs of every query that they receive. With sophisticated data mining, these pools of query logs can reveal sensitive information about a user’s lifestyle, habits, health, etc. [17, 34]. Similarly, the emergence of location based services (LBS) allows mobile users to browse points of interest (e.g. restaurants) in their surroundings. Since these queries are also logged at the LBS provider, a user’s location over a period of time can be tracked with very high accuracy [45].

Clearly, ordinary database queries involve an inherent privacy risk for users and, as a result, privacy preserving query processing is an emerging research field in the database community. A popular approach that enhances the level of privacy in certain applications, is anonymity. The central principle of anonymity is to inject sufficient noise into a query, so that the user has *plausible deniability* over the exact content of the query. A straightforward approach is for a client to combine the real query with several dummy ones (that are typically unrelated), and submit this aggregate query to the server. Algorithms based on anonymity have been proposed for both text search engines (e.g. [43, 44]) and location based services.
(e.g. [22, 42]). However, since the database server has access to the plaintext queries, it may be able to determine the real content of a query using background knowledge.

Encryption is another technique that can be leveraged to hide the content of a query. In this scenario, the server interacts with an encrypted version of the original database. Queries are also encrypted in a similar fashion and, thus, the server can not deduce any information about the query content. Research work in this area has focused on developing efficient encryption algorithms that facilitate exact query processing at the server side [14, 15]. The limitation of encryption schemes, however, is that two identical queries always produce the same encrypted result. Consequently, if the server has knowledge of the access patterns of the database records (i.e., their relative popularity), it can extract some information about a query through the records included in the result set.

Private information retrieval (PIR) is currently the only available solution that can be leveraged to build algorithms that provide perfect privacy. In particular, PIR protocols [20] allow a client to retrieve any record from a database, while making it impossible for a computationally bounded server to determine which record was retrieved. Note that, when PIR is employed, the server cannot perform the actual query processing. Instead, the client accesses (privately) the disk-resident index structure at the database server and resolves the query locally through a series of PIR retrievals [45]. However, PIR retrievals are computationally expensive and, for large databases, they may lead to query processing times of several tens of seconds.

In this dissertation, we will investigate two approaches for achieving efficient performance in PIR. In the first part, we will use secure hardware between the server and the client that will assist in achieving a trade-off between the provided privacy and the computational cost. We will give a formal definition of our proposed algorithm and illustrate its practical application. In the second part, we will design a software library that will provide its clients with a ”black-box” implementation of one of the most representative PIR algorithms to date.
Our library will operate on clusters of nodes of various sizes and distribute the computational cost of the algorithm across the cluster. We will also design a demonstration application for using our library in a client-application that will allow private access to a key-value dataset. Such datasets are common in various disciplines and private access is often required if they contain sensitive information.

1.1 Preliminaries

1.1.1 Anonymity

The goal of anonymity is to provide plausible deniability. This is achieved mainly by hiding the actual query into a pool of decoy queries that are generated by the client. In the context of location based services, Ref. [21, 23, 37] follow this approach, by having the mobile client generate random locations on the data space. The LBS server processes all queries and returns the aggregate result set to the client. Finally, the client computes the actual result by discarding the redundant records. In the spatial $k$-anonymity framework [26, 29, 35, 42], a trusted third-party (called the anonymizer) acts as a proxy between the mobile clients and the LBS server. First, the client sends the query with its actual location to the anonymizer. The anonymizer replaces the query location with an anonymizing region (typically a rectangular area) that includes at least $k - 1$ other users. Then, the anonymizer forwards the query to the LBS server, which evaluates it on the whole anonymizing region, instead of the individual location. After the anonymizer receives the set of candidate records, it filters out the false hits and returns the result set to the client.

Recently, anonymity has also been considered in the field of text search engines. Murugesan and Clifton [43] construct groups of canonical queries, such that every group of $k$ canonical queries covers $k$ diverse topics. When a client issues a keyword search query,
it is replaced by the closest canonical query and the corresponding group is submitted to the server. Consequently, the client has $k$-plausible deniability, since it could have initiated any of the $k$ queries in the group. Pang et al. [44] do not create query groups, but rather embellish the client’s query with decoy terms. They ensure that groups of decoy terms are semantically related, so that the server cannot identify the actual query.

In general, anonymity based approaches are computationally efficient, because the server evaluates the queries on the plaintext database. However, they do not provide a strong notion of privacy, since (i) the level of uncertainty is low and (ii) the server has access to the plaintext queries and may be able to link a query to a specific user, using background knowledge.

1.1.2 Data Encryption

Encryption based schemes emerged from the database outsourcing model. In this setting, the data owner delegates its database functionality to a service provider, who is then responsible for processing client queries on behalf of the owner. Since the database may contain confidential information that needs to be kept secret, the owner usually encrypts the database before sending it to the service provider. Research work in this area has focused primarily on developing searchable encryption schemes that allow the server to process queries on the encrypted data. Note that, queries need also be transformed to reflect the encrypted data and, thus, query privacy is protected.

The order preserving encryption scheme (OPES) [15] preserves the order of numeric values and facilitates the evaluation of simple relational operators on the encrypted data. Agrawal et al. [14] introduce a solution based on Shamir’s secret sharing scheme [46]. Specifically, data is divided into $n$ shares, and each share is stored at a different server. Shares are calculated as follows. First, the data owner chooses a random polynomial of degree $k - 1$, where the constant term is the database value. It also chooses a set of $n$ (secret) random points on the
polynomial, each corresponding to one of the servers. The share at each server is calculated based on its secret point. Queries are encoded accordingly and, during query processing, any set of $k$ servers must reply in order for the client to reconstruct the result. Wong et al. [52] introduce an asymmetric scalar-product-preserving encryption (ASPE) scheme that targets LBS applications. The proposed encryption preserves the relative distances of all the database points to any given (encrypted) query point. This property enables the processing of nearest neighbor queries, by comparing every database point with the query.

Encryption based approaches provide better privacy than anonymity based ones, as both the data and the queries are unreadable by the server. Nevertheless, they are vulnerable to access pattern attacks, because two identical queries always return the same ciphertexts. Therefore, if a server has knowledge of the relative access frequencies of the database records, it can extract some information about a query through the ciphertexts included in the result set.

### 1.1.3 Private Information Retrieval

PIR was first introduced by Chor et al. [20], and is formally defined as follows. The server holds a database, which is assumed to be a binary string $X$ of length $n$. The client wants to retrieve the $i$-th bit ($x_i$) of the database, without the server knowing the value of index $i$. In general, PIR protocols can be classified into three main categories: information theoretic, computational, and hardware based.

First, *information theoretic* PIR [18, 20, 30, 53] ensures that the query discloses no information about the retrieved bit, even if the server has unbounded computational power. However, these protocols are not practical, as they require that the database be replicated into $k$ non-colluding servers. On the other hand, *computational* PIR protocols [19, 27, 38, 39] work with a single server, and employ well known cryptographic primitives that guarantee query privacy for a computationally bounded server. Nevertheless, these protocols are ex-
tremely expensive for large databases, as they require at least one modular multiplication for every bit of the database.

Finally, secure hardware PIR \([33, 49, 50, 51]\) relies on a tamper resistant CPU (located at the server side), which acts as a proxy between the clients and the server. These protocols are significantly faster than computational PIR, because they do not need to scan the whole database for every query. Wang et al. \([49]\) utilize the internal storage of the secure hardware that can hold \(k\) out of \(n\) database pages. Every request inserts a new page into the secure storage and, when the storage capacity is reached, the database is re-shuffled. Therefore, the amortized computational cost of this approach is \(O(n/k)\). Ref. \([33, 50, 51]\) leverage the Oblivious RAM model \([31]\), which arranges the database pages into a pyramid-like structure. To achieve access pattern privacy, (i) every level of the structure is accessed during a page retrieval and (ii) the pyramid levels are periodically re-shuffled by the secure hardware. Iliev and Smith \([33]\) propose a method with \(O(\sqrt{n} \log n)\) amortized computational cost, while Williams and Sion \([50]\) improve this amortized cost to \(O(\log^2 n)\). Currently, the state-of-the-art approach is due to Williams et al. \([51]\), and provides an amortized logarithmic computational cost of \(O(\log n \log \log n)\). However, due to the periodic re-shuffling of the pyramid levels, the response time of a single PIR retrieval may vary from milliseconds to hundreds of seconds.

PIR based solutions have been explored previously in the context of spatial nearest neighbor queries. In particular, Khoshgozaran et al. \([36]\) and Papadopoulos et al. \([45]\) utilize secure hardware protocols, while Ghinita et al. \([28]\) employ an expensive computational PIR algorithm \([38]\). Ref. \([45]\) is a more general and comprehensive study on the applicability of PIR protocols on multi-level index structures. The authors introduce a solution that provides perfect privacy, and also present a detailed experimental evaluation based on secure hardware \([50]\) simulations. Their results show that query processing may require tens of seconds, even for moderate databases, due to the large number of PIR retrievals on the underlying disk-
resident index structures. Motivated by this fact, we propose an alternative approach that sacrifices some degree of privacy in order to reduce significantly the query processing cost.

1.2 Information Theoretic PIR

In Information Theoretic PIR the user privacy is guaranteed. Chor et al. [20] proved that the lower bound for any information theoretic PIR protocol is $O(n)$. This corresponds to the trivial and prohibitively non-efficient solution of the client simply retrieving the entire $n$-bit database in a single query. In order to reach a non-trivial solution the problem has to be relaxed. In particular, it is usually assumed that there exist several identical replicated clones of the database servers, each of which is independent. Chor et al. [20] presented a scheme with two replicated database servers with $O(n^{1/3})$ communication complexity. Ambainis[16] provided a k-database scheme with communication complexity of $O(n^{1/(2k-1)})$. Beimel et al.[18] provided further improved this limit to $O\left(\frac{\log \log k}{k \log k}\right)$.

1.3 Computational PIR

1.3.1 The Gentry-Ramzan scheme

Gentry and Ramzan[27] presented a single-database server PIR scheme with $O(w + B)$ communication complexity, based on the $\phi$-hiding assumption. Here, $w \geq \log n$ is a security parameter based on the size of the database $n$ and $B$ is the length in bits of the database block which is requested.

The database is split into $t$ $l$-bit blocks $C_t$ and the Chinese Remainder Theorem is utilized to compute the smallest integer $e \equiv C_t(\mod{\pi_t})\forall t$, where $\pi_t$ are prime powers associated with each block $C_t$. These prime powers are calculated as $\pi_i = p_i^{c_i}$, where $p_i$ are small and distinct primes and $c_i = \lceil l/\log_2 p_i \rceil$. This step can be pre-computed and stored in advance and the
same $e$ can be used in the following queries. It will have to be re-computed again though when an update of the database occurs (i.e. insertion, deletion or edit of a record). The user, in order to generate a query for the $i$-th block, chooses two large primes $p$ and $q$, such that $p = 2\pi r + 1$ and $q = 2st + 1$, where $r, s, t$ are large random integers. After setting $m = pq$, a random element $g \in \mathbb{Z}_m^*$ with order $\pi q$ is selected, where $gcd(\pi, q) = 1$. Finally, $(g, m)$ is sent to the server but $p, q$ are kept private. Upon receipt, the server has to compute $g_e = g^e$ and send it back to the user. Ultimately, extraction of the block is possible at the user side by computing $h_e = g_q^e$ and afterwards the discrete logarithm $\log_h h_e$ within $H \subset G$ of order $\pi_i$ with Pohlig-Hellman. At this point, if $p_i$ primes were small (which are chosen to be), the discrete logarithm calculation will be efficient.

1.3.2 The Kushilevitz-Ostrovsky scheme

Kushilevitz and Ostrovsky\cite{38} introduced the first PIR scheme that eliminated the need for replicating the database. It relies on the Quadratic Residuosity Assumption, according to which if $p_1$ and $p_2$ are large primes, then determining if a number is a Quadratic Residue (QR) $(mod\ p_1p_2)$ or a Quadratic Non Residue (QNR), is computationally hard. This approach represents the database as a matrix, which makes it suitable for a parallel scenario.

The server rearranges the database of size $n$ into a $\sqrt{n} \cdot \sqrt{n}$ matrix $M$. The client, who is interested in element $M_{r,b}$ of the matrix, picks two (secret) large primes $p_1$ and $p_2$ and prepares a row of $\sqrt{n}$ elements $y_1, ..., y_{\sqrt{n}} \in \mathbb{Z}_{n+1}$; all of them being quadratic residues in $mod\ p_1p_2$, with the exception of the one $y_b$ located in the position of the cross-column $b$ which contains the desired element. At this position a non quadratic residue is chosen. This row is transmitted to the server.

The server on its turn operates on every row $i$ of the matrix, producing for each a product $z_i = \prod_{j=1}^{\sqrt{n}} y_j y_j^{1-M_{i,j}}$, for $1 \leq i \leq \sqrt{n}$. In this set of numbers, $z_r$ will be a QR if $M_{r,b} = 0$ or a QNR if $M_{r,b} = 1$. 

In the example of Figure 1.1 we can observe that there are three occurrences of the same pair of bits (0,1) in the first, second and fourth rows. These pairs, along with the particular \( y_0, y_1, y_2, y_3 \) vector, will obviously contribute the same values to the computation of \( z_0, z_1 \) and \( z_3 \), which implies that if we save this value the first time we encounter them (i.e. during the \( z_0 \) computation), we can avoid their needed operations in the next two occurrences. We can expand this idea to help us significantly increase performance by avoiding a number of operations. Consider parsing the database in bytes; a byte has 256 possible bit combinations. As we keep "scanning" the database matrix for computing the \( z_i \) values, we can gradually store the derived value from each of these 256 cases we encounter and before starting computation on the next byte, we can check if it already exists in our cache. Considering even a 12.5 Megabyte (100 million bits) database, which will have 10,000 rows, very soon the 256 possible byte values will start reappearing.
Chapter 2

Hardware based PIR

In this chapter, we address this drawback and introduce a novel approach that sacrifices some degree of privacy in order to provide fast and constant query response times. Our method leverages the internal cache of the secure hardware to constantly reshuffle the database pages in order to create sufficient uncertainty regarding the exact location of an arbitrary page.

We give a formal definition of the privacy level of our algorithm and illustrate how to enforce it in practice. Based on the performance characteristics of the current state-of-the-art secure hardware platforms, we show that our method can provide low page access times, even for very large databases.

We introduce a novel approach that sacrifices some degree of privacy in order to provide fast and constant query response times. The goal is to design a system that balances the trade-off between computational cost and privacy guarantees. In other words, we aim to provide a much stronger notion of privacy compared to anonymity or encryption based schemes, but with a computational cost that is considerably lower compared to existing PIR techniques. Such a system would benefit applications that do not require perfect privacy, but are instead satisfied with a sufficient level of uncertainty.

Our algorithm initially encrypts and obliviously permutes the database pages. Each
page is then retrieved efficiently by accessing (through the secure hardware) its encrypted version from the server’s disk. To further enhance the privacy of our approach, we introduce a randomized algorithm that constantly reshuffles the underlying pages in order to create sufficient uncertainty regarding the exact location of an arbitrary page. The algorithm works by randomly moving every requested page to a new location on the disk. In particular, it leverages a built-in cache at the secure hardware that stores a fixed number of previously retrieved pages. Reshuffling occurs during each page request, with a random page from the cache being written to a new location on the disk. We give a formal definition of the privacy level of this approach and illustrate how to enforce it in practice. Based on the performance characteristics of the current state-of-the-art secure hardware platforms, we show that our method can provide low page access times, even for very large databases. In summary, the contributions of our work are the following.

- We propose a novel architecture, based on state-of-the-art secure hardware, that reduces significantly the cost of private page retrievals compared to existing PIR based techniques.

- We formally define the privacy level of our approach and use analytical models to derive the corresponding security parameter.

- We evaluate the performance of our method, using (i) analytical results from a secure hardware deployment and (ii) measurements from a software implementation. We show that, given sufficient secure storage capacity, our system can achieve sub-second query response times, even for TB-sized databases.

The remainder of this chapter is organized as follows. Section 2.1 describes the architecture of our approach and outlines the underlying adversarial model. Section 2.2 introduces our private page retrieval algorithm and Section 2.3 presents the analytical results from a secure hardware implementation. Finally, Section 2.4 concludes the paper.
2.1 Preliminaries

Section 2.1.1 describes the basic architecture of our approach and Section 2.1.2 outlines the underlying threat model.

2.1.1 System Architecture

Figure 2.1 illustrates the proposed system architecture. The secure hardware is a *tamper resistant* CPU, such as the IBM 4764 PCI-X secure coprocessor\(^1\). It is attached at the server machine, but it can be trusted to operate without any interference from the server. Specifically, it includes tamper detecting and responding circuitry that, in the event of an attack, destroys all the critical keys and certificates. The secure hardware incorporates an internal cache (up to 64MB for the IBM 4764 secure coprocessor) that is inaccessible by the server, and also has direct access to the server’s disk.

\[\text{Figure 2.1: System architecture}\]

Note that the secure hardware is only necessary in the *three-party* querying model, i.e., when any client (including the adversary) is allowed to query the database server. Nevertheless, our methods are also applicable in the *two-party* querying model. The two-party

\(^1\)[http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml]
model applies to the *database outsourcing* paradigm, where the data owner is the only client that accesses the database. In this setting, the owner outsources its data to a third-party service provider and wishes to access these data in a private manner. Since the data owner is the sole client in this architecture, there is no need for a secure hardware platform at the service provider. Instead, the functionality of the secure hardware can be implemented entirely at the owner’s side (physically isolated from the adversary), using any standard server configuration. We explore the feasibility of this approach in Section 2.3.

In our problem formulation, we consider a database consisting of $n$ pages (Table 2.1 summarizes the symbols used throughout the paper). Each page is a tuple $\langle \text{id}, \text{data} \rangle$, where the $\text{id}$ attribute uniquely identifies the page. Prior to query processing, the secure hardware encrypts and obliviously permutes the database pages. It utilizes a *symmetric-key* encryption algorithm, such as AES [25], and the encryption key is secret from both the database server and the clients. Clients communicate with the secure hardware via secure SSL connections. A client query $Q(i)$ is simply a request to retrieve the page with $\text{id} = i$ from the database (we assume pages are assigned $\text{id}$ values ranging from 0 to $n - 1$). To facilitate query processing, the secure hardware stores in its cache a look-up table that maps each page $\text{id}$ to its actual position on the disk. After identifying the corresponding position, the secure hardware retrieves the page from the disk, decrypts it, and finally transmits it to the client via the secure connection.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>Database size (number of pages)</td>
</tr>
<tr>
<td>$k$</td>
<td>Block size (number of pages)</td>
</tr>
<tr>
<td>$T$</td>
<td>Number of blocks in database ($= n/k$)</td>
</tr>
<tr>
<td>$m$</td>
<td>Cache capacity (number of pages)</td>
</tr>
<tr>
<td>$B$</td>
<td>Page size (bytes)</td>
</tr>
</tbody>
</table>

To provide perfect query privacy, previous approaches apply periodically an oblivious
permutation algorithm to reshuffle the database pages. Note that, after the reshuffling operation, every database page has an equal probability ($= 1/n$) of landing in any of the $n$ available locations. Consequently, any query that accesses a new page from the disk becomes indistinguishable from any other query. In this work, we aim to relax this stringent constraint and allow pages to land in different disk locations according to a non-uniform distribution.

Unlike prior methods, we do not reshuffle the entire database at once; instead, during each request instant, one previously retrieved page (that resides temporarily inside the cache) is relocated to a new position on the disk. In particular, for any value $c \geq 1$, we introduce the notion of $c$-approximate PIR as follows.

**Definition 1** A scheme provides $c$-approximate PIR if, after moving a single page $p$ to a new location on the disk and for any pair of disk locations $l_i, l_j$, the probability of $p$ landing in location $l_i$ is at most $c$ times larger than the probability of landing in location $l_j$.

The value $c$ is the privacy parameter of our approach, as it determines the variability of the distribution that models the individual page relocation process. Smaller values of $c$ result in better privacy, while the case $c = 1$ offers perfect privacy (i.e., equivalent to PIR).

### 2.1.2 Adversarial Model

We assume that the adversary is the server itself, and its goal is to derive any non-trivial information regarding the id of a requested page. Because of the underlying secure SSL connections, both the client queries and the generated replies are unreadable by the server. Nevertheless, the server can see the accessed locations on the disk and has knowledge of all the algorithms that are implemented inside the secure hardware. We also assume that the server can only perform polynomial time computations and is “curious but not malicious” (i.e., it will not tamper with the actual data).
2.1.3 Security Analysis

The page retrieval algorithm works by spreading the accesses for a single page over multiple disk locations. Once a page is requested and moves into the cache, it will be relocated to a new position during a subsequent request. Consequently, an adversary can only track probabilistically the location of an arbitrary page within the server’s disk. Our goal is to properly adjust the block size \( k \), in order to meet the privacy requirements of the \( c \)-approximate PIR definition (Section 2.1.1).

Consider a sequence of client requests at instants \( t = 0, 1, 2, \ldots \). Assume that page \( p \) is copied into the cache during a client request at \( t = 0 \). Then, the probability that it moves back to the disk at time \( t \geq 1 \) is computed as:

\[
P^t = \left( 1 - \frac{1}{m} \right)^{t-1} \cdot \frac{1}{m}
\]

Therefore, if the secure hardware accesses a set of \( k \) locations (from a single block) \( \mathcal{L}_t = \{l_1, l_2, \ldots, l_k\} \) during the request at time \( t \), the probability that page \( p \) is relocated to position \( l_j \) (\( 1 \leq j \leq k \)) is equal to:

\[
P^t_{p \rightarrow l_j} = \left( 1 - \frac{1}{m} \right)^{t-1} \cdot \frac{1}{m} \cdot \frac{1}{k}
\]

The value \( k \) is the security parameter of our approach, since it controls the time interval \( T = n/k \) (given as number of requests) that is required to scan every location on the disk exactly once through the round-robin schedule. Note that Equation (2.2) is a monotonically decreasing function, so the \( k \) locations that are accessed at \( t = 1 \) have the highest probability of hosting page \( p \). Specifically, for the locations \( l_j \in \mathcal{L}_1 \), the probability that \( p \) is relocated there is:

\[
P^1_{p \rightarrow l_j} = \sum_{i=0}^{\infty} \left( 1 - \frac{1}{m} \right)^{T \cdot i} \cdot \frac{1}{m} \cdot \frac{1}{k}
\]
Similarly, the locations \( l_j \in \mathcal{L}_T \) have the lowest probability of storing page \( p \):

\[
P^T_{p \rightarrow l_j} = \sum_{i=0}^{\infty} \left( 1 - \frac{1}{m} \right)^{(i+1)T-1} \cdot \frac{1}{m} \cdot \frac{1}{k} \tag{2.4}
\]

Consequently, the value of \( k \) can be determined by setting

\[
\frac{P^1_{p \rightarrow l_j}}{P^T_{p \rightarrow l_j}} = \frac{1}{(1 - \frac{1}{m})^{T-1}} = \frac{1}{(1 - \frac{1}{m})^{\frac{k}{k-1}}} = c \tag{2.5}
\]

Solving the above equation, we get:

\[
k = \frac{n}{\log(1/c)} \log(1 - 1/m) + 1 \tag{2.6}
\]

Note that, the value \( c = 1 \) corresponds to the trivial case of PIR, i.e., when the whole database is read for every request \( (k = n) \). On the other hand, a value such as \( c = 2 \) would indicate that any location is at most twice as likely to host a previously cached page as any other location on the disk. For a given database size \( n \) and privacy parameter \( c \), the value of the security parameter \( k \) is determined by the available cache capacity. As evident in Equation (2.5), for a fixed value of \( T \), the privacy parameter \( c \) converges towards 1 as the value of \( m \) increases.

### 2.1.4 Database Updates

A final remark concerns the handling of database updates in our system architecture. Similar to query processing, the database owner interacts only with the secure hardware through a secure SSL connection. Our system can handle trivially any type of updates, including insertions, deletions, and page modifications. In particular, every database update is treated as a regular query, i.e., the secure hardware (i) retrieves \( k + 1 \) pages from the disk, (ii) swaps
one page from the cache with one of the retrieved pages, and (iii) writes the $k+1$ pages back
to the disk after re-encrypting them. Consequently, the type of update operation performed
on the database is kept secret from the server.

Deletions are handled as cache hits, i.e., the $(k+1)$-th page is selected randomly. Additionally, if the deleted page is stored inside the cache, it is always selected to swap positions
with one of the $k$ pages in the block. Finally, the position attribute of the pageMap entry for that page is set to a reserved value (e.g., all 1’s) that signifies the deletion event.
Note that, if there are numerous page deletions on the database, the owner may choose to
reshuffle (offline) the whole database in order to physically remove the deleted pages. Page
modifications are handled as regular queries, i.e., they can either produce a cache hit (if the
page is stored inside the cache) or a cache miss. In any case, the original page is replaced
with the new version.

To handle insertion operations, the secure hardware should reserve in advance sufficient
storage space in its internal data structures. Therefore, during the initial reshuffling stage,
the secure hardware should create numerous dummy pages that may be utilized to store
the newly inserted pages. These pages are marked as deleted, so pages that are explicitly
deleted by the data owner may serve the same purpose as well. When a new page is created
in the database, the secure hardware accesses the next block of $k$ pages as usual. However,
the $(k+1)$-th page is always a deleted page. The newly inserted page is then stored inside
the cache, replacing one of the pages therein. Finally, the deleted page swaps positions with
one of the $k$ locations of the retrieved block, and the evicted page is copied over the deleted
page.
2.2 Private Page Retrieval Algorithm

Section 2.2.1 describes the page retrieval algorithm, while Section 2.1.3 provides an analytical model that quantifies its privacy level. Section 2.1.4 illustrates the database update procedure.

2.2.1 Algorithm

Our approach leverages the built-in cache at the secure hardware to obliviously mix a pool of database pages and copy them into random positions on the disk. We assume that the cache can store a total of $m$ pages and employs a randomized cache replacement policy. Note that the purpose of the cache is not to improve the page retrieval time, but to facilitate this continuous page reshuffling process.

During each page request, the algorithm retrieves a fixed number of $k + 1$ pages, where
$k$ is the security parameter. In particular, the secure hardware initially reads (in a round-robin manner) a block of $k$ contiguous pages. Specifically, on the first request it accesses the database pages at locations 0 through $k - 1$, next the pages at locations $k$ through $2k - 1$, etc. The $(k + 1)$-th page that is read is either the page requested by the client or a random one (the detailed algorithm is explained shortly). The reason for reading multiple pages is to guarantee that any cached page has a non-negligible probability of being written to any location on the disk (discussed in Section 2.1.3). If $n$ is not a multiple of $k$, the secure hardware inserts an appropriate number of dummy pages during the initial reshuffling stage.

Figure 2.2 shows the data structures maintained at the secure hardware. First, the cache is implemented as a vector (pageCache) holding $m$ database pages. pageMap is a vector of size $n$ and corresponds to the look-up table for all the database pages. Each entry in pageMap is a tuple $(inCache, position)$. Attribute inCache uses a single bit that, when set, indicates that the corresponding page is stored inside the cache. Attribute position is an integer value that has a dual interpretation: if inCache = 1, it represents the index in the pageCache vector where the page is stored; otherwise, it identifies the location of that page at the server disk under the current permutation order (see Figure 2.2). Finally, serverBlock is the vector (of size $k + 1$) that temporarily stores the pages that are written to or read from the disk. In the sample configuration of Figure 2.2, $n = 100$, $m = 10$, and $k = 4$.

The page retrieval algorithm is shown in Figure 2.3, and operates as follows. First, the client sends a query to the secure hardware, containing the id of the required page (e.g., page $i$). The secure hardware then reads and stores into serverBlock the next block of $k$ pages, according to the round-robin schedule. Next, it accesses pageMap[$i$] and identifies the current location of that page. If page $i$ is located at the server and is not included in the serverBlock vector, the page is retrieved from the corresponding location on the disk and stored into serverBlock. If, on the other hand, page $i$ is included in the serverBlock vector, the secure hardware selects a random page from the database that is not currently
CHAPTER 2. HARDWARE BASED PIR

Retrieve(i)

// read next block (of size k) in a round-robin fashion
1: serverBlock[0..k − 1] ← read(nextBlock)
2: if (pageMap[i].inCache or i ∈ serverBlock)
   // select a random page that is not cached
   // and is not retrieved in serverBlock
3:    do
4:       p ← random(0, n − 1)
5:      while (pageMap[p].inCache or p ∈ serverBlock)
6:    if (pageMap[i].inCache)
7:       result ← pageCache[pageMap[i].position]
   // else use requested page
8:  else
9:    p ← i
// read page p from the disk
10: serverBlock[k] ← read(pageMap[p].position)
// decrypt all pages in serverBlock
11: decrypt(serverBlock)
12: if (!pageMap[i].inCache)
13:   q ← index of page i in serverBlock
14:   result ← serverBlock[q]
15: else
16:   q ← k
// select a random page from the block
17:   r ← random(0, k − 1)
18: swap(serverBlock[r], serverBlock[q])
// select random page from cache
19:   s ← random(0, m − 1)
20: swap(pageCache[s], serverBlock[r])
// re-encrypt all pages in serverBlock (with a new nonce)
21: encrypt(serverBlock)
22: write updated pages at the disk
23: write(serverBlock)
// update pageMap (3 pages)
24: update(pageMap[pageCache[s]])
25: update(pageMap[serverBlock[r]])
26: update(pageMap[serverBlock[q]])
// send page i to the client over the SSL connection
27: return result

Figure 2.3: The page retrieval algorithm

cached or stored into serverBlock.

Subsequently, the secure hardware decrypts all k + 1 pages in serverBlock and extracts the requested page. It then selects a random page from the cache and replaces it with the newly requested page i. Similarly, the cached page is copied into serverBlock, overwriting page i. However, to ensure that the cached page is moved to any of the k locations in the read block (corresponding to the first k pages in serverBlock) with equal probability, the requested page initially swaps places with a random page in the block (line 18). Next, the pages in serverBlock are re-encrypted with a new random nonce, and are eventually
transferred back to the server’s disk. Finally, the secure hardware modifies the necessary entries (for the swapped pages) at the pageMap vector.

In the case where the requested page produces a cache hit, the secure hardware retrieves a random page \( p \) from the disk and repeats the same steps as above, i.e., as if page \( p \) was requested by the client. To summarize, during every query, the requested page (or a random page, in the case of a cache hit) is stored into the cache and a random page from the cache is moved to one of the \( k \) locations in the block that was accessed as part of that request. Note that, due to the randomized cache replacement policy, a certain cached page may be evicted while it is being requested by the client. Also, to avoid timing attacks, a cached page is not returned immediately to the client, because that would reveal the cache hit to the adversary.

2.3 Secure Hardware Deployment

In this section we analyze the storage requirements and query processing cost of our methods in a secure hardware deployment. Our analysis is based on the configuration shown in Table 2.2, which is similar to the ones assumed in related studies [45, 50].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure hardware cache</td>
<td>64MB</td>
</tr>
<tr>
<td>Disk seek time ( (t_s) )</td>
<td>5ms</td>
</tr>
<tr>
<td>Disk read/write ( (r_d) )</td>
<td>100 MB/s</td>
</tr>
<tr>
<td>Secure hardware link bandwidth ( (r_b) )</td>
<td>80 MB/s</td>
</tr>
<tr>
<td>Encryption/decryption ( (r_{ed}) )</td>
<td>10 MB/s</td>
</tr>
</tbody>
</table>

Secure storage requirements The page retrieval algorithm necessitates the storage of the three vectors depicted in Figure 2.2, inside the secure hardware cache. Given a database of \( n \) pages, each of size \( B \) bytes, the pageCache vector stores exactly \( m \) pages, thus consuming \( m \cdot B \) bytes. The serverBlock vector stores the \( k+1 \) pages that are read from the server, i.e., it
requires \((k + 1) \cdot B\) bytes of space. Finally, the pageMap vector maintains information about the position of all database pages, plus an additional bit that indicates whether a page is currently cached. Consequently, it requires \(n \cdot (\log n + 1)\) bits of storage space. Summarizing, the total storage cost of our approach (in bytes) is given as:

\[
S = n \cdot \left\lceil \frac{(\log n + 1)}{8} \right\rceil + (m + k + 1) \cdot B \tag{2.7}
\]

**Page retrieval cost**  For every client query the secure hardware needs to perform 4
random accesses at the server’s disk. Two of those correspond to the read operations (one for reading the next block, and one for the additional page), while the remaining two are performed for writing back the re-encrypted pages. The $k + 1$ accessed pages are transferred twice between the secure hardware and the server (read/write) and are also processed twice by the encryption/decryption circuitry inside the secure hardware. Therefore, the query processing time at the server for retrieving a single page from the disk is:

$$Q_t = 4 \cdot t_s + 2 \cdot (k + 1) \cdot B \cdot \left( \frac{1}{r_d} + \frac{1}{r_b} + \frac{1}{r_{rd}} \right)$$ (2.8)

Figures 2.4 and 2.5 show some sample configurations for retrieving 1KB and 10KB pages, respectively, from databases of different sizes (with a privacy parameter $c = 2$). Specifically, they depict the page retrieval times and storage space requirements at the secure hardware as a function of the cache size $m$. For a 1GB database, a single secure coprocessor can retrieve privately 1KB pages in 27 ms and 10KB pages in 94 ms. Note that, unlike existing secure hardware PIR schemes that feature amortized cost, the processing times shown here are constant. For larger databases, we may leverage multiple coprocessors at the server site to increase the secure storage capacity. This will boost the value of $m$, thus reducing considerably the security parameter $k$. For instance, with 1 coprocessor (up to 64MB of storage space) and a 10GB database, we can retrieve 1KB pages in 197 ms and 10KB pages in 731 ms. On the other hand, combining the storage space of 2 coprocessors can reduce those times to 65 ms and 378 ms, respectively.

Larger databases cannot be trivially handled by the current technology of tamper-resistant CPUs, due to the minimal storage resources that they provide. Consequently, 100GB databases will require 10 coprocessors to retrieve 1KB pages in 197 ms and 10KB pages in 613 ms. Even though this is an entirely feasible solution, it may increase considerably the monetary cost of PIR. Finally, for 1TB databases, sub-second page retrieval times (727 ms
Figure 2.5: Page retrieval costs for 10KB pages ($c = 2$)

for 1KB pages and 907 ms for 10KB pages) are only feasible with over 4GB of secure storage. With the current technology, this capacity translates to over 70 coprocessor units. This excessive cost is mainly due to the pageMap data structure that maintains the location of every database page on the disk. However, this is an unavoidable cost because, unlike previous approaches that use hash functions to permute the entire database, our scheme reshuffles on a per page level and necessitates each page to be stored individually.

Figure 2.6 depicts the query response time as a function of the privacy parameter $c = 1 + \varepsilon$. We consider 1KB pages and set the cache sizes for the different databases to their largest
values shown in Figure 2.4. Clearly, there is a trade-off between the privacy level of our approach and the computational cost. If we wish to provide better privacy, we need to retrieve more pages per block (increase $k$) in order to reduce the value of $T$. As shown in Equation (2.5), this will essentially decrease the value of the privacy parameter $c$. Nevertheless, our algorithm is efficient under strict privacy requirements and, for databases up to 100GB, sub-second query response times are achievable even for $c = 1.1$.

Despite the restrictions of current secure hardware technology, our methods are also applicable in the two-party querying model, as explained in Section 2.1.1. In this setting,
the functionality of the secure hardware can be implemented on a powerful server (physically isolated from the adversary), thus allowing for much larger cache sizes. Consequently, our page retrieval algorithm can be implemented efficiently even for TB-sized databases. To verify the efficiency of this approach, we measured the page retrieval costs from a real implementation\(^2\) of the two-party model. We set up the service provider and the owner to run on two different machines that were connected through a WiFi network. The network round-trip time (RTT) was set to 50ms and was simulated with the `sleep` function. Figure 2.7 illustrates the query response time and storage cost at the data owner as a function of the cache size \(m\). With 6GB of storage space, the system can accommodate 2 million pages in its cache, achieving a query response time of 0.737\(ms\) (for 1KB pages). Note that the bottleneck in this architecture is the network transfer cost, since our algorithm necessitates the transfer of \((k+1)\) database pages twice between the owner and the service provider. As a result, retrieving larger pages (10KB) requires a significant amount of storage space (to reduce the value of the security parameter \(k\)) and, as shown in Figure 2.7(b), over 10GB of space is necessary to achieve a query response time of 1.3s.

\[\text{Figure 2.7: Page retrieval costs for 1TB database (}\ c = 2)\]

\(^2\)We used the Boost.Asio library for the networking primitives and the Crypto++ library for the AES implementation.
2.4 Conclusions

In this chapter, we introduced a novel approach that provides a much stronger notion of privacy compared to anonymity or encryption based schemes, but with a computational cost that is considerably lower compared to existing PIR approaches. Our methods are built on top of a secure hardware that acts as a proxy between the clients and the server. The secure hardware encrypts and constantly reshuffles the database pages, in order to create sufficient uncertainty regarding the exact location of an arbitrary page. We give a formal definition of the privacy level of our algorithm and illustrate how to apply it in practice. Based on the performance characteristics of the current state-of-the-art secure hardware platforms, we show that our method is computationally efficient, even for very large databases.
Chapter 3

A library for efficient computational PIR

In recent years, the ability to ”lease” computational power and/or online storage has evolved in one of the main paradigms of computing. Services like Amazon EC2[1] and Microsoft Azure[7] can provide both processing power and storage capacity in the form of a personalized rented computing cluster. A very common use of such cloud services facilitates storage purposes; a user can have her data stored remotely with the benefits of both accessing them ubiquitously and having a remote backup as well. However, such availability of an on-demand number of processors can also be utilized in parallelizing expensive computational tasks and easily escalating the level of parallelization by adding more processors from the service provider.

In this chapter, we investigate possible performance improvement gained by parallelizing computational-PIR algorithms. In particular, we attempt to design a generic library which will be able to provide parallelized versions of various PIR algorithms. The implementation will be based on the MPI framework, which can utilize the increasingly available resources of today’s cloud computing paradigm. Apart from improving performance, such a library can
also provide researchers with a "black box" implementation of arbitrary PIR algorithms, useful for further research topics.

A (potentially) commercial use case for such a library can be applied as a cloud service, therefore introducing the concept of PIR-as-a-service (PIRaaS). In such a setting, a cloud provider will be able to provide users/clients with the ability to perform (potentially paid) private queries instead of plain-text ones. Subsequently, the database providers will be able to upload the data as key-value stores in the cloud and the queries will be applied there directly. Security-wise, the key-value stores can be encrypted using symmetric ciphers and the users performing the queries will have to be authenticated from the database provider, therefore being able to have the encryption key.

We further investigate a proof-of-concept case for such a generic library, using key-value stores and the Gentry-Ramzan PIR scheme.

Recently, there has been research on taking advantage of parallel computational power for improving the efficiency of computational PIR schemes. Most of the existing work focuses on improving particular PIR algorithms by parallelizing specific performance bottlenecks or commonly used expensive operations, such as modular exponentiations.

In [48], Unal and Savas present a bandwidth optimized version of Lipmaa’s computational PIR scheme[40], by employing multi-core architectures.

In [47], Topcuoglu et al., propose a PIR scheme which parallelizes multiple modular exponentiations, as a whole multi-exponentiation, on multi-core processors.

In [41], Mayberry et al., present PIRMAP, a PIR scheme adapted to the widely used MapReduce cloud paradigm.

Our proposed library, can offer a generic framework inside which such existing approaches can be used for various schemes and unifying them through a common API.
3.1 Preliminaries

3.1.1 Message Passing Interface

Message Passing Interface (MPI) [6] is a message-passing library interface specification [24], defined by a community of parallel computing vendors, computer scientists and application developers, which is aimed at developing parallel applications and widely used in high performance computing. In its core, the library provides a set of routines which allow processes concurrently running in separate processors to exchange messages from each one’s address space to the other’s, thus orchestrating parallel workload scenarios. MPI is a specification with multiple implementations (i.e. [9], [8], [3]). The MPI standard officially supports programming language bindings for Fortran and ISO C. However, bindings for other popular languages are generally becoming available. Two of the main benefits MPI offers is portability and scalability to larger clusters.

Datatypes

Often processes need to exchange information during a parallel run. MPI provides its own datatypes to encapsulate the actual data being sent and received. The generic datatypes provided correspond to the most common types of data. Based on these generic datatypes, more complex ones can be derived from existing data structures (vectors, structs, ...).

Point-to-point Communication

One of the basic MPI operations is point-to-point communication. A message of some generic or derived datatype is transmitted between a distinct pair of processes, through an orchestrated pair of send and receive routines. The transmission can be blocking or non-blocking, whether the recipient process is allowed to continue with the rest of its instructions before the transmission is completed or not.
Collective Communication

The second major MPI type of operations is collective communication. Here, more than two processes participate in the message exchange. Examples of collective communication include a broadcast (one-to-all), a gather (all-to-one) and others. Two very common use cases for collective communications is at the beginning of a program, where a process distributes the data to the rest so that they can subsequently work in parallel, and at the end of the program where a process gathers the results for output or saving.

Communication context

MPI groups the processes of a cluster in "teams" called communicators. A communicator is consisted of a group of processes and a context. The context can be simplistically thought of as an ID of the particular group. Processes inside a communicator can pass messages to each other. The default communicator simply includes all the available processes, but more can be defined. Communicators can be useful in cases where distinct subgroups of the total available processes need to perform different and probably independent parts of the computation from the rest.

Process Topologies

A feature which is derived from communicators is the virtual topologies of processes. Virtual topologies map the program’s structure in a network. This virtual structure is independent of the actual hardware network though. The two main types of topology supported are Cartesian and Graph topologies. Cartesian defines an N-dimensional grid of processes, while Graph topologies define a graph where the nodes are the processes and the edges are communication links between them.
Parallel I/O

Parallel I/O is implemented on layers due to complexity. There is a parallel file system (i.e. Lustre[5], IBM Spectrum Scale[4], etc) in the bottom layer and a MPI-IO implementation on top of it. And an even higher level I/O library can be on top of it. Generally, parallel I/O will allow simultaneous access to data files from different processes/sources. High performance is also achieved through support for higher-speed network technologies and maximum I/O bandwidth.

Process Windows

During a collective operation, processes can create a window in their memory which will be accessible to remote processes.

Attribute Caching

Apart from a process group and a context, additional information can be associated with a MPI communicator. This feature, known as attribute caching, allows us to attach simple or complex data values to a communicator and processes belonging in this communicator can set and get the attributes’ values. MPI-2 introduced attribute caching for datatypes and windows as well.

3.2 Design

In designing a parallel library, there are some additional issues that have to be taken into consideration besides the general design principles which apply to conventional libraries [13][32]. In principle, in a parallel environment there are multiple resources that have to be controlled and possibly shared across the processing entities, as well as information that has to be exchanged among them. The partitioning of the data domain, communication between
the participating elements, load balancing across the network and I/O are some of the major issues that have to be resolved. Performance, scalability, isolation of communication and error handling are common requirements in any parallel library design [32]. The debugging of parallel software can also be significantly hard, especially as scalability increases.

A parallel library will generally be called on a communicator of processes from the host application. This original communicator will have to be duplicated so that the library operates on the isolated context of the duplicate copy. This action will provide a private space for the library’s processes to communicate without the risk of accidentally sending a message outside of the library’s scope. However, it will not ensure that multiple library invocations inside the same duplicate communicator will not accidentally communicate, thus the library will not be reentrant.

![Figure 3.1: PPIR overview](image)

**Figure 3.1: PPIR overview**

### 3.2.1 Database Partitioning

In our proof of concept design of the Gentry-Ramzan algorithm (See 1.3.1) we will initially modify the client setting for receiving multiple consecutive blocks at a time. Such set of
blocks will form a page of the database.

Each MPI process will be assigned a part of the database as its local workload and all the processes will operate on their assigned parts independently. For a database with block size $B$, a page size $k = xB$ (for integer $x$), $N$ available processes in the cluster and $k \geq N$, then each process with rank $r$, for $0 \leq r \leq N - 1$, will be assigned the blocks

$$\{C_t | t = dr + kv + \nu, 0 \leq v \leq k - 1, 0 \leq \nu \leq d - 1, \ d = \lceil \frac{k}{N} \rceil\}.$$ 

A sample database of 16 blocks is shown as an example in Figure 3.2(a). The chosen page size is $k = 4$. Figures 3.2(b),3.2(c) and 3.2(d) demonstrate how its blocks will be assigned to 4, 3 and 2 processes respectively. If $k < N$, it follows that $d < 1$, practically meaning that a block would have to be "spread" across more than one processes. In this case, we decrease the block size and re-split the database. It can easily be observed that the consecutive blocks which form a page are located in equivalent positions in each process.

\begin{center}
\begin{tabular}{cccccccccccc}
$C_0$ & $C_1$ & $C_2$ & $C_3$ & $C_4$ & $C_5$ & $C_6$ & $C_7$ & $C_8$ & $C_9$ & $C_{10}$ & $C_{11}$ & $C_{12}$ & $C_{13}$ & $C_{14}$ & $C_{15}$ \\
\hline
Process 0 & $C_0$ & $C_4$ & $C_8$ & $C_{12}$ \\
Process 1 & $C_1$ & $C_5$ & $C_9$ & $C_{13}$ \\
Process 2 & $C_2$ & $C_6$ & $C_{10}$ & $C_{14}$ \\
Process 3 & $C_3$ & $C_7$ & $C_{11}$ & $C_{15}$ \\
\end{tabular}
\end{center}

(a) Original database

\begin{center}
\begin{tabular}{cccccccccccc}
Process 0 & $C_0$ & $C_1$ & $C_6$ & $C_7$ & $C_{12}$ & $C_{13}$ \\
Process 1 & $C_2$ & $C_3$ & $C_8$ & $C_9$ & $C_{14}$ & $C_{15}$ \\
Process 2 & $C_4$ & $C_5$ & $C_{10}$ & $C_{11}$ & $00000000$ \\
\end{tabular}
\end{center}

(b) 4 processes

\begin{center}
\begin{tabular}{cccccccccccc}
Process 0 & $C_0$ & $C_1$ & $C_4$ & $C_5$ & $C_8$ & $C_9$ & $C_{12}$ & $C_{13}$ \\
Process 1 & $C_2$ & $C_3$ & $C_6$ & $C_7$ & $C_{10}$ & $C_{11}$ & $C_{14}$ & $C_{15}$ \\
\end{tabular}
\end{center}

(c) 3 processes

\begin{center}
\begin{tabular}{cccccccccccc}
Process 0 & $C_0$ & $C_1$ & $C_4$ & $C_5$ & $C_8$ & $C_9$ & $C_{12}$ & $C_{13}$ \\
Process 1 & $C_2$ & $C_3$ & $C_6$ & $C_7$ & $C_{10}$ & $C_{11}$ & $C_{14}$ & $C_{15}$ \\
\end{tabular}
\end{center}

(d) 2 processes

Figure 3.2: Database split for parallelized Gentry-Ramzan scheme (page size $k = 4$)

At this point, each process will calculate $e \equiv C_t (mod \pi_t)$ on its local data, using the Chinese Remainder Theorem. This value can be stored and reused for as long as the database
is not updated. If a change occurs (insertion, deletion, modification), the calculations will be repeated.

### 3.2.2 Query Response

Upon a query, the server will receive from the client $g$ and the modulus $m$ for the requested page, and will have to compute $g^e \pmod{m}$. In fact, each process will do so for its respective $e$ value. Recall that the processes have pre-computed $e$ using CRT through the database initialization process. Specifically, the master process of the cluster will broadcast $g, m$ to all the others, and gather the resulting individual $g^e$ values from them. These values will subsequently be sent to the client.

### 3.3 Implementation

#### 3.3.1 General Layout

Such a generic library should provide a compact and clean API for both the server and the client side. Examples of invocation are listed in Code Listings 3.1 and 3.2.

Listing 3.1: Sample server ppir invocation

```c
#include <mpi.h>
#include <ppir.h>

int main( int argc , char **argv ) {
    MPI_Init( &argc , &argv );

    const char *database = "/nfs/data/db.dat";
```
```c
PPIR_query query;
PPIR_query_result result;

/* Code to wait and receive 'query' from client */
PPIR_database_query( MPI_COMM_WORLD, database, query, &result);

/* Code to send 'result' to client */
PPIR_clear(query);
PPIR_clear(result);

MPI_Finalize();
return 0;
}
```

Listing 3.2: Sample client ppir invocation

```c
#include <ppir.h>

int main( int argc, char **argv ) {
    int X = atoi( argv[1] );
PPIR_query query;
PPIR_query_result result;
```
3.3.2 Public Parameters

The Gentry-Ramzan scheme (1.3.1) specifies some public parameters. These parameters remain the same for as long as the database is unmodified and should be available to every process in the cluster. The approach we take is to pre-compute them during the database initialization phase and subsequently use attribute caching in order to attach them to the communicator which will compute the query. The public parameters attached to the query communication are listed in Table 3.1. The length of the $p$, $S$ and $c$ arrays are equal to the number of processes $np$ in the communicator and are processed "in parallel". Every index in the three arrays corresponds to the process with the same rank (i.e. Process 2 is using $p[2]$, $S[2]$ and $c[2]$.)

The main benefit of caching the public parameters as a communicator attribute, is that
after they have been initialized during the first library call, subsequent calls (i.e. for other queries) can retrieve and reuse them, avoiding recalculation, until the duplicated communicator is freed. Therefore, if the library is called from a daemon host application (i.e. a server) the communicator may stay active continuously since the application can initialize the MPI environment and the communicators on startup and not finalize them until shut down. If the database is modified, the processes can recalculate the new values and update the cached communicator attribute.

In case the communicator is freed (i.e. in a server shutdown or reboot), the library API can provide an option to save them on disk and then reload them on the next initialization. This needs to be done only from one process, usually the root.

<table>
<thead>
<tr>
<th>Communicator</th>
<th>The duplicated communicator of ( np ) processes which calculates the query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database File</td>
<td>The location of the actual database in the cluster</td>
</tr>
<tr>
<td>( n )</td>
<td>The total size of the database in bytes</td>
</tr>
<tr>
<td>( l )</td>
<td>The block size in bytes</td>
</tr>
<tr>
<td>( t )</td>
<td>The total number of blocks</td>
</tr>
<tr>
<td>( mp )</td>
<td>The maximum bit length for any prime power</td>
</tr>
<tr>
<td>( p[np] )</td>
<td>Array of prime numbers associated with the blocks</td>
</tr>
<tr>
<td>( S[np] )</td>
<td>Array of prime powers</td>
</tr>
<tr>
<td>( c[np] )</td>
<td>Array of the primes’ exponents</td>
</tr>
</tbody>
</table>

Table 3.1: Public Parameters for the Gentry-Ramzan scheme

### 3.3.3 Datatypes

In dealing with large integers that are required, we use the GMP library[12]. Since the data exchanged between the processes will often be of \texttt{mpz\_t} type, a derived datatype is required for MPI communication. \texttt{mpz\_t} integers are represented internally as a dynamically allocated
array of limbs, each of a certain number of bytes in size. The exact number of limbs depends on the size of the integer. Therefore, the number of bytes in a mpz_t variable cannot be known in advance. Since the receiving process has to allocate space for the incoming message in advance, we send the integer in two steps. First, it is extracted in \( n \) pieces of unsigned long values and the count is sent to the receiver as a single generic MPI_INT value, so the required space can be allocated. Secondly, the extracted bytes are packed in a message of \( n \) MPI_UNSIGNED_LONG values and sent. The received buffer can finally be imported in a mpz_t variable on the receiving end.

A trivial implementation can be seen in Code Listing 3.3.

Listing 3.3: Sample mpz_t send and receive over MPI

```c
void mpz_send( mpz_t x, int dest, int tag, MPI_Comm comm ) {
    size_t count = mpz_size( x );
    MPI_Send( &count, 1, MPI_UNSIGNED_LONG, dest, tag, comm );
    unsigned long *num =
        mpz_export( NULL, NULL, 1, sizeof(unsigned long), 0, 0, x );
    MPI_Send( num, count, MPI_UNSIGNED_LONG, dest, tag, comm );
}

void mpz_recv( mpz_t x, int source, int tag, MPI_Comm comm,
               MPI_Status *status ) {
    unsigned long count;
    MPI_Recv( &count, 1, MPI_UNSIGNED_LONG, source, tag, comm, status );
}```
3.3.4 Database Initialization

The database initialization method is responsible for two jobs. It should partition the database in parts that will be concurrently read by the cluster’s processes and it should initiate the public parameters. Before performing any of the two though, it will first duplicate the MPI communicator in which it was called, in order to create a safe isolated environment for the processes involved.

In particular, it will first create an attribute key which will be associated with the public parameters attribute (Section 3.3.2). This key however will only be created if it does not already exist. In case it already exists, it means that the library has already been called before.

```c
int flag;
if ( db_keyval == MPI_KEYVAL_INVALID )
    MPI_Comm_create_keyval( MPI_NULL_COPY_FN, MPI_NULL_DELETE_FN,
                            &db_keyval, NULL );
MPI_Comm_get_attr(comm, db_keyval, (void **)&db, &flag);
if ( flag == 0 ) { /* Library has not been called before */ }
else { /* Library has been called before */ }
```
In the latter case, the initialization work can be skipped and just get the existing attribute. This feature can allow the initialization method to be called indirectly from the query computation method, thus making the API cleaner for the end user. Following this approach, the first call for a query computation will also initialize the required parameters of the database and subsequent query calls can just utilize the parameters calculated in the first call.

In the case where the initialization has not been previously performed or the database has been modified, the communicator of the participating processes will be duplicated and the copy will be saved in the public parameters (Table 3.1). The processes will read their respective parts of the database as described in 3.2.1 and 3.3.5, the public parameters will be computed and the communicator attribute will be cached.

For the initialization we need to commit two new MPI datatypes representing a database block and page. The MPI_Type_contiguous() routine can be used to commit these types as contiguous sets of bytes and blocks respectively. Subsequently, the processes will call the MPI_File_open() and MPI_File_set_view() to set the way they "view" the database file. Setting the view is critical for the database split, since each process will be able to read only its own view of it. This particular operation can be also further implemented as an autonomous part of the library, thus providing different "splittings" for various MPI schemes.

Finally, the processes can collectively call MPI_File_read_all() to read their respective parts of the database file and proceed with computations. After this point, each process operates "locally" on its own part of the database file.
3.3.5 Data Storage

In our design the database is stored in a single location accessible from the cluster. The processes will read their individual parts of the data from this location, in order to compute the $e$ value through the Chinese Remainder Theorem. An alternative would be to store their data parts locally and avoid the communication overhead. This approach however will also affect the scalability of the system. The cluster which is used may change in size and the storage capacity of each of the participating nodes cannot be known in advance. By storing the data in a single location, which can be independently backed up by third party solutions, allows the user of the library to practically utilize any available nodes for the computation regardless of their storage capabilities. Additionally, nodes can be added or removed without the danger of data loss or the overhead of redistributing the data. Any MPI cluster therefore can be used solely for amortizing the computational cost of PIR. Assuming that the database is stored in a central location, we can utilize the collective parallel I/O capabilities of MPI. At the library level, the processes can concurrently read the database and at the system level a parallel file system can further support this concurrency. During the initialization phase, we dynamically set a file view based on the number of available processes in the cluster, so that each process "sees" only the part of the database which it needs as a serial region. A collective call is then made for all the processes to concurrently read their view of the database, scan all the bytes and compute their local $e$.

3.3.6 Query Computation

The query method call is the main function that library client will call on the server-side. As explained in 3.2.1, the first query call will also partition the database and initialize the database parameters. For a query operation, the database client has to send its $g$ and $m$ values for the requested page. Upon receiving such a pair of values, the root process of the
cluster will broadcast them to all the participating processes. Each process will perform a modular exponentiation to compute \( g_e = g^e \pmod{m} \) and the results will be gathered back at the root.

### 3.3.7 Error handling

The most important case of failure in our setting, is when one or more processes abort for any reason in the midst of a computation. In example, if a process fails during a query computation, its \( g_e \) value will never be received in the root process, setting the query result incomplete.

MPI does not provide a fault tolerance mechanism and by default any error that may occur is considered fatal, causing the entire job to abort. Initially, this behavior has to be altered by setting the default error handler to continuing execution in case of an error. Subsequently, callback functions have to be defined to either handle the error or notify the library client, depending on the type of error. The root process determines which process and function caused the error. Following, a new process should be spawned dynamically. The root process will handle to the new child process the required data and the child will execute the failed function.
Chapter 4

A private key-value store

To demonstrate how our library can be used in a practical scenario, we design a simple key value store on which we will be able to perform private queries. A key value store is a simple form of non-relational database which contains a collection of records. Each record is associated with a key and lookups for specific records are performed based on their associated key. The keys are stored with an index structure, such as a hash table, a B-tree or a B+tree. A key lookup will search the index structure for the key and the latter will refer to or be stored along with the actual record. Key-value stores have been increasingly popular along with the move towards cloud computing in contrast with traditional SQL-based relational databases.

In certain data settings a non-relational database can potentially offer better performance, scalability and more straightforward implementations. Assume a data collection which keeps data of various categories and/or types entered at various times along with the individual owners of the data. Using a traditional relational model, various tables will normally be used to keep the distinct categories of data (i.e. audio, image, video, document, user, ...) as well as tables representing the users and date/time of archival. Additionally, appropriate relations between the tables will be formed. If for example, a client wishes to retrieve a
particular entry of a certain user, a number of joins will have to be performed in order to retrieve all the corresponding results associated with the user and the entry time. In web setting like a streaming service or a social network, where series of similar queries have to be served, the database server will have to perform potentially hundreds or thousands of queries and joins. Using a non-relational database a single entry can hold all its associated data and therefore be served in a single query.

Another, more related and applicable scenario to private information retrieval is a research or archival dataset in CSV format. Such datasets are usually stored in plain files with multiple lines, each line representing a single entry and containing a series of alphanumerical values describing properties of the entry. Various scientific fields rely on archived data of such form for record keeping, historical retrieval and analysis, machine learning. Such datasets offer the benefit of being minimal in size due to the mostly textual representation used, since thousands of entries can be stored in a small plaintext file, and in many scenarios privacy in extracting entries from them can be required if sensitive information is included (i.e. medical information) or for collaborative access between potentially non-trusting parties (i.e. finance data). Access to a client who wishes to extract certain entries from a CSV dataset can be provided through a private information retrieval scheme and the relatively small size of the dataset can balance the higher computational constraint added, resulting in non-prohibitive performance.

### 4.1 Design

In this section we design a proof-of-concept key-value datastore that can serve private requests using the library described in Chapter 3. We assume a text multi-line dataset, where each line represents a particular entry. Each line contains alpha-numerical values in a certain order, each value representing a corresponding property of the entry. The values are
delimited by a preselected character (comma, tab, ...). The dataset itself can optionally be encrypted.

Each entry in the dataset will be associated with at least one key. The keys will be indexed by using a hash table. The hash table along with the records will be stored in a two separate flat files, the first file being the hashtable of the keys and the second file being the unmodified dataset file itself, thus additionally allowing plain queries in the dataset without the privacy requirement. Each key in the hashtable will be stored along with a file offset. The latter will be the offset of the key’s associated entry in the dataset file.

For retrieving a key’s corresponding value, normally the requested key will be hashed in order to find its location on the hash table. This location will subsequently also contain the offset of the key’s record on the dataset file. Thus, for a non-private query, the record can be retrieved in a single request. The client asks for a particular key and assuming the key exists in the hashtable the server simply extracts its associated offset and immediately returns the appropriate entry of the dataset from this offset. For a private query however, both the particular key lookup and the record retrieval from the dataset need to be private, since the server should not learn neither. The lookup process therefore, needs to take place in at least two steps, each step being a distinct PIR query, to the hash table and the dataset.

4.1.1 Page size

Our ppir library is designed to operate on a fixed page size and each query requests a particular page number from the database. Therefore, we will use a fixed size for the retrieved values from the dataset, equal to the page size set in the library. The decision of the page size can be made accordingly with the client requirements or the dataset’s structure but it should remain fixed afterwards. If the client wishes to further process the retrieved records with additional software the page size should potentially comply with it. Otherwise it can be set to the size of one or more entries in the dataset.
CHAPTER 4. A PRIVATE KEY-VALUE STORE

The hashtable entries will consist of key/offset pairs, which will probably be smaller in size than their corresponding entries’ size. For this case, the hashtable and the dataset can be initialized with different page sizes with two instances of the library. The first query will return a page equal to size of the key/offset pair(s) and the second query a page equal to the dataset record.

4.1.2 Hashtable buckets

Each entry in the hash table is a bucket holding keys and offsets to the dataset file location where the record is stored. Initially all the offsets are empty. When adding a new record, the new key is hashed modulo the size of the hash table and appended in the resulting bucket. The capacity of each bucket relies on the chosen page size of the hash table.

For a key lookup, the hash of the key modulo the table size will give the bucket in which the key/offset is located. In case the bucket contains multiple keys, the correct one will be chosen and its associated offset will be used for the following dataset lookup. The general layout of the buckets in the hash table is shown in Fig. 4.1. In this example, the page size is set equal to the bucket. Since the key size and the offset size will be fixed, each bucket will contain a certain number of key/offset pairs.

When inserting a new record, the bucket number will be retrieved as \( \text{hash}(k) \% \text{hash_table_size} \) and the key should be appended in the bucket.

In case of a collision when inserting a new record, the offset will be appended in the bucket if there is space remaining, otherwise the hash table will have to be expanded to hold new buckets. The entries at this point will have to be rearranged according to the new hash table size by computing the key hashes modulo the new size.
CHAPTER 4. A PRIVATE KEY-VALUE STORE

4.1.3 Queries

A complete query in the dataset will require two distinct PIR queries. In the first part, the client will hash her requested key modulo the hashtable’s size which can be received from the server, and retrieve the exact bucket number in the hash table. A PIR query is issued for this bucket number and the corresponding page is retrieved. The retrieved bucket will potentially contain multiple key/offset pairs. The additional offsets can be cached on the client for future queries. In particular, the key hash can be checked against the cached ones and possibly avoid the initial offset request. If the requested key exists in the received bucket, a second PIR query will take place in the dataset file. The page requested this time will be the one starting at the previously received key’s offset.

If the received bucket does not contain the requested key, this key has not yet been added to the database.
Figure 4.2: A private key-value store
Chapter 5

Conclusions

In this dissertation we examine two different approaches for efficient Private Information Retrieval. In the first part of our work, we introduce a novel approach that provides a stronger notion of privacy compared to anonymity or encryption based schemes, but at the same time with a lower computational cost than existing PIR approaches. We utilize secure hardware that acts as a proxy between the clients and the server. The secure hardware encrypts and constantly reshuffles the database pages, in order to create sufficient uncertainty regarding the exact location of an arbitrary page. We give a formal definition of the privacy level of our algorithm and illustrate how to apply it in practice. Based on the performance characteristics of the current state-of-the-art secure hardware platforms, we show that our method is computationally efficient, even for very large databases.

In the second part, we examined the design and implementation techniques for a library which utilizes the MPI framework for providing various MPI scheme implementations, showing the Gentry Ramzan computational PIR algorithm as a proof of concept, for use on various sizes of clusters. Such a library can provide clients with a "black-box" PIR solution which can be embedded in applications that will allow private access to a dataset. As a demonstration
setting, we design a basic key-value dataset, holding fixed size records. Various parties wish to privately query the store for retrieving records. The clients will perform two queries, one in the index structure at the server to retrieve the location of the records in the dataset and a second in the dataset to retrieve the records.

\section*{5.1 Future Work}

The PIR library can be further enhanced by incorporating additional levels of parallelization. Certain parts of the computation still need to be executed sequentially. OpenMP [11] support can be added on an extensive degree to speedup these serial areas of the computation, especially on the client’s part. OpenMP can also be included the server computation and its cooperation with MPI should be carefully designed in order to complement it, in particular in loop sections. Another important area in high performance computing is GPU computations, with technologies like CUDA[2] and OpenCL[10]. Parallel processes can take advantage of existing multi-core GPUS and further split the workload on a per-node basis.

A complete key-value datastore which provides private queries is a significant future project. The design proposed can be implemented in a high quality setting, supporting all the operations provided by most of today’s key value databases with the addition of optional private queries.
Bibliography


[34] R. Jones, R. Kumar, B. Pang, and A. Tomkins, “‘I know what you did last summer’: Query logs and user privacy,” in *CIKM*, 2007.


