Master Thesis

University of Bergen

Department of Informatics

Simulating Private Information Retrieval on Amazon Web Services

Author:
Eirik Rygnestad Bergesen

Supervisors:
Hsuan-Yin Lin
Eirik Rosnes

September 15, 2023

Preface

I want to express my gratitude toward my supervisors, Hsuan-Yin Lin and Eirik Rosnes. This thesis could not exist without your guidance. Thank you for sharing your knowledge with me, and thanks for many good discussions.

I would also like to give my thanks to my friends and fellow students at the master's study halls Glassburet, and Algo at the University of Bergen. We have shared many laughs and good times and you have kept my spirit high.

Abstract

As our modern lives have gradually moved more and more online, companies and state actors have taken it upon themselves to gather and analyze our behavior online, and as these actors have gradually shown just how much they know about a private user, or a group of users, a concern for privacy has grown accordingly. A virtual private network service could help anonymize a user, but the providers of services usually log what services they provide, which can provide identifying information. Research in privacy measures have thus become a larger topic in recent time. Private information retrieval allows a user to query a database without revealing to the server any information about the information queried, and if effective enough, could provide perfect privacy to everyone. In this thesis, we examine a state-of-the-art efficient private information retrieval scheme and study every step in the protocol in a simulation implemented on Amazon's cloud computing services.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

The International Telecommunication Union estimates that there were 5.3 billion active internet users in 2022 [1], and we can assume this number has steadily increased. The internet has become a massive medium for sharing information, social media, communication, collaboration, and commerce. As our daily online presence increases, what one can learn about our footprint also increases. Many share a lot of information about themselves online, with less concern for privacy, in exchange for some convenience, like Facebook being able to recommend your next friend or TikTok recommending the entertainment you are statistically inclined to watch. There are people who do not like to give up their information and preferences. Some people feel like the amount of knowledge some companies have about them is frightening. For example, how many were frightened by how the American retailer Target can fairly accurately estimate whether a woman is pregnant and how far along they are based on purchases [2], while still acknowledging how creepy and discomforting this would be if known to the customer. A private information retrieval (PIR) scheme allows a user to retrieve a file from a database without revealing any information about the requested file. This was first introduced by Chor *et al.* in 1995 [3], who proved that when accessing a single database, to guarantee perfect privacy, the whole database had to be downloaded. In the same paper, Chor *et al.* proved and gave examples of multi-server schemes of communication cost as low as $\mathcal{O}\sqrt[3]{n}$. Since Chor *et al.* published their paper in 1995, there have been many studies on reducing communication costs further using multi-server schemes. As modern computers and cloud services have vastly improved over the last decade we aim to study the practical performance of one of these new private information retrieval schemes, and what challenges emerge when doing a practical implementation on Amazon's modern cloud infrastructure. We implement Tajeddine, Gnilke, and El Rouayheb's private information retrieval scheme for a distributed storage system for no colluding servers [4] on Amazon's cloud service Elastic Cloud Computing, to study its performance. We compare it to a direct download scheme and Chor *et al.*'s scheme implemented on the same architecture to provide some comparisons on performance. We study and compare each individual step in these protocols and comment on their performance.

1.2 Our Objective

Our objective is to study how a modern PIR scheme performs in a simulation implemented on Amazon's web services. We aim to provide details of our implementation of Tajeddine, Gnilke, and El Rouayheb's scheme for no colluding nodes, Chor *et al.*'s scheme, and a direct download scheme. We aim to provide details of how our servers have been implemented, and what challenges we have faced when managing our servers in the Amazon cloud computing service, Elastic Cloud Computing. We aim to compare the performance of our implementation of Tajeddine, Gnilke, and El Rouayheb's scheme using a Reed Solomon encoding for some selected different parameters, to Chor *et al.*'s scheme implemented using 2 and 3 servers, and a no privacy single server direct download scheme. We aim to explore the challenges of our implementation of these private information retrieval schemes, provide insight into practical performance, and illuminate potential points of congestion and other factors for performance issues, in all steps of our simulation.

1.3 Thesis Organization

- **Chapter 1:** An introduction to our thesis.
- **Chapter 2:** Introduction of concepts and terminology used in this thesis and an overview of relevant related work.
- **Chapter 3:** In-depth details of our implementation and simulation environment
- **Chapter 4:** Detailed examination of our results and findings
- **Chapter 5:** Suggestions for future work and some final remarks drawn from our findings.

Notation and Nomenclature

We introduce some notation and nomenclature that will be used within the body of this document further on.

Abbreviations

CGKS Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan

DSS Distributed storage system

MDS Maximum distance separable

PIR Private information retrieval

RS Reed Solomon

TGE Razane Tajeddine, Oliver W. Gnilke, and Salim El Rouayheb

Notation

Bold italic characters Reference a vector or a matrix

Italic characters Reference a symbol or integer

Nomenclature

α	The number of stripes in a divided file, $d-1$, $n-k$
β	Quotient of $\frac{n-k}{k}$
$\boldsymbol{A}$	A transposed Vandermonde matrix, generator matrix for Reed Solomon codes
$\boldsymbol{G}$	Generator matrix for a linear code
$\boldsymbol{W}$	The RS encoded database
ω	The number of bytes in one file
σ	The sub-packetization factor of Chor *et al.*'s scheme
$\lvert \cdot \rvert$	The total number of one's in a matrix or a vector
d	The minimum distance of an MDS code, $d = n - k + 1$
k	The message length of a linear code
M	The number of files in the database

n The block length of a linear code, and the number of server nodes in the schemes

q The field over which the linear code is defined, in our implementation, the prime field GF(257)

r Remainder of $\frac{n-k}{k}$

Chapter 2

Preliminaries

In this chapter, we review some of the terminologies and concepts relevant to this thesis.

2.1 Finite Fields

A finite field or Galois field is a set of elements that satisfy the rules of arithmetic, called field axioms. This means that the elements are closed under addition, subtraction, multiplication, and division (except for division by zero). The order or size of a field is the number of elements in the field and is always a prime number or a power of a prime. Finite fields are denoted by $\mathrm{GF}(p^r)$ or $\mathrm{GF}(q)$ where $q = p^r$ or by $\mathbb{F}_{p^r}$, for some positive integer r. $\mathrm{GF}(p)$ is called the prime field of order p and is the field of residue classes modulo p. The prime field $\mathrm{GF}(p)$ thus contains the elements $0, 1, 2, \ldots, p-1$.

2.2 Linear Codes

A linear code over a finite field of q elements $\mathbb{F}_q$ is a linear subspace of the vector space $\mathcal{C} \subset \mathbb{F}_q^n$. The vectors making up this subspace are called codewords. The code gains an error-correcting feature when the distance between the codewords is large enough. If a codeword is received with errors, the codeword can be recovered by finding the nearest neighbor codeword. How many errors can be corrected depends on the distance between the codewords. The sphere packing bound explains that a code with a minimum distance d between two arbitrary codewords can detect $d-1$ errors and can correct $\lfloor \frac{d-1}{2} \rfloor$ errors. In a linear code, all linear combinations of the codewords are also codewords.

A linear block code is a linear code where blocks of information of length k are encoded into n symbols. This can be achieved by multiplying the message block of length k by a matrix of dimensions $k \times n$. These matrices are called generator matrices, and the resulting codeword $\boldsymbol{x}$ is given by

$$\boldsymbol{uG} = \boldsymbol{x},$$

where $\boldsymbol{u}$ denotes the message and $\boldsymbol{G}$ is the code generator matrix.

When the rows of the generator matrix consist of linearly independent vectors in $\mathbb{F}_q^n$ a code is created. As such, linear block codes are denoted by the block length

n, the message length k, the order of the field in which they are defined q, and the minimum distance between codewords, d.

Maximum distance separable (MDS) codes are a subset of linear block codes that achieve the Singleton bound, i.e., they have the maximum distance separating property of any two codewords, hence the name. For codes achieving the Singleton bound, the minimum distance between any two arbitrary codewords in the code is $d = n - k + 1$.

Reed Solomon (RS) codes are MDS codes where all the codewords in the code can be described as a sequence of function values of a polynomial of degree less than k. RS codes are denoted by a block length n, message length k, and an alphabet size q, where $k < n \leq q$. The set of alphabet symbols is interpreted as the finite field of order q, where q is a prime power. To obtain the codeword related to some message of length k made up of the elements in $\mathbb{F}_q$, the message symbols are treated as coefficients of a polynomial p of degree less than k, of the finite field of q elements. The polynomial $p_{\boldsymbol{u}}$ of a given message vector $\boldsymbol{u}$ is evaluated at $n \leq q$ distinct points: $\{a_1, \ldots, a_n\}$, and the corresponding codeword $\boldsymbol{x}$ is generated as

$$\boldsymbol{x}(\boldsymbol{u}) = \big(p_{\boldsymbol{u}}(a_1), p_{\boldsymbol{u}}(a_2), \ldots, p_{\boldsymbol{u}}(a_n)\big),$$

where $p_{\boldsymbol{u}}$ is a polynomial over $\mathbb{F}_q$ of degree less than k. The codeword $\boldsymbol{x}(\boldsymbol{u})$ satisfies $\boldsymbol{x}(\boldsymbol{u}) = \boldsymbol{u}^\intercal \boldsymbol{A}$, where $\boldsymbol{A}$ is a transposed Vandermonde matrix of dimensions $n \times k$ with elements from $\mathbb{F}_q$.

$$\boldsymbol{A} = \begin{bmatrix} 1 & \ldots & 1 & \ldots & 1 \\ a_1 & \ldots & a_k & \ldots & a_n \\ a_1^2 & \ldots & a_k^2 & \ldots & a_n^2 \\ \vdots & & \vdots & & \vdots \\ a_1^{k-1} & \ldots & a_k^{k-1} & \ldots & a_n^{k-1} \end{bmatrix}$$

That is, as RS codes are linear codes, we can use a transposed Vandermonde matrix as a generator matrix for an RS code.

2.3 Private Information Retrieval (PIR)

Private information retrieval was introduced in 1995 by Chor *et al.* [3]. Private information retrieval means that any of the queried databases can not learn any information on the identity of the item requested from the query it received. As the only single server information-theoretical PIR (IT-PIR) scheme is to download the whole database, IT-PIR research often approaches this problem by querying a replicated database stored over several servers. By querying bits of information from different servers where the actual query is hidden in interference, a user can remove the interference and reconstruct the message from the replies. The use of multi-server schemes also allows for reducing the storage overhead through the use of coded distributed storage systems. The efficiency of private information retrieval schemes is measured in the communication cost, where the communication cost contains both the upload cost and the download cost. As the size of the downloaded data often far outweighs the size of the queries, the efficiency is often measured in only the download cost. The efficiency of the download cost is typically known as

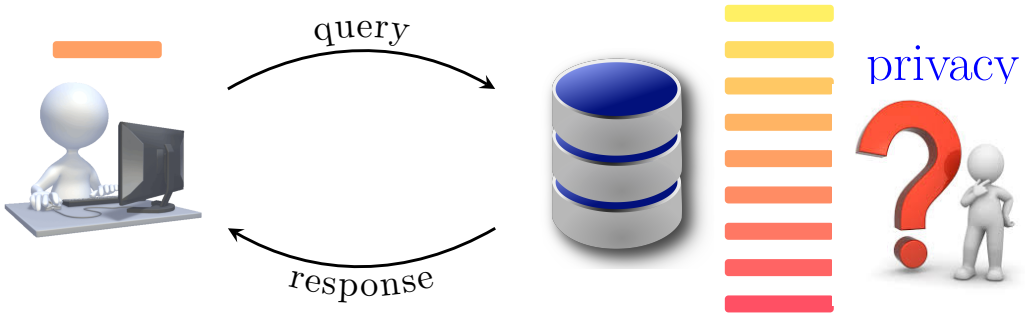

Figure 2.1: Illustration of PIR

the PIR rate. The PIR rate is the ratio of the number of symbols in the file requested and the number of symbols downloaded in the PIR scheme to retrieve a single file, i.e.,

$$R_{\text{PIR}} = \frac{\text{Number of symbols in file requested}}{\text{Number of symbols downloaded to retrieve requested file}}$$

The PIR capacity is the maximum PIR rate. Sun and Jafar [5] characterized the PIR capacity for the classical PIR model of replicated servers.

2.4 Chor *et al.*'s Scheme

In 1995, Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan (CGKS) released the paper *Private Information Retrieval* [3] detailing a scheme where the whole database is fully replicated across n servers. The scheme queries for a response containing information on the database and can reconstruct the desired file from the replies. In terms of communication cost, CGKS's scheme requires sending queries of size equal to the number of files in the database, M, to each of the databases. For two servers the size of the download from each of the servers is equal to the size of the largest file in the database. If the number of servers is more than two, we can query for a block of the file from each of them, reducing the download cost from each of the servers and reducing the total download cost. For a client to retrieve a file from this multi-server structure, it must generate n queries, where each query is of length equal to M, the number of files in the database. The servers compute responses of the database onto the query and send the results back to the client. The client must decode the file with a simple subtraction operation and a simple modulus operation.

In CGKS's scheme with n servers, the database is stored with full redundancy at all servers, $\boldsymbol{S}_1, \boldsymbol{S}_2, \ldots, \boldsymbol{S}_n$.

$$\boldsymbol{S}_l = \begin{bmatrix} \boldsymbol{X}_0 \\ \boldsymbol{X}_1 \\ \vdots \\ \boldsymbol{X}_{M-1} \end{bmatrix}, \quad l \in [n].$$

Now consider each file $\boldsymbol{X}$ divided into β sub-packets.

$$\boldsymbol{S}_l = \begin{bmatrix} x_{0,1} \\ x_{0,2} \\ \vdots \\ x_{0,\beta} \\ x_{1,1} \\ x_{1,2} \\ \vdots \\ x_{1,\beta} \\ \vdots \\ x_{M-1,1} \\ x_{M-1,2} \\ \vdots \\ x_{M-1,\beta} \end{bmatrix}, \quad l \in [n].$$

If the scheme is implemented with only two servers, then $\beta = 1$, and the sub-packet represents the whole file, but for $n > 2$, each file is divided into $\beta = n-1$ sub-packets. To retrieve a file i, $\boldsymbol{X}_i$, we need to retrieve all the sub-packets $x_{i,1}, x_{i,2}, \ldots, x_{i,\beta}$ to be able to reconstruct $\boldsymbol{X}_i$. To make a retrieval of $\boldsymbol{x}_i$, we generate our queries $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_n$ to send to the servers. We first generate a binary random vector $\boldsymbol{u} = [u_0, u_1, \ldots, u_{\beta M-1}]$, whose elements are chosen uniformly random from $\mathbb{F}_2$, as in either 0 or 1, with length equal to βM. We then generate β unit vectors $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_\beta$ of length βM, where the position correlating to the sub-packet we are trying to retrieve from the server database is equal to 1. To ensure we retrieve all the sub-packets $\boldsymbol{X}_i$ consists of, we need to retrieve the sub-packets $x_{i,1}, x_{i,2}, \ldots, x_{i,\beta}$.

$$\boldsymbol{S} = \begin{bmatrix} x_{0,1} \\ x_{0,2} \\ \vdots \\ x_{0,\beta} \\ \vdots \\ x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,\beta} \\ \vdots \\ x_{M-1,1} \\ x_{M-1,2} \\ \vdots \\ x_{M-1,\beta} \end{bmatrix}, \boldsymbol{e}_1 = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ 0 \\ \ldots \\ 0 \end{bmatrix}, \boldsymbol{e}_2 = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ 1 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \boldsymbol{e}_\beta = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Our queries $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_n$, where $n = \beta + 1$, are composed of the according unit vector $\boldsymbol{e}_i$ and the random vector $\boldsymbol{u}$ added together modulo 2, as our vectors are defined in $\mathbb{F}_2$, $\boldsymbol{u} + \boldsymbol{e}_1, \boldsymbol{u} + \boldsymbol{e}_2, \ldots, \boldsymbol{u} + \boldsymbol{e}_\beta$, while the last query remains as the random

vector $\boldsymbol{u}$.

$$\boldsymbol{Q}_1 = \begin{bmatrix} u_0 + e_{1,0} \\ u_1 + e_{1,1} \\ \cdots \\ u_{\beta M-1} + e_{1,\beta M-1} \end{bmatrix} \bmod 2, \quad \boldsymbol{Q}_2 = \begin{bmatrix} u_0 + e_{2,0} \\ u_1 + e_{2,1} \\ \cdots \\ u_{\beta M-1} + e_{2,\beta M-1} \end{bmatrix} \bmod 2, \ldots$$

$$\boldsymbol{Q}_\beta = \begin{bmatrix} u_0 + e_{\beta,0} \\ u_1 + e_{\beta,1} \\ \cdots \\ u_{\beta M-1} + e_{\beta,\beta M-1} \end{bmatrix} \bmod 2, \quad \boldsymbol{Q}_n = \begin{bmatrix} u_0 \\ u_1 \\ \cdots \\ u_{\beta M-1} \end{bmatrix}.$$

This means that any of queries $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_\beta$ and $\boldsymbol{Q}_n$ will always have magnitudes differing by 1. These queries are then sent to the servers $\boldsymbol{Q}_1 \to \boldsymbol{S}_1, \boldsymbol{Q}_2 \to \boldsymbol{S}_2, \ldots, \boldsymbol{Q}_n \to \boldsymbol{S}_n$. Note it does not matter what query is sent to what server as long as one query is sent to each server. To uphold the restrictions on download cost, the responses are computed in $\mathbb{F}_q$, where $\mathbb{F}_q$ is of a larger order than any unit of information in $\boldsymbol{X}_i$. The responses sent from the servers are

$$\begin{aligned}
\boldsymbol{r}_1 &= (u_0 + e_{1,0})x_{0,1} + (u_1 + e_{1,1})x_{0,2} + \cdots + (u_{\beta M-1} + e_{1,\beta M-1})x_{M-1,\beta} \bmod q \\
\boldsymbol{r}_2 &= (u_0 + e_{2,0})x_{0,1} + (u_1 + e_{2,1})x_{0,2} + \cdots + (u_{\beta M-1} + e_{2,\beta M-1})x_{M-1,\beta} \bmod q \\
&\vdots \\
\boldsymbol{r}_\beta &= (u_0 + e_{\beta,0})x_{0,1} + (u_1 + e_{\beta,1})x_{0,2} + \cdots + (u_{\beta M-1} + e_{\beta,\beta M-1})x_{M-1,\beta} \bmod q \\
\boldsymbol{r}_n &= (u_0 x_{0,1} + u_1 x_{0,2} + \cdots + u_{\beta M-1} x_{M-1,\beta}) \bmod q
\end{aligned}$$

As the client receives these answers, we can remove the interference and retrieve the sub-packets we requested. We compare each of the queries $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_\beta$ to the query $\boldsymbol{Q}_n$. If the magnitude of $\boldsymbol{Q}_n$ is larger than the query compared to it, we subtract the projection received from the query of the smallest magnitude from the one with the projection received from the query of the larger magnitude, modulo q, to retrieve the sub-packet. If $|\boldsymbol{Q}_1| > |\boldsymbol{Q}_n|$ then, $x_{i,1} = (\boldsymbol{r}'_1 - \boldsymbol{r}'_n) \bmod q$, otherwise $|\boldsymbol{Q}_1| < |\boldsymbol{Q}_n|$ and, $x_{i,1} = (\boldsymbol{r}'_1 - \boldsymbol{r}'_n) \bmod q$. Thus, we can retrieve all the sub-packets and restore the requested file, $\boldsymbol{X}_i$.

2.5 Tajeddine, Gnilke, and El Rouayheb's Scheme

In *Private Information Retrieval from MDS Coded Data in Distributed Storage Systems* [4], Razane Tajeddine, Oliver W. Gnilke, and Salim El Rouayheb introduce a PIR scheme for no colluding servers with a download cost of $\frac{1}{1-R}$ per unit of requested data, that achieves the theoretic optimal limit for linear schemes, where $R = \frac{k}{n}$ is the code rate, and n and k are the block length and the message length of the MDS code by which the database is encoded. This scheme is referred to as the TGE scheme in this thesis.

2.5.1 Database Structure

Consider a distributed storage system (DSS) that stores M files over n nodes. The DSS uses an (n, k, d) MDS code over $\mathbb{F}_q$ to store the files with redundancy at the n

nodes. To encode the files, they are all divided into α stripes, where each stripe is divided into k blocks. The files are thus divided into αk blocks. Let B be the ratio of the number of bytes in a file to αk. Then, block $\boldsymbol{x}_{1,1}$ contains the first B bytes of the file, block $\boldsymbol{x}_{1,2}$ contains the next B bytes of the file, and so on.

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_{1,1} \\ \boldsymbol{x}_{1,2} \\ \dots \\ \boldsymbol{x}_{1,k} \\ \boldsymbol{x}_{2,1} \\ \boldsymbol{x}_{2,2} \\ \dots \\ \boldsymbol{x}_{2,k} \\ \dots \\ \boldsymbol{x}_{\alpha,1} \\ \boldsymbol{x}_{\alpha,2} \\ \dots \\ \boldsymbol{x}_{\alpha,k} \end{bmatrix} \longrightarrow \begin{bmatrix} \boldsymbol{x}_{1,1} & \boldsymbol{x}_{1,2} & \dots & \boldsymbol{x}_{1,k} \\ \boldsymbol{x}_{2,1} & \boldsymbol{x}_{2,2} & \dots & \boldsymbol{x}_{2,k} \\ \dots & \dots & \dots & \dots \\ \boldsymbol{x}_{\alpha,1} & \boldsymbol{x}_{\alpha,2} & \dots & \boldsymbol{x}_{\alpha,k} \end{bmatrix}.$$

Each stripe is encoded separately by multiplying them with the generator matrix $\boldsymbol{G}$. This gives the encoded database $\boldsymbol{W}$. Our encoded database will thus have an $(\alpha M) \times n$ structure. This encoded database $\boldsymbol{W}$ is then divided by columns, where $\boldsymbol{W} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_n)$, and each column $\boldsymbol{w}_l$ is stored on a respective server $\boldsymbol{S}_l$. We assume the structure of the database is known to any user.

2.5.2 Queries

To construct our queries, we will first generate the interference pattern, and then we will generate the retrieval pattern. The interference pattern is a random matrix $\boldsymbol{U}$ of dimension $k \times \alpha M$, where each element is a random integer of $\mathbb{F}_q$. This will ensure private retrieval and conceal the retrieval pattern.

$$\boldsymbol{U} = \begin{bmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,\alpha M} \\ u_{2,1} & u_{2,2} & \dots & u_{2,\alpha M} \\ \dots & \dots & \dots & \dots \\ u_{k,1} & u_{k,2} & \dots & u_{k,\alpha M} \end{bmatrix}.$$

While $\boldsymbol{U}$ is a random non-deterministic matrix meant to confuse and conceal, our retrieval pattern $\boldsymbol{E}$ is a deterministic matrix, dependent on what file $\boldsymbol{X}$ we are retrieving. The retrieval pattern $\boldsymbol{E}_f$ will ensure that we will retrieve parts of $\boldsymbol{X}_f$ such that $\boldsymbol{X}_f$ can be restored from the responses to the query.

For querying the systematic nodes, we will first generate the pattern $\boldsymbol{E}_{f,1}$:

$$\boldsymbol{E}_{f,1} = \left[\begin{array}{c|c|c|c} \boldsymbol{0}_{k\times(f-1)\alpha} & \begin{matrix} \boldsymbol{I}_{r\times r} \\ \boldsymbol{0}_{(k-r)\times r} \end{matrix} & \boldsymbol{0}_{k\times\beta k} & \boldsymbol{0}_{k\times(M-f)\alpha} \end{array} \right].$$

$\boldsymbol{E}_{f,2}$ is obtained by a single downward cycle shift of the row vectors of $\boldsymbol{E}_{f,1}$, $\boldsymbol{E}_{f,2}$ is obtained by a single downward cycle shift of the row vectors of $\boldsymbol{E}_{f,2}$, …, and $\boldsymbol{E}_{f,k}$ is obtained by a single downward cycle shift of the row vectors of $\boldsymbol{E}_{f,k-1}$.

For the parity check nodes, we divide the last $n-k$ nodes into β number of groups of k nodes each, where all nodes in group s, i.e., nodes l where $sk+1 \leq l \leq sk+k$, receive the same query matrix.

$$\boldsymbol{E}_{f,l} = \left[\begin{array}{c|c|c} \boldsymbol{0}_{k\times(f-1)\alpha+r+(s-1)k} & \boldsymbol{I}_{k\times k} & \boldsymbol{0}_{k\times(\beta-s)k+(M-f)\alpha} \end{array} \right].$$

The remaining r parity nodes will not query for any specific stripe, and thus they receive $\boldsymbol{U}$ as the query.

After generating these patterns, $\boldsymbol{U}$ and $\boldsymbol{E}_f$, they are added together to form the query.

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{U} + \boldsymbol{E}_{f,1} \\ \boldsymbol{U} + \boldsymbol{E}_{f,2} \\ \cdots \\ \boldsymbol{U} + \boldsymbol{E}_{f,n} \end{bmatrix} \bmod q.$$

2.5.3 Retrieval

As we have constructed $\boldsymbol{Q}$, we can make the retrieval from the servers. Each $\boldsymbol{Q}_l$ will be sent to the respective server $\boldsymbol{S}_l$, which stores $\boldsymbol{w}_l$. Server $\boldsymbol{S}_l$ will compute and return a response, the response computed of $\boldsymbol{w}_l$ and the sub-queries $\boldsymbol{Q}_{l,1}, \boldsymbol{Q}_{l,2}, \ldots, \boldsymbol{Q}_{l,k}$. Each server will return with a response system, $\boldsymbol{r_l} = (\boldsymbol{r}_{l,1}, \boldsymbol{r}_{l,2}, \ldots, \boldsymbol{r}_{l,k}) = \boldsymbol{Q}_l \boldsymbol{w}_l$.

$$\begin{aligned} \boldsymbol{r}_1 &= (\boldsymbol{r}_{1,1}, \boldsymbol{r}_{1,2}, ..., \boldsymbol{r}_{l,k}) = \boldsymbol{Q}_1 \boldsymbol{w}_1 \\ \boldsymbol{r}_2 &= (\boldsymbol{r}_{2,1}, \boldsymbol{r}_{2,2}, ..., \boldsymbol{r}_{2,k}) = \boldsymbol{Q}_2 \boldsymbol{w}_2 \\ &\vdots \\ \boldsymbol{r}_n &= (\boldsymbol{r}_{n,1}, \boldsymbol{r}_{n,2}, ..., \boldsymbol{r}_{n,k}) = \boldsymbol{Q}_n \boldsymbol{w}_n \end{aligned}$$

Each of the responses contains information on $\boldsymbol{W}$. This means that we can remove the interference and decode the blocks from the responses. How to implement the decoding is not specified in TGE's paper [4].

2.6 Related Work

2.6.1 Classic Private Information Retrieval

The notion of PIR was introduced by Chor *et al.* in 1995 [3, 6]. Private information retrieval allows a user to retrieve an arbitrary file from a database, where the database could be stored on a single or over several servers, without revealing any information about the identity of the file requested. The classical PIR model explores schemes for a database of length m and a user that wants to privately retrieve the i-th bit of the database while considering the total communication cost. To retrieve something privately in an information-theoretic manner (perfect privacy), we consider the possibility that the server node has unlimited computing power and will still be unable to decode what we seek to retrieve. In the single server case, this can only be achieved by downloading the whole database. However, Chor *et al.* showed that perfect privacy is attainable with much less communication cost when the database is distributed over n non-colluding servers. As in, if the servers can under no circumstances communicate with each other, this perfect privacy is attainable. There has thus been extensive research and progress on the topic of further

reducing the communication cost of PIR. In [5], the maximum capacity of classical PIR schemes is explored and achieved. Kushilevitz *et al.* showed that replication of the database is not needed to achieve PIR and can be achieved using only a single server, assuming the hardness of the quadratic residuosity problem [7].

2.6.2 Coded Private Information Retrieval

As the classical PIR schemes had full replication of the database at n nodes, there has been done extensive research to reduce the storage overhead using coded databases in distributed storage systems (DSS), where the database is encoded using a linear code and then split up and stored on multiple servers [8–10]. This has been expanded upon by using maximum distance separable codes, as explored in [4, 11–15]. PIR schemes using arbitrary linear coded servers were explored in [16–19]. While a lot of research has been directed at reducing download cost, there has also been done research in the areas of optimal upload cost of PIR, to explore the minimum amount of required information contained in the query [20]. How many symbols needed to be accessed across all the servers (the access complexity) when privately retrieving a file was explored in [21]. The optimal download cost of PIR for arbitrary file size was explored in [22] and the trade-off between storage and download cost was explored in [23]. While classic PIR assumes no collaborating, unresponsive, or faulty server nodes, the notion that the servers might not be perfect in these aspects has also been studied. Scenarios including colluding servers has been studied in [4, 16, 18, 24–27], where robust PIR schemes for Byzantine or unresponsive servers are explored in [24, 28, 29].

2.6.3 Symmetric Private Information Retrieval

Symmetric PIR is that both the privacy of the user and the database is considered, i.e., when the database does not learn anything of what message the user is trying to retrieve and the user does not learn anything about the other messages in the database other than its requested message. This means the trivial solution of downloading the whole database is no longer applicable. Essentially symmetric PIR is a form of oblivious transfer, an important primitive block in cryptography. Symmetric PIR was explored in [30, 31].

2.6.4 Leaky/Weak Private Information Retrieval

Some researchers have also explored the idea of relaxing the perfect privacy assumption. Where applicable, relaxing the privacy could be beneficial as long as the amount of privacy is still within the user's requirement. A user might for example concede some information about whether they are actually watching a movie, rather than reading the news, if it would provide better download speed or otherwise better performance. This notion of relaxed PIR has been explored in [32–38]. The capacity of weak PIR has been studied in [39, 40] using a privacy metric related to differential privacy. The trade-offs that can be achieved by relaxing privacy have been studied in [37, 41]. Asymmetric leaky PIR which leaks privacy in both directions was studied in [35] and proposes an optimal upper bound on download cost for such schemes.

Chapter 3

Implementation Details

In this chapter, we will introduce our implementation of three schemes. We have implemented the TGE's scheme for no colluding servers, CGKS's scheme, and a direct download scheme with no privacy. We aim to provide information on our implementations to the degree that our results are replicable.

3.1 Programming Language and External Libraries

For simulating the PIR schemes, we have chosen to implement them in the programming language Python3 [42]. Python3 code is relatively easy to write and easy to read. When choosing the programming language, the factors that made us choose Python3 were our earlier experience with the language and how easy it was to read the language. The easiness of reading Python3 code was a very important factor to us as it allows for much more efficient collaboration. While other programming languages, such as Julia and Java, were discussed, we settled on a language we all had earlier experience with.

In our Python3 implementation, we make frequent use of some external libraries. External libraries are sets of functions, objects, etc., written to eliminate the need to write this code from scratch. All of these external libraries are easily imported into our implementation. We introduce some of the most critical or frequently used external libraries in our implementation and give insight into how they are exploited.

NumPy In our implementation, we frequently use the NumPy [43] library. The NumPy library is a set of mathematical operations implemented as optimized, pre-compiled C code and often performs vastly better than other mathematical implementations in Python3. We make frequent use of its functionality for its implementations of mathematical operations on matrices and vectors. This is used for computing the servers' responses and decoding the them.

SymPy SymPy [44] is a library containing functionality for symbolic mathematics that aims to become a full-featured computer algebra system. We make use of its functionality for inverting matrices.

Pillow Pillow [45] is a Python fork implementation of Python Imaging Library (PIL). PIL adds image processing capabilities to our Python interpreter, providing expanded file format support and image processing capabilities. Pillow allows us to easily import images (discussed in Section 3.4) and convert them to matrices of integers to be handled by our program.

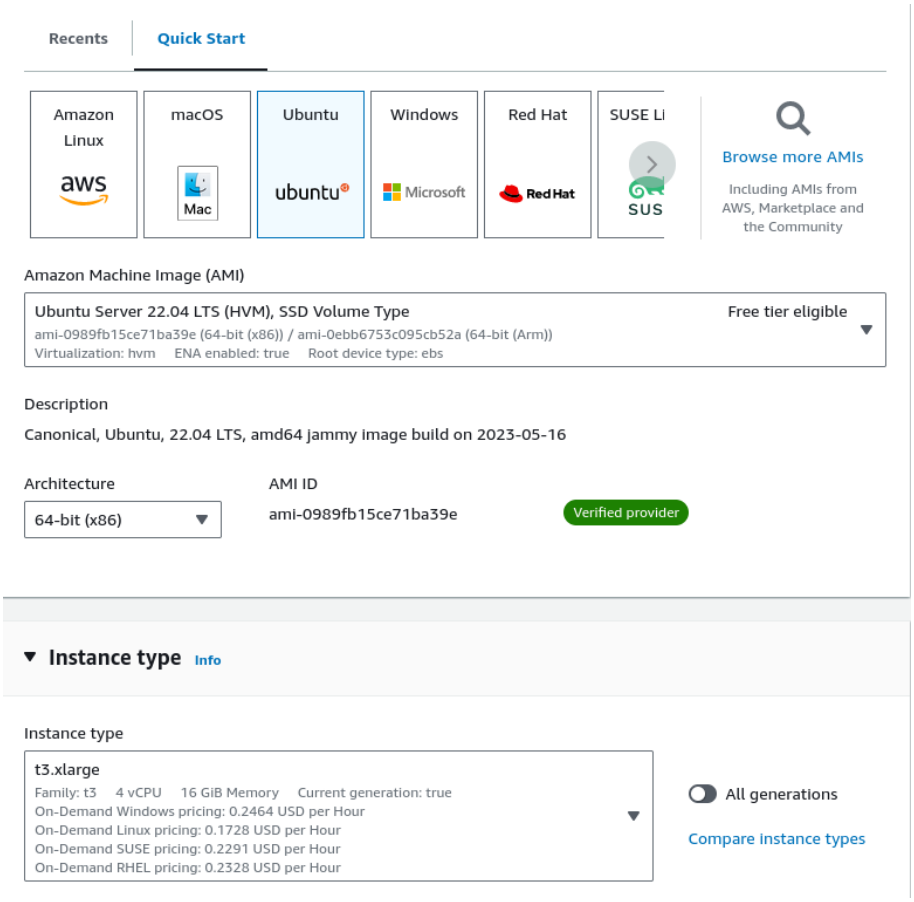

Figure 3.1: Launch properties of EC2

3.2 Amazon Web Services

Amazon Web Services (AWS) offers various web hosting services, including Elastic Compute Cloud (EC2). EC2 allows users to host virtual computers in their cloud infrastructure, called instances. AWS hosts web services in large server farms placed all around the world. In our implementation, our databases and server-side software are stored and run on EC2 instances hosted on AWS's servers in Stockholm. We have chosen the default Amazon Machine Images (AMI) for Ubuntu 22.04 LTS to launch our instances. In particular, we have used the Ubuntu AMI: ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-20230516. Our instance type is t3.xlarge, which holds 16GB of memory. For recreating our server setup, we would:

1. Create a user-account at `aws.amazon.com`.
2. Verify your email and log in to your user account.
3. Choose the region you would like your AWS services to be provided out of.
4. Navigate to the EC2 web page.
5. Launch new instance.
   (a) Create a new Amazon key-pair (RSA key).

(b) Select the same parameters when initiating instances as ours, as seen in Figure 3.1.

6. Connect to instance (through browser or ssh).

Consider:
- The default security group allows all traffic on all ports. Consider creating a new security group with fewer permissions.
- AWS will automatically generate an internet gateway for your virtual private cloud. This allows you to connect and communicate with your instances over the internet. The internet gateway has an associated cost, which is poorly explained when creating an account. When not using the service, consider detaching and deleting the internet gateway to minimize unnecessary costs.

3.3 Communication

Our program is implemented as a server-client architecture. Our client, which makes requests, is hosted by our personal computer, located at the University of Bergen, connected to the internet by wireless internet, and the server-side program is hosted on AWS's servers. Thus there is a need for some communication between our servers and our client. This is a crucial step in the implementation, as our measurement of the practical communication cost is reliant on this implementation. We have chosen the Python3 library Asyncio Streams [46]. Streams are high-level asynchronous await-ready primitives to work with network connections that allow sending and receiving data without using callbacks or low-level protocols and transports. We considered this to be the best option for this Python3 implementation, as true multi-threading is not available in Python3 because of the global interpreter lock [47], which enforces thread safety. As our readers and writers are not implemented in pure multi-threading, but asynchronously, we expect our findings in the section examining the upload speed (Section 4.2.2) and the section examining the download speed (Section 4.2.4) to be more skewed when more total data is being sent or received by our program.

3.4 Database Details

As our implementation needs a database to query, we have chosen to fill our database with images of various motives. We have a bundle of 10 images of about 1MB each, seen in Figure 3.2. These images are of dimensions 579×576 pixels, where three colors represent each pixel and where each color is represented by an 8-bit (1 byte) unsigned integer. This means that these images are each of $\approx$ 1MB.

$$579 \times 576 \times 3 = 1000512 \text{ bytes} \approx 1\text{MB}$$

Our database contains this bundle of 10 images, copied 100 times for our simulations. Thus, our unencoded database is 1000 images or $\approx$ 1GB. Our encoded database is encoded over GF(257), so we need 2 bytes to represent each encoded byte. To maintain the integrity of the data, we use the prime field GF(257) when computing responses from the servers and for our encoding/decoding, as this alphabet is large enough to cover all our color representations. This means that each byte encoded

Figure 3.2: Images used in experiments (Not to scale).

for transmission will need 2 bytes to be correctly represented. Each of our files is divided into lists of k sub-packets and then multiplied by our generator matrix of dimensions $n \times k$. As such, the size of our encoded database is

$$1\text{MB} \times 1000 \times 2 \times \frac{n}{k} = \frac{2000 \times n}{k} \text{ bytes}$$

For all of our schemes tested in this thesis, the images are encoded, computed into responses, and then reconstructed without any loss of information at the requesting user client. The whole database is loaded into memory before serving any requests. This time spent is not accounted for in our simulations.

3.5 Direct Download

The first scheme we introduce is our direct download scheme. This is a program that will download a picture from a single server. We can use this as a benchmark for how the other schemes are performing. This scheme does not have any privacy measures. The client sends a single integer, requesting the indexed file, to the server, and waits for a response.

```
import numpy as np
from random import randint
query = randint(0, 999)
reply = send_message_to_server_and_wait_for_reply(query)
```

The server receives the query and returns an image.

```
import numpy as np
query = wait_and_listen_for_a_query()
fetched_image = database[query]
flattened_image = np.asarray(fetched_image,
                    dtype=np.uint16).flatten().tobytes()
send_message_to_client(flattened_image)
```

The client gets the response it was waiting for and reconstructs the requested image. We presume the client knows of the dimension of the image it is trying to retrieve.

```
import numpy as np
import pillow as pil
```

```
reply = send_message_to_server_and_wait_for_reply(query)
flattened_image = np.frombuffer(reply, count=579*576*3,
                                dtype=np.int16)
image_as_array = np.reshape(flattened_image, (579, 576, 3))
received_pic = pil.Image.fromarray(
                        image_as_array.astype(np.uint8))
```

We have chosen the direct download scheme to encode the requested file as two bytes per byte. This decision was made to be able to make a better visual representation of a performance in the results Section 4.2.4 and put it more in line with the other schemes, as discussed in the file detail Section 3.4.

3.6 CGKS's Scheme

In this section, we detail our second scheme, CGKS's scheme. We provide a more specific example of the implementation over $n = 2$ servers and give further implementation details on each step in the protocol. The files are stored in a list of size $M = 1000$, where each index refers to a specific file. We assume the database structure is known to the client. For this scheme, an un-encoded database of 1000 images (as discussed in Section 3.4) are stored on $n = 2$ servers with full redundancy of all the data, $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$.

$$\boldsymbol{S}_1 = \begin{bmatrix} \boldsymbol{X}_0 \\ \boldsymbol{X}_1 \\ \dots \\ \boldsymbol{X}_{M-1} \end{bmatrix}, \boldsymbol{S}_2 = \begin{bmatrix} \boldsymbol{X}_0 \\ \boldsymbol{X}_1 \\ \dots \\ \boldsymbol{X}_{M-1} \end{bmatrix}$$

To retrieve a file $\boldsymbol{X}_5$, we need to generate our queries $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$. To do this we first generate a binary random vector $\boldsymbol{U} = [u_0, u_1, \dots, u_{999}]$, whose elements are chosen uniformly random as 0 or 1, with dimension equal to 1000.

```
import numpy as np
U = np.random.randint(0, 2, 1000)
```

We then copy $\boldsymbol{U}$ once so that we have a pair of equal vectors, $\boldsymbol{U}_1$ and $\boldsymbol{U}_2$. For retrieving file $\boldsymbol{X}_5$, we generate a unit vector $\boldsymbol{e}_5$ of dimension 1000 where the fifth entry is one. Our queries are then $\boldsymbol{Q}_1 = (\boldsymbol{U}_2 + \boldsymbol{e}_5) \mod 2$ and the random vector $\boldsymbol{Q}_2 = \boldsymbol{U}_1$. These queries are then sent to the servers $\boldsymbol{Q}_1 \to \boldsymbol{S}_1$ and $\boldsymbol{Q}_2 \to \boldsymbol{S}_2$. Both of the servers will then compute and return responses made from the database and the queries.

```
import numpy as np
query = wait_and_listen_for_query()
image_dimensions = (576, 579, 3)
response = np.zeros(image_dimensions, dtype = int)
for database_index, number in enumerate(query):
    if number == 1:
        response = np.add(response,
                          database_of_images[database_index])
response = (response % 257).astype(np.uint16)
flattened_response = np.asarray(response,
                          dtype=np.uint16).flatten().tobytes()
```

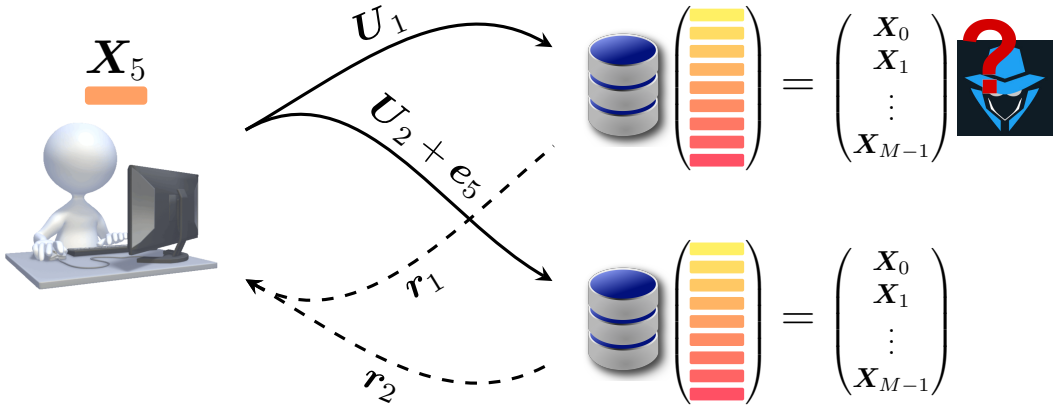

Figure 3.3: Illustration of CGKS's scheme with two servers, querying for $\boldsymbol{X}_5$

```
send_message_to_client(flattened_response)
```

When the client then receives these responses, it subtracts the response received from the query of the smallest magnitude from the one with the projection received from the query of the largest magnitude, modulo 257, to recover the file.

```
r1 = send_and_wait_for_reply_from_server(Q1)
r2 = send_and_wait_for_reply_from_server(Q2)
if sum(Q1) > sum(Q2):
    requested_image = np.subtract(r1, r2) % 257
else:
    requested_image = np.subtract(r2, r1) % 257
```

3.7 TGE's Scheme, RS$(5, 2)$

We provide a detailed example of TGE's scheme to provide insight into the implementation. We will have a look at the scheme when applied to querying a database encoded in RS$(5, 2)$ over the prime field GF(257). This means the encoded database $\boldsymbol{W}$ is split over $n = 5$ servers. Furthermore, for this example, $\alpha = 3$, and there are $M = 3$ files in the database.

3.7.1 Encoding the Database

To ensure systematic encoding of our database, we choose our generator matrix $\boldsymbol{G}$ as a Vandermonde matrix that is row reduced to row echelon form.

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 \end{bmatrix}, \quad \boldsymbol{G} = \begin{bmatrix} 1 & 0 & 256 & 255 & 254 \\ 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$

These are produced in Python3 using NumPy and SymPy.

```
import numpy as np
import sympy as sp
A = np.vander([l for l in range(n)], k,
```

```
                                    increasing=True).transpose()%257
G = np.asarray(sp.Matrix(A).rref()[0])%257
```

As per the parameters of our RS code, our files $\boldsymbol{X}$ are divided by $k = 2$ and $\alpha = 3$. Where block $\boldsymbol{x}_{1,1}$ contains the first $\frac{\omega}{6}$ bytes of the file, where ω is the number of bytes in the file, block $\boldsymbol{x}_{1,2}$ contains the next $\frac{\omega}{6}$ bytes of the file, and so on.

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_{1,1} & \boldsymbol{x}_{1,2} \\ \boldsymbol{x}_{2,1} & \boldsymbol{x}_{2,2} \\ \boldsymbol{x}_{3,1} & \boldsymbol{x}_{3,2} \end{bmatrix}$$

For this code consider $\alpha =$ ALPHA and $\omega =$ OMEGA.

```
import numpy as np
blocks = np.asarray(file).reshape(ALPHA, k, OMEGA / (ALPHA * k))
```

The files are then encoded by the $\alpha = 3$ stripes, i.e. multiplied by the generator matrix $\boldsymbol{G}$ creating the encoded database $\boldsymbol{W}$. Our encoded database $\boldsymbol{W}$ will thus have an $n \times \alpha m = 5 \times 9$ structure, where each column $\boldsymbol{w}_l$ would be stored at the respective server $\boldsymbol{S}_l$.

$$\boldsymbol{w}_1 = \begin{bmatrix} \boldsymbol{x}_{1,1} \\ \boldsymbol{x}_{2,1} \\ \boldsymbol{x}_{3,1} \\ \boldsymbol{x}_{1,1} \\ \boldsymbol{x}_{2,1} \\ \boldsymbol{x}_{3,1} \\ \boldsymbol{x}_{1,1} \\ \boldsymbol{x}_{2,1} \\ \boldsymbol{x}_{3,1} \end{bmatrix} \boldsymbol{w}_2 = \begin{bmatrix} \boldsymbol{x}_{1,2} \\ \boldsymbol{x}_{2,2} \\ \boldsymbol{x}_{3,2} \\ \boldsymbol{x}_{1,2} \\ \boldsymbol{x}_{2,2} \\ \boldsymbol{x}_{3,2} \\ \boldsymbol{x}_{1,2} \\ \boldsymbol{x}_{2,2} \\ \boldsymbol{x}_{3,2} \end{bmatrix} \boldsymbol{w}_3 = \begin{bmatrix} 256\ \boldsymbol{x}_{1,1} + 2\ \boldsymbol{x}_{1,2} \\ 256\ \boldsymbol{x}_{2,1} + 2\ \boldsymbol{x}_{2,2} \\ 256\ \boldsymbol{x}_{3,1} + 2\ \boldsymbol{x}_{3,2} \\ 256\ \boldsymbol{x}_{1,1} + 2\ \boldsymbol{x}_{1,2} \\ 256\ \boldsymbol{x}_{2,1} + 2\ \boldsymbol{x}_{2,2} \\ 256\ \boldsymbol{x}_{3,1} + 2\ \boldsymbol{x}_{3,2} \\ 256\ \boldsymbol{x}_{1,1} + 2\ \boldsymbol{x}_{1,2} \\ 256\ \boldsymbol{x}_{2,1} + 2\ \boldsymbol{x}_{2,2} \\ 256\ \boldsymbol{x}_{3,1} + 2\ \boldsymbol{x}_{3,2} \end{bmatrix},$$

$$\boldsymbol{w}_4 = \begin{bmatrix} 255\ \boldsymbol{x}_{1,1} + 3\ \boldsymbol{x}_{1,2} \\ 255\ \boldsymbol{x}_{2,1} + 3\ \boldsymbol{x}_{2,2} \\ 255\ \boldsymbol{x}_{3,1} + 3\ \boldsymbol{x}_{3,2} \\ 255\ \boldsymbol{x}_{1,1} + 3\ \boldsymbol{x}_{1,2} \\ 255\ \boldsymbol{x}_{2,1} + 3\ \boldsymbol{x}_{2,2} \\ 255\ \boldsymbol{x}_{3,1} + 3\ \boldsymbol{x}_{3,2} \\ 255\ \boldsymbol{x}_{1,1} + 3\ \boldsymbol{x}_{1,2} \\ 255\ \boldsymbol{x}_{2,1} + 3\ \boldsymbol{x}_{2,2} \\ 255\ \boldsymbol{x}_{3,1} + 3\ \boldsymbol{x}_{3,2} \end{bmatrix} \boldsymbol{w}_5 = \begin{bmatrix} 254\ \boldsymbol{x}_{1,1} + 4\ \boldsymbol{x}_{1,2} \\ 254\ \boldsymbol{x}_{2,1} + 4\ \boldsymbol{x}_{2,2} \\ 254\ \boldsymbol{x}_{3,1} + 4\ \boldsymbol{x}_{3,2} \\ 254\ \boldsymbol{x}_{1,1} + 4\ \boldsymbol{x}_{1,2} \\ 254\ \boldsymbol{x}_{2,1} + 4\ \boldsymbol{x}_{2,2} \\ 254\ \boldsymbol{x}_{3,1} + 4\ \boldsymbol{x}_{3,2} \\ 254\ \boldsymbol{x}_{1,1} + 4\ \boldsymbol{x}_{1,2} \\ 254\ \boldsymbol{x}_{2,1} + 4\ \boldsymbol{x}_{2,2} \\ 254\ \boldsymbol{x}_{3,1} + 4\ \boldsymbol{x}_{3,2} \end{bmatrix}.$$

3.7.2 Queries

We will generate a random matrix $\boldsymbol{U}$ when constructing our queries. $\boldsymbol{U}$ is a matrix of dimensions $k \times \alpha m$, where each element u is a random integer of GF(257). The random matrix $\boldsymbol{U}$ will be our interference pattern, ensuring that each server cannot deduce what file we are querying.

$$\boldsymbol{U} = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} & u_{1,5} & u_{1,6} & u_{1,7} & u_{1,8} & u_{1,9} \\ u_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & u_{2,5} & u_{2,6} & u_{2,7} & u_{2,8} & u_{2,9} \end{bmatrix}.$$

To make the retrieval of $\boldsymbol{X}_1$, we construct the retrieval pattern $\boldsymbol{E}_1$ for querying the encoded database. The retrieval pattern $\boldsymbol{E}_1$ will ensure that we will retrieve parts

of $\boldsymbol{X}_1$ from $\boldsymbol{W}$ such that $\boldsymbol{X}_1$ can be restored from the responses to the query. For our RS$(5, 2)$-code, $\beta = 1$, $r = 1$ and $\alpha = 3$.

For querying the systematic nodes, we will first generate the pattern $\boldsymbol{E}_{1,1}$:

$$\boldsymbol{E}_{1,1} = \left[\begin{array}{c:c:c:c} \boldsymbol{0}_{k\times(1-1)\alpha} & \begin{matrix}\boldsymbol{I}_{1\times1}\\ \boldsymbol{0}_{(2-1)\times1}\end{matrix} & \boldsymbol{0}_{2\times\beta2} & \boldsymbol{0}_{2\times(3-1)\alpha} \end{array}\right],$$

$$\boldsymbol{E}_{1,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

$\boldsymbol{E}_{1,2}$ is obtained by a single downward cycle shift of the row vectors of $\boldsymbol{E}_{1,1}$.

$$\boldsymbol{E}_{1,2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

For the parity check nodes, we divide the last $n-k = 5-2 = 3$ nodes into $\beta = 1$ group of k nodes. Where all nodes in group s, i.e., nodes l where $s2+1 \leq l \leq s2+2$, receive the same query matrix.

$$\boldsymbol{E}_{f,l} = \left[\begin{array}{c:c:c} \boldsymbol{0}_{2\times(1-1)3+1+(s-1)2} & \boldsymbol{I}_{2\times2} & \boldsymbol{0}_{2\times(1-s)2+(3-1)3} \end{array}\right],$$

$$\boldsymbol{E}_{1,3} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\boldsymbol{E}_{1,4} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

For the last node, it will not query for any specific stripe; hence, it will receive $\boldsymbol{U}$ as the query.

$$\boldsymbol{E}_{1,5} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

After generating these patterns, $\boldsymbol{U}$ and $\boldsymbol{E}_f$, they are added together to form the query.

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{U} + \boldsymbol{E}_{f,1} \\ \boldsymbol{U} + \boldsymbol{E}_{f,2} \\ \cdots \\ \boldsymbol{U} + \boldsymbol{E}_{f,n} \end{bmatrix} \bmod 257.$$

We add the retrieval pattern $\boldsymbol{E}$ to our random vector $\boldsymbol{U}$ and get our query $\boldsymbol{Q}$.

$$\boldsymbol{Q}_1 = \begin{bmatrix} u_{(1,1)}+1 & u_{(1,2)} & u_{(1,3)} & u_{(1,4)} & u_{(1,5)} & u_{(1,6)} & u_{(1,7)} & u_{(1,8)} & u_{(1,9)} \\ u_{(2,1)} & u_{(2,2)} & u_{(2,3)} & u_{(2,4)} & u_{(2,5)} & u_{(2,6)} & u_{(2,7)} & u_{(2,8)} & u_{(2,9)} \end{bmatrix} \bmod 257$$

$$\boldsymbol{Q}_2 = \begin{bmatrix} u_{(1,1)} & u_{(1,2)} & u_{(1,3)} & u_{(1,4)} & u_{(1,5)} & u_{(1,6)} & u_{(1,7)} & u_{(1,8)} & u_{(1,9)} \\ u_{(2,1)}+1 & u_{(2,2)} & u_{(2,3)} & u_{(2,4)} & u_{(2,5)} & u_{(2,6)} & u_{(2,7)} & u_{(2,8)} & u_{(2,9)} \end{bmatrix} \bmod 257$$

$$\boldsymbol{Q}_3 = \begin{bmatrix} u_{(1,1)} & u_{(1,2)}+1 & u_{(1,3)} & u_{(1,4)} & u_{(1,5)} & u_{(1,6)} & u_{(1,7)} & u_{(1,8)} & u_{(1,9)} \\ u_{(2,1)} & u_{(2,2)} & u_{(2,3)}+1 & u_{(2,4)} & u_{(2,5)} & u_{(2,6)} & u_{(2,7)} & u_{(2,8)} & u_{(2,9)} \end{bmatrix} \bmod 257$$

$$\boldsymbol{Q}_4 = \begin{bmatrix} u_{(1,1)} & u_{(1,2)}+1 & u_{(1,3)} & u_{(1,4)} & u_{(1,5)} & u_{(1,6)} & u_{(1,7)} & u_{(1,8)} & u_{(1,9)} \\ u_{(2,1)} & u_{(2,2)} & u_{(2,3)}+1 & u_{(2,4)} & u_{(2,5)} & u_{(2,6)} & u_{(2,7)} & u_{(2,8)} & u_{(2,9)} \end{bmatrix} \bmod 257$$

$$\boldsymbol{Q}_5 = \begin{bmatrix} u_{(1,1)} & u_{(1,2)} & u_{(1,3)} & u_{(1,4)} & u_{(1,5)} & u_{(1,6)} & u_{(1,7)} & u_{(1,8)} & u_{(1,9)} \\ u_{(2,1)} & u_{(2,2)} & u_{(2,3)} & u_{(2,4)} & u_{(2,5)} & u_{(2,6)} & u_{(2,7)} & u_{(2,8)} & u_{(2,9)} \end{bmatrix} \bmod 257$$

Retrieval

Table 1 illustrates what information is retrieved in the response of $\boldsymbol{W}$ and $\boldsymbol{E}_1$.

Blocks	$\boldsymbol{w}_1$	$\boldsymbol{w}_2$	$\boldsymbol{w}_3$	$\boldsymbol{w}_4$	$\boldsymbol{w}_5$
$\boldsymbol{x}_{(1,1)}$, $\boldsymbol{x}_{(1,2)}$	$\boldsymbol{E}_{1,1}$	$\boldsymbol{E}_{1,2}$	0	0	0
$\boldsymbol{x}_{(2,1)}$, $\boldsymbol{x}_{(2,2)}$	0	0	$\boldsymbol{E}_{1,3}$	$\boldsymbol{E}_{1,4}$	0
$\boldsymbol{x}_{(3,1)}$, $\boldsymbol{x}_{(3,2)}$	0	0	$\boldsymbol{E}_{1,3}$	$\boldsymbol{E}_{1,4}$	0
$\boldsymbol{x}_{(1,1)}$, $\boldsymbol{x}_{(1,2)}$	0	0	0	0	0
$\boldsymbol{x}_{(2,1)}$, $\boldsymbol{x}_{(2,2)}$	0	0	0	0	0
$\boldsymbol{x}_{(3,1)}$, $\boldsymbol{x}_{(3,2)}$	0	0	0	0	0
$\boldsymbol{x}_{(1,1)}$, $\boldsymbol{x}_{(1,2)}$	0	0	0	0	0
$\boldsymbol{x}_{(2,1)}$, $\boldsymbol{x}_{(2,2)}$	0	0	0	0	0
$\boldsymbol{x}_{(3,1)}$, $\boldsymbol{x}_{(3,2)}$	0	0	0	0	0

Table 3.1: Retrieval of $\boldsymbol{X}$, of an MDS(5,2,4) encoded database, where $M = 3$.

As we have constructed $\boldsymbol{Q}$, we can make the retrieval from the servers. Each $\boldsymbol{Q}_l$ will be sent to the respective server $\boldsymbol{S}_l$, which stores $\boldsymbol{w}_l$. Server $\boldsymbol{S}_l$ will compute and return a response, the responses of $\boldsymbol{w}_l$ and the sub-queries $\boldsymbol{Q}_{l,1}$ and $\boldsymbol{Q}_{l,2}$. For this code detailing the server-side implementation, computing the responses, consider $\alpha = \text{ALPHA}$, $\omega = \text{OMEGA}$ and $\boldsymbol{w}_l = \text{wl}$.

```
import numpy as np
import sympy as sp

query = wait_and_listen_for_query()
stripe_size = OMEGA / (ALPHA * k)

def make_response(query):
    response_li = np.zeros(stripe_size, dtype=int)
    for j, qj in enumerate(query):
            stripe = wl[j * stripe_size :
                        (j + 1) * stripe_size]
            response_li = np.add(response_li,
                                 np.multiply(stripe, qj))
    return response_li % 257

query = np.asarray(query).reshape(k, ALPHA * M)
responses = [make_response(query[i])
                        for i in range(k)]
flattened_responses = np.asarray(responses,
                    dtype=np.uint16).flatten().tobytes()
```

```
send_back_to_client(flattened_responses)
```

Each server will return with a response system, $\boldsymbol{r}_l = (\boldsymbol{r}_{l1}, \boldsymbol{r}_{l2}) = \boldsymbol{Q}_l \boldsymbol{w}_l$.

$$\begin{aligned}
\boldsymbol{r}_1 &= (\boldsymbol{r}_{1,1}, \boldsymbol{r}_{1,2}) = \boldsymbol{Q}_1 \boldsymbol{w}_1, \\
\boldsymbol{r}_2 &= (\boldsymbol{r}_{2,1}, \boldsymbol{r}_{2,2}) = \boldsymbol{Q}_2 \boldsymbol{w}_2, \\
\boldsymbol{r}_3 &= (\boldsymbol{r}_{3,1}, \boldsymbol{r}_{3,2}) = \boldsymbol{Q}_3 \boldsymbol{w}_3, \\
\boldsymbol{r}_4 &= (\boldsymbol{r}_{4,1}, \boldsymbol{r}_{4,2}) = \boldsymbol{Q}_4 \boldsymbol{w}_4, \\
\boldsymbol{r}_5 &= (\boldsymbol{r}_{5,1}, \boldsymbol{r}_{5,2}) = \boldsymbol{Q}_5 \boldsymbol{w}_5.
\end{aligned}$$

We can reorganize the responses by transposing them. Instead of 5×2 responses, we can turn them into 2×5 sub-responses.

$$\boldsymbol{r}_1^T = \begin{bmatrix} \boldsymbol{r}_{1,1} \\ \boldsymbol{r}_{2,1} \\ \boldsymbol{r}_{3,1} \\ \boldsymbol{r}_{4,1} \\ \boldsymbol{r}_{5,1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{q}_{1,1}\boldsymbol{w}_1 \\ \boldsymbol{q}_{2,1}\boldsymbol{w}_2 \\ \boldsymbol{q}_{3,1}\boldsymbol{w}_3 \\ \boldsymbol{q}_{4,1}\boldsymbol{w}_4 \\ \boldsymbol{q}_{5,1}\boldsymbol{w}_5 \end{bmatrix}, \boldsymbol{r}_2^T = \begin{bmatrix} \boldsymbol{r}_{1,2} \\ \boldsymbol{r}_{2,2} \\ \boldsymbol{r}_{3,2} \\ \boldsymbol{r}_{4,2} \\ \boldsymbol{r}_{5,2} \end{bmatrix} = \begin{bmatrix} \boldsymbol{q}_{1,2}\boldsymbol{w}_1 \\ \boldsymbol{q}_{2,2}\boldsymbol{w}_2 \\ \boldsymbol{q}_{3,2}\boldsymbol{w}_3 \\ \boldsymbol{q}_{4,2}\boldsymbol{w}_4 \\ \boldsymbol{q}_{5,2}\boldsymbol{w}_5 \end{bmatrix}.$$

This means that $\boldsymbol{r}_1^T$ is a function of $\boldsymbol{W}$ and the first sub-queries, and $\boldsymbol{r}_2^T$ is a function of $\boldsymbol{W}$ and the second sub-queries.

3.7.3 Decoding

As we know that each of the responses in the i-th sub-response system contains each of the columns in the encoded database computed by the i-th sub-query, we can make k inverse linear sub-response systems to solve for the blocks in each of them. To solve for these blocks, we can make an inverse linear sub-response system, which can be generated as follows. We will construct k $n \times n$ matrices containing the encoding and retrieval patterns. For each of them, the leftmost k columns are for solving for the interference. A transposed generator matrix will represent them.

$$\boldsymbol{G} = \begin{bmatrix} 1 & 0 & 256 & 255 & 254 \\ 0 & 1 & 2 & 3 & 4 \end{bmatrix}, \boldsymbol{G}^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 256 & 2 \\ 255 & 3 \\ 254 & 4 \end{bmatrix}.$$

The next $n - k$ columns represent the retrieval pattern in the i-th sub-queries. As we have k sub-response systems, each will contain α blocks. We generate a retrieval pattern $\boldsymbol{E}_{1,1}$ of our MDS(n, k) code, retrieving the first file of a database containing one item. We restructure this, by swapping the first and second axis, $\boldsymbol{E}_{1,1}^{(1,0,2)}$.

```
import numpy as np
retrieval_pattern = np.transpose(retrieval_pattern, (1, 0, 2))
```

$$\boldsymbol{E}_{1,1} = \begin{bmatrix} [[1,0,0],[0,0,0]] \\ [[0,0,0],[1,0,0]] \\ [[0,1,0],[0,0,1]] \\ [[0,1,0],[0,0,1]] \\ [[0,0,0],[0,0,0]] \end{bmatrix}, \boldsymbol{E}_{(1,1)}^{(1,0,2)} = \begin{bmatrix} [1,0,0] \\ [0,0,0] \\ [0,1,0] \\ [0,1,0] \\ [0,0,0] \end{bmatrix}, \begin{bmatrix} [0,0,0] \\ [1,0,0] \\ [0,0,1] \\ [0,0,1] \\ [0,0,0] \end{bmatrix}$$

For the next α columns, we will differentiate between the systematic nodes and the parity check nodes, and we will explain them row by row instead. The first k rows will be the first k rows in $\boldsymbol{E}_{1,1}^{(1,0,2)}$. The last α rows of the last α columns will be the last non-zero columns of $\boldsymbol{E}_{1,1}^{(1,0,2)}$ multiplied by $\boldsymbol{G}^T$ excluding the first k rows.

$$\boldsymbol{L} = \left[\left[\begin{array}{ccc:cc} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 256 & 2 & 0 & 256 & 2 \\ 255 & 3 & 0 & 255 & 3 \\ 254 & 4 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{ccc:cc} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 256 & 2 & 0 & 256 & 2 \\ 255 & 3 & 0 & 255 & 3 \\ 254 & 4 & 0 & 0 & 0 \end{array}\right]\right]$$

The modular inverse of $\boldsymbol{L}$ will be key to retrieving our blocks.

```
import numpy as np
import sympy as sp
L_modular_inverse = [np.asarray(sp.Matrix(Li).inv_mod(257),
                                dtype=int) for Li in L]
```

$$\boldsymbol{L}^{-1} \mod 257 = \left[\left[\begin{array}{ccccc} 0 & 87 & 0 & 0 & 171 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 170 & 0 & 0 & 86 \\ 0 & 170 & 3 & 255 & 86 \\ 0 & 256 & 2 & 256 & 0 \end{array}\right], \left[\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ 65 & 0 & 0 & 0 & 193 \\ 192 & 1 & 0 & 0 & 64 \\ 256 & 0 & 3 & 255 & 0 \\ 192 & 0 & 2 & 256 & 64 \end{array}\right]\right]$$

By multiplying $\boldsymbol{L}_i^{-1}$ with the i-th sub-response linear system, $\boldsymbol{r}_i^T$, we successfully solve for the blocks contained in the sub-response. Now consider the sub-responses of the 5 nodes to the first sub-query. Where $\boldsymbol{I}_l = \boldsymbol{u}_1^T \boldsymbol{w}_l$, where $l = 1, 2$ and $\boldsymbol{u}_1^T$ is the first sub-query (the first row of $\boldsymbol{U}$), is the interference.

$$\begin{aligned} \boldsymbol{r}_{11} &= \boldsymbol{I}_1 + \boldsymbol{x}_{11} \\ \boldsymbol{r}_{21} &= \boldsymbol{I}_2 \\ \boldsymbol{r}_{31} &= 256\boldsymbol{I}_1 + 2\boldsymbol{I}_2 + 256\boldsymbol{x}_{21} + 2\boldsymbol{x}_{22} \\ \boldsymbol{r}_{41} &= 255\boldsymbol{I}_1 + 3\boldsymbol{I}_2 + +255\boldsymbol{x}_{21} + 3\boldsymbol{x}_{22} \\ \boldsymbol{r}_{51} &= 254\boldsymbol{I}_1 + 4\boldsymbol{I}_2 \end{aligned}$$

$$\begin{bmatrix} 0 & 87 & 0 & 0 & 171 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 170 & 0 & 0 & 86 \\ 0 & 170 & 3 & 255 & 86 \\ 0 & 256 & 2 & 256 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{I}_1 + \boldsymbol{x}_{11} \\ \boldsymbol{I}_2 \\ 256\boldsymbol{I}_1 + 2\boldsymbol{I}_2 + 256\boldsymbol{x}_{21} + 2\boldsymbol{x}_{22} \\ 255\boldsymbol{I}_1 + 3\boldsymbol{I}_2 + +255\boldsymbol{x}_{21} + 3\boldsymbol{x}_{22} \\ 254\boldsymbol{I}_1 + 4\boldsymbol{I}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}_1 \\ \boldsymbol{I}_2 \\ \boldsymbol{x}_{11} \\ \boldsymbol{x}_{21} \\ \boldsymbol{x}_{22} \end{bmatrix}$$

Now let us consider the sub-responses of the 5 nodes to the second sub-query. Where $\boldsymbol{I}_l = \boldsymbol{u}_2^T \boldsymbol{w}_l$, where $l = 1, 2$ and $\boldsymbol{u}_2^T$ is the second sub-query (the second row of $\boldsymbol{U}$), is the interference.

$$\begin{aligned} \boldsymbol{r}_{12} &= \boldsymbol{I}_1 \\ \boldsymbol{r}_{22} &= \boldsymbol{I}_2 + \boldsymbol{x}_{12} \\ \boldsymbol{r}_{32} &= 256\boldsymbol{I}_1 + 2\boldsymbol{I}_2 + 256\boldsymbol{x}_{31} + 2\boldsymbol{x}_{32} \end{aligned}$$

$$\boldsymbol{r}_{42} = 255\boldsymbol{I}_1 + 3\boldsymbol{I}_2 + +255\boldsymbol{x}_{31} + 3\boldsymbol{x}_{32}$$
$$\boldsymbol{r}_{52} = 254\boldsymbol{I}_1 + 4\boldsymbol{I}_2$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 65 & 0 & 0 & 0 & 193 \\ 192 & 1 & 0 & 0 & 64 \\ 256 & 0 & 3 & 255 & 0 \\ 192 & 0 & 2 & 256 & 64 \end{bmatrix} \begin{bmatrix} \boldsymbol{I}_1 \\ \boldsymbol{I}_2 + \boldsymbol{x}_{12} \\ 256\boldsymbol{I}_1 + 2\boldsymbol{I}_2 + 256\boldsymbol{x}_{31} + 2\boldsymbol{x}_{32} \\ 255\boldsymbol{I}_1 + 3\boldsymbol{I}_2 + +255\boldsymbol{x}_{31} + 3\boldsymbol{x}_{32} \\ 254\boldsymbol{I}_1 + 4\boldsymbol{I}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}_1 \\ \boldsymbol{I}_2 \\ \boldsymbol{x}_{12} \\ \boldsymbol{x}_{31} \\ \boldsymbol{x}_{32} \end{bmatrix}$$

This leaves us with:

$$\begin{bmatrix} \boldsymbol{I}_1 \\ \boldsymbol{I}_2 \\ \boldsymbol{x}_{11} \\ \boldsymbol{x}_{21} \\ \boldsymbol{x}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{I}_1 \\ \boldsymbol{I}_2 \\ \boldsymbol{x}_{12} \\ \boldsymbol{x}_{31} \\ \boldsymbol{x}_{32} \end{bmatrix}$$

Of this new vector, we can simply disregard the first k elements, as they are interference. We are now left with k vectors of α blocks.

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{x}_{11} \\ \boldsymbol{x}_{21} \\ \boldsymbol{x}_{22} \end{bmatrix}, \begin{bmatrix} \boldsymbol{x}_{12} \\ \boldsymbol{x}_{31} \\ \boldsymbol{x}_{32} \end{bmatrix}$$

But we have to sort them correctly to recreate the file. We begin by generating a new retrieval pattern $\boldsymbol{E}_{1,1}$ of our MDS(n, k) code, retrieving one file of a database containing one item. We restructure this by swapping the first and second axis, and then swapping the second and third axis, $\boldsymbol{E}_{1,1}^{(1,2,0)}$.

```
import numpy as np
retrieval_pattern = np.transpose(retrieval_pattern, (1, 2, 0))
```

$$\boldsymbol{E}_{1,1}^{(1,2,0)} = \begin{bmatrix} 1,0,0,0,0 \\ 0,0,1,1,0 \\ 0,0,0,0,0 \end{bmatrix} \begin{bmatrix} 1,0,0,0,0 \\ 0,0,0,0,0 \\ 0,0,1,1,0 \end{bmatrix}$$

We sort the blocks $\boldsymbol{B}$ by applying the sorting Algorithm 1. $\boldsymbol{B}$ is now in the correct order and correctly represents the file we queried for.

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_{11} \\ \boldsymbol{x}_{12} \\ \boldsymbol{x}_{21} \\ \boldsymbol{x}_{22} \\ \boldsymbol{x}_{31} \\ \boldsymbol{x}_{32} \end{bmatrix}$$

Algorithm 1 An algorithm to sort blocks

$X \leftarrow []$ ▷ This is an empty list
$Xcounter \leftarrow [0] * k$
for `row in range` α **do**

```
for col in range n do

    for i in range k do

        if E_{1,1}^{(1,2,0)}[i][row][col] == 1 then
            X.append(B[i][Xcounter[i]])
            Xcounter[i]+ = 1
        end if
        end for
    end for
end for
```

Chapter 4

Findings

4.1 Considerations

During our experiments, we observed various unexpected behavior in our findings. We hope to bring some more light onto why these various strange behaviors exist and how they impact our findings.

4.1.1 Local Time Problem

During our experimentation, our servers and our client were quite far apart. We have observed some strange behavior when trying to measure the time spent when sending information between them. In some instances, we could observe in our findings that, somehow, the data was received before it was sent. This is, of course, impossible. We hypothesized that this was because the local clock on the server differed from the local clock of the client by amounts larger than the time spent sending the data. In search of a solution to remedy this problem, we found the Chrony. Chrony [48] is an NTP (Network Time Protocol) time-synchronizing daemon. This is used by Facebook to synchronize the time of their servers. [49] [50] Implementing Chrony into our simulation, we could no longer observe this strange behavior in our findings, but as the synchronizing of time between servers still remains a problem, we still consider this behavior when interpreting the findings in Section 4.2.2 and Section 4.2.4.

4.1.2 Averaged Statistics

Some of our plotted data are CDFs of average time spent per server. For each single iteration, the data collected is summed and divided by the number of servers. This makes for way better readability of the plots and makes comparisons easier. This does come at the cost of accuracy, however. If any single one of the servers is subject to any performance issues and performs worse than expected, they will skew the plot, creating an illusion of a worse-performing protocol.

4.2 Results

In this section, we present our findings gathered from our simulation, implemented as presented in Chapter 3. The findings in this section are gathered from simulations, simulating PIR on a database as discussed in Section 3.4. For parameters for our schemes, we have chosen to analyze the CGKS scheme with 2 and 3 servers. For the TGE scheme, we have chosen the parameters $(5, 2)$, $(5, 3)$, and $(5, 4)$ to examine schemes with different rates. We also chose the parameters $(4, 2)$, $(6, 3)$, and $(8, 4)$, to examine half-rate codes with different amounts of servers. This gives us a wide selection of rates to examine, while also allowing us to examine TGE's scheme with a fixed rate over different numbers of servers. We simulate CGKS's scheme using the parameters $n = 2$ and $n = 3$. This will allow us to compare the two PIR schemes. We compare the simulations of CGKS's scheme and TGE's scheme to our simulation of direct download.

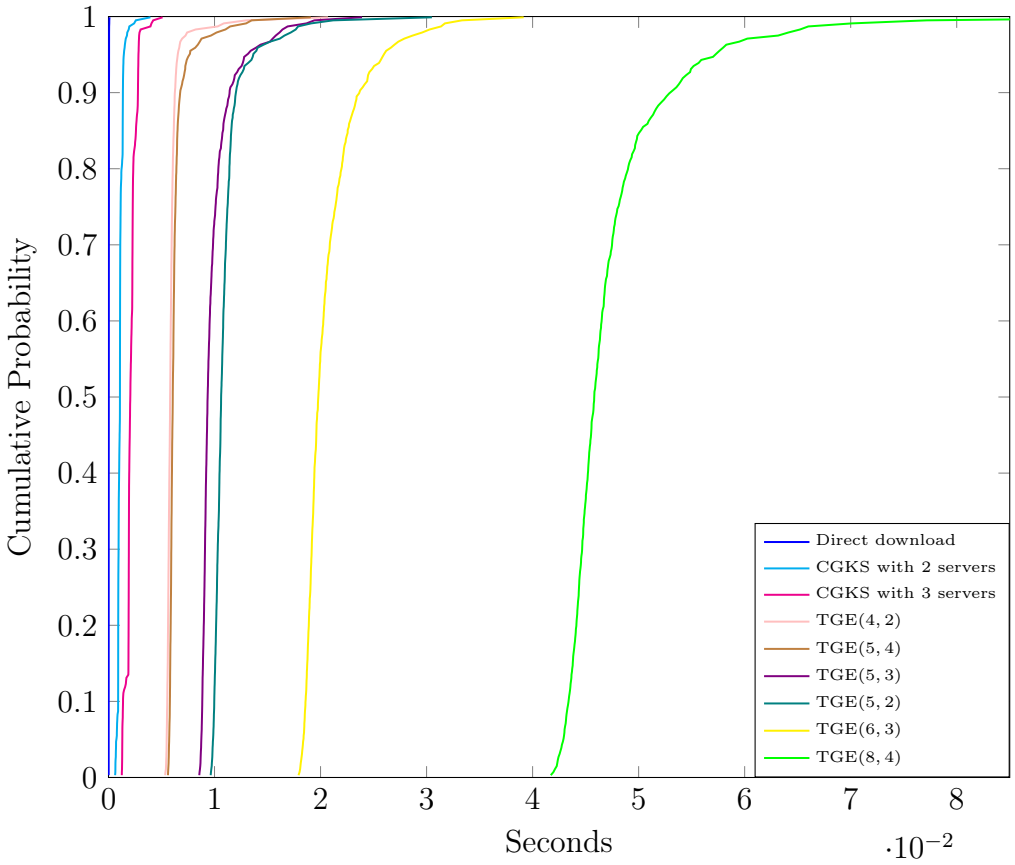

Figure 4.1: CDF of: Time spent generating queries. Querying 1 random file of a database of 1000 files. Files of 1MB. 1000 iterations. Labels according to performance.

4.2.1 Generating Queries

We expect the time spent creating the query to be related to how large the query is. The direct download scheme queries for a random entry in the database, and thus only generates a single random integer and might be hard to see in Figure 4.1. The CGKS scheme with 2 servers generates a vector of length M, and the CGKS scheme with 3 servers generates a vector of length $2M$. The TGE scheme generates a randomized matrix of size $k \times \alpha M$. Then it constructs the retrieval pattern of dimensions $n \times k \times \alpha M$ and adds the randomized matrix on top of the retrieval pattern. By comparing Table 4.1 and Figure 4.1 we argue our findings seem to support our expectation, of how the time spent creating the queries, is related to their size. The time spent generating the queries seems quite negligible. Even the worst performing, TGE$(8, 4)$, is on average computing a query in about 0.045 seconds, as observed in Figure 4.1. We argue this will not be noticeable to an average user.

Protocol	Size of interference	Size of retrieval pattern
Direct download	0	One random integer
CGKS with 2 servers	M	1 byte
CGKS with 3 servers	$2M$	2 bytes
TGE$(4, 2)$	$4M$	$16M$
TGE$(5, 4)$	$4M$	$20M$
TGE$(5, 3)$	$6M$	$30M$
TGE$(5, 2)$	$6M$	$30M$
TGE$(6, 3)$	$9M$	$54M$
TGE$(8, 4)$	$16M$	$128M$

Table 4.1: Size of a query, by protocol.

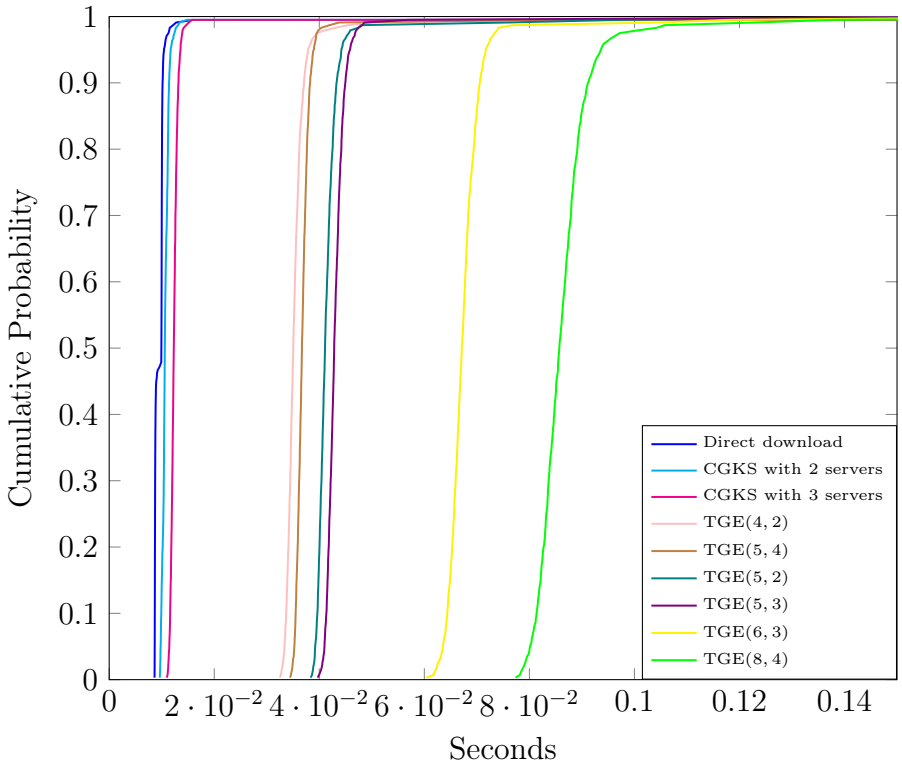

Figure 4.2: CDF of: Per server average time spent sending a query to a server. Querying 1 random file of a database of 1000 files. Files of 1MB. 1000 iterations. Labels according to average performance.

4.2.2 Upload

How much time each protocol spends uploading data is affected by how much data is sent, the physical distance between the client and the servers, and the infrastructure connecting the client to the servers. We argue, that as the physical distance between our client computer and our servers does not change during our simulations, and as the infrastructure is not altered (as we know of), the time spent sending the queries to the servers is directly related to the size of each query contains. We expect the time spent on uploading the queries for each protocol to be directly related to how much data is sent to each single server.

By comparing our findings in Figure 4.2 to the number of bytes sent to each server, as seen in Table 4.2, we argue that our expectations are close to our findings. We argue there might be deviation as discussed as the time problem in Section 4.1.1 and because sending and receiving data is also affected by how much total data is sent, as discussed in the Asynchronous python3 problem in Section 3.3.

Protocol	Bytes sent to each server	Bytes sent to all servers
Direct download	2	2
CGKS with 2 servers	M	$2M$
CGKS with 3 servers	$2M$	$6M$
TGE$(4, 2)$	$4M$	$16M$
TGE$(5, 4)$	$6M$	$20M$
TGE$(5, 2)$	$6M$	$30M$
TGE$(5, 3)$	$6M$	$30M$
TGE$(6, 3)$	$9M$	$54M$
TGE$(8, 4)$	$16M$	$128M$

Table 4.2: Bytes sent by each protocol

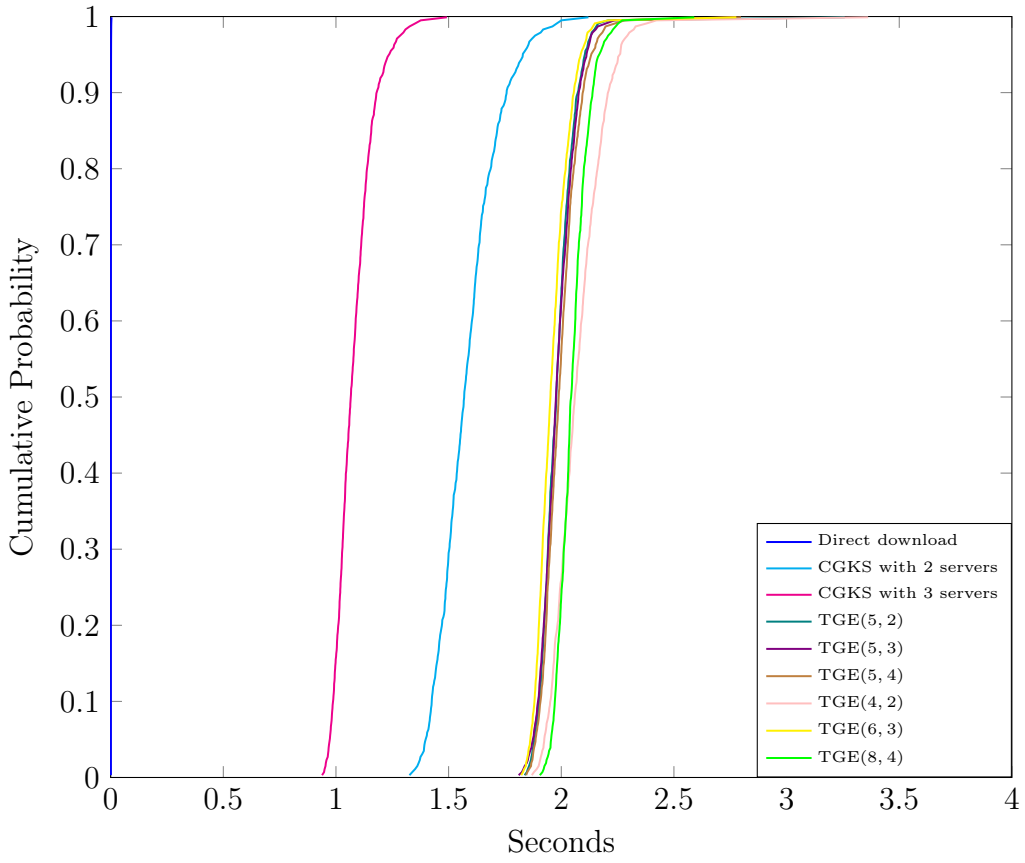

Figure 4.3: CDF of: Server average time spent on computing reply. Querying 1 random file of a database of 1000 files. Files of 1MB. 1000 iterations. Direct download omitted.

4.2.3 Server Computation

We argue the time spent on server computation is related to the workload on the server. For the direct download scheme, the workload on the server amounts to retrieving a file from an indexed database once and sending the file. On average, the time spent for a server computing for this scheme is 0.001239 seconds.

For CGKS's scheme, the workload consists of computing the response of the database onto the query. The CGKS scheme with 3 servers has double the amount of files in the database, but the file sizes are half the size of those in the CGKS scheme with 2 servers, they would both have the same computational complexity. As the queries are uniform random binary vectors, the expected number of 1's, signaling to include that indexed file in the response, would be $\frac{1}{2}M$. This means that the workload is to compute a response of $\frac{1}{2}M$ Megabyte (MB), or $\frac{1}{2}$ Gigabyte (GB).

In the TGE scheme, each server has to compute the response of $\boldsymbol{w}_l$ onto $\boldsymbol{Q}_l$. $\boldsymbol{W}$ is the encoded database of M 1MB files, and thus is of size $\frac{n}{k}M$ MB. $\boldsymbol{w}_l$ is thus of size $\frac{1}{k}M$ MB. The query $\boldsymbol{Q}_l$ is a composite of k sub-queries $\boldsymbol{q}_{l,i}$, where the server needs to compute the response of $\boldsymbol{w}_l$ onto each of the k sub-queries. This means that the workload of the TGE scheme is to compute a response of M MB, or one GB.

We can observe in Figure 4.3 that the TGE schemes are clustered around the

Protocol	Workload
Direct download	0
CGKS with 2 servers	Response of $\frac{1}{2}M$ MB
CGKS with 3 servers	Response of $\frac{1}{2}M$ MB
TGE$(4, 2)$	Response of M MB
TGE$(5, 2)$	Response of M MB
TGE$(5, 3)$	Response of M MB
TGE$(5, 4)$	Response of M MB
TGE$(6, 3)$	Response of M MB
TGE$(8, 4)$	Response of M MB

Table 4.3: Workload of a single server in a protocol.

2 seconds mark, and have little variation in performance. We observe in Figure 4.3 that the CGKS scheme with 3 servers performs at about 1 second on average and the CGKS scheme performs at about 1.5 seconds on average. Our expectation was for the two CGKS schemes to be clustered, as they seemingly have the same workload. We attribute the worse performance of the CGKS scheme with 2 servers, compared to the CGKS scheme with 3 servers, to larger file sizes to load into the NumPy array data structure.

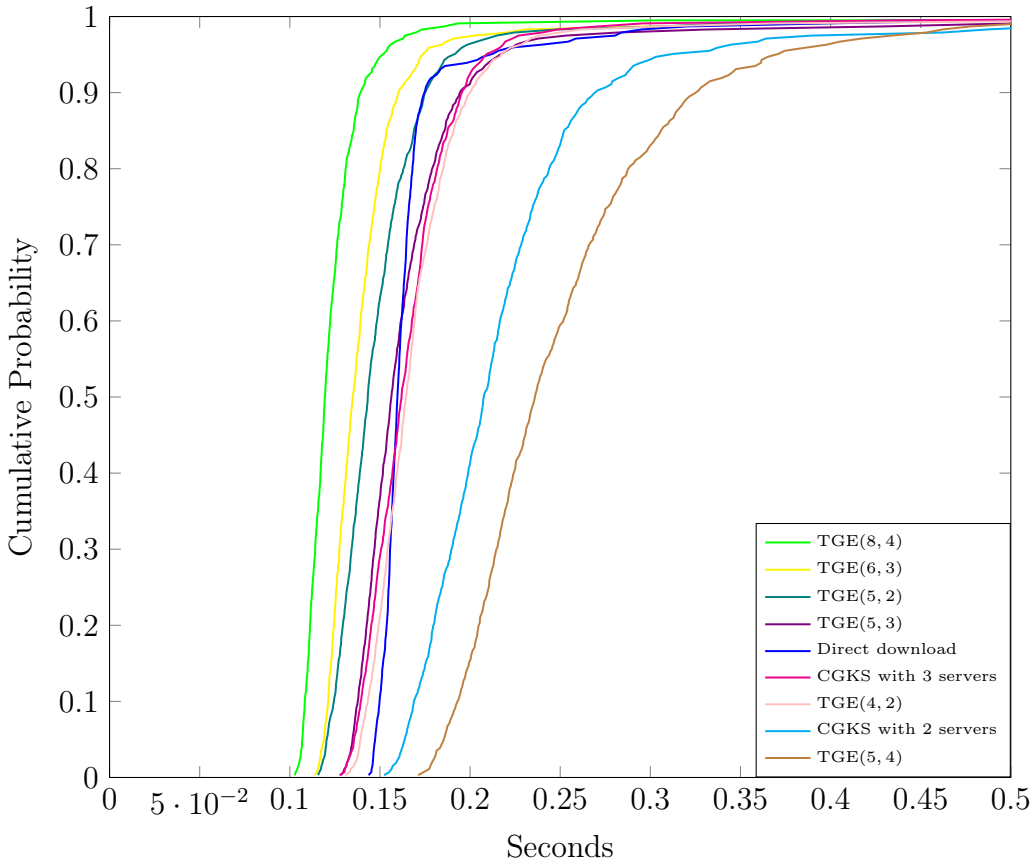

Figure 4.4: CDF of: Average time spent on downloading a reply from a server. Querying 1 random file of a database of 1000 files. Files of 1MB. 1000 iterations. Labels according to average performance.

4.2.4 Download

As we argue in Section 4.2.2, we assume our findings should be related to the number of bytes downloaded. Note that all of our download transmissions are encoded as 2 bytes per byte, as discussed in Section 3.4. The direct download scheme downloads 2 times the number of bytes in a file, 2ω, bytes. The CGKS schemes download $2\frac{\omega}{\sigma}$ per server, and $2\frac{n\omega}{\sigma}$ in total from all the servers. The amount of bytes sent by the servers in the TGE scheme is

$$\text{The number of bytes sent by each single server} = 2\frac{\omega}{\alpha}$$

$$\text{The number of bytes downloaded from all servers} = 2\frac{n\omega}{\alpha}$$

Based on this, we would expect the performance of the different protocols in our findings to reflect a ranking of performance similar to Table 4.4, which is the protocols ranked by how much data is downloaded from each server in the protocol. Our findings presented in Figure 4.4 are prone to the averaged data problem discussed in Section 4.1.2 and to the local time problem discussed in Section 4.1.1. We observe that our findings do not match our expectations. In our findings in Figure 4.4, only TGE$(8, 4)$, TGE$(5, 4)$, and CGKS with 2 servers perform according to our expected ranking. We argue the performance of the protocols to also be influenced by the to-

Protocol	Bytes downloaded from one server divided by two	Bytes downloaded from all servers divided by two
TGE$(8,4)$	$\frac{1}{4}\omega$	2ω
TGE$(5,2)$	$\frac{1}{3}\omega$	$\frac{5}{3}\omega$
TGE$(6,3)$	$\frac{1}{3}\omega$	2ω
CGKS with 3 servers	$\frac{1}{2}\omega$	$\frac{3}{2}\omega$
TGE$(4,2)$	$\frac{1}{2}\omega$	2ω
TGE$(5,3)$	$\frac{1}{2}\omega$	$\frac{5}{2}\omega$
Direct download	ω	ω
CGKS with 2 servers	ω	2ω
TGE$(5,4)$	ω	5ω

Table 4.4: Bytes downloaded from servers divided by two

tal amount of bytes downloaded, as can be explained by the lack of multi-threading in our implementation as discussed in Section 3.3.

Protocol	Bytes downloaded from all servers divided by two
TGE$(5, 2)$	$\frac{5}{3}\omega$
TGE$(8, 4)$	2ω
TGE$(6, 3)$	2ω
TGE$(4, 2)$	2ω
TGE$(5, 3)$	$\frac{5}{2}\omega$
TGE$(5, 4)$	5ω

Table 4.5: Total number of bytes downloaded from all servers in protocol divided by two.

4.2.5 Client Computation

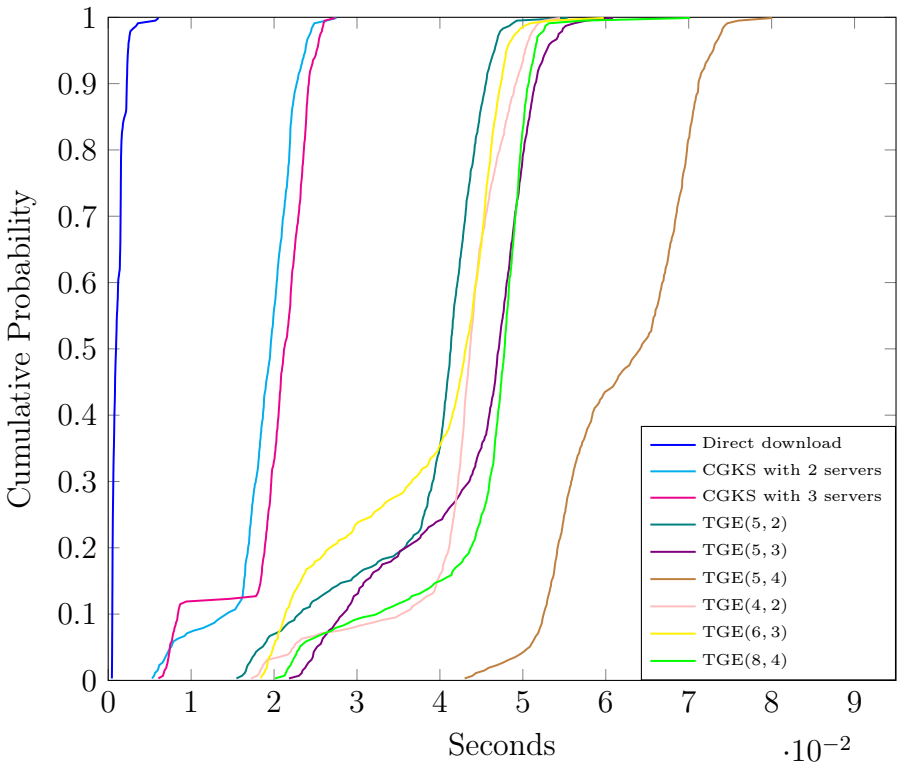

Figure 4.5: CDF of: Time spent reconstructing and decoding reply from servers. Querying 1 random file of a database of 1000 files. Files of 1MB. 1000 iterations.

After downloading the replies from the servers, the simulation reconstructs the file. Our findings presented in Figure 4.5 show the time each protocol spends after fully downloading the replies until the file is fully reconstructed. For the direct download scheme, the workload consists of restructuring an array to match the shape of the image, and then converting said array to a .png file, and this workload also affects the other schemes.

For the CGKS scheme, the process of retrieving the file from the responses is to subtract the interference from the response(s) and then do a modulus operation on all the data. For the TGE scheme, the workload consists of decoding and restructuring the data. We expect this workload to be dependent on the amount of bytes in

the replies. In our findings in Figure 4.5 we can observe that our findings do not match our expectations. For TGE$(5, 2)$, TGE$(5, 3)$, and TGE$(5, 4)$, our expectations seem to be correct, but our expectations seem to be flawed when observing the performance of the different half-rate codes, TGE$(4, 2)$, TGE$(6, 3)$, and TGE$(8, 4)$.

Based on our observations in Figure 4.5, we alter our assumptions. We speculate the decoding is reliant on the specific contents of our files. As we detail in Section 3.4, there are 10 different images in our database, although the database is filled with 1000 of them. We speculate the decoding of specific images takes different amounts of time, explaining the skewed performance of all the schemes.

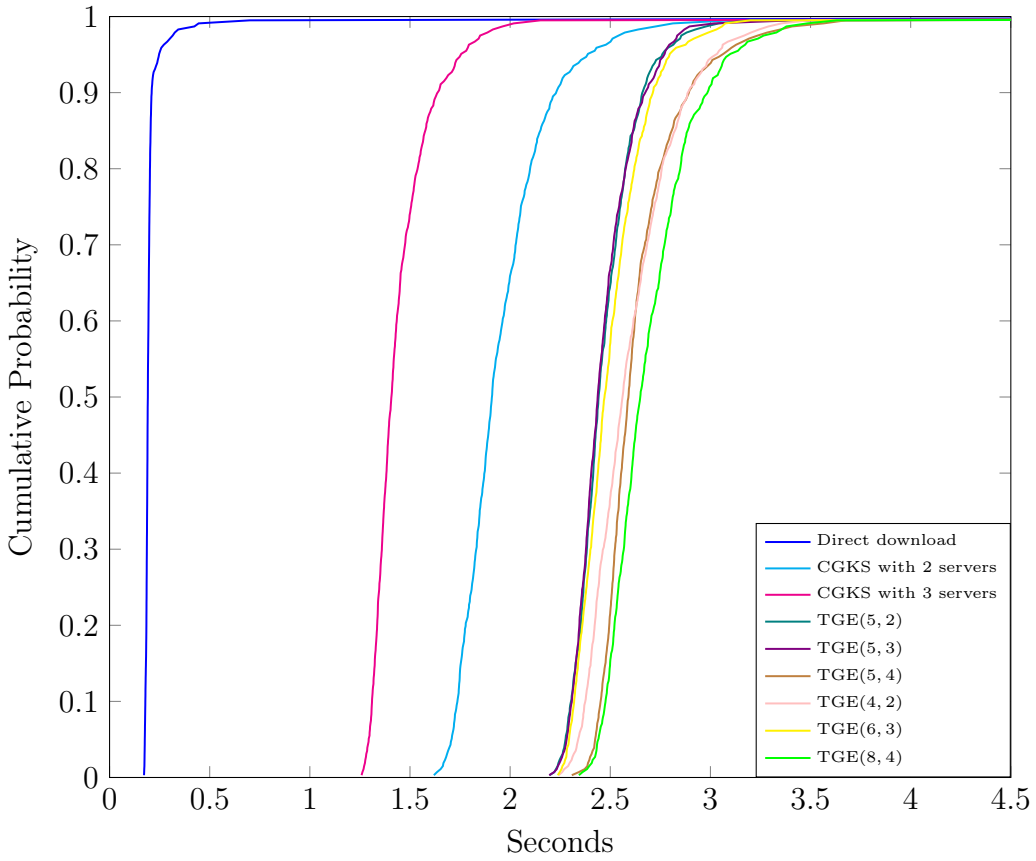

Figure 4.6: CDF of: Total time spent retrieving a file from server(s). Querying 1 random file of a database of 1000 files. Files of 1MB. 1000 iterations.

4.2.6 Total Time Spent

For the total time spent, throughout all of the sections of the protocols, we expected the direct download scheme with no privacy measures to outperform the TGE scheme and the CGKS scheme. We also assumed TGE's scheme to perform better than CGKS's scheme, based on the theoretical optimal download rate of the former. In our findings in Figure 4.6, we observe, that while the direct download scheme performs as expected, TGE performs worse than CGKS. For the PIR schemes, we expected the time spent on downloading and decoding to be the time frames of more significance in terms of limiting the performance of the protocols as a whole. The time spent for each server computing a reply was a larger part of the total time, and we conclude the time spent computing the replies to be the largest contributing factor to the total time spent per protocol. On this note, it is important to emphasize that the time cost of the server computation is expected to grow linearly with the number of files in the database.

Chapter 5

Conclusion

The initial objective of this thesis was to study the performance of a simulation of two PIR protocols, namely a protocol by Tajeddine *et al.* and a protocol by Chor *et al.*, and a direct download scheme with no privacy guarantee, implemented on AWS's servers. In the first part of this thesis, we detail the different schemes and how we have implemented them. We provide details on our simulation environment, and remark how our implementation affects our simulations. In the second part of this thesis, we provide the findings of our simulations and remark how our findings make us question and adjust our expectations. We conclude the time spent computing the responses for the PIR protocols was proportionally longer than we anticipated and conclude this is the main bottleneck in relation to the total time spent when doing a private retrieval. Our thesis details the many unforeseen challenges we have met when trying to simulate PIR. Doing a thorough review of all the possible sources affecting the time spent transmitting data over the internet has been a great challenge, and the list of issues we have encountered is possibly incomplete. The lack of pure multi-threading in Python3 has proved to be a large source of frustration when transmitting data and implementing server-side algorithms. We conclude that our implementation is fragile and small changes could affect the results.

5.1 Future Work

In this section, we aim to provide trajectories for further study.

5.1.1 Reducing Time Spent Downloading

As our implementation encodes over GF(257) our transmissions are twice the size they could have been, by encoding over GF(2^8). We propose encoding over GF(2^8) would still ensure the integrity of the data, and we assume this will reduce the time spent on download by half.

Our simulations of the multi-server schemes are plagued by the lack of pure multi-threading, for the readers and writers, when sending and receiving data. We propose implementing the reader and writer in a programming language that allows pure multi-threading, as we expect this to eliminate the assumed skewed data discussed in Section 3.3.

5.1.2 Compute Response of Servers in Parallel

In Section 4.2.3 and Section 4.2.6 we argue computing the responses of the database onto the queries is the main practical bottleneck for the PIR schemes. We propose computing the responses of the database in parallel, in a programming language that features pure multi-threading capabilities. We expect this would drastically reduce the time spent on server computation for the PIR schemes.

5.1.3 Study of Weak PIR

We propose a study of a simulation of weak PIR protocols compared to TGE's scheme. Reducing the privacy requirement in trade for better performance is the main objective of weak PIR. A study of a simulation of both a weak and strict PIR protocol would possibly give valuable insight into the difference in practical performance.

Bibliography

[1] "International Telecommunication Union facts and figures 2022." https://www.itu.int/itu-d/reports/statistics/2022/11/24/ff22-internet-use/. Accessed: 2023-08-25.

[2] C. Duhigg, "How companies learn your secrets," 2012.

[3] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pp. 41–50, 1995.

[4] R. Tajeddine, O. W. Gnilke, and S. El Rouayheb, "Private information retrieval from mds coded data in distributed storage systems," *IEEE Transactions on Information Theory*, vol. 64, no. 11, pp. 7081–7093, 2018.

[5] H. Sun and S. A. Jafar, "The capacity of private information retrieval," *IEEE Transactions on Information Theory*, vol. 63, no. 7, pp. 4075–4088, 2017.

[6] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 965–981, 1998.

[7] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: Single database, computationally-private information retrieval," in *Proceedings 38th annual symposium on foundations of computer science*, pp. 364–373, IEEE, 1997.

[8] N. B. Shah, K. V. Rashmi, and K. Ramchandran, "One extra bit of download ensures perfectly private information retrieval," in *2014 IEEE International Symposium on Information Theory*, pp. 856–860, 2014.

[9] T. H. Chan, S.-W. Ho, and H. Yamamoto, "Private information retrieval for coded storage," in *2015 IEEE International Symposium on Information Theory (ISIT)*, pp. 2842–2846, 2015.

[10] A. Fazeli, A. Vardy, and E. Yaakobi, "Codes for distributed pir with low storage overhead," in *2015 IEEE International Symposium on Information Theory (ISIT)*, pp. 2852–2856, 2015.

[11] K. Banawan and S. Ulukus, "The capacity of private information retrieval from coded databases," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1945–1956, 2018.

[12] H. Sun and C. Tian, "Breaking the mds-pir capacity barrier via joint storage coding," *Information*, vol. 10, no. 9, p. 265, 2019.

[13] J. Zhu, Q. Yan, C. Qi, and X. Tang, "A new capacity-achieving private information retrieval scheme with (almost) optimal file length for coded servers," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1248–1260, 2020.

[14] R. Zhou, C. Tian, H. Sun, and T. Liu, "Capacity-achieving private information retrieval codes from mds-coded databases with minimum message size," *IEEE Transactions on Information Theory*, vol. 66, no. 8, pp. 4904–4916, 2020.

[15] J. Li, D. Karpuk, and C. Hollanti, "Towards practical private information retrieval from mds array codes," *IEEE Transactions on Communications*, vol. 68, no. 6, pp. 3415–3425, 2020.

[16] S. Kumar, H.-Y. Lin, E. Rosnes, and A. Graell i Amat, "Achieving maximum distance separable private information retrieval capacity with linear codes," *IEEE Transactions on Information Theory*, vol. 65, no. 7, pp. 4243–4273, 2019.

[17] H.-Y. Lin, S. Kumar, E. Rosnes, and A. G. i. Amat, "Asymmetry helps: Improved private information retrieval protocols for distributed storage," in *2018 IEEE Information Theory Workshop (ITW)*, pp. 1–5, 2018.

[18] R. Freij-Hollanti, O. W. Gnilke, C. Hollanti, A.-L. Horlemann-Trautmann, D. Karpuk, and I. Kubjas, "t-private information retrieval schemes using transitive codes," *IEEE Transactions on Information Theory*, vol. 65, no. 4, pp. 2107–2118, 2019.

[19] J. Lavauzelle, R. Tajeddine, R. Freij-Hollanti, and C. Hollanti, "Private information retrieval schemes with product-matrix mbr codes," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 441–450, 2021.

[20] C. Tian, H. Sun, and J. Chen, "Capacity-achieving private information retrieval codes with optimal message size and upload cost," *IEEE Transactions on Information Theory*, vol. 65, no. 11, pp. 7613–7627, 2019.

[21] Y. Zhang, E. Yaakobi, T. Etzion, and M. Schwartz, "On the access complexity of pir schemes," in *2019 IEEE International Symposium on Information Theory (ISIT)*, pp. 2134–2138, 2019.

[22] H. Sun and S. A. Jafar, "Optimal download cost of private information retrieval for arbitrary message length," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 12, pp. 2920–2932, 2017.

[23] C. Tian, "On the storage cost of private information retrieval," *IEEE Transactions on Information Theory*, vol. 66, no. 12, pp. 7539–7549, 2020.

[24] H. Sun and S. A. Jafar, "The capacity of robust private information retrieval with colluding databases," *IEEE Transactions on Information Theory*, vol. 64, no. 4, pp. 2361–2370, 2018.

[25] R. Freij-Hollanti, O. W. Gnilke, C. Hollanti, and D. A. Karpuk, "Private information retrieval from coded databases with colluding servers," *SIAM Journal on Applied Algebra and Geometry*, vol. 1, no. 1, pp. 647–664, 2017.

[26] R. G. L. D'Oliveira and S. El Rouayheb, "One-shot pir: Refinement and lifting," *IEEE Transactions on Information Theory*, vol. 66, no. 4, pp. 2443–2455, 2020.

[27] L. Holzbaur, R. Freij-Hollanti, J. Li, and C. Hollanti, "Toward the capacity of private information retrieval from coded and colluding servers," *IEEE Transactions on Information Theory*, vol. 68, no. 1, pp. 517–537, 2021.

[28] K. Banawan and S. Ulukus, "The capacity of private information retrieval from byzantine and colluding databases," *IEEE Transactions on Information Theory*, vol. 65, no. 2, pp. 1206–1219, 2019.

[29] R. Tajeddine, O. W. Gnilke, D. Karpuk, R. Freij-Hollanti, and C. Hollanti, "Private information retrieval from coded storage systems with colluding, byzantine, and unresponsive servers," *IEEE Transactions on Information Theory*, vol. 65, no. 6, pp. 3898–3906, 2019.

[30] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin, "Protecting data privacy in private information retrieval schemes," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 151–160, 1998.

[31] H. Sun and S. A. Jafar, "The capacity of symmetric private information retrieval," *IEEE Transactions on Information Theory*, vol. 65, no. 1, pp. 322–329, 2019.

[32] D. Asonov and J.-C. Freytag, "Repudiative information retrieval," in *Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society*, pp. 32–40, 2002.

[33] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*, pp. 265–284, Springer, 2006.

[34] C. Dwork, "Differential privacy," in *International colloquium on automata, languages, and programming*, pp. 1–12, Springer, 2006.

[35] I. Samy, M. Attia, R. Tandon, and L. Lazos, "Asymmetric leaky private information retrieval," *IEEE Transactions on Information Theory*, vol. 67, no. 8, pp. 5352–5369, 2021.

[36] H.-Y. Lin, S. Kumar, E. Rosnes, A. Graell i Amat, and E. Yaakobi, "Weakly-private information retrieval," in *Proceedings IEEE International Symposium on Information Theory (ISIT)*, pp. 1257–1261, 2019.

[37] H.-Y. Lin, S. Kumar, E. Rosnes, A. Graell i Amat, and E. Yaakobi, "Multi-server weakly-private information retrieval," *IEEE Transactions on Information Theory*, vol. 68, no. 2, pp. 1197–1219, 2022.

[38] H.-Y. Lin, S. Kumar, E. Rosnes, A. Graell i Amat, and E. Yaakobi, "The capacity of single-server weakly-private information retrieval," *IEEE Journal on Selected Areas in Information Theory*, vol. 2, no. 1, pp. 415–427, 2021.

[39] I. Samy, R. Tandon, and L. Lazos, "On the capacity of leaky private information retrieval," in *2019 IEEE International Symposium on Information Theory (ISIT)*, pp. 1262–1266, 2019.

[40] H. Y. Lin, S. Kumar, E. Rosnes, A. G. i. Amat, and E. Yaakobi, "The capacity of single-server weakly-private information retrieval," in *2020 IEEE International Symposium on Information Theory (ISIT)*, pp. 1053–1058, 2020.

[41] Y. Yakimenka, H.-Y. Lin, E. Rosnes, and J. Kliewer, "Optimal rate-distortion-leakage tradeoff for single-server information retrieval." submitted to IEEE Journal on Selected Areas in Communications, June 2021.

[42] "python.org/about/ python3 about." https://www.python.org/about/. Accessed: 2023-09-14.

[43] "numpy.org learn." https://numpy.org/learn/. Accessed: 2023-09-14.

[44] "sympy.org about." https://www.sympy.org/en/index.html. Accessed: 2023-09-14.

[45] "pypi.org pillow." https://pypi.org/project/Pillow/. Accessed: 2023-09-14.

[46] "python.org streams." https://docs.python.org/3/library/asyncio-stream.html. Accessed: 2023-08-23.

[47] "python.org kernel globalinterpreterlock." https://wiki.python.org/moin/GlobalInterpreterLock. Accessed: 2023-08-23.

[48] "Ubuntu Manuals chronyc." https://manpages.ubuntu.com/manpages/impish/man1/chronyc.1.html. Accessed: 2023-09-15.

[49] "NextGen Network Synchronization packet timing: Network time protocol." https://link.springer.com/chapter/10.1007/978-3-030-71179-5_7#Sec9. Accessed: 2023-09-15.

[50] "Engineering at Meta building a more accurate time service at facebook scale." https://engineering.fb.com/2020/03/18/production-engineering/ntp-service/. Accessed: 2023-09-15.