Efficient Private Information Retrieval

KONSTANTINOS F. NIKOLOPOULOS
THE GRADUATE CENTER, CITY UNIVERSITY OF NEW YORK
KNIKOLOPOULOS@GRADCENTER.CUNY.EDU

COMMITTEE:
SPIRIDON BAKIRAS (MENTOR)
ABDULLAH UZ TANSEL
STATHIS ZACHOS
Introduction

Database queries can reveal sensitive information about an individual. Therefore, privacy-preserving query processing is an emerging research area in database systems.

- **Objective**: Hide the content of the query from the database server

Existing solutions
- Database encryption
- Query obfuscation
- Private information retrieval (PIR) – our work
Introduction

Database encryption
- Design appropriate encryption methods that facilitate query processing on the encrypted data (e.g., OPES)
- Queries are also encrypted, i.e., *query privacy* is inherent
- Limitation: *Access pattern attacks*

Query obfuscation
- Hide the actual query into a pool of decoy queries
- Limitation: *Weak privacy*

PIR
- Offers perfect privacy
- Limitation: *Computationally expensive*
Private information retrieval

Consider an $n$-bit database $DB = \{x_1, x_2, ..., x_n\}$. A PIR protocol allows a client to retrieve bit $x_i$, while keeping the value of the index $i$ secret from the server

- Can easily be adapted to retrieve blocks of data

Objective: Hide the content of the query – the database may not be encrypted

- Encryption is orthogonal – can be applied independently

There are three main categories of PIR protocols

- Information theoretic PIR
- Computational PIR
- Secure hardware based PIR – our work
Information Theoretic PIR

[Chor et al., *FOCS’95*]

Ensures that the query discloses no information about the retrieved bit, even if the server has \textit{unbounded} computational power.

For a single server and information theoretic privacy, all \( n \) bits of the database need to be transmitted to the client.

For sublinear communication cost, the database must be replicated into \( k \) \textit{non-colluding} servers.

\textit{Not practical} – the servers need to be trusted that they will not collude with each other.
Computational PIR

These protocols work with a single server

They employ well known cryptographic primitives that guarantee query privacy in the presence of a *computationally bounded* server

- Quadratic residuosity assumption [Kushilevitz and Ostrovsky, FOCS’97]
- $\varphi$-hiding assumption [Gentry and Ramzan, ICALP’05]
- *Extremely expensive* – need at least one modular multiplication for every bit of the database
Secure hardware PIR

Implements the PIR functionality through a specialized hardware, called secure co-processor (SCOP)

- This is a tamper-resistant CPU, which can be trusted to carry out its processing unmolested and unobserved, even if the adversary has physical access to it

[Wang et al., ESORICS’06]

- Database consists of $n$ pages
- The SCOP initially encrypts and obliviously permutes the database pages
- The SCOP’s cache can hold $k$ out of $n$ pages
  - Typical cache size is 64MB
Secure hardware PIR

[Wang et al., *ESORICS’06*] (cont’d)

- Clients communicate with the SCOP through a secure SSL connection
  - Each client requests a certain page that is retrieved by the SCOP and returned back to the client through the SSL connection
  - For every request the SCOP inserts a new page into the cache (even if it was a cache hit)
- When the storage capacity is reached, the database is re-shuffled
- The amortized computational cost of this approach is $O(n/k)$
Secure hardware PIR

[Williams and Sion, NDSS’08 and Williams et al., CCS’08]

- Amortized $O(\log^2 n)$ and $O(\log n \log \log n)$ computational costs, respectively (state-of-the-art)

- Implement the Oblivious RAM model on a SCOP
  - The SCOP maintains a pyramid data structure with $\log_4 n$ levels in the server’s disk, where level $i$ contains $4^i$ buckets
  - For every request, the SCOP accesses one bucket per level
  - Every requested page is inserted at the top of the pyramid structure
  - When a level becomes full, the SCOP empties it into the next level, after re-encrypting the pages, and obliviously re-permuting them
Motivation

Secure hardware PIR is currently the only practical approach for acceptable privacy

Existing solutions feature *amortized* costs
- Some queries may incur prohibitively large response times – 100s or 1000s of seconds
- This is due to the page re-shuffling process

Consider a database of $n$ pages
- After re-shuffling the database, any page has an equal probability ($= 1/n$) of being stored in any of the $n$ disk locations
Motivation

Some applications may not require such stringent privacy guarantees
- What if we allow pages to land in different disk locations according to a non-uniform distribution?

Our goals
- Reasonable page retrieval times
- Constant cost
- Adjustable privacy level
Our approach

Every requested page is temporarily stored inside the SCOP’s cache

Continuous re-shuffling of the database

- During each request instant, *one* previously retrieved page is relocated to a new position on the disk

We introduce the notion of *c-approximate PIR*

- 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*
Architecture

Adversary
- Polynomial time
- Can query the database
- Can see the accessed locations on the disk
- Knows all algorithms implemented inside the secure hardware
- Curious but not malicious
Page retrieval algorithm

During each page request, the SCOP

- Retrieves a block of \( k \) contiguous pages (in a round-robin manner)
  - \( k \) is the security parameter
- Retrieves the requested page or a random one (in the case of a cache hit)
- Moves a random page from the cache to one of the \( k \) locations corresponding to the retrieved block
- Stores the retrieved page into the cache

Database updates are handled similarly
Security analysis

What should the value of $k$ be to satisfy the privacy parameter $c$?
  - Assume a cache size of $m$ pages

Consider a page that is just stored into the cache
  - The probability that it is relocated on the subsequent block is

$$P_{max} = \sum_{i=0}^{\infty} \left( 1 - \frac{1}{m} \right)^{\frac{n}{k} \cdot i} \cdot \frac{1}{m} \cdot \frac{1}{k}$$

  - The probability that it is relocated on the block before that is

$$P_{min} = \sum_{i=0}^{\infty} \left( 1 - \frac{1}{m} \right)^{(i+1) \cdot \frac{n}{k} - 1} \cdot \frac{1}{m} \cdot \frac{1}{k}$$
Security analysis

To satisfy the privacy parameter \( c \), we want

\[
\frac{P_{max}}{P_{min}} = c \Rightarrow k = \frac{n}{\log\left(1/c\right)} + 1
\]

\( c = 1 \) corresponds to the trivial case of PIR \( k = n \)

Given \( c \) and \( n \), the the value of the security parameter \( k \) is determined by the available cache capacity

- Page retrieval times increase linearly with \( k \)
Secure hardware deployment

Storage cost
- Cached pages
- Look-up table (location of each page on the disk or cache)

Page retrieval cost
- Transfer cost between the SCOP and the server \((k+1\text{ pages})\)
- Encryption/decryption cost inside the SCOP (bottleneck)
Secure hardware deployment

Typical values used in our results

<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>5( ms )</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>
Response time for 1KB pages ($c=2$)
Response time vs $\epsilon = 1 + \varepsilon$

(a) 1GB ($m = 50000$)

(b) 10GB ($m = 100000$)

(c) 100GB ($m = 500000$)

(d) 1TB ($m = 500000$)
Software implementation (1TB, $c=2$)

(a) 1KB pages ($n = 10^9$)

(b) 10KB pages ($n = 10^8$)
Back to Computational PIR

Traditional Computational PIR relies on a single server.

Today’s infrastructure:
  ◦ Cloud Computing
  ◦ High Performance Clusters
  ◦ Parallel Programming
  ◦ Amazon EC2, Microsoft Azure, ...

Motivation
  ◦ Employ today’s increasingly available parallel computational resources
  ◦ Goal: Improve Computational PIR Efficiency
Parallel Computing

Communication
  ◦ Shared memory
  ◦ Message Passing

Computation
  ◦ Data parallelism
    ◦ Same Instructions, Multiple Data (SIMD)
  ◦ Operation parallelism
    ◦ Multiple Instructions, Multiple Data (MIMD)
    ◦ Single Program, Multiple Data (SPMD)
Message Passing

Process
- A processor running a program. Uses its own program counter and address space.

Inter-process communication
- Move data from one process’ local address space to another’s.
- Cooperation – Messages have to be sent and received.
Message Passing Interface (MPI)

What is MPI?
- A Message Passing library specification
- Many implementations
  - MPICH, OpenMPI, ...
- MIMD/SPMD parallelism
- Provides access to parallel hardware, while abstracting underlying details of it.
- Portable - Implemented in many platforms
- Languages: Mainly C and Fortran.
- MPI Standard: http://www.mpi-forum.org
MPI Basics

Datatypes
Point-to-point communication
Collective communication
Process topologies
Parallel I/O
Communicators, Groups, Processes

Communicator = Group + Context
Our proposed architecture

```cpp
#include "ppir.h"
...
PPIR_Query(DB, PIR_Scheme, Info)
...
PPIR_Update(DB, Operation, Info)
..."
Gentry Ramzan PIR

$O(w + B)$ communication complexity
- $w$: security parameter, based on database size $n$ and $B$.
- $B$: length of database blocks (in bits).

Relies on the Phi-hiding assumption

Server Operations
- Chinese Remainder Theorem
- Modular multiplications

Client Operations
- Discrete logarithm (Pohlig-Hellman)
  - efficient due to small primes used
Gentry Ramzan – Parallel Version
Database Initialization

Each process will independently handle a part of the database.

Operate on pages, not blocks.
- Page size: $k = xB$, for integer $x$.

$N$ available processes
- process rank: $0 \leq r \leq N-1$

$$\{C_t | t = dr + ku + v, 0 \leq u \leq k-1, 0 \leq v \leq d-1, d = \lceil \frac{k}{N} \rceil \}.$$
Gentry Ramzan – Parallel Version
Database Initialization Example

(a) Original database

<table>
<thead>
<tr>
<th>C_0</th>
<th>C_1</th>
<th>C_2</th>
<th>C_3</th>
<th>C_4</th>
<th>C_5</th>
<th>C_6</th>
<th>C_7</th>
<th>C_8</th>
<th>C_9</th>
<th>C_{10}</th>
<th>C_{11}</th>
<th>C_{12}</th>
<th>C_{13}</th>
<th>C_{14}</th>
<th>C_{15}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C_0</td>
<td>C_4</td>
<td>C_8</td>
<td>C_{12}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C_1</td>
<td>C_5</td>
<td>C_9</td>
<td>C_{13}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C_2</td>
<td>C_6</td>
<td>C_{10}</td>
<td>C_{14}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C_3</td>
<td>C_7</td>
<td>C_{11}</td>
<td>C_{15}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) 4 processes

<table>
<thead>
<tr>
<th>Process 0</th>
<th>C_0</th>
<th>C_1</th>
<th>C_6</th>
<th>C_7</th>
<th>C_{12}</th>
<th>C_{13}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>C_2</td>
<td>C_3</td>
<td>C_8</td>
<td>C_9</td>
<td>C_{14}</td>
<td>C_{15}</td>
</tr>
<tr>
<td>Process 2</td>
<td>C_4</td>
<td>C_5</td>
<td>C_{10}</td>
<td>C_{11}</td>
<td>0000000000</td>
<td></td>
</tr>
</tbody>
</table>

(c) 3 processes

<table>
<thead>
<tr>
<th>Process 0</th>
<th>C_0</th>
<th>C_1</th>
<th>C_4</th>
<th>C_5</th>
<th>C_8</th>
<th>C_9</th>
<th>C_{12}</th>
<th>C_{13}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>C_2</td>
<td>C_3</td>
<td>C_6</td>
<td>C_7</td>
<td>C_{10}</td>
<td>C_{11}</td>
<td>C_{14}</td>
<td>C_{15}</td>
</tr>
</tbody>
</table>

(d) 2 processes
Gentry Ramzan – Parallel Version
Query Generation & Extraction

Client generates pair \((G, g)\)

\(G\): cyclic group of order \(|G| = q^\pi_i\), for an integer \(q\)

\(g\): generator

Receives \(g_e = g^e \equiv G\) from server.

Computes \(h_e = g_e^q\)

\(\log_h h^e\) within \(H \setminus \text{in } G\) of order \(\pi_i\)

- Pohlig-Hellman
- Efficient, if \(p_i\) primes were small.
Server receives $g$ and the module $m$.
Distributes the values to the processes.
Gathers $g^e$ from all
Sends back to client

Many hidden MPI details behind these operations, partly due to length of numbers.
Gentry Ramzan – Parallel Version Overview
Kushilevitz Ostrovsky PIR

First computational PIR which eliminated database replication.

Relies on Quadratic Residuosity Assumption.

If $p_1, p_2$ are large primes, determining if a number is a Quadratic Residue (QR) or Quadratic Non Residue (QNR) in $(mod \; p_1p_2)$, is computationally hard.
**Kushilevitz Ostrovsky Example**

Example adapted from Tan’s presentation

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Get $M_{2,3}$

- $e=2$, $g=3$, $N=35$, $m=6$
- QNR=$\{3,12,13,17,27,33\}$
- QR=$\{1,4,9,11,16,29\}$

$$z_i = \prod_{j=1}^{t} y_j \cdot y_j^{1-M_{i,j}}$$

$z_2=\text{QNR} \implies M_{2,3}=1$

$z_2=\text{QR} \implies M_{2,3}=0$
Kushilevitz Ostrovsky
Improvements

Read chunks of bits and store in cache.
Keep all 256 possible combinations for a byte.
Further Work

Additional parallelization
  ◦ i.e. GPU for modular multiplications

Add more PIR schemes into framework

Collective run times of experiments.

Develop complete application.
Questions?


- MPI Standard: [http://www.mpi-forum.org](http://www.mpi-forum.org)