FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Evaluation of Percy++, A Private Information Retrieval Library
Student:	Arnold Stanovský
Supervisor:	Ing. Josef Kokeš, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

1) Research the Private Information Retrieval (PIR) problem. What it is, why do we want it, which solutions are available.
2) Describe the major PIR protocols.
3) Select a suitable real-world dataset, propose realistic use case scenarios and design a selection of tests for PIR against this dataset.
4) Using the Percy++ library, implement the proposed tests using various PIR protocols. Analyze the security properties as well as resource consumption.
5) Discuss your results.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 15 February 2023 in Prague.

Bachelor's thesis

EVALUATION OF PERCY++, A PRIVATE INFORMATION RETRIEVAL LIBRARY

Arnold Stanovský

Faculty of Information Technology
Department of Computers Systems
Supervisor: Ing. Josef Kokeš Ph.D.
June 29, 2023

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my supervisor for his patience and valuable insights. I would also like to thank anyone who supported me on this journey in any way.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on June 29, 2023 Arnold Stanovský

Abstract

In this thesis, I explore the Private Information Retrieval (`PIR`) problem. I describe situations in which this problem arises and the incentives for its solution. I then present several methods for retrieving online resources privately (i.e. while concealing which resource is being accessed) and discuss real world use-cases for these methods.

To verify my claims about the feasibility of such use cases, I also present the design of several benchmarks to measure the performance impact of choosing specific methods, using queries over an SQL database as an example of a resource to retrieve. Finally, I describe my implementation of these benchmarks using the `Percy++` library and discuss their results.

Keywords private information retrieval, PIR, evaluation, benchmark, SQL

Abstrakt

V této práci se věnuji problému Private Information Retrieval (`PIR`). Popisuji situace, ve kterých tento problém vzniká a motivace k jeho řešení. Následně představuji několik metod získávání online zdrojů soukromě (tedy způsobem, který skrývá, ke kterému zdroji je přistupováno) a diskutuji reálné případy použití těchto metod.

Abych ověřil má tvrzení o uskutečnitelnosti těchto případů použití, prezentuji návrh několika sad testů měřících dopad specifických metod na výkon. V těchto testech používám dotazy nad SQL databází jako příklad zdroje, který lze získat. Závěrem popisuji mou implementaci těchto testů za použití knihovny `Percy++` a diskutuji jejich výsledky.

Klíčová slova private information retrieval, PIR, evaluace, výkonostní testování, SQL

List of Abbreviations

`AG`	Aguilar-Melchor's algorithm
`CHOR`	Chor's algorithm
`CPIR`	Computational `PIR`
`GF2`N	Goldberg's algorithm over $GF\left(2^N\right)$
`HYB`	Devet & Goldberg's hybrid algorithm
`ITPIR`	Information-theoretic `PIR`
`PIR`	Private Information Retrieval
`R_AG`	Recursive `AG`
SQL	Structured Query Language
`ZZ_P`	Goldberg's algorithm over $\mathbb{Z}_P$

Introduction

Preserving the anonymity of users online has always been a challenging task. While several technologies for concealing the users' identity have been developed and are commonly used (privacy networks, anonymizers), there are cases in which this is simply not sufficient.

One can, for example, imagine a situation in which their car breaks down while traveling abroad. One might need to find someone who can repair the car, speaks the same language language and preferably can get to one's location shortly. In this situation, a search engine or perhaps a dedicated database might be used to find such a person. While there are several ways to conceal one's IP address and other identifying information, some of it (such as one's location, the make of one's car or the language that one speaks) will have to be present in one's query. Should one wish to conceal this information, finding someone who can help them is going to become much more difficult.

We can see that there could be a desire by the user to retrieve the information without revealing some sensitive parts of the query. Does the server maintainer have an incentive to allow them to do this? Indeed, collecting user information can be a powerful source of revenue for the maintainer, either through the direct sale of said information, or indirectly through (micro-) targeted advertisement. However, a maintainer wishing to respect the privacy of the user does have some alternative sources of income. First, instead of making their business model reliant on income from advertisers, they could shift towards a subscription based platform, where the main source of the income is the user. A privacy conscious user would definitely be willing to pay a premium for this service. Second, advertising on such a platform might be interesting for companies looking to reach privacy conscious customers. While this targeted advertising would be less specific and therefore less profitable for the maintainer, it could supplement the income from the users' subscriptions.

Having established motivations of both the server maintainer and the user, we can now explore technical solutions to this problem. Trivially, the server could transfer the entire database to the client and let the client perform the query themself. However, this approach has some limitations. Namely, the database might be too large for the transfer or even for the client's device's memory. Therefore, an approach that minimizes the amount of data transferred is necessary.

This problem is known in the literature as **Private Information Retrieval** (`PIR`). In this thesis, I first describe the solutions to this problem that have been implemented in the `Percy++` library. After a brief formal introduction to the problem and the library itself, I compare the computational and transfer costs of each one of these solutions and analyze their respective strengths and weaknesses.

Further on, I describe various deployment scenarios, such as the one of a privacy conscious database operator described earlier, or a quasi anonymizer. I give relevant real-world examples where appropriate. For each of those scenarios, I choose a `PIR` method that would be the most appropriate for it, and explain why.

Finally, I verify my claims by performing several benchmarks. The benchmarks are performed using a model SQL database and with the aim of obscuring sensitive parts of SQL queries made against it. The aim of the benchmarks is to conclusively state whether or not the performance of my proposed scheme is sufficient and whether or not the client can obtain better performance by revealing some information about their private constant. I discuss the results and suggest further research.

Chapter 1
Private Information Retrieval

In this chapter, I describe the problem of **Private Information Retrieval** and some of its solutions. Without further ado, let us formally define this problem, using the definition from [1]:

▶ **Definition 1.1.** *Given k databases each holding $x \in \{0,1\}^n$ and a user holding $i \in \{1,\ldots,n\}$, a solution to the **Private Information Retrieval** problem is a protocol that allows the user to retrieve the value at i without leaking any information about i to the database.*

While this is the most general and well known definition, it isn't really practical for our use-case. Instead of retrieving a single bit from the database, we will want to extract a block of l bits, using the extension of classic `PIR` proposed by [2]. While this could be accomplished by simply invoking one bit retrieval l times, we can reduce the overhead of this operation significantly.

Directly derived from this definition is the main security issue that all schemes must address, the distinguishability attack. A scheme must guarantee, aside from general structural security, that two queries for different items cannot be distinguished from one another.

Let us now take a look at the three families of `PIR` that are implemented in `Percy++`: computational `PIR`, information-theoretic `PIR`, and hybrid `PIR`.

1.1 Computational PIR

Computational `PIR` (`CPIR`) schemes offer the user a lower level of protection while also making the fewest assumptions about the servers. In this family of schemes, there is only one server ($k = 1$) and it is computationally bounded. It is not impossible for the server to learn the client's secret, it is only computationally infeasible.

1.1.1 Aguilar-Melchor and Gaborit 2007

In 2007, Sion and Carbunar [3] concluded that `CPIR` schemes were computationally impractical (i.e. are orders of magnitude slower than the trivial transfer of the entrire database). However, the very same year, Aguilar-Melchor and Gaborit [4] introduced a scheme that, while having a slightly higher communication cost than the previous `CPIR` schemes, drastically reduced the computational complexity of server reply generation. The scheme's performance has been further improved [5] by running using a GPU resulting in a total of three orders of magnitude speedup over previous schemes. These results have been empirically validated in [6]. While the `Percy++` implementation does not use the GPU, it still performs better than the trivial transfer in situations where network speed is limited.

1.1.1.1 Algorithm

I will now describe the algorithm from [4]. The algorithm has three user-chosen integer parameters: N is the size of matrices used by the algorithm, w_{AG} is the size of a single word the algorithm operates on and should be equal to $\lceil \log(n \times N) \rceil + 1$, and p is a prime that defines the field in which the arithmetic operations of the algorithm will take place ($\mathbb{Z}_p$). It needs to be larger than $2^{3w_{AG}}$.

Aguilar-Melchor and Gaborit [4] suggest the following values for these parameters: $N = 50$, $w_{AG} = 20$, $p = 2^{60} + 325$. This means that n can be at best just below 21000. To overcome this limitation, we can increase the value of w_{AG} or use recursion. The use of recursion in the context of this algorithm will be discussed later. We also note $q = 2^{w_{AG}}$.

The general idea of the algorithm is to generate a set of n matrices containing noise (disturbed matrices), one of which contains a much stronger noise than the others. The index of the matrix shall be the index of the record the client is seeking. This set shall then be sent to the server, where each matrix is multiplied by a record in the database and the results added together into a single vector. This vector is then sent back to the client, who filters out the soft noise and retrieves their desired record. I will now describe the algorithm in greater detail:

First, the client generates a query.

input : index i_0 of the record client wishes to retrieve
output: query in the form of ordered list of matrices to be sent to the server

1 $A \leftarrow$ random invertible matrix of size $N \times N$ over $\mathbb{Z}_p$;
2 $B \leftarrow$ random matrix of size $N \times N$ over $\mathbb{Z}_p$;
3 $M \leftarrow [A|B]$;
4 **for** $i \in \{1, \ldots, n\}$ **do**
5 | $P_i \leftarrow$ random invertible matrix;
6 | $M_i'' = [A_i|B_i] \leftarrow P_i \cdot M$;
7 **end**
8 $\Delta \leftarrow$ random $N \times N$ diagonal matrix over $\mathbb{Z}_p$; `// scrambling matrix`
9 **for** $i \in \{1, \ldots, n\} \setminus i_0$ **do**
10 | $D_i \leftarrow N \times N$ random matrix over $-1, 1$; `// soft noise matrix`
11 | $M_i' \leftarrow [A_i|B_i + D_i\Delta]$; `// soft disturbed matrix`
12 **end**
`// generate hard noise matrix`
13 $D_{i_0} \leftarrow$ soft noise matrix with each diagonal term replaced by q;
`// compute hard disturbed matrix`
14 $M_{i_0}' \leftarrow [A_{i_0}|B_{i_0} + D_{i_0}\Delta]$;
15 $\mathcal{P} \leftarrow$ random permutation of columns;
16 **for** $i \in \{1, \ldots, n\}$ **do**
17 | $M_i \leftarrow \mathcal{P}(M_i')$;
18 **end**
19 **return** *ordered set* $\{M_1, \ldots, M_n\}$

Algorithm 1: The algorithm for client query generation

The previously mentioned noise is introduced into the matrices by the addition of noise matrices $(D_1, \ldots, D_n)$ on lines 11 and 14. However, the noise is also scaled by the scrambling matrix Δ to obscure the hard noise. In order to later filter out the noise, the client will need to save the scrambling matrix. Similarly, permutation of columns is intended to make the identification of the hard disturbed matrix harder. The permutation must also be saved by the client, as does M.

Next, a `PIR` server computes its reply. We assume each record is $w_{AG} \times N$ bits long. If they are not, the records are split and/or padded to fit.

input : ordered set of n matrices from the client
output: vector V of size $2N$ over $\mathbb{Z}_p$
1 **for** $m_i \in$ *database records* **do**
2 | split m_i into N w_{AG}-bit integers $\{m_{i1}, \ldots, m_in\}$;
3 **end**
4 **for** $i \in \{1, \ldots, N\}$ **do**
5 | $v_i \leftarrow \sum_{j=1}^{N} m_{ij} M_{ij}$;
| // M_{ij} denotes the j-th row of M_i
6 **end**
7 **return** $\sum_{j=1}^{n} v_i$

Algorithm 2: The algorithm for server reply generation

The reply generation is very straightforward, the properly transposed matrix is multiplied by the appropriate record which is treated as a vector. The resulting vectors are then summed together and sent to the client.

Finally, the client decodes the response.

input : vector V of size $2N$ over $\mathbb{Z}_p$
output: record i_0
1 $V' \leftarrow= \mathcal{P}^{-1}(V)$;
2 ${V_D}' \leftarrow$ disturbed half of V';
3 ${V_U}' \leftarrow$ undisturbed half of V';
4 $E \leftarrow {V_D}' - {V_U}' A^{-1} B$; // scrambled noise
5 $E' \leftarrow E\Delta^{-1}$; // unscrambled noise
6 **for** ${e_j}' \in E'$ **do**
7 | **if** ${e_j}' \mod q < q/2$ **then**
8 | | $\epsilon \leftarrow {e_j}' \mod q$;
9 | **else**
10 | | $\epsilon \leftarrow ({e_j}' \mod q) - q$;
11 | **end**
12 | $e''_j \leftarrow e'_j - \epsilon$;
13 **end**
14 **for** $j \in \{1, \ldots, n\}$ **do**
15 | $m_j \leftarrow e''_j q^{-1}$;
16 **end**
17 **return** $\{m_1, \ldots, m_n\}$

Algorithm 3: The algorithm for server reply decoding

The decoding is performed by first reversing the operations the client has performed and extracting the noise (lines 1–5). Next, the soft noise is filtered out by detecting whether the sum of soft noise was negative (line 7) and appropriately modifying the noise.

As an example, I now present a possible run of the algorithms. All the figures in this example are taken from [4].

The client wishes to retrieve the second record of a database containing two six-bit records. They begin by generating random matrices as per lines 1–8, 10, and 13 of algorithm 1. Next, they generate the disturbed matrices (lines 5, 11, and 14), and finally permute the columns of the matrices as per lines 16–18. The result is sent to the server.

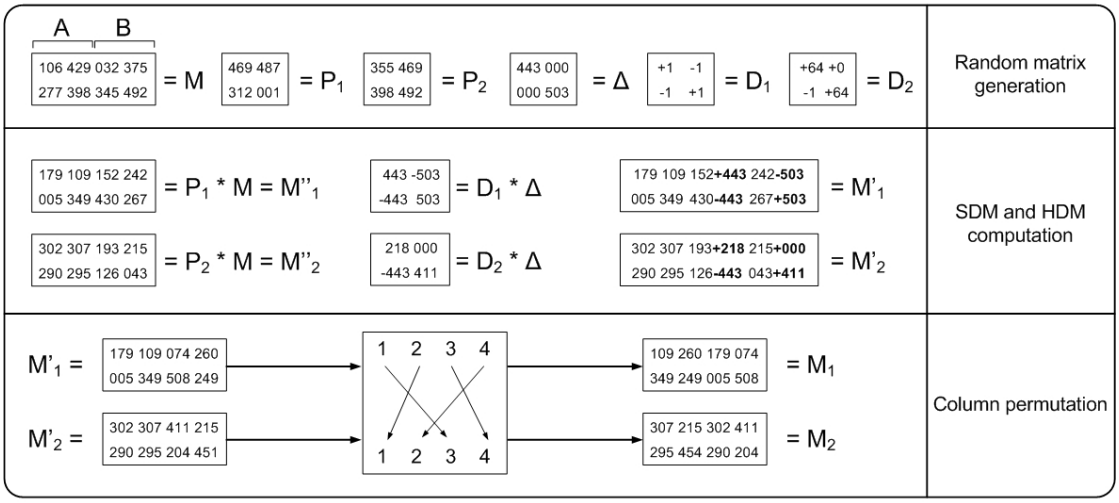

Figure 1.1 Diagram showing an example run of algorithm 1. Source: [4]

Upon receiving the reply, the server generates its reply following algorithm 2 and sends it to the client.

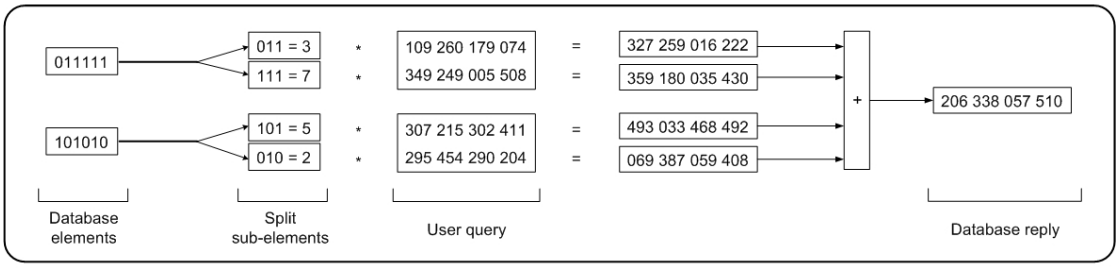

Figure 1.2 Diagram showing an example run of algorithm 2. Source: [4]

Finally, the client decodes the data according to algorithm 3. The following diagram shows the decoding process. Lines relevant to each panel are noted in the top right corner of the panel.

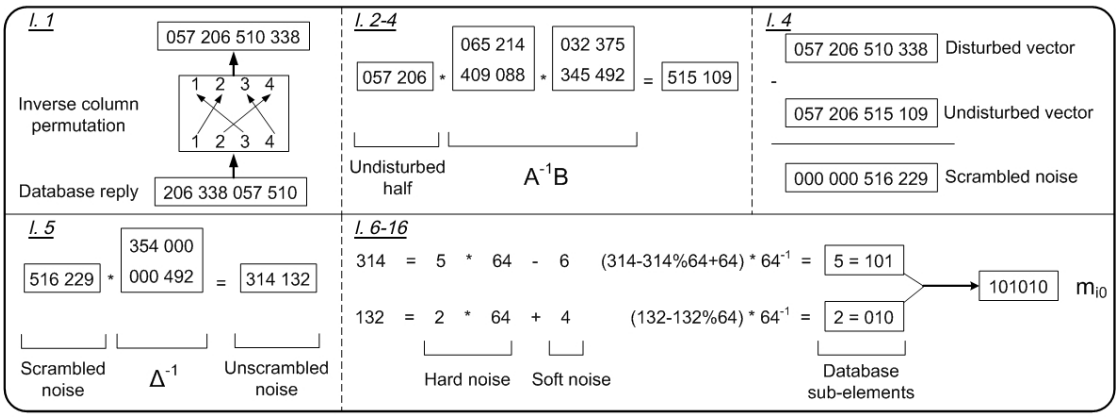

Figure 1.3 Diagram showing an example run of algorithm 3. Source: [4], relevant line numbers added

1.1.1.2 Recursion

An important observation can be made about the properties of the above algorithm. The parameter w_{AG} limits the number of accessible records of the database. Indeed, we could just increase this parameter arbitrarily, but this is inefficient. Instead, Aguilar-Melchor and Gaborit propose a recursive approach [4]. First, choose d to be the depth of recursion. Next, at each level, split the database into $\sqrt[d]{n}$ virtual records, each containing $n/\left(\sqrt[d]{n}\right)^d$ actual records. Let the user calculate which virtual record contains their desired record and also let them query for it. Do not send the reply to that query to the client, instead hold on to it and treat it as the database in the next step, unless the maximum depth has been reached, in which case send the result to the client and let them calculate the result.

A diagram for $d = 3$ can be seen in figure 1.4. First, the client calculates the records location at each level of recursion. Next, the client sends the server a sequence of 3 queries, Q_1, Q_2, Q_3, one for each level of recursion, and stores their respective invert transforms. For each query except the final one, the server holds on to the result and applies the next query to it instead of the entire database. After applying the final query, the server sends the result back to the client. The client then performs the inverse operations for the queries in the reverse order to retrieve their desired record.

1.1.1.3 Security

The scheme relies on the Hidden Lattice Problem (HLP) and the Differential Hidden Lattice Problem (DHLP) to defend against structural and distinguishability attacks respectively. These problems were not very well researched at the time of publication. While Aguilar-Melchor and Gaborit [4] relate HLP to a known NP-complete problem in coding theory [7], they only note that lattice based attacks against DHLP are very unlikely. This hints at a potential vulnerability in the system, and in recent works, Aguilar-Melchor has abandoned this approach in favor of more standard problems (namely RLWE) [8].

Indeed, such a vulnerability has been found. In 2016, Liu and Bi [9] showed that there is a "hidden linear relationship between the public matrices and noisy matrices" and that the server can learn which record the client requested as long as the number of records in the database is small. This means the scheme is effectively broken. This also impacts the security of the hybrid protocol, but that shall be discussed furher on.

1.2 Information theoretic PIR

It has been proven [2] that single server ($k = 1$) `PIR` scheme in which it is *impossible* for the server to learn anything about the client's query would require $\Omega(n)$ bits of communication, therefore making it worse than a trivial transfer of the entire database. Because of this, all information theoretic `PIR` schemes assume $k > 1$. They can also assume that some of the servers might not be responding incorrectly or not responding at all. Another assumption that can be made is that each of the servers only holds a part of the database, and that only a coalition of server can learn its full content. However, these assumptions are beyond the scope of this thesis. Instead, let us only assume t-privacy:

▶ **Definition 1.2** (t-privacy)**.** *A `PIR` scheme is t-private iff of the k servers, up to t can collude and still learn nothing about the contents of the query.*

1.2.1 General idea: Vector multiplication of a matrix

In general, both `ITPIR` schemes follow the same simple idea. We take the database as an $n \times l$ matrix, with each element of the database being w bits long. To retrieve the record at index i_0,

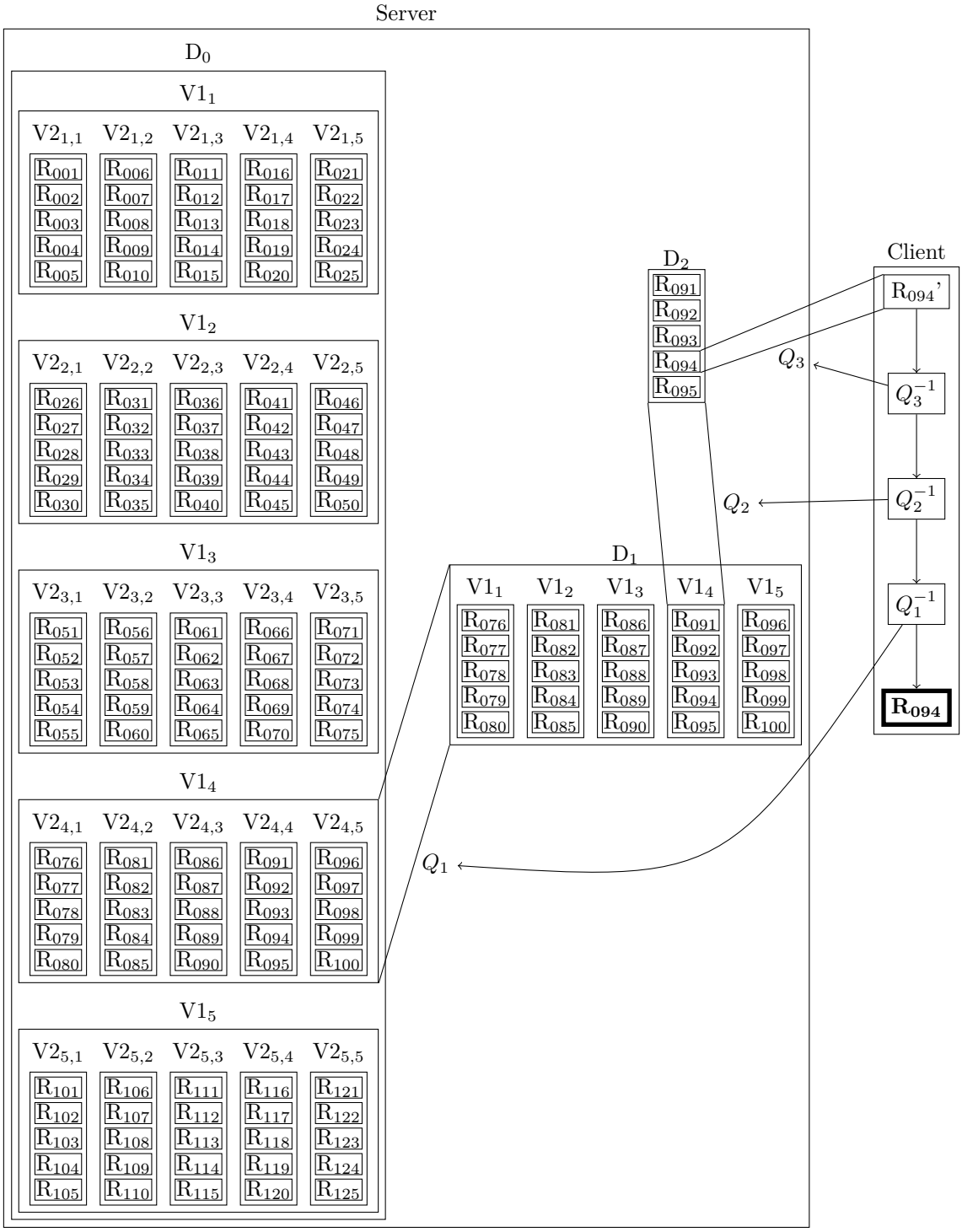

Figure 1.4 Example of recursive retrieval of a single record for $d = 3$

the database needs to be multiplied by a row vector consisting of only zeroes except for a single one in the i_0-th position. If the server was to perform this operation, the client's privacy would be compromised, as the server would know which index the client wished to retrieve.

$$\begin{bmatrix}0 & 0 & \dots & 1 & \dots & 0\end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1l} \\ w_{21} & w_{22} & \dots & w_{2l} \\ \vdots & \vdots & & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nl} \end{bmatrix} = \begin{bmatrix} w_{i_0 1} & w_{i_0 2} & \dots & w_{i_0 l} \end{bmatrix}$$

Therefore, we need to distribute the query in a way that achieves t-privacy.

1.2.2 Chor 1995

Chor et al. [2] propose a simple algorithm. We take $w = 1$, i.e. each element of the database matrix is one bit long. We also assume that each of the $k \geq 2$ servers hold a full copy of the database.

We begin by generating the queries.

input : vector v_0 of length n containing only zeroes and a single one at i_0, the index of the record the client wishes to retrieve
output: a set of k vectors of length n to be sent to servers
for $i \in \{1, \dots, k-1\}$ **do**
  $v_i \leftarrow$ uniformly random n-length vector over $GF(2)$;
end
$v_k \leftarrow v_0 \oplus v_1 \oplus \dots \oplus v_{k-1}$;
return $\{v_1, v_2, \dots, v_k\}$

The client then sends one query to each server. The server then multiplies their database by the vector they received and sends the result back to the client. This result is essentially a XOR of the records whose index happened to hold a one in the query. Once the client has all the responses, they can easily calculate the final result by XORing them together.

We can see that this protocol is $k-1$-private, i.e. all the servers would have to collude to learn anything about the client's query. This is the strongest protection against collusion an `ITPIR` protocol can provide. In terms of communication and computation costs, it also far outperforms the other protocols implemented in `Percy++`. However, the protocol is not robust, meaning that if even just one server fails to respond or responds incorrectly, the client will fail to retrieve the record.

1.2.3 Goldberg 2007

To address the robustness limitations of [2], Goldberg [10] proposed his own algorithm which I will now describe.

1.2.3.1 Building blocks

Before describing Goldberg's algorithm itself, let us define some necessary prerequisites.

1.2.3.1.1 Shamir's secret sharing Shamir's secret sharing ($SSS(k,t)$) is a scheme that allows data D_0 to be shared among k participants in such a way that no coalition of up to t of them can assemble D_0 and any coalition of at least $t+1$ can assemble D_0. It works in the following way: To obtain D_0, participants need to evaluate $f(0)$. To do this, they need to

input : data D_0 as element of finite field $\mathbb{F}$
output: set of tuples to be distributed among participants
for $i \in \{1, \ldots, k\}$ **do**
  $\alpha_i \leftarrow$ random non-zero index from $\mathbb{F}$;
end
$\{a_1, \ldots, a_t\} \leftarrow$ uniformly random elements of $\mathbb{F}$;
$f(x) \leftarrow$ polynomial in the form $D_0 + a_1 x + a_2 x^2 + \cdots + a_t x^t$;
return $\{(\alpha_1, f(\alpha_1)), \ldots, (\alpha_k, f(\alpha_k))\}$

reconstruct the polynomial using Lagrange interpolation, which is only possible if at least $t+1$ of them share their tuples. Otherwise, they cannot learn anything about D_0. We can, however, see that should some of the participants provide an incorrect tuples, reconstruction will fail. To correct for this, error correcting codes are used.

1.2.3.1.2 Reed-Solomon codes To recover from missing or incorrect responses from the servers, Goldberg's protocol treats the responses from servers as a list of strings to be decoded using Reed-Solomon decoding. For a set of $R_1, \ldots, R_k$ responses from servers and their respective indices $\alpha_1, \ldots, \alpha_k$, this problem turns into finding a polynomial that passes through at least t pairs of α_i and R_i. There are several efficient algorithms that solve this problem, either finding a unique solution [11], or a list of possible solutions [12].

The original paper uses the algorithm from [12]. In 2012, Devet et al [13] improved the scheme to also include the algorithm from [11] and their own algorithm utilizing simultaneous decoding of multiple polynomials at the same time. A brute-force algorithm was also included, as well as an algorithm to select the most efficient of those to apply to a given set of responses.

1.2.3.2 Algorithm

Having defined the necessary prerequisites, let us now look at the algorithm itself. Once again, the idea is to allow the server to multiply the database by a vector, and we also assume each of the $k \geq 2$ servers hold the full copy of the database. However, unlike in Chor's case where the field of operation was $GF(2)$, we will now be working in a larger field $\mathbb{F}$. `Percy++` allows the field to be one of $GF\left(2^8\right)$, $GF\left(2^{16}\right)$, or $\mathbb{Z}_p$. Also, the elements of the database matrix shall be larger, $w = \log |\mathbb{F}|$. Each row of the matrix will therefore contain $s = l/w$ elements.

The client begins by generating s random Shamir polynomials. The data to be shared via these polynomials (i.e. the constant term) shall be 0 for the indices the client is not interested in, and 1 for the one they are. The client also generates k Shamir indices $\alpha_1, \ldots, \alpha_n$ and evaluates the polynomials at them yielding a set of s values to be sent to each server. These are then sent to them.

Each server, receiving a set of n values from the client ($\{\delta_1, \ldots, \delta_n\}$), evaluates the following for each column of the database. d_{ij} denotes the j^{th} item of the i^{th} row.

$$R_j = \sum_{i=1}^{n} \delta_j d_{ij}$$

The resulting set $\{R_1, \ldots, R_s\}$ is sent back to the client.

Finally, the client has to decode the responses. The following properties hold for any two Shamir polynomials f, g and any constant $c \in \mathbb{F}$:

$$f(0) = a,\ g(0) = b \implies (f+g)(0) = a + b$$
$$f(0) = a \implies (c \cdot f)(0) = c \cdot a$$

For each column of the database, the client receives from each server a point on a curve of a polynomial that is defined as a sum of $n-1$ polynomials whose constant term is 0 and one whose constant term is the desired value for a given column. Therefore, client has to simply reconstruct the polynomial, using Lagrange interpolation if there are no misbehaving servers, or resorting to some form of error correction if there are, and evaluate it at zero. Once the client repeats this process for each column of the database, they will recover the full result of their query.

This scheme relies on Shamir's secret sharing for a guarantee of privacy and robustness against non-responding servers. The client only needs t Shamir responses to reconstruct the result to their query. However, the servers also only need t shares to learn the contents of the client's query. The client can, however, choose t to be equal to k, thus making the protocol $k-1$-private. This makes the protocol reach the same level of privacy as that of Chor's algorithm described in section 1.2.2, but eliminates the robustness property.

In terms of communication costs, this scheme performs slightly worse than Chor's, as it needs to send an element of $\mathbb{F}$ instead of a single bit per record in the database in the client's query. The response is the same length unless the record needs to be padded to a multiple of w.

Performance is also slightly worse, as the operations performed in $\mathbb{F}$ are more computationally expensive than a simple XOR.

1.3 Hybrid PIR

We have observed that both `CPIR` and `ITPIR` their issues. While `CPIR` only requires a single server, it does not offer perfect privacy in the information theoretic sense. It is also generally slower. On the other hand, `ITPIR` schemes fail as soon as more than t servers collude. There is also no way for recursion to be applied.

Realizing these shortcomings of these approaches, one might wish to somehow mitigate them by combining both into a hybrid scheme. While [10] already proposed such a scheme, its performance was 3-4 orders of magnitude lower than that of plain `ITPIR`. The scheme relied on Pailler cryptosystem. At the time, this was a reasonable choice, because the scheme did require the `CPIR` component to be additively homomorphic and thus enable the use of recursion. However, the computational cost of the Pailler cryptosystem made this scheme unlikely to be useful.

1.3.1 Devet & Goldberg 2014

In 2014, Devet and Goldberg [14] proposed an improved hybrid scheme. It is very similar to the previous scheme by Goldberg in its structure, but instead of using the Pailler cryptosystem, it relies on Aguilar-Melchor and Gaborit's [4] scheme. However, it does not require it specifically, any recursive `CPIR` scheme will suffice. Similarly, while it does use the `ITPIR` scheme of [10], it does not require this scheme specifically, and any other multi-server `ITPIR` scheme would suffice.

1.3.1.1 Algorithm

Let us now denote Ψ the `ITPIR` component of the hybrid protocol and Φ the `CPIR` component.

The hybrid scheme works in the following way: First, similarly to 1.1.1.2, a recursion depth d is chosen in a way that guarantees communication cost at worst equal to that of Ψ[1] and the database is split into virtual blocks. The client calculates the first level virtual block that holds their desired record and queries the servers for it using Ψ. The servers do not send the reply and instead hold on to it. Next, the client recursively queries each of the servers using Φ, until the maximum recursion depth is reached. At this point, the servers send back their replies. The

[1] The worst case is reached if there is no communication cost benefit to using recursion and `CPIR`. If this is the case, the hybrid scheme devolves into Ψ.

client then applies the decoding algorithm of Φ to each of the replies and finally, using Ψ, recovers the desired record from the decrypted replies.

1.3.1.2 Security

Due to its highly modular design, the hybrid scheme's security relies on the security of underlying protocols. Let us now explore some possible failure states for this scheme.

1.3.1.2.1 `CPIR` broken This is the state that the `Percy++` implementation is in due to the fact that Aguilar-Melchor and Gaborit's [4] scheme has been broken by [9]. This means that each server can, with relative ease, learn the position of the record the client was interested in withing the first level virtual block. This impacts the client's privacy severely, as instead of knowing nothing about the client's query, each server can now determine that it was for one of $\sqrt[d]{n}$ records. However, some level of privacy is still retained due to the non-collusion assumption.

1.3.1.2.2 Server collusion If at least $t+1$ servers collude, but the computational hardness of the `CPIR` protocol still holds, the servers can learn which first level virtual block the client wanted to retrieve. This has an equal or lower impact on client's privacy than `CPIR` compromise, as it only reduces the number of possible records the client might have been interested in to $(\sqrt[d]{n})^{d-1}$. However, this attack by the servers might be more likely, due to the fact that it requires no protocol breakage and instead only needs cooperation from the servers.

1.3.1.2.3 Full compromise If both `CPIR` breakage and server collusion co-occur, client's private query is revealed. We can see that the hybrid protocol's defence in depth approach yields additional security for the client compared with situations where only one of the protocols is used at an insignificant computational cost.

1.4 Performance analysis

Having established the major families of `PIR` protocols and some members of these families, let us now analyze their performance. Our investigations will lead two along axes: communication and computation.

1.4.1 Communication costs

The cost of communication of a `PIR` protocol is made up of two components: the cost of the query and the cost of the reply. The costs of retrieving a single record for all the previously established protocols are given in a table below. I assume Goldberg's scheme is being used as Ψ and Aguilar-Melchor and Gaborit' recursive scheme as Φ for the hybrid protocol.

	Chor 1995	Goldberg 2007	Devet & Goldberg 2014	A&G 2007	A&G 2007 rec
Query	kn	knw	$kd\left(6N^2w_{AG}\right)^{\frac{d-1}{d}}\ \sqrt[d]{w}\sqrt[d]{n}$	$\left(6N^2w_{AG}\right)n$	$d\left(6N^2w_{AG}\right)\sqrt[d]{n}$
Reply	kl	kl	$k6^{d-1}l$	$6l$	6^dl

Table 1.1 Communication costs of different protocols

As we can see, all the replies are linear in l and k (A&G can only use one server), meaning that as the number of servers and the length of the records increases, so does the cost of the reply. We can also observe that the cost of the query is linear in n for all protocols that do not use recursion and linear in k for all servers (again, $k = 1$ for A&G). We also note that the protocols that do use recursion will perform better on databases that contain a large number

of small records. The depth of the recursion is algorithmically chosen to be optimal, i.e. to minimize communication.

1.4.2 Computational cost

The cost of computation for a single query is hard to estimate, but we can generally estimate that `ITPIR` protocols will be faster than `CPIR`, as the `ITPIR`'s server side computation only involves a single multiplication of a matrix by a vector whereas `CPIR` needs to perform n such multiplications (albeit on smaller matrices). The hybrid protocol will likely perform better than its components, but will perhaps underperform when recursion is not used due to the overhead of estimating the optimal depth.

Chapter 2
Use cases

Having described the different types of `PIR` available in `Percy++`, I will now present some general use cases for `PIR`. For the rest of the chapter, I will use O_0 to denote the original source of data. Its copies will likewise be labeled as such in diagrams, and its partial copies shall be labeled as O_i `PIR` servers shall be denoted by S_i, and `PIR` traffic will be marked by dashed arrows. Any other traffic will be shown with solid line arrows. Clients will be denoted by C_i.

2.1 Privacy conscious database operator

This use case is similar to the one I presented in the introduction. It only has a single source of data O_0 and a single `PIR` server S_1, possibly running on the same machine. There can be multiple clients accessing S_1. As per the business model discussed in the introduction, the operator would cover the extra costs arising from using `PIR` by charging the clients for using the service, and possibly further supplement this by advertising.

As there is only one server in this case, `CPIR` would need to be used. This might incur heavy computational costs for S_1 as the number of queries grows. To mitigate this, S_1 could aggressively rate limit the clients. An alternative solution would be to distribute the database across multiple servers. This will be discussed in a following section.

This setup assumes it is the client's responsibility to take appropriate measures to protect their query en route to the server. Indeed, end-to-end encryption will be supported by the server, but a user wishing to also conceal the ends of the communication from a curious network operator will have to take additional measures such as using an onion network.

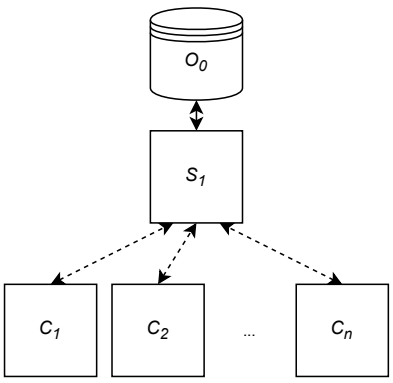

Figure 2.1 Diagram of a single server scheme described above

A real-world example of this setup could be a private DNS iterative resolution where we view each iteration as a resource retrieval that can be made private with **PIR**. While even today solutions for private transport of a DNS query exist that do not severely impact performance (e.g. DNS-over-HTTPS) [15], the privacy of the contents of the query is rarely considered. Indeed, the current infrastructure of DNS servers does not lend itself too well to trivial transfer of the entire DNS database, because it is large, decentralized and quite volatile. However, by allowing the client to perform each iterative step of the retrieval using **PIR** with keyword search based on [1], we can achieve data transfer costs lower than that of trivial transfer.

Indeed, in this case, changes to how a DNS query is handled would need to be made, both on the server and the client side, slowing the entire process down and making it more expensive. Recursive resolvers that stand between the client and the domain name servers would need to be dropped, along with their ability to cache results for multiple clients, slowing the operation down further[1]. However, benefits for the client would be tremendous, essentially going from possibly being tracked online all the time to tracking being impossible. Whether the costs would be acceptable to the client and the server remains to be seen.

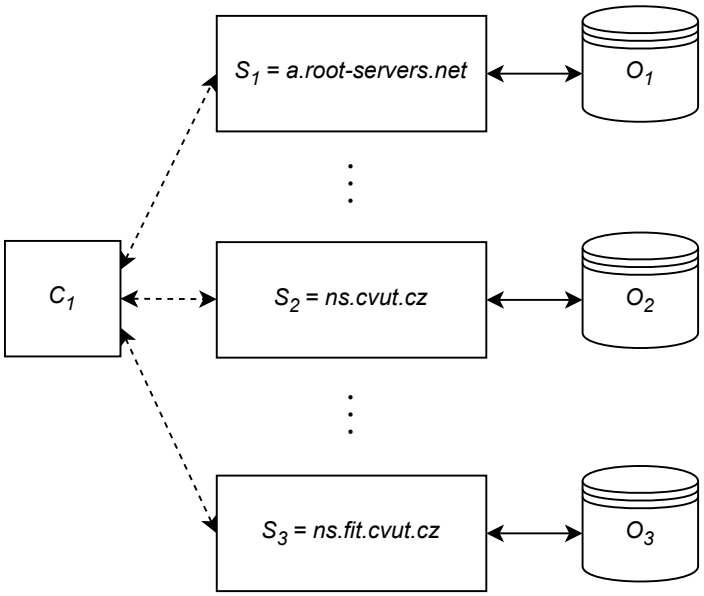

Figure 2.2 Diagram of client iteratively retrieving the IP address for progtest.fit.cvut.cz (not all servers contacted shown)

2.1.1 Variant: Untrusted cloud

A variation on this scheme is the situation where the entity controlling O_0, S_1 and clients is the same and the hardware that O_0 and S_1 are running on is not directly controlled by this entity or is in general not trusted. The entity wishes to hide the data in O_0 from the hardware operator. This is achieved by encrypting the data at rest and only decrypting it once it reaches the client. The entity also wishes to conceal the access patterns, e.g. to make it harder for the hardware operator to extract important data by focusing their attack on blocks of data that are accessed more often. To do this, the entity employs a **PIR** scheme, essentially masking out any meaningful information that could be extracted by the hardware operator, except for the time of access and the identity of the client (unless the identity is also masked, in which case only the time of retrieval is learned).

[1]For an approach utilizing multi-server recursive DNS resolution see section 2.3.1

A real-world example of this use case could be a company handling large amounts of extremely sensitive data (e.g. medical or banking records) lacking on-premise computatons power or perhaps needing extra computation power to handle all the data. In these cases, cloud computing would have to be used. As long as the access to the data is not critically time-sensitive and does not happen often, using `PIR` and encryption at rest would be a viable option for concealing the data and the access patterns (as I will show in the following chapter).

Like the previous use case, this one has only a single server and therefore needs to use `CPIR`. While there could be multiple servers set up for `ITPIR` or hybrid scheme, it is unlikely that the additional privacy gained would in fact outweigh the cost. A case could be made for a distributed scheme if performance was critical, as an `ITPIR` or hybrid scheme would be faster and use less bandwidth.

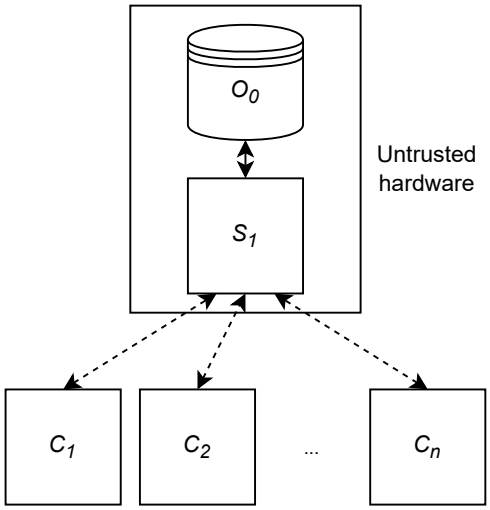

Figure 2.3 Diagram of single server scheme with untrusted hardware

2.1.2 Variant: Law enforcement

Another variation one might consider is a situation in which a database operator stores personal information on subjects in the database. We could, for example, imagine that the database contains records of people accommodated in a certain hotel. It is clear that such information might be valuable to law enforcement if they were looking for a suspect who might have stayed in that hotel. Indeed, law enforcement could simply copy the entire database and then search it themselves, but the database might be large or contain records for whose retrieval the law enforcement agency has no right. We can see that even a trivial transfer in this case might be problematic.

Why not let law enforcement query the database directly then? This would eliminate the need for ad-hoc solutions via trivial transfer. However, it would also disclose very sensitive information to the database operator. In our example, the database operator would learn that the law enforcement agency is looking for the suspect. This might thwart the agency's investigation, as well as infringe upon the privacy of the suspect. This means that allowing the agency to directly query the database will also not work.

A PIR enabled database partially solves this issue. PIR guarantees that the operator learns nothing about the contents of the query, and therefore ensures that law enforcement can privately retrieve the data they desire. However, conventional PIR protocols do not guarantee that the client will not learn anything besides what they asked for. This could lead to the law enforcement agency learning some information it was not authorized to learn.

To adress this issue, several Symetrical PIR (SPIR) protocols have been developed [16, 17]. These protocols, besides having the guarantees of standard PIR protocols, also guarantee that the client learns nothing about the database except for the record they are retrieving. They do, however, incur computational and communication costs.

2.2 Distributed PIR network

In the previous example, we have seen that the operator of O_0 can have a legitimate interest in allowing the client to perform their query privately. However, the computational costs of running a single server CPIR could deter the operator. Simply putting up multiple servers is also not enough as it completely removes the privacy guarantees of ITPIR (the operator would definitely collude with themself). Therefore, it is necessary to involve parties other then the clients.

Let us imagine a situation where we have n independent PIR servers labeled $S_1, \ldots, S_n$, each operated by a different and independent entity. These servers need access to the data at O_0. They will first need to perform a trivial transfer of the entire database when first joining the network, and then be updated when the data changes. This update can be initiated by O_0 simply telling the servers there has been a change, the server itself periodically checking if the database has changed, or even the client detecting outdated response from the server thanks to the properties of robust ITPIR (although this approach could introduce some data staleness issues if a majority of servers has outdated data).

This setup introduces benefits for the operator of O_0 in the form of reduced traffic (assuming new servers join infrequently and updates to the database are not both large and frequent) and computational cost (simply serving the data without running PIR on them is surely cheaper than doing both). Clients also benefit, gaining reduced latency and access to the benefits of ITPIR. Additionally, it gives them the option of using whichever PIR protocol they happen to like the best, as long as the servers support it. PIR server operators can monetize their service and thus cover their costs and potentially turn a profit.

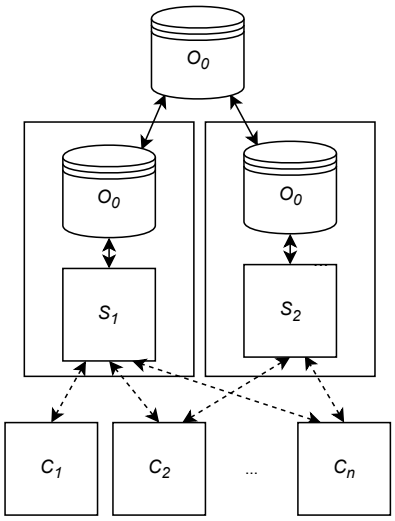

Figure 2.4 Diagram of distributed PIR network

2.2.1 Variant: Partially distributed database

The database from O_0 does not need to be fully replicated at each of the n servers. Servers can choose to only copy the part of it which they are interested in. While this may impact the users' ability to retrieve some records conveniently, it reduces the storage requirements for the servers.

Furthermore, the properties of some ITPIR protocols allow for the database to be distributed in such a way that no coalition of servers (up to a certain size) can learn the full contents of the database. While this would incur some overhead on O_0 and limit the clients' choice to ITPIR or hybrid schemes, it guarantees at least some privacy to O_0.

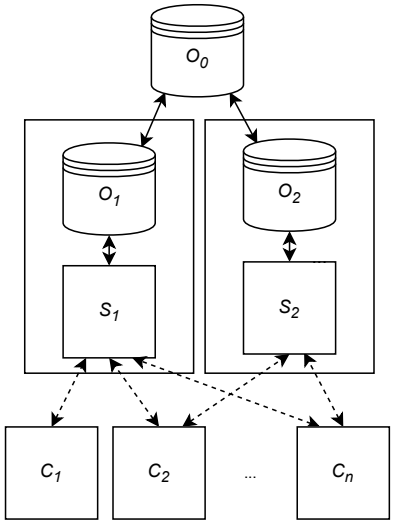

Figure 2.5 Diagram of distributed PIR network with partially distributed database

2.3 Adversarial database access, quasi-anonymizers

In the previous examples, we have assumed that O_0 is cooperative and willing to provide the entire database in any form. In the real world scenarios, this is rarely the case, as most operators choose to build their business models around free access with tracking and targeted advertisement. Users wishing to avoid tracking and profiling will often have to use an anonymizer of sorts. By anonymizer, I mean a service that takes a user's query and relays it to O_0, removing or replacing any sensitive parts (i.e. the IP address) and then relaying O_0's response back to the client.

An anonymizer achieves improved privacy for the user as long as two assumptions hold. Firstly, the anonymizer must be more trustworthy than O_0 itself. Were this not the case, the clients would still be subject to tracking, just by someone else. Secondly, there needs to be enough clients using the anonymizer. For example, if there was just a single client accessing O_0 through the anonymizer, O_0 could easily identify and track this user, even though their traffic would be coming from a different IP address. The anonymizer would be acting as a simple proxy in this case. Let now k denote the number of clients using the anonymizer. As k gets higher, it becomes harder and harder for O_0 to distinguish between the k users. Some information leakage is still present, namely the time of the access and the resource accessed. These will, however, always be present, as we have established O_0 as averse to allowing the users to access its resources privately. Therefore, using a trustworthy anonymizer, k users can become indistinguishable in the eyes of O_0, that is achieve what is in the literature known as k-anonymity [18].

Improvements to privacy are not the only reason for using an anonymizer. Assuming clients mostly retrieve similar resources, the anonymizer can use caching to improve latency. It can also

further enhance the user experience of the clients by only serving the content they are interested, e.g. serving a user who wishes to watch a video just the video file instead of a website laden with tracking scripts etc. This also decreases the amount of information O_0 has access to, since not all queries actually reach it.

Implementing a fully PIR-enabled solution for the anonymizer is very hard, probably impossible. Scraping the entire database will not be possible due to rate limiting (and the sheer size of it), fetching only the records clients request will also not work as that would require revealing the query to the server, preemptively caching some of the content would essentially be guesswork and likely in vain. Let us instead investigate a partially PIR enabled anonymizer.

As long as many clients do indeed wish to retrieve the same resources, the amount of information an anonymizer learns can be minimized with usage of PIR and caching. A client can learn whether a given resource is in the anonymizer's cache privately, using either trivial or non-trivial PIR. If it is, they can also retrieve it privately. If it is not, they have a choice of either not retriving the resource or revealing which resource they wanted to access so that the anonymizer can access it for them and cache it. Any subsequent queries for the resource will be private. This can be seen in figure 2.6, where C_1 retrieves an item without concealing which item it retrieved, and subsequent retrievals of the same item by other clients are private.

This approach minimizes the need to trust the anonymizer while also maintaining k-anonymity for the users.

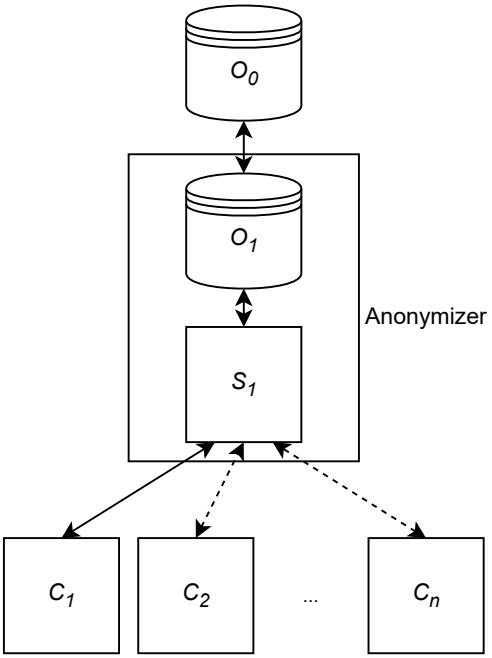

Figure 2.6 Diagram of partially PIR enabled anonymizer

It is unclear whether this scheme can be used in practice. Anonymizers typically do not generate any revenue from serving ads or collecting personal information, and are instead maintained by volunteers supported by donations. Multiple instances of anonymizers for popular social media are hosted publicly [19, 20, 21], which means that the general idea of an anonymizer as a service has some economic merit to it. Whether it would be economically possible even with the added costs of using PIR remains to be seen.

Likewise, it is uncertain whether a multi-server version of the partially PIR-enabled anonymizer would be feasible. Indeed, in order for a multi-server retrieval to succeed, a sufficient number of servers would need to have cached the content the client is requesting. If this is not the case,

the client will have to reveal their query to at least one of the servers and retrieve the resource trivially. The server must then inform the other servers, so that they can cache the resource. This way, cache size grows on all the servers.

2.3.1 Range based approaches to anonymizers

Much like the anonymizer which masks user's queries by mixing them with those of other users, a user can mask their own query by also querying for records they are not interested in. While this does reveal some information to whoever the client is querying, it can quite naturally extend the previously presented singler-server partially `PIR`-enabled anonymizer. By having the client query for multiple records, of which only some are of interest to them, privacy of the whole system is improved in two ways: the client making the queries is better protected against tracking by the anonymizer, and all other clients benefit from the increased entropy of queries made by the anonymizer against O_0. It should be noted that this is going to result in larger cache size for the anonymizer.

The idea of querying for resources the client is not interested in has an even bigger impact in a multi-server setup. For example, Zhao's [22] two-server scheme for private DNS queries [2] is quite interesting in this regard. It is similar to Chor's scheme described in section 1.2.2 in that it utilizes the properties of the XOR function. In this scheme, assuming the client wishes to retrieve an `A` record for a given domain name, the client first generates a set of random domain names. They send this set to one of the servers, and to the other, they send the same set with the domain name they are interested in appended. Each of the servers then resolves each of the domain names they received, and XORs the resulting IP addresses together. Thus obtained results are then sent back to the client. The client once again XORs the two results together to obtain the final result.

Unless the two servers collude, they each only learn that with a 50 % certainty, the requested domain name is one of the ones they received. This is of course far from the perfect privacy guaranteed by `ITPIR`, but it might be enough for some clients. The scheme is also very fast, given that only a handful of XORs are needed at each of the servers and the client, and, with some slight modifications, might even be suitable for privately retrieving multimedia. It is, however, susceptible to correlation attacks by O_0. It is also not robust against misbehaving servers. Whether this could be improved by employing an approach similar to Goldberg's [10] is a question for further research.

[2] Contrast with single server iterative solution suggested in section 2.1

Chapter 3
Benchmarks

In this chapter, I will describe the design and results of benchmarks I performed against `Percy++`. I will also describe the dataset I chose to perform these benchmarks against. For convenience, I will refer to different protocols implemented in `Percy++` as modes from now on. To further save space, I will assign each mode an abbreviation, which is also used internally in `Percy++`:

- **`AG` and `R_AG`** for Aguilar-Melchor's algorithm described in section 1.1.1 and its recursive variant respectively,
- **`GF28`, `GF216`, `ZZ_P`** for variations of Goldberg's scheme in section 1.2.3,
- **`CHOR`** for Chor's scheme described in section 1.2.2,
- **`HYB`** for Devet & Goldberg's hybrid scheme which I describe in section 1.3.1.

3.1 Benchmark design

Taking inspiration from [23], I have decided to design a suite of tests where an application built with `Percy++` is acting as a proxy between a client and an SQL database. This is similar to the use-cases described in sections 2.2, with the caveat that the machines running the `PIR` servers don't actually contain the full database, but instead pass queries to a central SQL database server. This prevents a failure state where the copies of the database on the `PIR` server machines are inconsistent with one another. In many ways, this is also similar to the scenario described in section 2.3, but we assume the database to be either cooperating or under the application's direct control.

3.1.1 Dataset

I chose Czech State Institute for Drug Control's Database of Medical Products [24] as a basis for the SQL database over which the queries shall be performed. I believe there is value to examining the possibilities of private information retrieval against it, because I consider personal medical information to be especially sensitive and potentially extremely harmful to the individual if seized by an adversary. Information about medication an individual is using is definitely such information. Should an individual wish to learn something about a medication they are taking, e.g. the active ingredients, they would need to consult a database similar to the one made available at [24]. Without using `PIR`, they will inevitably reveal some information about the medication to the database operator.

Another reason for using this database was the need to mimic a real world scenario. While generating a database full of random data is certainly an option, it does not convey the expressiveness of private SQL queries very well.

However, there are some downsides to using this dataset. The most obvious one is the uneven distribution of lengths of data in some fields. For example, as one can see in figure 3.1, the column `nazev` containing the name of the medical product includes items that are mostly shorter than 25 characters, with a few outliers going as high as 70 characters. This is unfortunate because all the shorter values will need to be padded to the length of the longest one and as a result a large part of the transported data will be padding. To mitigate this, the client can employ methods similar to those described in section 3.3.3.3.

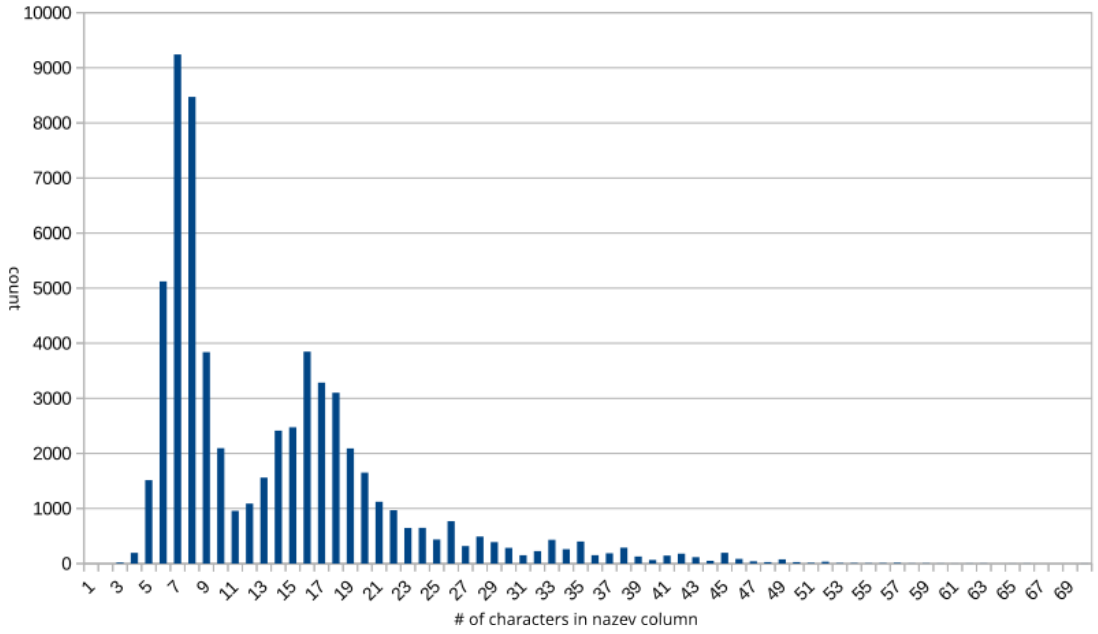

Figure 3.1 Histogram of lengths of the nazev field

The schema for this database can be retrieved at the Institute's website[1]. I will be using it in query examples in the following sections.

3.1.2 Hiding private constants

The proxying application is going to be masking out sensitive constants in an SQL query. As an example, suppose we want to perform the following query

```
SELECT kod_sukl, nazev, sila, forma
FROM dlp_lecivepripravky
WHERE nazev='ATORIS' AND (sila = '40MG' OR sila = '20MG');
```

Code listing 3.1 Original query containing the private constant in the nazev column

but hide the value of the `nazev` column (revealing the information about the `sila` column is not of any concern to us). Trivially, we can see that we could modify this query to

```
SELECT kod_sukl, nazev, sila, forma
FROM dlp_lecivepripravky
```

[1] https://opendata.sukl.cz/?q=katalog/databaze-lecivych-pripravku-dlp

```
    WHERE sila = '40MG' OR sila = '20MG';
```

Code listing 3.2 Modified query with the private constant in the nazev column removed

and then perform the search for rows with correct values ourselves. This is what we could consider a trivial `PIR`. However, the result of this query might be too large to transfer over the network or for our device to handle. Therefore we need to deploy a more sophisticated `PIR` protocol.

We will utilize the results of the modified query from 3.2. However, instead of sending it to the client directly, we will treat it as a database to run a `PIR` server against, with each row of the result acting as one record in this database. Merely running a `PIR`server will not be enough, because the client does not know what the result contains and therefore also does not know the index of the record they want to retrieve. Furthermore, the client does not know how many records match their private constraints (e.g. there can be more than one row matching the `WHERE nazev = 'ATORIS'` constraint). Let us now explore some ways of dealing with these issues.

3.1.2.1 Point queries

Let us first examine the case where the concatenation of columns containing the private constants is unique for each query. In our example, this would mean that there is exactly one row that contains `ATORIS` in its `nazev` column. To allow the client to translate their private constant to a record number to be retrieved via `PIR`, the server could send the client an ordered set of concatenations of columns containing the private constants and allow the client to find the number of the record they are interested in themselves. However, this set could be quite large, and its transfer problematic. Because of this, we will employ a Minimal Perfect Hash Function.

Minimal Perfect Hash Function (mPHF) is defined in [25] as:

▶ **Definition 3.1** (mPHF)**.** *Given a set n distinct elements of set $X = \{x_1, \ldots, x_n\}$, a function f is a Minimal Perfect Hash Function iff*

- $\forall i \in \{1, \ldots, n\} : f(x_i) \in \{1, \ldots, n\}$ *and*
- $\forall a, b \in \{1, \ldots, n\} : f(x_a) = f(x_b) \Leftrightarrow x_a = x_b$.

Because mPHF uniquely maps each element of X onto a set of n consecutive integers, it is very well suited for our needs. The server needs only to find such a function, sort the result of the modified query according to its evaluation and send its parameters to the client who then evaluates it on their private constant. This yields the index of record for the client to retrieve via `PIR`. This evaluation can be performed multiple times, allowing the client to retrieve multiple records without the need to wait for a newly generated mPHF each time.

One might wonder if the mPHF parameter could exceed the size of the set of private constant containing columns. The `emphf` library based on [25]

guarantees the size of parameters will not exceed $2.61 \cdot n$ bits in all but the smallest of cases. This is definitely smaller than what the size of the data would have been by itself and therefore using mPHF will be preferred in all cases.

It should also be noted that the client will need to verify they received the correct data. Indeed, even if the record the client is seeking is not be present in the result of the modified query 3.2, the mPHF constructed over this result will still return a record index. Upon querying for this index, the client might find out that they have received data different from the data they were originally looking for.

3.1.2.2 Generic queries

It is not always the case that the columns containing the private constants will concatenate to unique fields. In such a case, a mPHF cannot be used. As was established previously,

sending the columns in their entirety will also not be possible due to communication cost or computational limits. Therefore, to successfully retrieve a record, the server will need to construct some sort of structure over the database and allow the client to privately query over it. The most straightforward way to achieve this is simply ordering the database and letting the client perform binary search on it.

While this increases the communication cost of retrieval of a single record from a single query to $\log n$ queries, it also allows us to perform much more expressive queries. For example, suppose the client wants to perform the following query while hiding the date range they are interested in:

```
SELECT kod_sukl, nazev, v_platdo
FROM dlp_lecivepripravky
WHERE v_platdo >= '2030-01-01' AND v_platdo < '2031-01-01'
```

Code listing 3.3 Date range query

Clearly, if the server simply retrieves the necessary columns from the database and orders the result by `v_platdo`, the client can then very easily retrieve the desired records utilizing binary search. Note that similar approach can be applied to `LIKE` clause operating on strings. Similarly, we can also address private constants in multiple columns by performing the search operation against the concatenation of them.

It should be noted that the communication cost of this method will be higher in all cases where the following holds (assuming n as the lower bound for communication per query): $n \log(n) > 3.61 \cdot n$, i.e. where $n > 13$. We can safely assume that in the cases where this would come into play (cases with unique keys for at most twelve records) will be sufficiently small and the difference unimportant.

3.1.2.3 Complex queries, joins

While it is true that the majority of query types used by the average user can be reduced to a `SELECT` statement with a single `WHERE` or `HAVING` clause, this is not always desirable. For example, a user trying to find out whether or not a given chemical is present in a medical drug they are taking from the database described in section 3.1.1, they would need to perform a query that would look something like this

```
SELECT *
FROM dlp_lecivepripravky AS p
NATURAL JOIN dlp_slozeni AS sl
     JOIN dlp_synonyma AS sy ON sl.kod_latky = sy.kod_latky
WHERE sy.nazev = 'BENZOPEROXIDUM' AND p.nazev = 'ALGIFEN'
```

Code listing 3.4 Multijoin query

While there is technically nothing wrong with this query, once the `WHERE` clause containing the private constants is removed, the result of the query is over 20 gigabytes.

```
SELECT *
FROM dlp_lecivepripravky AS p
NATURAL JOIN dlp_slozeni AS sl
     JOIN dlp_synonyma AS sy on sl.kod_latky = sy.kod_latky;
```

Code listing 3.5 Multijoin query with private constants removed

This is certainly too large for a simple transfer, but it is also too large to run `PIR` upon. Therefore, the client must perform each of the two subqueries from the original query themself and then join the results accordingly.

```
SELECT *
FROM dlp_lecivepripravky AS p
NATURAL JOIN dlp_slozeni AS sl;
% privately retrieve records where p.nazev = 'ALGIFEN'

SELECT *
FROM dlp_synonyma AS sy;
% privately retrieve records where sy.nazev = 'BENZOPEROXIDUM'

% join the results
```

Code listing 3.6 Split multijoin query with private constants removed

While this requires certain computational capabilities from the client, it allows for a richer set of possible queries.

3.2 Implementation

I implemented the application described above in Python 3.10. The application has a client side and a server side. The protocol between the two sides is the following:

- The client sends their requested `PIR` mode, the value t for their desired t-privacy, their SQL query with the private constants removed and the index of the column that contains the private constant to the server.
- The server performs the sanitized SQL query. It then examines the column the client chose as the one that will contain the private constant. It informs the client that the column either does or does not contain only unique data. It also informs the client about the maximum length in each column and the total length of the original data.
- Based on this information, the client then calculates the total transfer cost of the `PIR` operation, taking into account the padding necessary and the either the cost of transferring the mPHF parameters or performing $\log n$ queries instead of just one. If the client concludes that the trivial transfer would be less costly, they perform it. Otherwise, they carry on with regular `PIR`.
- At this stage, the server either performs the trivial transfer and terminates, or, if the client requested it, continues with the `PIR` portion of the protocol. If the data in client's chosen column is unique, it generates an mPHF[2] for the values in the column and sorts the result according to this mPHF. Otherwise, it simply orders the result according to the client's chosen column. Finally, it pads the the ordered result so that the records are uniform and then writes them into a file.
- The server then runs a number of instances of appropriate `Percy++ PIR` server binaries that will allow the client to privately retrieve information contained in the file, passing the necessary options selected by the client to them. It then sends the parameters of the `PIR` servers and, if it was generated, the parameters of the mPHFto the client.
- The client then invokes their own external `Percy++` binary and begins to retrieve the records they are interested in using `PIR`. If they have received the mPHF parameters, they need only to evaluate it against their private constants and retrieve the resulting indices as described in section 3.1.2.1. Otherwise, they perform binary search for their desired records, as described in section 3.1.2.2.

[2]The mPHF does not need to be generated every time, but can instead be cached. This reduces the overall time in a situation where multiple queries are made, either by the same client, or different clients.

- After the client finishes the retrieval of their desired records, they inform the server. The server shuts down the instances of `PIR` server it has started and deletes the files it has created.

The implementation aims to mimic the options available to the client in a distributed environment described in 2.2, as it allows for the testing of `CPIR`, `ITPIR` and hybrid protocols. While in a true multi-server environment where each server has their own copy of the database, the client would need to operate with a different set of parameters for each server as each server would also need to generate their own mPHF, I chose not to feature this in my experiments. Instead, I chose to present a setup that minimizes transfers between `PIR` servers and clients as it would likely be costly (e.g. routed through an onion network) by generating the mPHF only once at the database server and then distributing it to the `PIR` servers. This is simulated in my application by first performing the preprocessing centrally and then invoking the `Percy++` binaries.

Another difference between my tests and a potential real world scenario is that instead of deleting the files it has produced, the server would instead keep them, effectively caching them to be later used for other clients. To simulate this, I used larger amounts of requests from the client than they would normally need. This serves to amortize the time cost of mPHF generation or sorting and file system access.

3.3 Methodology

In this section I will describe the methods with which I acquired my data.

3.3.1 Hardware

I ran my benchmarks on an HP laptop equipped with an Intel Core i7-8750H CPU running Arch Linux kernel version 6.2.10. I used MariaDB 10.11.2 for my DBMS.

3.3.2 Evaluation factors

I chose three main factors to evaluate across various scenarios:

- communication cost
- computational cost
- information revealed to the server

I anticipated them all to be linked in a way where a decrease in one would lead to an increase in the others. I theorized that the relationship between them would heavily depend on the chosen type of `PIR` (`CPIR`, `ITPIR` or hybrid) and also on the chosen algorithm. I also expected communication and computation costs to be dependent upon the size of the database against which `PIR` is run.

3.3.3 Scenarios

To obtain representative data, I designed several scenarios similar to the ones a typical user of a public database might go through. I used a single server setup for `CPIR` and a two server setup for `ITPIR` and hybrid modes.

3.3.3.1 Point queries

The first few scenarios concern themselves with measuring performance when using the mPHF approach described in 3.1.2.1. I performed measurements using an increasing number of queries per mPHF generated. The main point of these scenarios was to measure at which point a trivial transfer would become preferable, and also how much information could be retrieved in a reasonable time, as well as to compare the performance of different protocols.

More specifically, I ran 2500 random unique queries over the `dlp_synonyma` table. I chose this table to demonstrate the unfortunate consequence of the design of the application, namely that the majority of data transferred was padding. Indeed, without padding, the table is not much larger than the `dlp_lecivepripravky` table (both roughly 10 MB). However, when both are padded for `PIR`, the size of `dlp_synonyma` increases to more than five times that of `dlp_lecivepripravky` (260 and 46 MB respectively). While this might be concerning to the user (only less than four percent of the data they are retrieving is useful to them!), it is the price they have to pay for perfect privacy. Why the user might not always want that shall be further discussed in section 3.3.3.3.

The query that was performed is the following:

```
SELECT concat(kod_latky, "-", sq), nazev
FROM dlp_synonyma
WHERE concat(kod_latky, "-", sq) = $PRIVATE CONSTANT$
```

Code listing 3.7 Unmodified point query

The private constant is the concatenation of the two parts of the key. I performed the retrievals with each of `CHOR`, `GF28`, `GF216`, and `HYB` modes. My preliminary testing showed that the only other multi-server mode, `ZZ_P`, performs roughly two orders of magnitude worse than the remaining modes and so I chose not to include it in the test. I also ran a smaller batch of 25 retrievals with `R_AG`, to compare the performance of a single-server mode against multi-server ones. Again, I chose not to include `AG` in the test, because it was performing orders of magnitude worse compared to the other modes.[3]

3.3.3.2 General queries

Next, my aim was to measure how impactful using general binary search queries would be compared with using mPHF, and to measure how effective the retrieval of non-unique indexes would be while maintaining maximum privacy.

In contrast to the previously mentioned point queries, a smaller table was chosen for general queries. I chose the `dlp_lecivepripravky` table for its prominence in the database and its convenient size. Even though the table is smaller, the number of retrievals still had to be decreased to achieve reasonable times, and so only 250 retrieval were performed for multi-server modes and 10 were performed for `R_AG`. The query was constructed like this:

```
SELECT *
FROM dlp_lecivepripravky
WHERE nazev = $PRIVATE CONSTANT$
```

Code listing 3.8 Unmodified generic query

3.3.3.3 Revealing private constants

The previous two sets of scenarios were designed to test the computational and communication costs of various modes of operation while revealing as little information to the server as possible.

[3] A comparison of the performance of all modes in a test described in section 3.3.3.3 can be found in file `privacy_results.txt`.

This scenario is aimed at showing that these costs can be drastically decreased if the client is willing to reveal some part of their private constants to the server.

The scenario is designed in such a way that the client reveals a suffix [4] of the key they are interested in and thus reduces the total size of the database over which `PIR` is run. Thus, the total time of retrieval is reduced.

As an example, let us look at the following query:

```
SELECT kod_sukl, nazev, sila, forma
FROM dlp_lecivepripravky
WHERE nazev='RITALIN';
```

Code listing 3.9 Query containing a full private constant

Following the procedure described in 3.1, the query could be transformed into:

```
SELECT kod_sukl, nazev, sila, forma
FROM dlp_lecivepripravky;
```

Code listing 3.10 Query with priate constant entirely removed

This query yields the selected columns from the entire table. This could be a problem, since the time required for successful retrieval increases with the size of the database. The only way for the client to reduce the size of the database is to reveal a part of their private constant, transforming the original query into:

```
SELECT kod_sukl, nazev, sila, forma
FROM dlp_lecivepripravky
WHERE nazev LIKE '%IN';
```

Code listing 3.11 Query revealing the final two characters of the private constant

This way, obtaining the result will be much faster, but the server will learn some information about the client's query. This can be problematic, as the client has no effective control over how much information is actually revealed to the server. For example, if there was only a single record that ended with this suffix, the client would effectively reveal their entire secret constant to the server. One way to prevent this could be for the server to send the client the list of unique values that are present in the concerned columns. However, for a larger database, this will become impractical, as the client might not be able to process the list effectively, or its transfer might be too slow. Therefore, I will assume some intuitive knowledge on the part of the client about the distribution of the data in the table.

Another way to prevent revealing too much could be to reveal some information about the constant instead of revealing parts of it. One such information to be revealed could be the length of the constant. This neatly deals with the issues outlined in section 3.1.1, reducing the total amount of padding and the overall database size. An example query could look like this:

```
SELECT kod_sukl, nazev, sila, forma
FROM dlp_lecivepripravky
WHERE LENGTH(nazev) < 20 AND LENGTH(nazev) > 10;
```

Code listing 3.12 Modified query revealing that the constant has a certain length

Depending on the extent of the intuition attributed to the client and their desired privacy, the range for the length could be broadened or narrowed. Indeed, both this approach and the approach of revealing parts of the constant could be combined and expanded in a myriad of ways (e.g. ranges and divisibility for numeric constants), but as general examples I believe these will suffice.

[4] While I use a suffix in my examples, any other `LIKE` clause would work just as well.

It should be noted that throughout this section, I assumed that the client has an intuitive knowledge of the distribution of the data in the database. If such intuitive knowledge is not available to the client or easily obtainable by them, or if the known distribution is not favourable to the client's query (e.g. they wish to retrieve a record whose key does not share many characteristics with the rest of the set), the client can attempt to smooth out the distribution by querying for the hash of the key instead of the key directly. Similar techniques can be used by the client to reveal only parts of the hash, some of its divisors etc.

I ran tests for both the revealing of parts of the secret constant and ranges for its length. I arbitrarily chose a record in the `dlp_lecivepripravky` with the most common length (7) of the `nazev` field. I then performed a series of queries revealing its suffix starting at no suffix revealed and ending when the effective reveal of the private constant was achieved (at 5 out of 7 characters revealed). Similarly, I performed a series of queries revealing the range of the constant's length. I started with the range $[0,\ 2 \times length(\$SECRET\ CONSTANT\$)]$ and in each step narrowed the range by one at each end, ultimately reaching a range only containing results of the same length as the private constant.

I performed these queries for all of the available modes with the exception of the combination of `AG` and no suffix reveal, because this test could not be performed on my machine due to insufficient memory.

3.4 Results

In this section, I will present and discuss the results of my measurements.

3.4.1 Performance

The main determining factor for the performance of my `PIR` based retrieval seems to be the uniqueness of the key. The worst case network transfer cost of performing $\log n$ queries per record vastly overshadows that of having to transfer the parameters of mPHF. As far as time is concerned, the time needed for sorting of the result set or mPHF generation will not play a role, as both the sorting order and the mPHF parameters can be efficiently cached. It will naturally take longer to perform $\log n$ queries than to perform a single one.

Having established that queries for unique and non-unique keys are indeed not comparable in any meaningful sense, let us now look at each one separately. Full results for all modes can be found in file `performance_results.txt` in the attached archive. Detailed measurements are included in the file `performance_measurements.csv`.

3.4.1.1 Communication costs for unique keys

The communication costs for all multi-server modes and trivial transfer can be seen in the following graph:

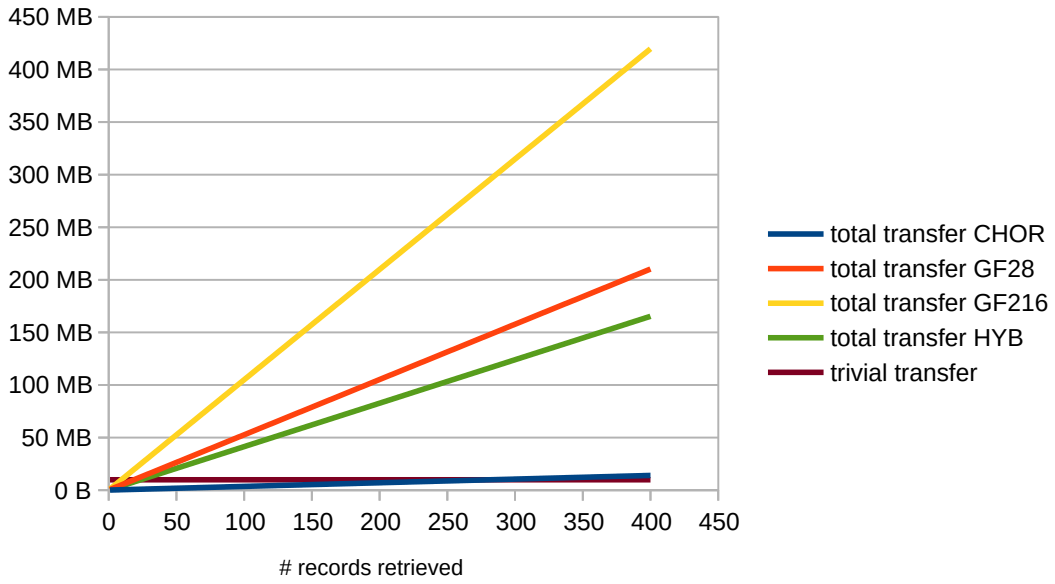

Figure 3.2 Data transferred per protocol for unique keys

We can see that `CHOR` is clearly the most communication efficient of all the modes. Of note is also the fact that `HYB` is distinctly more communication efficient than `GF28`, which is the mode used as the `ITPIR` part of `HYB`. `GF216`'s cost is almost exactly double the cost of `GF28`, which is to be expected, as the query contains the same number of elements for both of these modes, but the elements of `GF216` are twice as big.

Next, let us observe a section of the graph featuring all the previous modes, as well as `R_AG`:

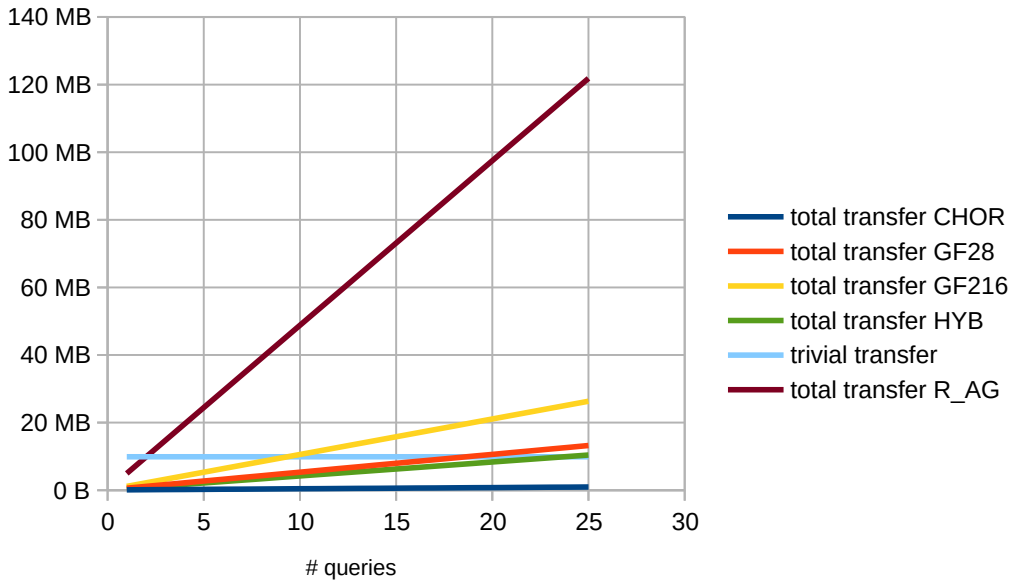

Figure 3.3 Data transferred per protocol for unique keys, including `R_AG`

We can see that `R_AG`'s scaling is much worse, yet for a single query it does perform better

than trivial transfer. This is no surprise, as the protocol was designed with much larger records in mind.

3.4.1.2 Communication costs for non-unique keys

The following graph shows the costs of communication for all the multi-server modes and trivial transfer for small number of non-unique queries:

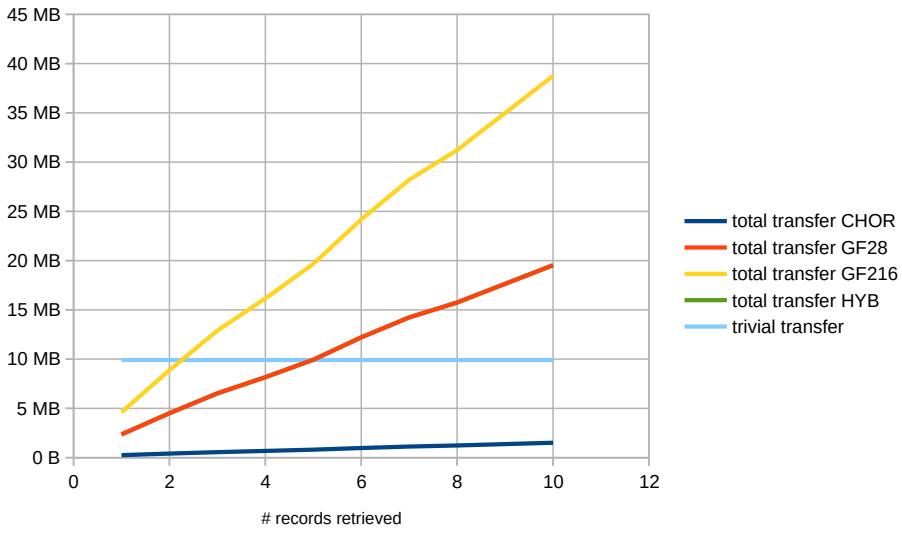

Figure 3.4 Data transferred per protocol for non-unique keys

We can see that even in this case, `CHOR` vastly outperforms all the other modes, as well as the trivial transfer. `GF28` and `HYB` have exactly the same performance in this case. This is due to the fact that `HYB` would not lower its transfer cost by using recursion, and effectively becomes `GF28`. Once again, `GF216`'s cost is double the cost of `GF28`.

Finally, let us also take a look at `R_AG`'s performance compared to the other modes.

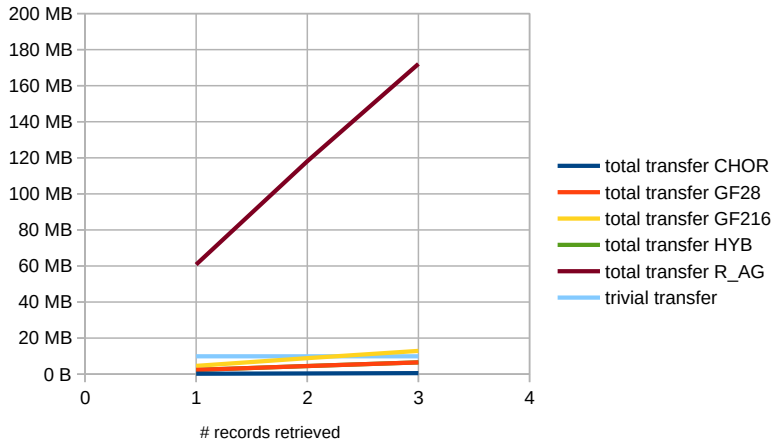

Figure 3.5 Data transferred per protocol for non-unique keys, including `R_AG`

We can see that the cost of using R_AG is much higher than that of a trivial transfer.

3.4.1.3 Time

Let us now compare the times it took to retrieve the records for each of the modes and the scenarios designed. I will consider time to be a good enough measurement of computational complexity, I will not be comparing the times to trivial transfer, as the time needed for that would heavily depend on network speed as well as client's capabilities, both of which will inevitably vary wildly between deployment situations. The time presented will be total time of the operation. However, the vast majority of this time was spent performing PIR processing. Time in seconds can be seen in the following table:

# records retrieved	R_AG	GF216	GF28	HYB	CHOR
Unique 2500	-	400.3681	415.2385	299.4892	225.7498
Non-unique 250	-	98.8406	314.1123	312.7850	34.6995
Unique 25	362.1292	-	-	-	-
Non-unique 3	153.9076	-	-	-	-

Table 3.1 Timing for queries in different modes

The following conclusions can be drawn from the data:

- CHOR, aside from being the most communication efficient, is also the fastest mode.
- R_AG is orders of magnitude slower than the other modes due to its computational nature.
- If recursion is used, HYB is faster than the GF2N modes. If it is not, it performs the same as GF28.
- Unlike the communication cost, the processing time for GF216 is lower than for GF28.

However, we can observe that, as we established earlier, the size of the database also influences the performance, not just the chosen mode. To quantify this, I ran a smaller benchmark to see how the different modes would perform on a database of changing size. I only changed the number of records in the database, since the client will never be retrieving any columns it does not need, and therefore in a database with ideally spaced records, the size of each record will be predetermined. I performed 20 unique queries for each of GF216, GF28, HYB, and CHOR for database sizes ranging between 2500 and 200000 records of uniform sizes. The results of the measurements can be seen in figure 3.6.

We can see that the growths are linear with some unpredictable spikes for GF28 and GF216. HYB mostly follows GF28, but at 150,000 records drops down and continues from there. This is the point at which HYB switches to using R_AG as well as GF28. Also of note is the performance of CHOR, which seems to be the fastest of all the protocols. The full results can be found in file *db_results.txt*.

In conclusion, all the protocols become slower with higher database sizes. I will discuss the utility of this in section 3.4.2.

3.4.1.4 Protocol choice

Which protocol should the client choose then? In the case that multiple servers can be operational and robustness is not required, CHOR seems to be the only reasonable choice.

If robustness is required, the client should choose HYB, and depending on whether they wish to optimize for time of bandwidth, choose either GF216 or GF28 as the ITPIR component respectively. It should be noted, however, that R_AG, operating as the CPIR component of HYB, is not secure. Until a suitable replacement is implemented in Percy++, the client should use the ITPIR component on its own for complete privacy.

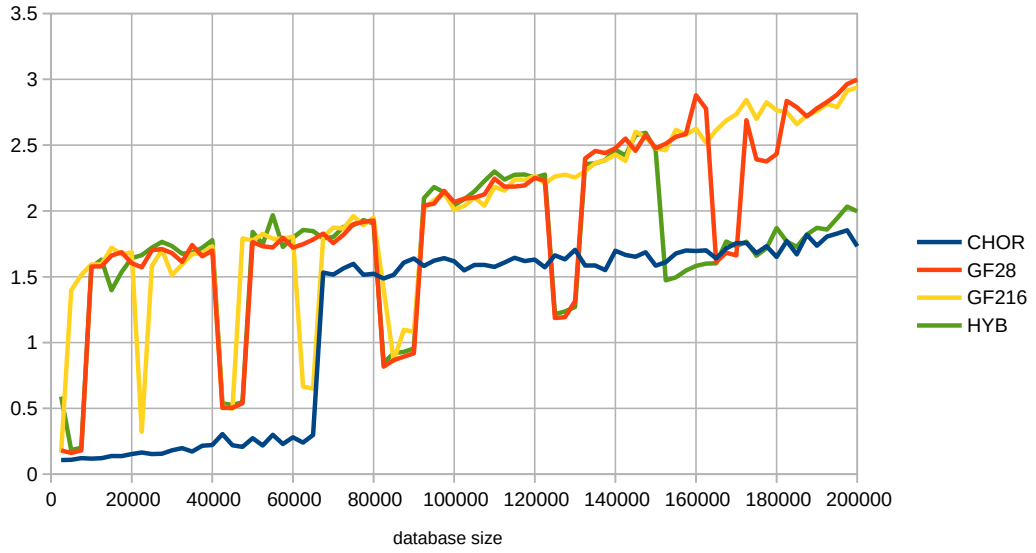

Figure 3.6 Time in seconds per mode for database size

`R_AG` and `AG` are both not secure, and therefore **`Percy++`** offers no single-server solution. Should such a solution be desired, it must be sought elsewhere, e.g. [8]. Even if that weren't the case, their performance is not good enough for real world deployment.

3.4.2 Privacy

Let us now take a look at how clients can influence the performance of their queries by revealing information about their secret constants. As I have demonstrated earlier, the size of the database determines the per query speed of retrieval. If a non-unique key is used, the total performance is further influenced by the need to perform multiple queries. Therefore, limiting the size of the database is a good way to improve performance, both in terms of computation and communication. As was mentioned earlier, limiting the size of the padded records is not an option, and thus only the number of records can be meaningfully influenced.

I will demonstrate this on the results of the benchmark of revealing suffix and length range of the private constant for `ZZ_P`. The full results for all modes can be found in `privacy_results.txt` file in the attached archive.

bench	*time/s*	*total_comm*	*data_size*	*pir_db_size*	*% useful*	*saved/s*	*worst*
suf-0	120.86	4.04 MB	9.79 MB	45.93 MB	21.32		16
suf-1	7.23	465.70 KB	1.42 MB	4.66 MB	30.43	113.63	14
suf-2	1.06	99.70 KB	235.55 KB	690.95 KB	34.09	6.17	11
suf-3	0.04	7.30 KB	11.97 KB	22.70 KB	52.71	1.02	7
suf-4	0.02	4.09 KB	2.05 KB	3.33 KB	61.68	0.02	4
suf-5	0.01	1.44 KB	0.26 KB	0.28 KB	93.62	0.01	1
len-0	38.91	1.79 MB	5.39 MB	24.80 MB	21.73		16
len-1	47.25	2.23 MB	5.01 MB	23.09 MB	21.70	-8.34	16
len-2	44.39	2.13 MB	4.76 MB	22.00 MB	21.64	2.86	15
len-3	24.34	1.16 MB	4.59 MB	21.23 MB	21.63	20.05	15
len-4	33.42	1.62 MB	4.44 MB	20.54 MB	21.61	-9.08	15
len-5	27.61	1.84 MB	4.09 MB	13.76 MB	29.74	5.81	15
len-6	20.08	1.40 MB	3.30 MB	11.03 MB	29.95	7.53	15
len-7	8.67	583.64 KB	1.33 MB	4.43 MB	30.04	11.41	14

Table 3.2 Effects of revealing information about secret constant for `ZZ_P`

The *bench* column displays the name of the benchmark. The number indicates how long the revealed suffix is for suffix reveal and how much the range has shrunk for length range. The total amount of communication between the server and the client can be seen in the *total_comm* column. Columns *data_size* and *pir_db_size* contain the sizes of the query result and the result padded for `PIR` respectively. The percentage of useful data in the padded result is expressed in the *% useful* column. Time saved over the previous row can be seen in the *save/s* column, and the final column contains the number of queries necessary to retrieve a record in the worst case scenario.

We can clearly see that revealing information about the secret constant does in fact improve performance by virtue of lowering the database size. While the trend isn't uniform (sometimes, the record will end up at a worse position in terms of binary search), it is clearly present. The performance improvement is much steadier for the length reveal, and the overall performance remains lower than when a suffix is revealed. However, the secret constant is never revealed, unlike suf-5, in which the only remaining records in the database match the client's secret constant. Also, in all cases except for suf-4 and suf-5, trivial transfer is, communication-wise, worse than the `PIR` transaction performed.

benchmark	*record_size*	*num_records*
suf-0	736	62,400
suf-1	507	9,189
suf-2	418	1,653
suf-3	264	86
suf-4	222	15
suf-5	141	2
len-0	679	36,520
len-1	677	34,107
len-2	676	32,546
len-3	675	31,457
len-4	674	30,481
len-5	488	28,191
len-6	483	22,838
len-7	479	9,243

Table 3.3 Effects of revealing information about secret constant for `ZZ_P` on the number of records and their size

In table 3.3, we can see that the effect of revealing information to the server is quite unpredictable. For example, we see a large jump in record size between len-4 and len-5, which also corresponds to a large increase in percentage of effective data in the database which we see in the corresponding rows of table 3.2. Indeed, it means that a record that was much larger than the rest had been dropped, but the client had no way of predicting that this was going to happen. While they could have, in theory, gained some knowledge about the distribution of the data in the table by performing a trivial transfer of certain columns in advance, the generality of this approach is dubious and also runs the risk of disclosing sensitive information to the server.

In conclusion, paired with clever trivial transfers, the client can craft a query that does reduces the client's privacy but increases the performance. This might be a worthwhile trade-off, but generalizing this approach might not be easy. Further research is needed in this area.

Chapter 4
Conclusion

I have successfully completed the goals of this thesis. I have described all solutions to the **PIR** problem available in the **Percy++** library. I have analyzed their properties, described the security guarantees they present, compared their respective strengths and weaknesses.

Building on the insight I have gained researching this topic, I have designed several generic deployment scenarios. I have analyzed their specifics and given concrete real world examples of situations in which these generic scenarios might might find use. While I did not give it much attention in the later parts of my thesis, I find the topic of **PIR** enabled anonymizers very interesting. I would like to continue my research by enhancing existing anonymizers with **PIR**, utilizing what I have learned over the course of writing this thesis.

I have also built a testing suite designed to mimic some of the designs proposed in the previous parts. On a suitable dataset, I have performed tests to verify my earlier claims. I have performed performance analysis of the different protocols. While my approach turned out to be efficient enough in a multi-server environment, its performance in a single-server setting was not sufficient to outperform trivial transfer except for very small cases. In further research, I would like to focus on integrating other existing `CPIR` protocols into the framework of **Percy++**.

In the framework of my SQL mock-up framework, I have devised a way for the client to gain some performance by sacrificing parts of their privacy. While it did turn out that the gain in performance is substantial, I also noticed that the amount of lost privacy cannot be easily controlled by the client. This is due to the fact that the client has only limited knowledge of the database and few ways to obtain more information about it. I would like to further improve the ways in which the client might choose to limit their privacy for a gain in performance.

Overall, I have shown that **PIR** is a useful piece of technology. I have shown that it can be easily integrated into existing infrastructures utilizing public facing SQL databases and provide improved experience for the clients. I have shown that for very small cases, single-server solution can be employed. If a client needs to perform more than a few queries, a multi-server solution is necessary. Unless robustness against misbehaving servers is necessary, Chor's algorithm performs the best on all metrics. If robustness is necessary, then Goldberg and Devet's hybrid algorithm should be chosen.

Bibliography

1. CHOR, Benny; GILBOA, Niv; NAOR, Moni. *Private information retrieval by keywords*. Citeseer, 1997.
2. CHOR, Benny; KUSHILEVITZ, Eyal; GOLDREICH, Oded; SUDAN, Madhu. Private information retrieval. *Journal of the ACM (JACM)*. 1998, vol. 45, no. 6, pp. 965–981.
3. SION, Radu; CARBUNAR, Bogdan. On the computational practicality of private information retrieval. In: *Proceedings of the network and distributed systems security symposium*. Internet Society Geneva, Switzerland, 2007, pp. 2006–06.
4. AGUILAR-MELCHOR, Carlos; GABORIT, Philippe. A lattice-based computationally-efficient private information retrieval protocol. *Cryptology ePrint Archive*. 2007.
5. AGUILAR-MELCHOR, Carlos; CRESPIN, Benoit; GABORIT, Philippe; JOLIVET, Vincent; ROUSSEAU, Pierre. High-speed private information retrieval computation on gpu. In: *2008 Second International Conference on Emerging Security Information, Systems and Technologies*. IEEE, 2008, pp. 263–272.
6. OLUMOFIN, Femi; GOLDBERG, Ian. Revisiting the computational practicality of private information retrieval. In: *Financial Cryptography and Data Security: 15th International Conference, FC 2011, Gros Islet, St. Lucia, February 28-March 4, 2011, Revised Selected Papers 15*. Springer, 2012, pp. 158–172.
7. WIESCHEBRINK, Christian. Two NP-complete problems in coding theory with an application in code based cryptography. In: *2006 IEEE International Symposium on Information Theory*. IEEE, 2006, pp. 1733–1737.
8. AGUILAR-MELCHOR, Carlos; BARRIER, Joris; FOUSSE, Laurent; KILLIJIAN, Marc-Olivier. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*. 2016, pp. 155–174.
9. LIU, Jiayang; BI, Jingguo. Cryptanalysis of a fast private information retrieval protocol. In: *Proceedings of the 3rd ACM international workshop on ASIA public-key cryptography*. 2016, pp. 56–60.
10. GOLDBERG, Ian. Improving the robustness of private information retrieval. In: *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 2007, pp. 131–148.
11. CYCLOTOMICS INC. *Error correction for algebraic block codes*. Inventor: Lloyd R. WELCH; Elwyn R. BERLEKAMP. Publ.: 1986-12-30. Appl.: 1983-09-27. US 4 633 470 A. Available also from: `https://patents.google.com/patent/US4633470A`.
12. GURUSWAMI, Venkatesan; SUDAN, Madhu. Improved decoding of Reed-Solomon and algebraic-geometric codes. In: *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*. IEEE, 1998, pp. 28–37.

13. DEVET, Casey; GOLDBERG, Ian; HENINGER, Nadia. Optimally Robust Private Information Retrieval. In: *USENIX Security Symposium*. 2012, pp. 269–283.

14. DEVET, Casey; GOLDBERG, Ian. The best of both worlds: Combining information-theoretic and computational PIR for communication efficiency. In: *Privacy Enhancing Technologies: 14th International Symposium, PETS 2014, Amsterdam, The Netherlands, July 16-18, 2014. Proceedings 14*. Springer, 2014, pp. 63–82.

15. BÖTTGER, Timm; CUADRADO, Felix; ANTICHI, Gianni; FERNANDES, Eder Leão; TYSON, Gareth; CASTRO, Ignacio; UHLIG, Steve. An Empirical Study of the Cost of DNS-over-HTTPS. In: *Proceedings of the Internet Measurement Conference*. 2019, pp. 15–21.

16. KUMAR MISHRA, Sanjeev; SARKAR, Palash. Symmetrically private information retrieval. In: *Progress in Cryptology—INDOCRYPT 2000: First International Conference in Cryptology in India Calcutta, India, December 10–13, 2000 Proceedings 1*. Springer, 2000, pp. 225–236.

17. NARAYANAM, Krishnasuri; RANGAN, C. A novel scheme for single database symmetric private information retrieval. In: *Proceedings of Annual Inter Research Institute Student Seminar in Computer Science (IRISS)*. Citeseer, 2006, pp. 803–815.

18. SWEENEY, Latanya. k-anonymity: A model for protecting privacy. *International journal of uncertainty, fuzziness and knowledge-based systems*. 2002, vol. 10, no. 05, pp. 557–570.

19. INVIDIOUS PROJECT CONTRIBUTORS. *List of public Invidious instances* [online]. 2023-05. [visited on 2023-05-11]. Available from: `https://docs.invidious.io/instances/`.

20. NITTER PROJECT CONTRIBUTORS. *List of public Nitter instances* [online]. GitHub [visited on 2023-05-11]. Available from: `https://github.com/zedeus/nitter/wiki/Instances`.

21. PROXITOK PROJECT CONTRIBUTORS. *List of public ProxiTok instances* [online]. GitHub [visited on 2023-05-11]. Available from: `https://github.com/pablouser1/ProxiTok/wiki/Public-instances`.

22. ZHAO, Fangming; HORI, Yoshiaki; SAKURAI, Kouichi. Two-servers PIR based DNS query scheme with privacy-preserving. In: *The 2007 International Conference on Intelligent Pervasive Computing (IPC 2007)*. IEEE, 2007, pp. 299–302.

23. OLUMOFIN, Femi; GOLDBERG, Ian. Privacy-preserving queries over relational databases. In: *Privacy Enhancing Technologies: 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings 10*. Springer, 2010, pp. 75–92.

24. STÁTNÍ ÚSTAV PRO KONTROLU LÉČIV. *Databáze léčivých přípravků DLP* [online]. 2023-04. [visited on 2023-04-26]. Available from: `https://opendata.sukl.cz/?q=katalog/databaze-lecivych-pripravku-dlp`.

25. BELAZZOUGUI, Djamal; BOLDI, Paolo; OTTAVIANO, Giuseppe; VENTURINI, Rossano; VIGNA, Sebastiano. *Cache-Oblivious Peeling of Random Hypergraphs*. 2013. Available from arXiv: `1312.0526 [cs.DS]`.

Contents of the attached archive

- `README` — brief description of the attached archive
- `impl` — implementation and library source code
  - `README` — instructions for building and running the code
  - `globals.py` — settings for client and server
  - `server.py` — server side of the application
  - `client.py` — client side of the application
  - `benchmark.py` — utility program to perform benchmarks
  - `tests`
    - `tests.py` — definitions of various tests
  - `emphf` — emphf library with some modifications
  - `percy++-1.0.0` — Percy++ library
- `data` — data for the benchmarks
  - `create.sql` — SQL script to create the necessary tables
  - `populate.sql` — SQL script to populate the tables with data
  - `table_data` — CSV data to populate the tables with
- `results` — results of the benchmarks
- `thesis` — thesis LaTeX sources
- `thesis.pdf` — text of the thesis in PDF