Equality Operators for Constant-weight Codewords with Applications in (Keyword) PIR

by

Rasoul Akhavan Mahdavi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Homomorphic encryption allows computation to be performed on data while in encrypted form. However, the computational overhead of a circuit that is run using homomorphic encryption depends on the number of multiplications and multiplicative depth. For example, equality checks which are a common step in many tasks, have a multiplicative depth that depends on the bit-length of the numbers. In this work, we propose *constant-weight equality operators*, which compare constant-weight codewords using a circuit that has a multiplicative depth that depends solely on the Hamming weight of the constant-weight code, not the size of the operands.

Private Information Retrieval (PIR) is one task where equality operations are a solution. In a PIR protocol, a user wishes to query a database without revealing which element is queried to the server. In this thesis, we also detail an architecture for PIR which was previously assumed to be impractical. At the heart of this architecture is the constant-weight equality operator.

Our experiments show how constant-weight equality operators outperform existing equality operators and can be used for practical purposes. We also conduct experiments to show the practicality of PIR using our approach and our results show how constant-weight PIR outperforms existing work in aspects of scale such as large domain sizes and large responses.

Acknowledgements

First and foremost, I would like to thank Florian Kerschbaum for his outstanding supervision during this period. His patience, diligence, and care in working with me as an inexperienced graduate student, with all the missteps and mistakes, was admirable. His rigour, not just in his field but also in being considerate and patient, was undoubtedly a lesson for me in my career to come.

Maintaining sanity during this period of uncertainty, both in the research process and at the time of a raging pandemic, could not have been possible without the love and support of a wonderful group of friends. A complete list would require a thesis of its own, but I specifically thank members of the CrySP lab who spared no effort to make me feel welcome, included, and supported. Just a handful of CrySP lab members that I want to thank includes — but is not limited to — Thomas, John, Shannon, Lindsey, Emily, Jason, Bailey, Miti, Matthew, Nils, Masoumeh, Simon, Nik, and of course one affiliated CrySP member, Kiernan. Thank you, not only for the professional help but for making the graduate experience fun, exciting and also just making me a better person.

This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo. All experiments in Chapter 4 were conducted using RIPPLE machines.

Dedication

To my parents, whose unconditional love shall never be explained by science.

Table of Contents

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Our Contributions . . . 2

1.2 Organization . . . 3

2 Background & Related Work 4

2.1 Homomorphic Encryption . . . 4

2.1.1 Fan–Vercauteren (FV) Cryptosystem. . . . 5

2.1.2 Microsoft SEAL Library . . . 5

2.2 Private Information Retrieval . . . 6

2.3 Single-Server computational PIR . . . 7

2.3.1 SEALPIR . . . 8

2.3.2 MulPIR . . . 10

2.4 Equality Operators . . . 10

2.4.1 PIR using Equality Operators . . . 12

2.5 Keyword PIR . . . 13

3 Our Constructions **14**
3.1 Equality Operator for Constant-weight Codewords . . . 15
3.2 Mappings to Constant-weight Codewords . . . 18
3.3 PIR using Constant-weight Codewords . . . 20
3.3.1 Setup . . . 21
3.3.2 Query . . . 22
3.3.3 Process Query . . . 22
3.3.4 Extract . . . 25
3.3.5 PIR for Sparse Databases . . . 25
3.3.6 Probabilistic Keyword PIR . . . 26

4 Evaluation **27**
4.1 Comparing Equality Operators . . . 27
4.1.1 Plain Operators . . . 29
4.1.2 Arithmetic Operators . . . 30
4.2 Comparing PIR Protocols . . . 32
4.2.1 Implementation and Experimental Details . . . 32
4.2.2 Packed Database Experiments . . . 33
4.2.3 Varying Domain Size . . . 37
4.2.4 Varying Response size . . . 40

5 Applications, Limitations, and Future Work **43**
5.1 Applications . . . 43
5.1.1 First Practical Solution . . . 43
5.1.2 Setup-Free (Update Friendly) . . . 44
5.1.3 Less Overhead for Decentralized Database . . . 44
5.1.4 Anti-Fishing . . . 45
5.2 Limitations . . . 46
5.3 Future Work . . . 46

6 Conclusion 48

References 49

List of Figures

4.1 Encoding size as a function of multiplicative depth 40
4.2 Runtime of constant-weight PIR and MulPIR for large response sizes . . . 42

List of Tables

2.1 Cost of operations in SEAL 3.6 6

3.1 Stages of PIR using constant-weight codewords 21

4.1 Properties of circuits implementing equality operators 28
4.2 Runtimes for plain equality operators 30
4.3 Runtimes for arithmetic equality operators 31
4.4 Parameters for different PIR protocols 34
4.5 Runtimes for Folklore and Constant-weight PIR 35
4.6 Runtime of Constant-weight PIR, SEALPIR, and MulPIR 36
4.7 Runtime of Constant-weight PIR executed with parallelization 37
4.8 Parameters for sparse PIR protocols 38
4.9 Bit-length of the query in different protocols 39
4.10 Breakdown of runtime for constant-weight PIR over a sparse databases . . 41
4.11 Parameters for different PIR protocols with large response size 42

5.1 Number of operations in single and multi-DB PIR 45

Chapter 1

Introduction

Homomorphic encryption is a tool that permits computation over data in encrypted form. Complex functions over the input data are calculated using basic arithmetic operations. For example, the FV cryptosystem allows addition and multiplication. However, there is an imbalance between the cost of these operations. In the case of FV, multiplications are up to 20× slower than additions. In addition, the maximum number of sequential multiplications, commonly referred to as the *multiplicative depth*, depends on the parameters of the cryptosystem and performing multiplications beyond that results in an undecryptable ciphertext. Larger parameters for the cryptosystem are required when the circuit has a higher multiplicative depth, which in turn makes the circuits slower and less practical. Consequently, the multiplicative depth and number of multiplications in a circuit directly impact the efficiency of that circuit when run using homomorphic encryption.

Equality checks are a necessary step in many tasks. For example, in the task of Private Set Intersection (PSI) the objective is to find the intersection of two private sets held by two parties, without the two parties learning anything about the sets beyond the intersection. Elements from the sets must be compared to determine if they fall in the intersection.

One main obstacle when comparing two numbers that are encrypted using homomorphic encryption is the high multiplicative depth of the circuits that are evaluated. For example, the simple *folklore* equality circuit has a multiplicative depth of $\log_2 \ell$ when comparing two ℓ-bit elements.

In this work we propose *constant-weight equality operators*, used to compare constant-weight codewords. Circuits for such a task have a multiplicative depth that depends only on the Hamming weight of the code, not the length of the codewords. Our theoretical and experimental evaluation shows how these circuits compare to existing equality operators.

Specifically, the *plain constant-weight* equality operator is up to 10 times faster than the equivalent folklore operator. Moreover, the *arithmetic constant-weight* equality operator is up to 10 times faster than the equivalent folklore operators when parallelized.

Private Information Retrieval (PIR) is another task that requires equality checks at its core. In PIR, a user wishes to retrieve an element from a database such that it is not revealed which element it wishes to access. There are many practical solutions in the multi-server setting, where the database is held by multiple non-colluding servers. However, the practicality of the PIR in a single-server setting was disputed. Lattice-based cryptosystems provided solutions that performed better than the trivial solution of downloading the entire database. XPIR [1] was the first practical solution that utilized an additive homomorphic encryption system. However, the communication was still on the order of the number of records in the database. More recent work, namely SEALPIR [6] and MulPIR [4], reduced the communication cost and provided even faster solutions.

All existing solutions for single-server PIR avoid performing equality operations due to the high computational cost. We propose the first practical solution to PIR using equality circuits. Our solution uses the equality operator over constant-weight codewords at its core for efficiency. Our protocol is also applicable in *keyword PIR* where the user retrieves an element from the database using a keyword, not the physical address.

We evaluate our protocol in comparison with existing work and show its practicality through our experiments. Our evaluations focus on how protocols behave in different aspects of scale, such as large domains and large response sizes. Our experiments show how our protocol is a suitable solution in the devised scenarios. Finally, we elaborate on how the results of our experiments translate to real-world properties and applications.

1.1 Our Contributions

The contributions of this thesis can be summarized as follows:

- We propose *constant-weight equality operators*, used to compare constant-weight codewords with a multiplicative depth that depends only on the Hamming weight of the code. We also provide mappings to efficiently map elements from other domains to constant-weight codewords.
- We detail a PIR architecture using constant-weight codes which was previously assumed to be impractical. We also explain how it can be extended to the case of keyword PIR.

- We compare constant-weight equality operators with existing equality operators, both theoretically and through experiments. We compare elements with various bit-lengths and also examine the effect of parallelization.
- We also compare the PIR protocol proposed in this paper with that of related work in terms of scalability. For this, we perform experiments in three different scenarios.
- Finally, we elaborate on applications where our constructions are beneficial compared to existing solutions based on the observations from our experiments.

1.2 Organization

In Chapter 2, the necessary background regarding homomorphic encryption, private information retrieval and single-server private information retrieval solutions are summarized. This chapter also includes descriptions of existing equality operators and their properties.

In Chapter 3, we present our constructions, the constant-weight equality operators, mapping to constant-weights, and a PIR protocol using constant-weight codes at its core.

In Chapter 4, we perform experiments to compare constructions from Chapter 3 with those from Chapter 2.

Finally, in Chapter 5, we discuss potential applications for the constructions of Chapter 3 based on the observations from Chapter 4.

Chapter 2

Background & Related Work

2.1 Homomorphic Encryption

Homomorphic Encryption allows computation on encrypted data, without the need for decryption or access to the secret key. This maintains the secrecy of the data while computation is performed. One use case is a client delegating computation on its data to a remote, untrusted server.

The concept of homomorphic encryption was introduced by Rivest et al. [30] where they proposed a cryptosystem that permits one operation, e.g. addition, on data in encrypted form. Later work showed how to construct an encryption scheme that allows additions and one level of multiplication [9]. In 2009, Gentry proved the existence of a *fully homomorphic* cryptosystem based on lattices that can evaluate arbitrary functions on encrypted data [21]. The security of lattice-based cryptosystems is due to a small, random *noise* that is added to the plaintexts. This noise accumulates as each homomorphic operation is performed, to the point where the ciphertext is not decryptable if any more operations are performed. Gentry proposed an expensive bootstrapping procedure to reduce the noise in a ciphertext and allow more operations to be performed [21].

Multiple lattice-based cryptosystems were proposed following the seminal work of Gentry which improved the efficiency drastically [11, 12, 24, 36]. However, the bootstrapping procedure remains impractical and cryptosystems are used in a *leveled* fashion. A leveled homomorphic cryptosystem allows only a predefined number of sequential multiplications, determined by the parameters of the cryptosystem, and does not perform the bootstrapping procedure. The Fan–Vercauteren cryptosystem is an example that we explain in the next subsection.

2.1.1 Fan–Vercauteren (FV) Cryptosystem.

The Fan–Vercauteren cryptosystem [20] is a lattice-based cryptosystem where plaintexts are elements from the polynomial ring $R_t = \mathbb{Z}_t[x]/(x^N + 1)$. The *polynomial modulus degree*, N, is a power of two and t is the *plaintext modulus*. Messages must be encoded as a polynomial in the field before they can be encrypted. An FV ciphertext is an array of polynomials, each from $R_q = \mathbb{Z}_q[x]/(x^N + 1)$, where q is called the *coefficient modulus*. In the simplest case, the ciphertext is only two polynomials. Let $\mathcal{C}$ denote the ciphertext space. N and q determine both the security parameter and how many homomorphic operations can be performed on ciphertexts before decryption (or bootstrapping) is necessary.

In addition to the standard operations for a cryptosystem, i.e. key generation, encryption and decryption, FV supports homomorphic operations over the ring as well. Four of these operations are listed below. All operations over plaintexts are in the ring R_t.

- **Addition:** Given ciphertexts $c_1(x), c_2(x) \in \mathcal{C}$ that encrypt $m_1(x), m_2(x) \in R_t$, respectively, output $c_A(x)$ which encrypts $m_1(x) + m_2(x)$.
- **Plain Multiplication:** Given $m_1(x) \in R_t$ and $c_2(x) \in \mathcal{C}$ that encrypts $m_2(x) \in R_t$, output $c_{PM}(x)$ which encrypts $m_1(x)m_2(x)$.
- **Multiplication:** Given ciphertexts $c_1(x), c_2(x) \in \mathcal{C}$ that encrypt $m_1(x), m_2(x) \in R_t$, respectively, output $c_M(x)$ which encrypts $m_1(x)m_2(x)$.
- **Substitution:** Given $c(x) \in \mathcal{C}$ that encrypts $m(x)$ and an integer k, output $c_S(x)$ which encrypts $m(x^k)$.

2.1.2 Microsoft SEAL Library

The SEAL library [31] implements the FV cryptosystem and supports all the operations mentioned above. Specifically, the implementation for the substitution operation in this library was first introduced by Angel et al. [6] based on the plaintext slot permutation technique discussed by Gentry et al. [23]. One FV plaintext can encode $N \log_2(t)$ bits of data. Also, the size of the smallest ciphertext that encrypts a plaintext is $2N \log_2(q)$ bits. An important parameter is the *expansion factor* which is the ratio between the size of a ciphertext and the largest plaintext that can be encrypted and is equal to

$$F = \frac{2\log(q)}{\log(t)}. \tag{2.1}$$

Table 2.1 compares the four described operations in terms of speed and noise grown, as implemented in SEAL 3.6.

Table 2.1: Cost of operations in SEAL 3.6, for $N \in \{2048, 4096, 8192, 16384\}$ and the default ciphertext modulus. * Time and Noise growth in plain multiplication also depends on the value of the unencrypted operand.

Operation	**Time** (μs)				**Noise Growth**
	$N = 2048$	$N = 4096$	$N = 8192$	$N = 16384$	
Addition	6	19	67	435	Additive
Plain Multiplication*	12–135	30–529	105–2201	509–9647	Multiplicative
Multiplication	-	3823	15744	66908	Multiplicative
Substitution	-	768	4137	26047	Additive

2.2 Private Information Retrieval

Private Information Retrieval [16] is a protocol where a user retrieves an element from a database, such that the owner of the database cannot determine which element was retrieved. There are two forms of PIR protocols. In the first form, which we denote *index PIR*, the user holds the *physical* address of the item, e.g., the row in a database table or the index in a public registry. In the second form, the physical address of the desired item may not be known and it is only accessible by an identifier pertaining to the sought item, e.g., the name of a file. The latter is referred to as *keyword PIR* or *sparse PIR*, first introduced by Chor et al. [15].

The privacy guarantee of a PIR protocol can be information-theoretic or computational. Information-theoretic PIR (IT-PIR) is private even in the presence of a computationally unbounded adversary and usually requires computationally inexpensive operations (additions, XOR, etc.) and achieves communication sublinear in the size of the database [5, 8, 16, 17]. However, these solutions require replication of the database across multiple non-colluding servers. The assumption of non-collusion is hard to enforce in practice. If only a single server is used to avoid the non-collusion assumption, Chor et al. proved that any solution would require communication at least the size of the database for information-theoretic privacy [16].

Computational PIR (CPIR) relaxes the assumption to an adversary with bounded computational power. In the single-server setting, which is the focus of this paper, solutions rely on some intractability assumption, e.g., the hardness of determining the quadratic residuosity modulo composite numbers [25, 27] or the security of lattice-based cryptosystems [1, 4, 6, 18, 19, 22, 38].

In CPIR solutions, each item in the database has to be processed at least once, otherwise, it can be trivially excluded from the list of potential queries and compromise privacy.

Sion and Carbunar argued that the time required for any single-server CPIR protocol would exceed the time required for the trivial solution of simply downloading the entire database [32]. Later work by Aguilar-Melchor et al. showed this argument to be incorrect with the use of lattice-based cryptosystems, which have smaller per-bit computation cost when used in a batched fashion [1]. They were able to show that PIR is a faster alternative to downloading the database over networks with less bandwidth.

2.3 Single-Server computational PIR

Single-server computational PIR solutions aim to perform better than the *trivial* solution of downloading the entire database. In the trivial solution, the *download cost* for the user is equal to the size of the database, with no *upload cost* for the user. Downloading the entire database also comes at almost no computational burden for the server, i.e., the *computational cost* is zero. We examine some single-server CPIR protocols in this subsection and the following subsections to compare them with the trivial solution and each other, based on their upload, download, and computational cost.

CPIR protocols utilizing homomorphic encryption are the most practical solutions to date [1, 4, 6]. All these solutions expand on a baseline method that works as follows:

Baseline PIR method. Let $\mathbb{DB}$ denote the database with n rows and $\mathbb{DB}[i]$ denote the i^{th} row in this database. Also, throughout this thesis, define $[n] = \{0, 1, ...n-1\}$, for any $n \in \mathbb{N}$. When the goal is to retrieve row q, a response r_q is derived as

$$r_q = \sum_{i \in [n]} \mathbb{I}(i == q) \cdot \mathbb{DB}[i]. \tag{2.2}$$

where $\mathbb{I}(\cdot)$ denotes an indicator function which is one when the input evaluates to true and zero otherwise. It is easy to verify that if $q \in [n]$ then $r_q = \mathbb{DB}[q]$. Equation (2.2) is an inner product between the database and a vector of bits called the *selection vector*. For obtaining element q in the database, the selection vector is one in index q and zero otherwise.

PIR protocols realizing Equation (2.2) encrypt the bits of the selection vector with a homomorphic encryption scheme that supports addition and plaintext multiplication and perform the operations in Equation (2.2) over ciphertexts. In XPIR [1] and SEALPIR [6], two recent practical solutions, an additive homomorphic encryption scheme is used. MulPIR [4]

is the first practical solution using a fully homomorphic encryption scheme, which is also the case for our work.

The server requires ciphertexts of the bits of the selection vector, i.e., $\mathbb{I}(i == q)$, to realize Equation (2.2). There are two general approaches for the server to acquire the encrypted bits of the selection vector: 1) Communicating the selection vector 2) Equality Operators.

In the first approach, the user generates the selection vector locally, encrypts it and transmits it to the server. XPIR, SEALPIR, and MulPIR all take this approach. XPIR uploads the entire selection vector but provides experiments to show the practicality of this approach [1]. Despite its practicality, the upload cost of XPIR is on the order of the number of rows in the database which limits scalability.

Recursion is a method to reduce the upload cost to sublinear in the size of the database. It was first used by Kushilevitz and Ostrovsky [27] and later Stern [34]. This approach is also used in SEALPIR and MulPIR. In the next section, we describe how recursion is done in SEALPIR, which is conceptually similar to prior work.

2.3.1 SEALPIR

SEALPIR [6] is a PIR scheme based on the SEAL library which uses *recursion* and a *query compression* technique to reduce the upload cost. They also use additive homomorphic encryption in a layered fashion.

In SEALPIR, to communicate fewer ciphertexts, the user encodes multiple bits into one plaintext, which is called the query compression technique. Specifically, for a selection vector $(s_i)_{i\in[n]}$, the user constructs the plaintext $p(x) = \sum_{i\in[n]} s_i x^i$ and encrypts it. Recall that in SEAL, plaintexts are polynomials of degree at most N, so if the size of the selection vector exceeds N, more than one plaintext is used. For n bits in the selection vector, at least $\lceil n/N \rceil$ ciphertexts are needed. As a consequence of the compression technique, SEALPIR performs a novel *oblivious expansion* on the server to extract a vector of ciphertexts such that each bit of the selection vector is in a separate ciphertext. SEALPIR uses the substitution operation to perform the oblivious expansion. Algorithm 1 depicts this procedure for expanding one ciphertext into a vector of 2^c ciphertexts, for $c \in [0, \log_2 N]$.

To further reduce the communication, SEALPIR uses a technique called *recursion* in which the database is restructured into a d-dimensional table. The size of the i^{th} dimension is d_i such that $\prod d_i \geq n$. Then instead of one selection vector, d selection vectors are sent to the server, one for each dimension. We refer to d as the *recursion level*. The total size of the query is at least $d \lceil \sqrt[d]{n} \rceil$ which is sublinear in n for any $d \geq 2$.

Algorithm 1 SEALPIR Oblivious Expansion

Input: $ct(x) \in \mathcal{C}$, compression Factor $c \in [0, \log_2 N]$

$cts \leftarrow [ct(x)]$
for $a \in [c]$ **do**
    for $b \in [2^a]$ **do**
        $c_0 = cts[b]$
        $c_1 = x^{2^{-a}} \cdot c_0$
        $cts[b] = c_0 + \texttt{Sub}_{N/2^a+1}(c_0)$
        $cts[b + 2^a] = c_1 + \texttt{Sub}_{N/2^a+1}(c_1)$
$inv = (2^{-c} \mod t)$
for $i \in [2^c]$ **do**
    $cts[i] \leftarrow inv \cdot cts[i]$

Output: $cts \in \mathcal{C}^{2^c}$

As an example, assume $d = 2$ and $d_1 = d_2 = \lceil \sqrt{n} \rceil$, and assume that the user's desired query, q, is now at row q_1 and column q_2. Equation (2.3) depicts the formula used to derive the results in this case.

$$r_q = \sum_{i=1}^{n} \mathbb{I}(i == q) \cdot \mathbb{DB}[i] = \sum_{i_1=1}^{\lceil \sqrt{n} \rceil} \mathbb{I}(i_1 == q_1) \left(\sum_{i_2=1}^{\lceil \sqrt{n} \rceil} \mathbb{I}(i_2 == q_2) \cdot \mathbb{DB}\left[i_1 \left\lceil \sqrt{n} \right\rceil + i_2 \right] \right) \tag{2.3}$$

In SEALPIR, an additive homomorphic encryption scheme is used so the first multiplication is performed as a plaintext multiplication. However, the second multiplication is between two ciphertexts, which is not supported. To overcome this issue, one ciphertext is treated as a plaintext in the multiplication. This is referred to as *layered encryption* and results in the size of the response multiplying by a factor of F where F is the expansion factor of the ciphertext. More generally, the size of the response is multiplied by a factor of F^{d-1} for recursion level equal to d. Overall, SEALPIR performs $\sum_{i=0}^{d-1} n^{\frac{d-i}{d}} F^i$ plaintext multiplications for recursion level $d \geq 1$ and expansion factor of F for the ciphertext.

Ali et al. proposed three optimizations to SEALPIR to reduce the communication cost: compressing the uploaded ciphertexts by encrypting using the secret key instead of the public key, compressing the response ciphertexts using modulus switching, and a modified oblivious expansion to fit more bits into the one ciphertext [4]. Throughout this thesis, *SEALPIR* denotes this modified version of the protocol.

2.3.2 MulPIR

MulPIR [4] replaces the layered encryption in SEALPIR with homomorphic multiplications. This reduces the download cost drastically compared to SEALPIR. However, it comes at the cost of increased computation for the server since homomorphic multiplications are more expensive than plain multiplications and larger parameters are required to allow more homomorphic multiplications. Algorithm 2 shows the query evaluation process performed by the server after query expansion for a recursion level of $d = 2$. The multiplications on lines 2 and 3 are plain and homomorphic, respectively.

Algorithm 2 MulPIR Query Evaluation

Input: $n' = \lceil \sqrt{n} \rceil$, $cts_0, cts_1 \in \mathcal{C}^{n'}$

1: **for** $i \in [n']$ **do**

2: $\quad c_i = \sum_{j \in [n']} cts_0[j] \cdot \mathbb{DB}[i \cdot n' + j]$

3: $c = \sum_{i \in [n']} c_i \cdot cts_1[i]$

Output: $c \in \mathcal{C}$

Overall, MulPIR performs n plaintext multiplications and $\sum_{i=1}^{d-1} n^{\frac{d-i}{d}}$ homomorphic multiplications, for a recursion level $d \geq 1$.

In SEALPIR, due to the expansion in the response, the server can not perform any post-processing on the output which is a disadvantage of the protocol. Examples of post-processing include deriving functions of the user's query or conjunctive and disjunctive PIR queries. In contrast to SEALPIR, the output of the MulPIR protocol can be post-processed before being sent back to the user. Ali et al. [4] describe how to perform conjunctive and disjunctive queries in their paper.

2.4 Equality Operators

Checking the equality of two values is an integral step in many tasks over encrypted data such as secure search [2, 3], secure pattern matching [10, 37], PSI [13, 26], and PIR [16].

We define an *equality operator* as follows.

Definition 1 (Equality Operator)**.** *A procedure f is an equality operator over a domain D if $\forall x, y \in D$,*

$$f(x,y) = \begin{cases} 1 & if \;\; x = y \\ 0 & o.w. \end{cases} \tag{2.4}$$

If Equation (2.4) holds with some probability $1-\epsilon$, the procedure f is an ϵ-probabilistic equality operator.

We now define some equality operators over their respective domains and derive the multiplicative depth of a circuit implementing each one. When working with an element $x \in \{0,1\}^\ell$, we treat it as a string of bits and refer to the bits of the string by indexing, i.e., $x[i]$ denotes the i^{th} bit of x.

Arithmetic Folklore Equality Operator. The first operator is used to compare two numbers in binary format. For a domain $D = \{0,1\}^\ell$, define f_{AF} as

$$f_{AF}(x,y) = \prod_{i=0}^{\ell-1} \left(1 - (x[i] - y[i])^2\right) \tag{2.5}$$

for $x, y \in \{0,1\}^\ell$. This operator is correct when operating over any field such as $\mathbb{Z}_p$. The multiplicative depth of a circuit realizing this operator is equal to $\log_2(\ell) + 1$, where ℓ is the bit-length of the operands.

The arithmetic folklore operator is oblivious to both input operands. This is critical in some applications, e.g. comparing two encrypted or secret shared numbers. When one operator is public, the arithmetic folklore equality operator can be modified such that it has a smaller multiplicative depth. The modified operator is as follows.

Plain Folklore Equality Operator. For a domain $D = \{0,1\}^\ell$, define f_{PF} as

$$f_{PF}(x,y) = \prod_{y[i]=0} (1 - x[i]) \prod_{y[i]=1} x[i] \tag{2.6}$$

for $x, y \in \{0,1\}^\ell$. This operator depends on the public operand, which is y in this case. The multiplicative depth of a circuit realizing this operator is equal to $\log_2(\ell)$, where ℓ is the bit-length of the operands.

When operating over a binary field, the arithmetic folklore equality operator can again be modified to have a lower multiplicative depth.

Binary Folklore Equality Operator. For a domain $D = \{0,1\}^\ell$, define f_{BF} as

$$f_{BF}(x,y) = \prod_{i=0}^{\ell-1}(1 + x[i] + y[i]) \mod 2 \tag{2.7}$$

for $x, y \in \{0,1\}^\ell$. The multiplicative depth of a circuit realizing this operator is equal to the logarithm of the bit-length of the elements in the domain, i.e., $\log_2(\ell)$.

Binary-Raffle Equality Operator. Next, we define a probabilistic equality operator inspired by the methods of Razbarov and Smolenski [29, 33]. They propose a low degree approximation of the logical-OR function which can be used as described below to construct an equality operator.

This operator is used to compare elements in binary format over a binary field. For a domain $D = \{0,1\}^\ell$, sample $N(\epsilon)$ random elements from $\{0,1\}^\ell$, which we denote as r_j for $j \in \{1, 2, \cdots, N(\epsilon)\}$, then define $f_{BR}(x,y)$ as

$$f_{BR}(x,y) = \prod_{j=1}^{N(\epsilon)} \left(1 + \sum_{r_j[i]=1} (x[i] + y[i])\right) \mod 2 \tag{2.8}$$

for $x, y \in \{0,1\}^\ell$.

The output of this operator is correct with probability $1 - \epsilon$ if $N(\epsilon) = \log_2(1/\epsilon)$. The multiplicative depth of a circuit realizing this operator is equal to $\log_2(N(\epsilon))$ which solely depends on the failure probability of this operator.

2.4.1 PIR using Equality Operators

As mentioned in Section 2.3, equality operators are another approach to PIR. In this approach, the user's query is encoded into some domain, encrypted and sent to the server. The server computes each bit of the selection vector using an equality operator between the user's encrypted query and each identifier in the database. Then the server, using the encrypted bits of the selection vector, derives the encryption of r_q using Equation (2.2), which is then sent back to the user for decryption.

PIR with this approach using the folklore equality operator has the smallest upload cost amongst all non-trivial approaches. In this approach, only the optimal logarithmic

binary encoding of the query is encrypted and uploaded. However, the computation cost is prohibitively high due to the multiplicative depth which depends on the size of the database. In general, PIR using equality operators is assumed to be impractical due to the high multiplicative depth of equality circuits as parameters scale [4, 6].

2.5 Keyword PIR

In keyword PIR, also referred to as sparse PIR, a user retrieves an element from a database using a keyword or identifier pertaining to the sought item. Another way to phrase this is that in index PIR, all addresses correspond to an element in the database whereas in keyword PIR, some identifiers may not correspond to any element and those entries are empty, hence the name *sparse PIR*.

Previous work has suggested solutions for keyword PIR which all basically reduce keyword PIR to index PIR at their core. Chor et al. suggested two solutions where the user interactively queries the server to obtain the physical address of the desired item, given the identifier. A common solution also proposed by Ali et al. involves a probabilistic hashing technique to map identifiers into a small domain such that index PIR is feasible.

PIR using equality operators is another approach to keyword PIR, where the users query is compared to all the keywords present in the database. However, since the cost of comparing keywords is prohibitively high for large keywords, this approach is assumed to be impractical. This work proposes a PIR protocol for index PIR which can be easily extended to keyword PIR with minimal change. Moreover, the practical cost of comparison results in a practical keyword PIR protocol.

Chapter 3

Our Constructions

In this section, we describe our constructions. First, we propose equality operators for constant-weight codewords. Then we describe mappings from other domains to constant-weight codes to facilitate the use of our proposed operator in other contexts. Finally, we explain PIR using constant-weight codewords in detail.

Constant-weight Code. A *constant-weight code*, also known as *m-of-n code*, is a form of error detecting code where all codewords share the same Hamming weight. A *binary constant-weight code* has the additional condition that all codewords are binary strings. The one-hot (unary) code and the *balanced code* are two examples of a binary constant-weight code. In a balanced code, the number of ones is equal to the number of zeros in all codewords.

The length of a code is the maximum bit-length of its codewords and the size of the code is the number of distinct codewords. For a binary constant-weight code of length m and Hamming weight of k, the size is $\binom{m}{k}$. For a fixed Hamming weight k, to have a binary constant-weight code with a size of at least n, we must choose the length, m, such that $\binom{m}{k} \geq n$. By one approximation, we have $m \in \Omega\left(\sqrt[k]{k!n}\right)$. We denote the binary constant-weight code with length m and Hamming weight k by $CW(m, k)$.

In all the constructions, k and m denote the Hamming weight and length of the code, respectively.

3.1 Equality Operator for Constant-weight Codewords

We propose three variants of the equality operator over constant-weight codewords.

Plain Constant-weight Equality Operator. For two constant-weight codewords $x, y \in CW(m, k)$,

$$f_{PCW}(x, y) = \prod_{y[j]=1} x[j] \tag{3.1}$$

is the *plain equality operator*. This operator is oblivious to the first operand but depends on the second. A circuit realizing this operator performs k multiplications with a multiplicative depth of $\lceil \log_2 k \rceil$.

Arithmetic Constant-weight Equality Operator. For two constant-weight codewords $x, y \in CW(m, k)$, Algorithm 3 describes the *arithmetic equality operator* over constant-weight codewords. Algorithm 3 operates over any field in which $k!$ has a multiplicative inverse.

Algorithm 3 Arithmetic Constant-weight Equality Operator

Input: $x, y \in CW(m, k)$

1: $k' = \sum_{i \in [m]} x[i] \cdot y[i]$

2: $e = \frac{1}{k!} \prod_{i \in [k]} (k' - i)$

Output: $e \in \{0, 1\}$

Theorem 1. *For $x, y \in CW(m, k)$, if $f_{ACW}(x, y)$ is the output of Algorithm 3, then $f_{ACW}(x, y)$ is an equality operator.*

Proof. If x and y are equal, the position of bits equal to one in their encodings are identical, and consequently, the inner product, k', will be equal to k. When they are not equal, the inner product will be in the set $\{0, 1, ..., k-1\}$. Also, based on the definition of e on line 2 of Algorithm 3, it holds that

$$e = \begin{cases} 1 & k' = k \\ 0 & k' \in \{0, 1, ..., k-1\} \end{cases} \tag{3.2}$$

Putting these two together, e will be one, if and only if x and y are equal and zero otherwise. □

A circuit realizing this operator performs $m + k$ multiplications with a multiplicative depth of $1 + \lceil \log(k) \rceil$.

Binary Constant-weight Equality Operator. The final variant is an equality operator over a binary field. This operator is described in Algorithm 4. In this Algorithm, the binary representation of k is shown as $(\overline{b_h ... b_1 b_0})_2$.

Algorithm 4 Binary Constant-weight Equality Operator

Input: $x, y \in CW(m, k)$, $k = (\overline{b_h ... b_1 b_0})_2$, $h = \lfloor \log k \rfloor$

for $i \in [m]$ **do**
    $z_0[i] = x[i] \cdot y[i]$
$b'_0 = \sum_{j \in [m]} z_0[j] \mod 2$
for $i \in \{1, 2, ..., h\}$ **do**
    $s_i[0] = 0$
    for $j \in \{1, ..., m-1\}$ **do**
        $s_i[j] = s_i[j-1] + z_{i-1}[j-1] \mod 2$
    for $j \in [m]$ **do**
        $z_i[j] = z_{i-1}[j] \cdot s_i[j] \mod 2$
    $b'_i = \sum_{j \in [m]} z_i[j] \mod 2$
$e = \prod_{i=0}^{h} (1 + b_i + b'_i) \mod 2$

Output: $e \in \{0, 1\}$

Theorem 2. *For $x, y \in CW(m, k)$, if $f_{BCW}(x, y)$ is the output of Algorithm 4, then $f_{BCW}(x, y)$ is an equality operator over $CW(m, k)$.*

Proof. For $x \in \{0, 1\}^\ell$ define $I(x)$ as follows:

$$I(x) = \{i : x[i] = 1\} \tag{3.3}$$

which denotes the set of indices where x is one. We first prove a lemma:

Lemma 1. $|I(z_i)| = \lfloor |I(z_{i-1})|/2 \rfloor$

To prove the lemma, assume

$$I(z_{i-1}) = \{i_1, ..., i_{k_{i-1}}\}. \tag{3.4}$$

where $i_1 < i_2 < ... < i_{k_{i-1}}$. Now based on the definition of s_i from line 7 of Algorithm 4, we have

$$I(s_i) = \{j : i_{2t-1} < j \leq i_{2t}, 1 \leq t \leq k_{i-1}/2\}. \tag{3.5}$$

Based on line 9 of Algorithm 4, z_i is equal to one in the indices where z_{i-1} and s_{i-1} are both equal to one so combining (3.4) and (3.5) we have

$$I(z_i) = \{j : j = i_{2t}, 1 \leq t \leq k_{i-1}/2\} \tag{3.6}$$

So $|I(z_i)| = \lfloor k_{i-1}/2 \rfloor = \lfloor |I(z_{i-1})|/2 \rfloor$ and the lemma is proven.

Next we prove that b'_i calculated in line 10 of Algorithm 4 is the i^{th} bit of $I(z_0)$. In other words, if $I(z_0) = (\overline{b''_h ... b''_1 b''_0})_2$, we will prove that $b''_i = b'_i$. Based on the lemma, we know that $I(z_i) = (\overline{b''_h ... b''_{i+1} b''_i})_2$ so

$$b'_i = \sum_{j=1}^{m} z_i[j] = I(z_i) \mod 2 = b''_i \tag{3.7}$$

So $I(z_0) = (\overline{b'_h ... b'_1 b'_0})_2$. To finish the proof, we know that x and y are equal, if and only if $I(z_0) = k$ and line 11 of Algorithm 4 evaluates the folklore binary operator between the bitwise representation of $I(z_0)$ and k, so the theorem is proven. □

Overall, Algorithm 4 performs $m + (m+1)\lfloor \log_2 k \rfloor$ multiplications. The multiplicative depth of circuit realizing this operator is given in the following theorem.

Theorem 3. *The multiplicative depth of a circuit equivalent to Algorithm 4 is equal to*

$$1 + \lfloor \log_2 k \rfloor + \lceil \log_2 (\lfloor \log_2 k \rfloor + 1) \rceil. \tag{3.8}$$

Proof. Initially, on line 1, one multiplication is performed for each of the bits of the codewords. Then in the for loop of line 4, one multiplication is performed in each of the $h = \lfloor \log_2(k) \rfloor$ iterations. Additions are also performed in each iteration so the multiplications can only be done in sequence. Finally, on line 11, $h + 1$ multiplications are done in a tree-style fashion with a depth of $\lceil \log_2 (h+1) \rceil$. Hence, the total depth is $1 + h + \lceil \log_2 (h+1) \rceil$ and the theorem is proven. □

Remark. If k is a power of two, line 11 in Algorithm 4 can be simplified to $e = b'_h$ and the last term in Equation (3.8) can be dropped. The multiplicative depth of the corresponding circuit is equal to

$$1 + \lfloor \log_2 k \rfloor \,. \tag{3.9}$$

3.2 Mappings to Constant-weight Codewords

The equality operators described in the previous section are all over constant-weight codewords. To benefit from these constructions in a setting where we want to compare elements from other domains, we also propose two mappings from other domains to constant-weight codewords. The goal is for the mapping (and inverse mapping) procedure to be efficient and less expensive than storing an equivalence table.

Perfect Mapping. This mapping is used to map numbers in the set $[n]$ to $CW(m, k)$ such that it is injective and has an inverse. To have the injective property, the code size must be at least n, i.e., $|CW(m, k)| = \binom{m}{k} \geq n$. The mapping procedure is given in Algorithm 5.

Algorithm 5 Perfect Mapping

Input: $x \in [n]$, $m, k \in \mathbb{N}$ such that $\binom{m}{k} \geq n$

1: $r = x$
2: $h = k$
3: $y = 0^m$
4: **for** $m' = m - 1, ..., 1, 0$ **do**
5: **if** $r \geq \binom{m'}{h}$ **then**
6: $y[m'] = 1$
7: $r = r - \binom{m'}{h}$
8: $h = h - 1$

Output: $y \in CW(m, k)$

Intuitively, this procedure is assigning the i^{th} valid codeword from a sorted list of codewords to the number i. Creating this list and extracting the mapping corresponding to a number would be prohibitively expensive with an average complexity of $\theta\left(\binom{m}{k}\right)$. The complexity of our mapping procedure is $O(m + k)$.

Since the mapping is one-on-one, there also exists an inverse mapping which is described in Algorithm 6. Similar to the mapping, the complexity of the inverse mapping procedure to $O(m+k)$.

Algorithm 6 INVERSE PERFECT MAPPING

Input: $y \in CW(m,k)$

$x = 0$
$h = 1$
for $m' \in [m]$ **do**
  if $y[m'] = 1$ **then**
    $x = x + \binom{m'}{h}$
    $h = h + 1$

Output: $x \in \mathbb{N}_0$

The perfect mapping also preserves the order between the mapped elements. This is useful in applications where it is important to preserve the ordering of elements in the domain, e.g., comparison operators.

Lossy Mapping. In some cases, we may need to map elements of some large domain to constant-weight codewords but the size of the domain is too large to assign a distinct codeword to each element. Recall that if S is the domain, the code length, m, needs to be chosen such that $\binom{m}{k} \geq |S|$ which results in a prohibitively large m.

To address this issue, we propose a lossy mapping inspired by Bloom filters. The procedure for the lossy mapping is given in Algorithm 7.

Based on the definition, a probability exists that unequal elements of the domain are mapped to the same codeword which is formalized in the following theorem.

Theorem 4. *In Algorithm 7, assume $(H_i : S \mapsto [m])_{i\in\mathbb{N}}$ is a series of uniformly random hash functions and $M_{m,k}(x)$ is the output of the algorithm for input x, m, and k with $(H_i)_{i\in\mathbb{N}}$ as the parameters. For two randomly chosen elements $x, y \in S$ such that $x \neq y$,*

$$\mathbb{P}\left[M_{m,k}(x) = M_{m,k}(y)\right] = \frac{1}{\binom{m}{k}}. \tag{3.10}$$

Proof. To prove this theorem, it suffices to prove that for any given codeword in the range of $M_{m,k}(x)$ such as c,

$$\mathbb{P}\left[M_{m,k}(x) = c\right] = \frac{1}{\binom{m}{k}}.$$

Algorithm 7 LOSSY MAPPING

Parameters: Series of uniformly random hash functions $(H_i : S \mapsto [m])_{i \in \mathbb{N}}$
Input: $x \in S, m, k \in \mathbb{N}$

```
1: cnt ← 0
2: i ← 1
3: y ← 0^m
4: while cnt < k do
5:     m' = H_i(x)
6:     if y[m'] = 0 then
7:         y_[m'] = 1
8:         cnt = cnt + 1
9:     i = i + 1
```

Output: $y \in CW(m, k)$

We prove this by induction over k. For $k = 1$, it is easy to see that

$$\mathbb{P}\left[M_{m,1}(x) = c\right] = \frac{1}{m}$$

for any $c \in Range(M_{m,1}(x))$.

Let $I(c)$ denote the positions in the codeword c where the bit is set to one. For $k > 1$, the probability that $H_1(x) \in I(c)$ is equal to $\frac{k}{m}$. By induction, the probability that set of the next $k-1$ distinct outputs in the series $(H_i(x))_{i \geq 2}$ is equal to $I(c) - \{H_1(x)\}$ is equal to $\frac{1}{\binom{m-1}{k-1}}$. Hence

$$\mathbb{P}\left[M_{m,k}(x) = c\right] = \frac{k}{m} \frac{1}{\binom{m-1}{k-1}} = \frac{1}{\binom{m}{k}}.$$

□

Due to the lossy nature of the mapping, an inverse mapping is not available for the lossy mapping.

3.3 PIR using Constant-weight Codewords

In this section, we will describe our protocol for PIR using constant-weight codewords. Our protocol follows the approach using equality operators with the plain constant-weight

Table 3.1: Stages of PIR using constant-weight codewords

Stage	Performed by	Offline/Online	Functionality	Computational Complexity
Setup	Server	Offline	Set Parameters, Put DB in plaintext format	$O(n)$
Query	Client	Online	Construct query, Send to Server	$O(m)$
Process	Server	Online	Query Expansion	$O(m)$
			Selection Vector Calculation	$O(n)$
			Inner product with DB	$O(ns)$
Extract	Client	Online	Decrypt & decode the server's response	$O(s)$

equality operator at its core and is the first practical and scalable PIR protocol using the equality operator approach.

The PIR protocol is conducted between a server and user. The server holds a database, $\mathbb{DB}$, with n entries. Each entry is accessible by a unique identifier. We denote the set of identifiers in the database by $\mathbb{ID}$. The user holds a query q from the domain of identifiers which we denote by $S(\mathbb{ID})$. We know by definition that $\mathbb{ID} \subseteq S(\mathbb{ID})$, but the user's query might not necessarily be in the database. Previous work, including SEALPIR and MulPIR, focuses mainly on PIR when $|S(\mathbb{ID})| = |\mathbb{ID}| = n$, i.e., index PIR, and keyword PIR is reduced to index PIR. In contrast, our work is applicable for both index and keyword PIR. We describe index PIR first and expand on keyword PIR in Sections 3.3.5 and 3.3.6.

The protocol consists of four main stages: Setup, Query, Process, and Extract. The Setup is an offline stage, whereas the other three stages happen online. An offline stage does not depend on the user's query and the server can perform such as stage before the user sends its query to reduce latency. Table 3.1 summarizes the stages of our PIR protocol. In the following sections, we will describe each stage in detail.

3.3.1 Setup

In this stage, parameters for homomorphic encryption system are chosen such that they meet the security requirements. The payload data within each row of the database is then converted into FV plaintexts. For this protocol, only the contents of each database row must be converted to plaintexts, not the set of identifiers. This stage only depends on the choice of encryption parameters and can be done without regard to the user's query. After this offline stage the server holds a table of plaintexts with n rows and at most s plaintexts in each row, for some $s \geq 1$.

3.3.2 Query

In this stage, the user constructs its query in the appropriate format and sends it to the server. First, parameters for the user's query are chosen. The Hamming weight, k, is chosen and then the code length m, is derived such that $\binom{m}{k} \geq n$. The user then constructs its query as depicted by Algorithm 8. Let $q \in S(\mathbb{ID})$ denote the user's query. The user maps its query to a constant-weight code from $CW(m, k)$. Let E_q denote the mapping of q. E_q is then converted to FV plaintexts as shown in lines 2–4 of Algorithm 8. The compression factor, c, indicates how many bits of the user's query are in each plaintext. Specifically, for $c \in \{0, 1, ..., \log_2 N\}$, exactly 2^c bits are in each plaintext. A higher compression factor reduces the upload cost but requires more computation for decompression, as we will see the next stage. Finally, the plaintexts are encrypted using the user's secret key. The client sends the output of Algorithm 8 along with m, k, and c to the server for the next stage.

Algorithm 8 QUERY

Input: $q \in S(\mathbb{ID})$, $m, k \in \mathbb{N}$, $c \in \{0, 1, ..., \log_2 N\}$

1: $E_q \leftarrow \texttt{MapToConstantWeightCode}(q, m, k)$
2: $h = \lceil \frac{m}{2^c} \rceil$
3: **for** $i \in [h]$ **do**
4: $\quad m_i(x) = \sum_{j \in [2^c]} 2^{-c} \cdot E_q[i2^c + j] \cdot x^j$
5: **for** $i \in [h]$ **do**
6: $\quad ct_i(x) = \texttt{Enc}(\texttt{sk}, m_i(x))$

Output: $(ct_i(x))_{i \in [h]}$

3.3.3 Process Query

The server conducts this stage which consists of three steps: Query Expansion, Selection Vector Calculation, and Inner Product.

Query Expansion. In the first step, the server expands the ciphertexts received by the user such that each bit of the user's query is in a separate ciphertext. Algorithm 9 describes the query expansion procedure, which is a modified version of Algorithm 1. We replace the use of two substitutions and one plaintext multiplication in the inner loop of Algorithm 1

with one substitution and two plaintext multiplications. Since substitution is slower compared to plain multiplication, as indicated in Table 2.1, there is an overall speedup. This modification in the expansion algorithm was first adopted in the implementation of MulPIR from the OpenMined community.[1]

Algorithm 9 Query Expansion

Input: $(ct_j(x)) \in \mathcal{C}^{\lceil \frac{m}{2^c} \rceil}$, $m \in \mathbb{N}$, $c \in \{0, 1, ..., \log_2 N\}$

1: $h = \lceil \frac{m}{2^c} \rceil$
2: $ctxts \leftarrow []$
3: **for** $j \in [h]$ **do**
4: $\quad cts \leftarrow [ct_j]$
5: $\quad$ **for** $a \in [c]$ **do**
6: $\quad\quad$ **for** $b \in [2^a]$ **do**
7: $\quad\quad\quad c_0 \leftarrow cts[b]$
8: $\quad\quad\quad c_0 \leftarrow \mathtt{Sub}_{N/2^a+1}(c_0)$
9: $\quad\quad\quad c_1 \leftarrow x^{-2^a} \cdot c_0$
10: $\quad\quad\quad cts[b + 2^a] \leftarrow x^{-2^a} \cdot cts[b]$
11: $\quad\quad\quad cts[b] \leftarrow cts[b] + c_0$
12: $\quad\quad\quad cts[b + 2^a] \leftarrow cts[b + 2^a] - c_1$
13: $\quad ctxts \leftarrow ctxts||cts$

Output: $ctxts \in C^m$

We prove the correctness of this procedure by showing it is equivalent to Algorithm 1, which has been proven to be correct by Angel et al. [6]. The for loop on line 6 of Algorithm 9 can be executed in parallel.

Theorem 5. *The output of Algorithm 9 is identical to that of Algorithm 1.*

Proof. To prove the correctness of the oblivious expansion in Algorithm 9, we prove it is equivalent to the oblivious expansion of SEALPIR, shown in Algorithm 1. For this, we prove that line 4–7 of Algorithm 1 is equivalent to line 7–12 of Algorithm 9.

In Algorithm 1, denote $cts[b]$ on line 4 by $m(x)$ for simplicity. By executing lines 4 to

[1] https://github.com/OpenMined/PIR

7, of the protocol, we can see that the new values for $cts[b]$ and $cts[b + 2^a]$ are

$$\begin{aligned} cts[b] &\leftarrow m(x) + \texttt{Sub}_{N/2^a+1}(m(x)) \\ cts[b + 2^a] &\leftarrow x^{-2^a} \cdot m(x) + \texttt{Sub}_{N/2^a+1}(x^{-2^a} \cdot m(x)) \end{aligned}$$

Similarly for Algorithm 9 and denoting $cts[b]$ on line 7 as $m(x)$, by executing lines 7 to 12, the new values for $cts[b]$ and $cts[b + 2^a]$ are

$$\begin{aligned} cts[b] &\leftarrow m(x) + \texttt{Sub}_{N/2^a+1}(m(x)) \\ cts[b + 2^a] &\leftarrow x^{-2^a} \cdot m(x) - x^{-2^a} \cdot \texttt{Sub}_{N/2^a+1}(m(x)) \end{aligned}$$

So $cts[b]$ gets the same value after both protocols. To show that $cts[b + 2^a]$ also gets the same value, it suffices to show that $\texttt{Sub}_{N/2^a+1}(x^{-2^a} \cdot m(x)) = -x^{-2^a} \cdot \texttt{Sub}_{N/2^a+1}(m(x))$ which can be proven as follows:

$$\begin{aligned} \texttt{Sub}_{N/2^a+1}(x^{-2^a} \cdot m(x)) &= (x^{N/2^a+1})^{-2^a} \cdot m(x^{N/2^a+1}) \\ &= x^{-N-2^a} \cdot m(x^{N/2^a+1}) \\ &= -x^{-2^a} \cdot m(x^{N/2^a+1}) \\ &= -x^{-2^a} \cdot \texttt{Sub}_{N/2^a+1}(m(x)) \end{aligned}$$

□

The output of this step is a vector of m ciphertexts, where each ciphertext contains one of the bits of E_q, i.e., the encoded query.

Selection Vector Calculation. In this step, the server creates the selection vector using the expanded query from the output of the previous step. For this, the server iterates over $\mathbb{ID}$, the set of identifiers in the database, maps each identifier to a constant-weight codeword and performs the equality operator between the mapped identifier and the user's query. We use the plain constant-weight equality operator since one of the operators is unencrypted. Algorithm 10 depicts this step with the output from the query expansion as input.

This is the most computationally expensive step of the protocol, however, it can be done in parallel across the identifiers in the database. The output of this stage is an encrypted selection vector of size n, with each bit in a separate ciphertext.

Algorithm 10 Selection Vector Calculation

Input: $ctxts \in \mathcal{C}^m$

1: $sel \leftarrow []$
2: **for** $i \in [n]$ **do**
3: $\quad E \leftarrow \texttt{MapToConstantWeightCode}(\mathbb{ID}[i], m, k)$
4: $\quad sel[i] = \prod_{E[j]=1} ctxts[j]$

Output: $sel \in \mathcal{C}^n$

Inner Product. In the last step of this stage, an inner product is performed between the selection vector derived from the previous step and the database. Each row of the database contains at most s plaintexts from the setup phase, hence s inner products are performed and s ciphertexts are sent to the user as the response. Each inner product operation includes n plaintext multiplication which can be done in parallel. The s inner products can also be done in parallel when s is large to enhance performance. The output of the inner product is sent to the user for the next stage.

3.3.4 Extract

In the last stage, the user decrypts the ciphertext(s) received from the server. The results are extracted from the decrypted messages by the client.

3.3.5 PIR for Sparse Databases

Recall that $\mathbb{ID}$ is the list of identifiers in the database, and $S(\mathbb{ID})$ refers to the domain of identifiers, i.e., the set of all possible identifiers. By definition, $\mathbb{ID} \subseteq S(\mathbb{ID})$. In the previous sections, we have discussed PIR in the case where $\mathbb{ID} = S(\mathbb{ID})$. Related work has also mainly focused on PIR under this assumption [4, 6]. A sparse database, however, specifies the case where $\mathbb{ID}$ where is much smaller than $S(\mathbb{ID})$. In this case, not all identifiers in the domain are associated with an element in the database.

The architecture described in this section is applicable when the database is sparse, with computation on the order of the database size, not the domain size. For this, the following changes must be made to the protocol.

- In the query stage, the code length, m, and Hamming weight, k, are chosen such that $\binom{m}{k} \geq |S(\mathbb{ID})|$.
- In the selection vector calculation step, encrypted bits of the selection vector are generated only for identifiers in the database, i.e., the for loop on line 4 of Algorithm 10 is performed only over the identifiers in the database.
- Similarly in the inner product step, we only perform plain multiplications and sum for identifiers in the database.

PIR solutions based on selection vectors have a computational complexity that depends on the domain size, which makes them unsuitable for sparse PIR.

3.3.6 Probabilistic Keyword PIR

In some cases, the domain of identifiers is so large that a sparse domain is not feasible either, i.e., the code size, m, is prohibitively large. For example, the identifiers might be a string denoting the name of a file. A probabilistic version of the protocol is applicable in this scenario, which we denote *probabilistic keyword PIR*. For this, we apply the following changes to the protocol:

- In the query stage, we use the lossy mapping. For a database with n rows and a chosen maximum failure rate, α, we choose the code length, m, and Hamming weight, k, such that $n/\binom{m}{k} \leq \alpha$. The user then uses the lossy mapping to construct the query.
- In the process stage, identifiers in the database are also mapped using the lossy mapping.

Failure in probabilistic keyword PIR results in an incorrect response. This occurs when the user's query maps to the same code as an identifier in the database, while not being equal to it. The following theorem holds regarding the failure rate.

Theorem 6. *Let $\mathbb{DB}$ be a database with n rows. For probabilistic keyword PIR using constant-weight codewords, failure rate is at most $\texttt{ERR} = n/\binom{m}{k}$ where m is the code length and k is the Hamming weight.*

Proof. The probability of the user's query colliding with any one of the identifiers is derived using Theorem 4. By applying a union bound, the theorem can be proven. □

Chapter 4

Evaluation

For our evaluation, we aim to conduct experiments for two purposes. First, compare equality operators proposed in this work with existing operators. Second, to compare PIR using constant-weight codewords with existing PIR protocols.

In the first subsection, we compare equality operators in two categories, plain operators (operators where one operand is public) and arithmetic operators (operators oblivious to the inputs which can be performed over an arbitrary field). Plain operators are at the core of the PIR protocols discussed in Chapter 3. Arithmetic operands are also useful in computation over encrypted data such as PSI, when circuits must be oblivious to both operands. For example, the PSI protocol proposed by Kacsmar et al. [26] performs an equality check on a many pairs of number in parallel, hence the circuit used for this purpose can not depend on any of the inputs.

In the next subsection, we evaluate PIR protocols as parameters such as the domain and response size scale. We compare the performance of PIR protocols discussed in this work in terms of runtime and communication, in multiple scenarios. In our experiments, we also apply parallelization, where applicable, to observe the effect on runtime. We argue that some protocols achieve a higher speedup when run in parallel compared to others.

4.1 Comparing Equality Operators

The equality operators discussed in this thesis fall into three categories:

- Plain equality operators, where one operand is public, i.e., the circuit depends on

Table 4.1: Properties of circuits implementing equality operators mentioned in this work. Properties include the number of homomorphic and plain multiplications, multiplicative depth, and size of encoded elements. Parameters are chosen such that the size of the domain is at least n. M denotes a homomorphic multiplication.

Operator	Domain	# of Operations	Multiplicative Depth	Conditions
Plain Folklore	$\{0,1\}^\ell$	$\ell \cdot \mathtt{M}$	$\lceil \log_2 \ell \rceil$	$\ell \geq \log_2 n$
Plain Constant-weight	$CW(m,k)$	$k \cdot \mathtt{M}$	$\lceil \log_2 k \rceil$	$\binom{m}{k} \geq n$
Arithmetic Folklore	$\{0,1\}^\ell$	$2\ell \cdot \mathtt{M}$	$\lceil 1 + \log_2 \ell \rceil$	$\ell \geq \log_2 n$
Arithmetic Constant-weight	$CW(m,k)$	$(m+k) \cdot \mathtt{M}$	$\lceil \log_2 k \rceil$	$\binom{m}{k} \geq n$
Binary Folklore	$\{0,1\}^\ell$	$\ell \cdot \mathtt{M}$	$\lceil \log_2 \ell \rceil$	$\ell \geq \log_2 n$
Binary Constant-weight	$CW(m,k)$	$m(\log_2 k + 1) \cdot \mathtt{M}$	$\lceil 1 + \log_2 k \rceil$	$\binom{m}{k} \geq n$
Binary Raffle	$\{0,1\}^*$	$N(\epsilon) \cdot \mathtt{M}$	$\lceil \log_2 N(\epsilon) \rceil$	-

one of the operands. We consider two candidates in this category: the plain folklore and the plain constant-weight equality operator.

- Arithmetic equality operators, where the circuit is oblivious to both operands and operates over an arbitrary field. We consider the folklore arithmetic and the constant-weight arithmetic equality operators in this category.

- Binary equality operators, where the circuit is oblivious to both operands and operates over a binary field. We consider the binary folklore, the binary constant-weight, and the binary raffle equality operators in this category.

Table 4.1 summarizes these operators, along with the properties of circuits that implement each of them. We include properties that significantly influence the runtime such as the number of homomorphic operations (plain and homomorphic multiplications), and the multiplicative depth. Note that different circuits operate over different domains, which are stated in Table 4.1, but for a fair comparison, we select parameters such that all domains are of the same size. Conditions for parameters are also in the table. Mappings (such as those proposed in Section 3.2) can be used to compared elements from domains other than those stated in the table.

The experiments in this section cover plain and arithmetic operators since the homomorphic encryption library that we use, SEAL, is not a suitable choice when operating over a binary field. We leave implementation of the binary operators with a suitable library for future work.

In the experiments, we vary the domain size to observe the effect on the performance of the equality circuits. We implement all circuits using C++ and the SEAL library. Within the SEAL library, we use three different encryption parameters specified by N, the polynomial modulus degree, where $N \in \{4096, 8192, 16384\}$. The default ciphertext modulus is used to achieve 128-bit security. We also run all experiments both in single-thread and in parallel across multiple cores. The goal is to see the amount of speed-up in each circuit when run in parallel.

All circuits are run in a SIMD fashion using the batch encoding functionality of SEAL. Using this feature, N elements can be compared at the same time. In plain operators, since the circuit depends on the plain operands, this means that N elements are compared to the same operand in the clear. This is not the case for the arithmetic operand. N pairs of numbers are compared simultaneously. The runtime can be divided by N to achieve the amortized cost of one equality check.

We run all experiments on an Intel Xeon E5-4640 @ 2.40GHz server with 1/2 TB RAM and 32 cores running Ubuntu 16.04.

4.1.1 Plain Operators

Table 4.2 summarizes the results of our experiments for plain equality operators. We report the results for the plain constant-weight operator in four categories based on the relationship between ℓ and k. For a given ℓ, runtimes using different Hamming weights are within the same column.

The constant-weight plain operator consistently outperforms the folklore operator in terms of running time. The advantage is greater when smaller homomorphic encryption parameters (namely N) can be used. This is possible due to a smaller multiplicative depth compared to the folklore circuit in cases where $k < \ell$. However, the advantage exists even when using the same homomorphic encryption parameters. This can be attributed to the fewer multiplications that are performed when a small Hamming weight is used. To achieve the best runtime, the Hamming weight must be chosen according to ℓ.

Faster runtimes for the plain constant-weight circuit come at the cost of higher memory usage during the protocol. The memory usage depends on the code length, also specified in the table. Depending on the application, the code length determines the communication complexity if operands are communicated over the network.

Parallelization offers roughly up to 10× speedup for both circuits and there is no noticeable difference in the advantage that parallel implementation offers for both circuits.

Table 4.2: Runtimes for plain equality operators in milliseconds. Dashes indicate cases where the ciphertext was undecryptable due to homomorphic noise.

		Single-thread									Parallel (64 threads)								
		ℓ	4	8	16	32	64	128	256	512	ℓ	4	8	16	32	64	128	256	512
Plain Folklore		Mult Depth	2	3	4	5	6	7	8	9	Mult Depth	2	3	4	5	6	7	8	9
		$N = 4096$	-	-	-	-	-	-	-	-	$N = 4096$	-	-	-	-	-	-	-	-
		$N = 8192$	120	274	541	-	-	-	-	-	$N = 8192$	175	205	258	-	-	-	-	-
		$N = 16384$	513	1140	2428	5027	10309	21030	42251	83924	$N = 16384$	462	748	962	1323	1943	2795	4314	7612
Plain Constant-weight	$\ell = k$	Hamming weight	4	8	16	32	64	128	256	512	Hamming weight	4	8	16	32	64	128	256	512
		Mult Depth	2	3	4	5	6	7	8	9	Mult Depth	11429	3	4	5	6	7	8	9
		Code Length (m)	7	12	22	43	85	168	334	665	Code Length (m)	7	12	22	43	85	168	334	665
		$N = 4096$	-	-	-	-	-	-	-	-	$N = 4096$	-	-	-	-	-	-	-	-
		$N = 8192$	118	270	566	-	-	-	-	-	$N = 8192$	135	186	283	-	-	-	-	-
		$N = 16384$	492	1126	2595	4985	10131	20557	40852	81773	$N = 16384$	415	584	1026	1248	1946	2670	4093	6975
Plain Constant-weight	$\ell = 2k$	Hamming weight	2	4	8	16	32	64	128	256	Hamming weight	2	4	8	16	32	64	128	256
		Mult Depth	1	2	3	4	5	6	7	8	Mult Depth	1	2	3	4	5	6	7	8
		Code Length (m)	7	11	19	36	68	132	261	517	Code Length (m)	7	11	19	36	68	132	261	517
		$N = 4096$	10	-	-	-	-	-	-	-	$N = 4096$	37	-	-	-	-	-	-	-
		$N = 8192$	41	119	272	551	-	-	-	-	$N = 8192$	86	150	180	240	-	-	-	-
		$N = 16384$	171	489	1142	2445	5019	10437	21519	41291	$N = 16384$	189	375	759	874	1398	2185	2839	4180
Plain Constant-weight	$\ell = 4k$	Hamming weight	1	2	4	8	16	32	64	128	Hamming weight	1	2	4	8	16	32	64	128
		Mult Depth	0	1	2	3	4	5	6	7	Mult Depth	0	1	2	3	4	5	6	7
		Code Length (m)	16	24	37	64	117	221	427	838	Code Length (m)	16	24	37	64	117	221	427	838
		$N = 4096$	0.12	10	-	-	-	-	-	-	$N = 4096$	0.09	27	-	-	-	-	-	-
		$N = 8192$	37	41	119	249	493	-	-	-	$N = 8192$	0.45	58	111	225	252	-	-	-
		$N = 16384$	1.9	173	486	1136	2444	5037	10077	21062	$N = 16384$	2.05	189	499	759	1038	1364	1822	2598
Plain Constant-weight	$\ell = 8k$	Hamming weight		1	2	4	8	16	32	64	Hamming weight		1	2	4	8	16	32	64
		Mult Depth		0	1	2	3	4	5	6	Mult Depth		0	1	2	3	4	5	6
		Code Length (m)		256	363	569	968	1749	3290	6349	Code Length (m)		256	363	569	968	1749	3290	6349
		$N = 4096$		0.11	8	-	-	-	-	-	$N = 4096$		0.12	28	-	-	-	-	-
		$N = 8192$		0.41	38	103	252	545	-	-	$N = 8192$		0.46	67	145	220	270	-	-
		$N = 16384$		2.7	178	500	1147	2425	5039	10438	$N = 16384$		2.64	293	537	733	1087	1434	1824

4.1.2 Arithmetic Operators

Table 4.3 summarizes the results of our experiments for arithmetic equality operators. Similar to before, we report the results for the arithmetic constant-weight operator in four categories based on the relationship between ℓ and k. For a given ℓ, runtimes using different Hamming weights are within the same column.

Unlike the plain operators, the constant-weight arithmetic operator is not always faster than the equivalent folklore arithmetic equality circuit. This is due to the large number of homomorphic multiplications that are required in a constant-weight arithmetic circuit $(m + k)$. Specifically, when the encoding size m is large due to a small Hamming weight k, the number of multiplications can be very high compared to the folklore. However, in some

Table 4.3: Runtimes for arithmetic equality operators in milliseconds. Dashes indicate cases where the ciphertext was undecryptable due to homomorphic noise.

		Single-thread									Multi-thread								
		ℓ	4	8	16	32	64	128	256	512	ℓ	4	8	16	32	64	128	256	512
Arithmetic Folklore		Mult Depth	3	4	5	6	7	8	9	10	Mult Depth	3	4	5	6	7	8	9	10
		$N = 4096$	-	-	-	-	-	-	-	-	$N = 4096$	-	-	-	-	-	-	-	-
		$N = 8192$	241	490	-	-	-	-	-	-	$N = 8192$	274	433	-	-	-	-	-	-
		$N = 16384$	1014	2237	4636	9197	19050	37283	74451	149282	$N = 16384$	1068	1757	3098	5604	10397	20165	37921	73966
Arithmetic Constant-weight	$\ell = k$	Hamming weight	4	8	16	32	64	128	256	512	Hamming weight	4	8	16	32	64	128	256	512
		Mult Depth	3	4	5	6	7	8	9	10	Mult Depth	3	4	5	6	7	8	9	10
		Encoding size	7	12	22	43	85	168	334	665	Encoding size	7	12	22	43	85	168	334	665
		$N = 4096$	-	-	-	-	-	-	-	-	$N = 4096$	-	-	-	-	-	-	-	-
		$N = 8192$	385	692	-	-	-	-	-	-	$N = 8192$	204	289	-	-	-	-	-	-
		$N = 16384$	1621	3065	6048	12029	23677	46960	93774	186824	$N = 16384$	722	1072	1418	2013	2788	4483	8224	14369
Arithmetic Constant-weight	$\ell = 2k$	Hamming weight	2	4	8	16	32	64	128	256	Hamming weight	2	4	8	16	32	64	128	256
		Mult Depth	2	3	4	5	6	7	8	9	Mult Depth	2	3	4	5	6	7	8	9
		Encoding size	7	11	19	36	68	132	261	517	Encoding size	7	11	19	36	68	132	261	517
		$N = 4096$	-	-	-	-	-	-	-	-	$N = 4096$	-	-	-	-	-	-	-	-
		$N = 8192$	310	538	-	-	-	-	-	-	$N = 8192$	176	220	366	-	-	-	-	-
		$N = 16384$	1291	2252	4340	8247	16161	31195	63830	123168	$N = 16384$	448	845	1113	1671	2629	3997	6565	9561
Arithmetic Constant-weight	$\ell = 4k$	Hamming weight	1	2	4	8	16	32	64	128	Hamming weight	1	2	4	8	16	32	64	128
		Mult Depth	1	2	3	4	5	6	7	8	Mult Depth	1	2	3	4	5	6	7	8
		Encoding size	16	24	37	64	117	221	427	838	Encoding size	16	24	37	64	117	221	427	838
		$N = 4096$	152	-	-	-	-	-	-	-	$N = 4096$	67	-	-	-	-	-	-	-
		$N = 8192$	572	854	1346	-	-	-	-	-	$N = 8192$	124	296	423	-	-	-	-	-
		$N = 16384$	2562	4385	6474	11429	21156	40279	78214	154572	$N = 16384$	492	701	1081	1570	2839	4276	7173	12479
Arithmetic Constant-weight	$\ell = 8k$	Hamming weight		1	2	4	8	16	32	64	Hamming weight		1	2	4	8	16	32	64
		Mult Depth		1	2	3	4	5	6	7	Mult Depth		1	2	3	4	5	6	7
		Encoding size		256	363	569	968	1749	3290	6349	Encoding size		256	363	569	968	1749	3290	6349
		$N = 4096$		2078	-	-	-	-	-	-	$N = 4096$		447	-	-	-	-	-	-
		$N = 8192$		8483	11998	19033	-	-	-	-	$N = 8192$		818	1151	1621	-	-	-	-
		$N = 16384$		41138	58104	91925	156589	282136	533596	1064621	$N = 16384$		3045	4540	7078	12449	20359	37048	73267

cases, the smaller Hamming weight results in a lower multiplicative depth, which allows the use of smaller homomorphic encryption parameters (N). For example, for $\ell = 16$ and Hamming weight of 4, the constant-weight using $N = 8192$ is about 4 times faster than the folklore using $N = 16384$. The amortized cost is also about 2 times faster.

Similar to the plain equality operators, high memory usage is also an issue with the arithmetic constant-weight equality operator and it requires much more memory than the equivalent folklore operator.

The effect of the parallelization is however substantially different between folklore and constant-weight operators. The folklore circuit runs at most 2 times faster with parallelization, whereas the constant-weight circuit has more than a $10\times$ speedup in some cases. The speedup is larger as ℓ grows. The speedup is mainly due to the m homomorphic multiplications that can be done in parallel.

4.2 Comparing PIR Protocols

In this section, we present the results of experiments conducted to compare the PIR protocols mentioned in this work. Specifically, we compare four protocols:

- PIR using the folklore equality circuit (which we call folklore PIR)
- PIR using the constant-weight equality circuit (which we call constant-weight PIR)
- SEALPIR [4]
- MulPIR [4]

SEALPIR and MulPIR are based on the approach where the selection vector is communicated to the server, whereas folklore PIR and constant-weight PIR make use of equality operators. We aim to compare the two general methods (selection vectors vs. equality circuits) while also evaluating constant-weight PIR against folklore PIR.

Unary Approach. Note that SEALPIR and MulPIR with $d = 1$ are equivalent to constant-weight PIR when $k = 1$. Hence, we refer to this configuration as the *unary* approach and report runtimes separately as a baseline for the other methods.

4.2.1 Implementation and Experimental Details

Constant-weight PIR is implemented as described in Section 3.3. We also implement folklore PIR using the same architecture and consisting of the same steps described in Section 3.3. However, we use a logarithmic binary encoding for indices and the equality operator is replaced with a plain folklore equality operator per definition in Equation (2.6). In experiments where parallelization is activated, the process stage is performed in parallel across the rows in the database (not across the multiplications in the equality circuit).

We implement all protocols using C++ and SEAL[1] (version 3.6) as the homomorphic encryption library. For SEALPIR and MulPIR, we use the implementation by the OpenMined community.[2] This implementation does not include parallelization so we exclude SEALPIR/MulPIR from experiments where the effect of parallelization is taken into account.

[1] https://github.com/microsoft/SEAL
[2] https://github.com/OpenMined/PIR

We select homomorphic encryption parameters such that it satisfies 128-bit security. Specifically, we use $N \in \{4096, 8192, 16384\}$ and the default coefficient modulus in SEAL for 128-bit security. Each protocol is run with the smallest parameter set which produces decryptable results. Specifically, SEALPIR uses $N = 4096$, whereas MulPIR, folklore PIR, and constant-weight PIR require $N = 8192$.

We run all experiments on an Intel Xeon E5-4640 @ 2.40GHz server with 1/2 TB RAM and 32 cores running Ubuntu 16.04.

We conduct experiments for three scenarios which vary three parameters: the number of rows (number of keywords which correspond to existing payload data in the database), the domain size (the size of the set of all possible queries), and the response size (the maximum size of the payload data within each row of the database). The scenarios are as follows:

- PIR over a database in which we vary the number of rows
- PIR over a database with a fixed number of rows, but with a varying domain size
- PIR over a database with a fixed number of rows, but with varying response size

In the next subsections, we explain each scenario in more detail, provide results for the three scenarios outlined above, and discuss the conclusion derived from each one.

4.2.2 Packed Database Experiments

In this scenario, we perform PIR using a database with the four protocols mentioned above. We assume each database row contains payload data equal to the size of precisely one plaintext, hence the name *packed*. This is the scenario that is focused on in related work. While some real-world databases might not initially meet this requirement, they are restructured using the packing techniques for this purpose [4, 7]. Another way to express a packed database is with the condition $|S(\mathbb{ID})| = |\mathbb{ID}|$. This implies that all database rows are full (in contrast to the next scenario where some rows might be empty). Table 4.4 compares the properties of the four aforementioned protocols.

In our experiments, the size of a plaintext depends on the homomorphic encryption parameters used in each approach. So, for a fair comparison, we compare runtimes for different protocols when the database size (in MB) is roughly the same.

We report the results in two tables. In Table 4.5, approaches using equality circuits, folklore and constant-weight PIR are compared. In Table 4.6, we compare constant-weight

Table 4.4: Parameters for PIR using SEALPIR, MulPIR, folklore PIR, and constant-weight PIR when $|S(\mathbb{ID})| = |\mathbb{ID}| = n$. The ciphertext expansion factor is denoted by F. PM, M indicate plain multiplication and homomorphic multiplication, respectively.

Method	**Mult Depth**	**Query Bit-length**	**# of Operations**	**Download Cost (in ciphertexts)**
SEALPIR	$d-1$	$d\left\lceil \sqrt[d]{n} \right\rceil$	$\sum_{i=0}^{d-1} n^{\frac{d-i}{d}} F^i \cdot \texttt{PM}$	F^{d-1}
MulPIR	$d-1$	$d\left\lceil \sqrt[d]{n} \right\rceil$	$(n \cdot \texttt{PM} + \sum_{i=1}^{d-1} n^{\frac{d-i}{d}} \cdot \texttt{M})$	1
Folklore PIR	$\lceil \log_2 \lceil \log_2 n \rceil \rceil$	$\lceil \log_2 n \rceil$	$n \lceil \log_2 n \rceil \cdot \texttt{M} + n \cdot \texttt{PM}$	1
Constant-weight PIR	$\lceil \log k \rceil$	$O\left(\sqrt[k]{k!n}\right)$	$nk \cdot \texttt{M} + n \cdot \texttt{PM}$	1

PIR with approaches using selection vectors, i.e., SEALPIR and MulPIR. The runtimes in these two tables are not accelerated using parallelization.

Table 4.5 shows the folklore is much slower than the other protocols in the same table. At $\ell = 512$, the parameters of the homomorphic cryptosystem must be increased from $N = 8192$ to $N = 16384$ to produce valid, decryptable results. Larger parameters increase the runtime drastically.

The unary approach is the fastest approach amongst the three in Table 4.5, since the number of homomorphic operations is the fewest (no homomorphic multiplications). However, the communication cost, specifically the upload cost, increases very quickly and the expansion step constitutes the bulk of the runtime due to a large query size. Constant-weight PIR with $k = 2$ lies between folklore PIR and the unary approach. The upload cost is the same for all database sizes in Table 4.5. The runtime, albeit higher than the unary approach, is an order of magnitude less than folklore PIR. Consequently, constant-weight PIR is the first practical PIR protocol using equality operators.

Table 4.6 compares constant-weight PIR to SEALPIR and MulPIR. This tables includes runtimes and communication sizes. However, communication sizes for SEALPIR and MulPIR are reported from the paper [4] since the implementation does not report the sizes of the messages. Runtimes in this table are much smaller so we can examine larger database sizes as well. SEALPIR has a large communication cost, specifically because of the larger download cost compared to the other protocols. However, the runtime is less than that of constant-weight PIR and MulPIR for the reported database sizes. MulPIR and constant-weight PIR have the smallest communication cost, with MulPIR having a smaller runtime in this scenario. This can be attributed to the smaller number of homomorphic operations in MulPIR compared to constant-weight PIR.

Table 4.5: Runtimes for Folklore and Constant-weight PIR

# DB Rows (n)	DB Size (MB)	Query Bit-length	Communication (KB)		Time (ms)				Valid Response
			Query	Response	Expansion	Sel. Vec. Calculation	Inner Product	Total Server	
Folklore ($N = 8192$)									
256	5.242	8	216	103	66	57748	815	58829	✓
512	10.485	9	216	103	125	132093	1618	134038	✗
1024	20.971	10	216	103	120	296654	3209	300197	✗
2048	41.943	11	216	103	121	658535	6354	665211	✗
4096	83.886	12	216	103	134	1456388	12964	1469687	✗
Folklore ($N = 16384$)									
4096	167.772	12	913	224	764	7086351	54440	7143207	✓
8192	335.544	13	913	224	813	15682684	113848	15799042	✓
16384	671.088	14	913	224	781	34603717	241270	34847343	✓
Unary ($N = 4096$)									
256	2.621	256	46	46	509	9	174	734	✓
512	5.242	512	46	46	973	20	376	1411	✓
1024	10.485	1024	46	46	1840	51	777	2704	✓
2048	20.971	2048	46	46	3688	140	1626	5491	✓
4096	41.943	4096	46	46	7468	447	3253	11207	✓
8192	83.886	8192	92	46	15080	1945	7043	24110	✓
16384	167.772	16384	185	46	29362	6622	12918	48951	✓
32768	335.544	32768	371	46	58691	23925	26590	109271	✓
65536	671.088	65536	743	46	117832	86717	55412	260057	✓
131072	1342.177	131072	1486	46	243702	334502	112959	691343	✓
Constant-weight ($k = 2, N = 8192$)									
256	5.242	24	216	103	266	8351	807	9621	✓
512	10.485	33	216	103	510	16656	1628	19005	✓
1024	20.971	46	216	103	518	33250	3248	37212	✓
2048	41.943	65	216	103	1051	66570	6288	74111	✓
4096	83.886	92	216	103	1296	132686	12703	146886	✓
8192	167.772	129	216	103	2227	265632	24595	292664	✓
16384	335.544	182	216	103	2416	538861	53444	595024	✓
32768	671.088	257	216	103	4466	1087970	110094	1202754	✓
65536	1342.177	363	216	103	5110	2242229	219007	2466662	✓

In Table 4.7, runtimes with parallelization are given to further demonstrate the practicality of constant-weight PIR. This table shows that when executed in parallel, constant-PIR has a 10× speedup and is faster than the existing implementation of MulPIR. A more detailed comparison of the effect of parallelization requires reimplementation of SEALPIR and MulPIR with parallelization in mind, which we leave for future work.

To summarize, in a packed database, SEALPIR and MulPIR are the best options in terms of runtime and communication cost, respectively and constant-weight PIR is not

advantageous in this scenario.

Table 4.6: Runtime of Constant-weight PIR, SEALPIR, and MulPIR

# DB Rows (n)	DB Size (MB)	Encoding Size	Communication (KB)		Time (ms)				Valid Response
			Query	Response	Expansion	Sel. Vec. Calculation	Inner Product	Total Server	
Constant-weight ($k = 2, N = 8192$)									
256	5.242	24	216	103	266	8351	807	9621	✓
512	10.485	33	216	103	510	16656	1628	19005	✓
1024	20.971	46	216	103	518	33250	3248	37212	✓
2048	41.943	65	216	103	1051	66570	6288	74111	✓
4096	83.886	92	216	103	1296	132686	12703	146886	✓
8192	167.772	129	216	103	2227	265632	24595	292664	✓
16384	335.544	182	216	103	2416	538861	53444	595024	✓
32768	671.088	257	216	103	4466	1087970	110094	1202754	✓
65536	1342.177	363	216	103	5110	2242229	219007	2466662	✓
SEALPIR ($d = 2, N = 4096$)									
512	4.98	46	61.4	307	-	-	-	344	✓
1024	9.96	64	61.4	307	-	-	-	463	✓
2048	19.92	92	61.4	307	-	-	-	809	✓
4096	39.85	128	61.4	307	-	-	-	1239	✓
8192	79.69	182	61.4	307	-	-	-	2244	✓
16384	159.38	256	61.4	307	-	-	-	3765	✓
32768	318.77	364	61.4	307	-	-	-	7025	✓
65536	637.53	512	61.4	307	-	-	-	12535	✓
131072	1275.07	726	61.4	307	-	-	-	24696	✓
262144	2550.14	1024	61.4	307	-	-	-	50722	✓
524288	5100.27	1450	61.4	307	-	-	-	100965	✓
1048576	10200.55	2048	61.4	307	-	-	-	199522	✓
2097152	20401.09	2898	61.4	307	-	-	-	430533	✓
MulPIR ($d = 2, N = 8192$)									
256	4.98	32	122	119	-	-	-	2384	✓
512	9.96	46	122	119	-	-	-	4125	✓
1024	19.92	64	122	119	-	-	-	6859	✓
2048	39.85	92	122	119	-	-	-	12759	✓
4096	79.69	128	122	119	-	-	-	22887	✓
8192	159.38	182	122	119	-	-	-	44018	✓
16384	318.77	256	122	119	-	-	-	83304	✓
32768	637.53	364	122	119	-	-	-	163926	✓
65536	1275.07	512	122	119	-	-	-	318856	✓
131072	2550.14	726	122	119	-	-	-	634210	✓
262144	5100.27	1024	122	119	-	-	-	1256404	✓
524288	10200.55	1450	122	119	-	-	-	2531407	✓

Table 4.7: Runtime of Constant-weight PIR executed with parallelization

# DB Rows (n)	DB Size (MB)	Encoding Size	Communication (KB)		Time (ms)				Valid Response
			Query Comm.	Response Comm.	Expansion	Sel. Vec. Calculation	Inner Product	Total Server	
256	5.242	24	216	103	137	392	174	915	✓
512	10.485	33	216	103	174	749	400	1523	✓
1024	20.971	46	216	103	153	1545	770	2669	✓
2048	41.943	65	216	103	192	2959	1688	5044	✓
4096	83.886	92	216	103	244	5621	3083	9150	✓
8192	167.772	129	216	103	300	11051	6483	18038	✓
16384	335.544	182	216	103	325	21704	13094	35332	✓
32768	671.088	257	216	103	507	42647	26456	69832	✓
65536	1342.177	363	216	103	526	85765	50539	137080	✓
131072	2684.354	513	216	103	948	172771	105093	279110	✓
262144	5368.709	725	216	103	1195	345965	223047	570666	✓

4.2.3 Varying Domain Size

In the previous section, we examined PIR over databases where all keywords/identifiers in the domain of keywords correspond to some payload data in the database. This is not always the case and sparse databases (keyword PIR) are an example of this.

In this scenario, we perform PIR over a sparse database by using different domain sizes but with a fixed number of rows. We examine the effect of a sparse database on SEALPIR, MulPIR and constant-weight PIR, and perform experiments for constant-weight PIR. SEALPIR and MulPIR are not specifically designed for a sparse domain so we can only estimate their performance. A more detailed comparison can be done with reimplementation of the protocols with sparsity in mind, which we leave to future work.

Table 4.8 shows the number of operations adjusted for when the database is sparse. n denotes the number of rows in the database, whereas $|S|$ refers to the size of the domain from which the query is selected.

We argue that constant-weight PIR is minimally affected by sparsity in the database and it is a suitable solution for sparse PIR. Table 4.8 which counts the number of operations performed on the server for all protocols supports this argument, as the number of operations does not depend on the size of the domain. The number of operations in this table does not include the expansion step. Note that we exclude folklore PIR from this section since it is strictly slower than constant-weight PIR.

Table 4.8 also shows the query bit-length of each method. The query bit-length determines the communication cost in the protocol and also affects the computation cost, specifically the expansion step. This parameter is affected by the domain size. The query

Table 4.8: Parameters for sparse PIR using SEALPIR, MulPIR, and constant-weight PIR. The ciphertext expansion factor is denoted by F. PM and M indicate plain multiplication and homomorphic multiplication, respectively.

Method	Mult Depth	Query Bit-length	# of Operations	Download Cost (in ciphertexts)
SEALPIR	$d-1$	$d\left\lceil \sqrt[d]{\lvert S\rvert} \right\rceil$	$n \cdot \texttt{PM} + \sum_{i=1}^{d-1} \lvert S\rvert^{\frac{d-i}{d}} F^i \cdot \texttt{PM}$	F^{d-1}
MulPIR	$d-1$	$d\left\lceil \sqrt[d]{\lvert S\rvert} \right\rceil$	$n \cdot \texttt{PM} + \sum_{i=1}^{d-1} \lvert S\rvert^{\frac{d-i}{d}} \cdot \texttt{M}$	1
Constant-weight PIR	$\lceil \log k \rceil$	$O\left(\sqrt[k]{k!\lvert S\rvert}\right)$	$nk \cdot \texttt{M} + n \cdot \texttt{PM}$	1

bit-length in constant-weight PIR is equal to the size of the constant-weight code that is used. SEALPIR and MulPIR use the same type of encoding for PIR queries which essentially calculates the position of the desired row of the database when restructured into a d-dimensional table. We denote this a *dimension-wise* encoding in this section.

Table 4.9 shows the number of bits required to represent a query using a constant-weight code and a dimension-wise encoding. The logarithmic binary encoding, used in folklore PIR, is given as a reference in the second column and is the most space-efficient representation of a query. In the next four columns, the constant-weight code size is shown for different values of k, the Hamming weight. Finally, In the last three columns, we derive the bit-length of the dimension-wise encoding. The depth refers to the multiplicative depth in a PIR protocol using the set of parameters in that column.

There are multiple observations from this table. Firstly, larger k or d (and higher multiplicative depth in turn) drastically reduces the bit-length of the query. Given this observation, a fair comparison between the constant-weight code and dimension-wise encoding is comparing those with the same multiplicative depth since the multiplicative depth directly impacts the performance. For the same multiplicative depth, the constant-weight code is smaller than the dimension-wise code. Figure 4.1 visualizes this for even larger domain sizes and higher multiplicative depths. Note that the scale on the vertical axis is logarithmic and the gap between the size of the codes increases as the domain size increases and a larger multiplicative depth is used.

The size of the query can also affect the server runtime in the protocol. Table 4.10 shows this effect by providing a breakdown of the server's runtime in constant-weight PIR. We fix the size of the database to roughly 330 MB.

The runtime of the protocol consists of the expansion step, and the iteration step

Table 4.9: Bit-length of the query in different protocols

Domain Bitlength	Log2 Binary Encoding	Constant-weight code size				Dimension-wise		
		depth=0	depth=1	depth=2		depth=0	depth=1	depth=2
		k=1	k=2	k=3	k=4	d=1	d=2	d=3
4	2	16	7	6	7	16	8	9
6	3	64	12	9	8	64	16	12
8	3	256	24	13	11	256	32	21
10	4	1024	46	20	15	1024	64	33
12	4	4096	92	31	20	4096	128	48
14	4	16384	182	48	27	16384	256	78
16	4	65536	363	75	37	65536	512	123
18	5	262144	725	118	52	262144	1024	192
20	5	-	1449	186	73	-	2048	306
22	5	-	2897	295	102	-	4096	486
24	5	-	5794	467	144	-	8192	768
26	5	-	11586	740	202	-	16384	1221
28	5	-	23171	1174	285	-	32768	1938
30	5	-	46342	1862	403	-	65536	3072
32	5	-	92683	2955	569	-	131072	4878
34	6	-	185365	4690	803	-	262144	7743
36	6	-	370729	7444	1135	-	524288	12288
38	6	-	741456	11816	1605	-	1048576	19506
40	6	-	-	18756	2268	-	-	30966
42	6	-	-	29773	3207	-	-	49152
44	6	-	-	47261	4535	-	-	78024
46	6	-	-	75021	6413	-	-	123858
48	6	-	-	119088	9068	-	-	196608

(which is selection vector calculation and inner product combined). We report numbers for $k \in \{2, 3, 4\}$ since we know that $k = 1$ produces an encoding size that is prohibitively large. Table 4.10 shows the runtimes of PIR with a varying domain size using constant-weight PIR, accelerated using parallelization. Initially, for a domain size up to 2^{27}, $k = 2$ has the smallest runtime. However, when the domain size approaches 2^{28}, the expansion time constitutes a significant portion of the runtime and a switch to $k = 3$ results in a smaller overall runtime. Similarly, when the domain bit-length reaches 41 bits, a switch to $k = 4$ produces the best results.

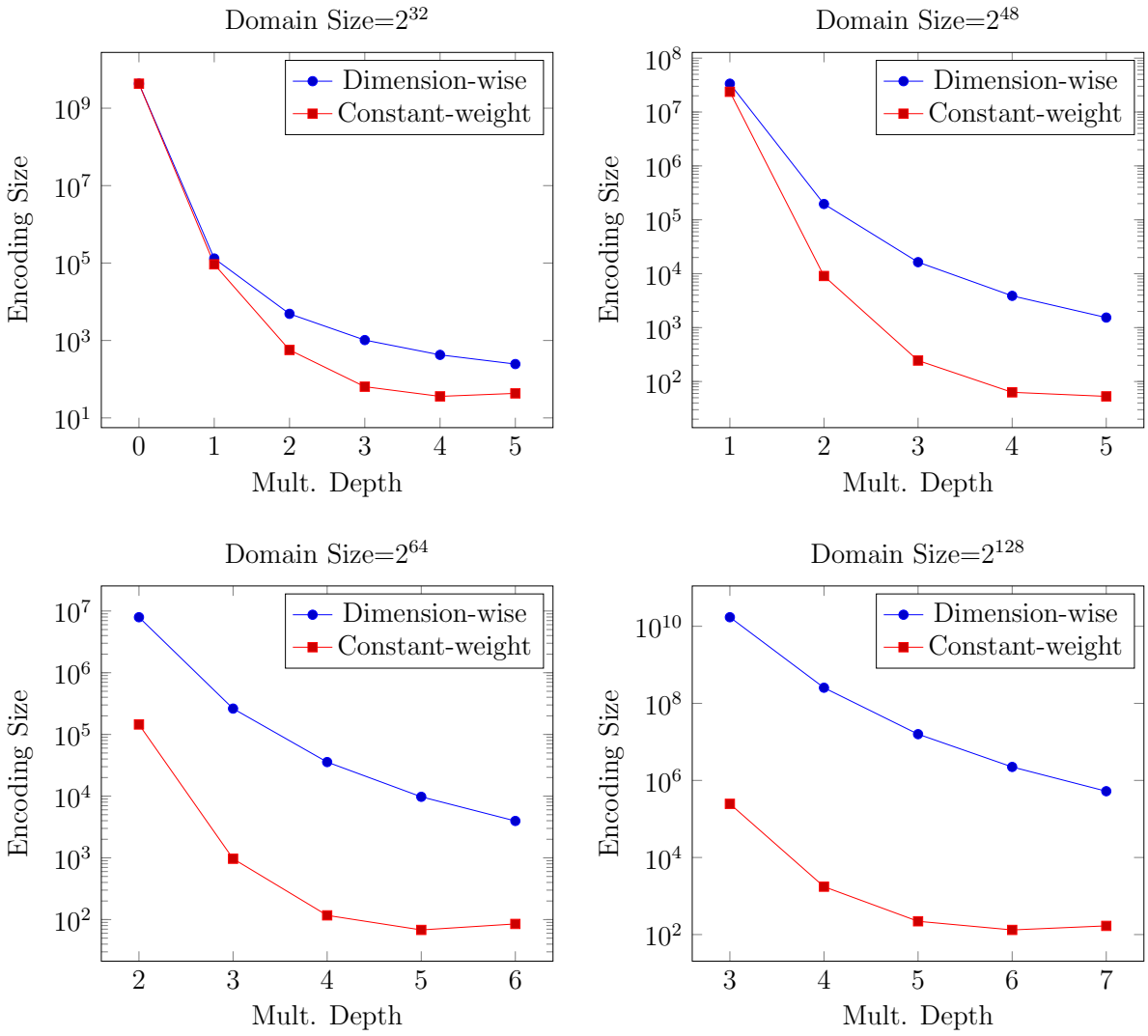

Figure 4.1: Encoding size as a function of multiplicative depth

4.2.4 Varying Response size

In the previous scenarios, we performed experiments under the assumption that the response is at most one ciphertext. In some applications, the response size might exceed the size of one ciphertext [28]. In this section, assume the payload data corresponding to each row can be fit into at most s plaintexts, with one row requiring exactly s plaintexts.

Table 4.10: Breakdown of runtime for constant-weight PIR over sparse databases.

	Hamming Weight = 2					Hamming Weight = 3					Hamming Weight = 4				
$\log_2 \lvert S \rvert$	Code Size	Query Size (in cts)	Expansion Time (ms)	Iteration Time (ms)	Server Total Time (ms)	Code Size	Query Size (in cts)	Expansion Time (ms)	Iteration Time (ms)	Server Total Time (ms)	Code Size	Query Size (in cts)	Expansion Time (ms)	Iteration Time (ms)	Server Total Time (ms)
14	182	1	446	33422	34149	48	1	146	54365	54788	27	1	104	75401	75762
15	257	1	447	33810	34510	60	1	160	54148	54566	32	1	116	75858	76220
16	363	1	508	33623	34384	75	1	266	54542	55066	37	1	158	75143	75551
17	513	1	856	33975	35082	94	1	181	54835	55273	44	1	146	75492	75885
18	725	1	810	33038	34103	118	1	233	54825	55306	52	1	142	75400	75792
19	1025	1	1819	33910	35985	148	1	258	55308	55836	62	1	142	75834	76231
20	1449	1	2049	33718	36022	186	1	358	54647	55254	73	1	173	75255	75674
21	2049	1	3578	33860	37699	234	1	310	55656	56226	86	1	205	75824	76284
22	2897	1	3882	33018	37152	295	1	505	55538	56299	102	1	264	74264	74810
23	4097	1	5859	34369	40485	371	1	546	55531	56328	121	1	252	75294	75801
24	5794	1	7560	33528	41340	467	1	501	55507	56257	144	1	316	75081	75645
25	8193	2	7959	34288	42503	588	1	843	55273	56372	170	1	314	75505	76075
26	11586	2	12649	33275	46191	740	1	1042	55081	56366	202	1	359	75510	76122
27	16385	3	16117	33776	50155	932	1	1033	55731	57020	240	1	299	75355	75900
28	23171	3	23921	34585	58762	1174	1	1907	55070	57233	285	1	523	76564	77339
29	32769	5	33171	34088	67529	1478	1	1851	56117	58213	339	1	439	76776	77464
30	46342	6	45910	35789	81971	1862	1	2374	54611	57237	403	1	534	75708	76492
31	65537	9	62272	34539	97090	2346	1	3490	54301	58046	478	1	540	75181	75980
32	92683	12	87382	34898	122573	2955	1	3410	54470	58132	569	1	820	75069	76140
33	131073	17	114097	36645	151056	3723	1	4815	55327	60395	676	1	1085	75140	76470
34						4690	1	6174	56817	63250	803	1	899	76363	77508
35						5909	1	7401	56036	63694	955	1	966	75651	76874
36						7444	1	7897	55919	64080	1135	1	1759	75115	77131
37						9379	2	10107	56242	66604	1350	1	1979	75011	77247
38						11816	2	12381	55660	68294	1605	1	2059	76309	78622
39						14887	2	15333	56644	72240	1908	1	1891	75342	77482
40						18756	3	19917	57105	77282	2268	1	3406	77031	80765
41						23631	3	21794	54999	77053	2697	1	3992	76771	81012
42						29773	4	32581	57650	90501	3207	1	4064	75299	79613
43						37511	5	37536	55374	93182	3814	1	4719	75822	80786
44						47261	6	47000	56874	104155	4535	1	6608	75685	82538
45						59545	8	59277	56286	115844	5393	1	6724	76674	83650
46						75021	10	70884	58772	129944	6413	1	7519	76362	84140
47						94521	12	88535	55770	144599	7626	1	8747	77852	86850
48						119088	15	114411	57590	172302	9068	2	9877	78812	88938

In SEALPIR and MulPIR, the server's computation (excluding the expansion step) must be performed s times. Constant-weight PIR is different in that regard and only one step of the protocol, the *inner product* step, is repeated s times. Table 4.11 shows how the response size affects the number of operations. While all the operations are multiplied by s for SEALPIR and MulPIR, only those corresponding to the inner product step are multiplied by s in constant-weight PIR.

We perform experiments for PIR over a database with a fixed number of rows and varying response sizes. We fix the number of rows to $2^{14} = 16384$ and measure the response size in terms of the number of ciphertexts returned by the server. To compare, we use MulPIR with $d = 2$ and constant-weight PIR with $k = 2$, which have a similar communication complexity as shown in the previous section. Also, comparison of the two protocols when $d = k = 1$ is irrelevant since the two protocols reduce to unary approach.

Figure 4.2 visualizes the runtimes for MulPIR and constant-weight PIR. The implementation of MulPIR used in the experiments does not support response sizes larger than

Table 4.11: Parameters for PIR using SEALPIR, MulPIR, and constant-weight PIR when $|S(\mathbb{ID})| = |\mathbb{ID}| = n$ and response is s plaintexts large. The ciphertext expansion factor is denoted by F. `PM`, `M` indicate plain multiplication and homomorphic multiplication, respectively.

Method	**Mult Depth**	**Query Bit-length**	**# of Operations**	**Download Cost (in ciphertexts)**
SEALPIR	$d-1$	$d\lceil \sqrt[d]{n} \rceil$	$(\sum_{i=0}^{d-1} n^{\frac{d-i}{d}} F^i \cdot \texttt{PM}) \cdot s$	$F^{d-1}s$
MulPIR	$d-1$	$d\lceil \sqrt[d]{n} \rceil$	$(n \cdot \texttt{PM} + \sum_{i=1}^{d-1} n^{\frac{d-i}{d}} \cdot \texttt{M}) \cdot s$	s
Constant-weight PIR	$\lceil \log k \rceil$	$O\left(\sqrt[k]{k!n}\right)$	$nk \cdot \texttt{M} + ns \cdot \texttt{PM}$	s

one ciphertext, so we resort to an approximation of the runtime of MulPIR (without parallelization). The approximation is indicated by the blue dotted line. The yellow line shows the runtime of constant-weight PIR (without parallelization). We include constant-weight PIR with parallelization for reference. As seen in the figure, the runtime of constant-weight PIR is initially worse than MulPIR but overtakes it as the response size increases. Constant-weight PIR outperforms MulPIR when the response size is at least 12 ciphertexts which is 245.76 KB given the encryption parameters that are used. This corresponds to a database size of about 4.02 GB in this experiment.

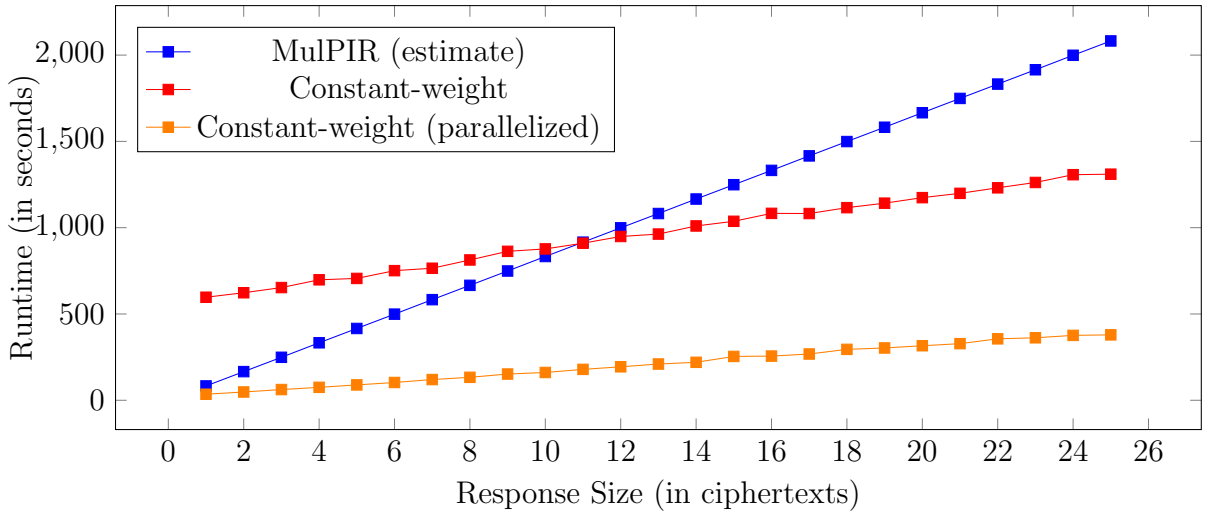

Figure 4.2: Runtime of constant-weight PIR and an estimation of the runtime of MulPIR for large response sizes.

Chapter 5

Applications, Limitations, and Future Work

5.1 Applications

In this section, we elaborate on properties of constant-weight codes and constant-weight PIR and explain how these properties distinguish them from related work. We also describe scenarios where the constructions proposed in this work are useful and beneficial.

5.1.1 First Practical Solution

PIR using equality circuits has generally been deemed an impractical approach due to the high cost of equality circuits using homomorphic encryption [4, 6]. Constant-weight equality operators and constant-weight PIR are the first practical realization of this approach. Concretely, plain constant-weight operators are up to 10 times faster than equivalent

The computational cost of this approach is inevitably higher than state of the art approaches for PIR over a packed database. However, it proves the practicality of this approach. Moreover, constant-weight PIR is the first practical protocol which supports keyword PIR without the need to reduce the problem to index PIR.

5.1.2 Setup-Free (Update Friendly)

Solutions for keyword PIR discussed in Section 2.5 all reduce it to index PIR [4, 15]. The most common solutions map elements of a large, sparse domain to a small array using a probabilistic hashing technique. Index PIR is then used on the small array. This poses multiple problems. First, a setup phase is needed in which the server selects parameters for the mapping such that there are no collisions, or the number of collisions is minimized. Second, when elements are added to the database, the mapping must be recalculated to account for new collisions. If the range of the mapping is too small, parameters must be increased accordingly. Change in parameters must be coordinated amongst the servers which hold the database and communicated to all users which interact with the server.

In keyword PIR using equality circuits, a setup is not required. Any update to the database (insertion, delete, etc) is done without any coordination between the user and the server. When using a mapping such as the order preserving mapping to construct a query, an update in the database size is backwards compatible and previous queries are valid.

5.1.3 Less Overhead for Decentralized Database

In the previous chapters, we only examined PIR in the setting where the database is held by a single server. Our protocol can also be extended to a setup with multiple data owners, each holding a portion of the database. This differs from the well known multi-sever setup described in information theoretic PIR where multiple non-colluding servers hold the same database. In this setting, which we name *multi-DB PIR*, there are h servers, namely $s_1, ..., s_h$ where s_i holds a database $\mathbb{DB}_i$ of size n_i. Define the union of $\mathbb{DB}_i$ as $\mathbb{DB}$. In contrast to the multi-server setup, there is no non-collusion assumption in this setting.

Assume a user needs to query the union of the databases, $\mathbb{DB}$. One solution is to aggregate the databases and perform PIR over $\mathbb{DB}$. However, aggregating the data might not be feasible. While the data might be public, the aggregated database might be considered intellectual property or dangerous to release. For example, in the case of Password Checkup [35], companies such as Google may not want to share the list of compromised credentials they have obtained, albeit through public sources. The company may also have a list of passwords from its own database that might have been compromised, but is not willing to give this list to other companies. Conversely, the data might be aggregated initially but to reduce latency and increase response time, multiple servers are used in parallel where each server computes on part of the database.

Table 5.1: Total number of homomorphic operations in single-DB and multi-DB PIR for MulPIR (with recursion level of d) and constant-weight PIR over a database with n rows. In the multi-DB case, the database is spread over h servers.

Method	Single-DB	Multi-DB	
	# Mults	# Mults Per DB	# Total Mults
MulPIR	$O\left(n^{\frac{d-1}{d}}\right)$	$O\left((\frac{n}{h})^{\frac{d-1}{d}}\right)$	$O\left(h^{\frac{1}{d}}n^{\frac{d-1}{d}}\right)$
Constant-weight PIR	$O(n)$	$O(n/h)$	$O(n)$

We argue that in multi-DB PIR solutions based on equality circuits incur no additional overhead, except aggregating the response from the databases. In contrast, solutions based on selection vectors such as MulPIR incur excessive overhead when the database is spread our across multiple servers.

We justify this argument by counting the number homomorphic multiplications needed in MulPIR and constant-weight PIR. Based on Table 4.8, the number of plaintext multiplications in both MulPIR and constant-weight PIR is equal to n in both approaches, hence we omit that from our comparison.

In multi-DB PIR, the client transmits its encrypted query $Enc(q)$ to each of databases (or transmits it to one and its forwarded to the rest). Then each server i derives $PIR(DB_i, q)$ and the response is equal to the sum of the responses, i.e., $PIR(DB, q) = \sum_{i=1}^{h} PIR(DB_i, q)$. The aggregation can be done by one of the servers. The overall number of homomorphic multiplications done by all the servers is shown in Table 5.1. This shows that multi-DB PIR using MulPIR induces additional overall overhead compared to the single-DB setting with a database of the same size. Constant-weight PIR has no additional overhead .

5.1.4 Anti-Fishing

PIR protocols grant access to users to query a database privately. In some applications, allowing users to query the database is beneficial compared to releasing the entire database. One example is password checkup, mentioned in the previous section. While compromised passwords may be sourced from publicly available data, releasing the aggregate data would empower attackers for credential stuffing attacks [35]. In another hypothetical use case, assume a private file storage system [28] that stores private, encrypted records of individuals and grants access to the records through a private retrieval system. This database could be a hospital storing patient records. While users only store encrypted records on the server,

and records can only be decrypted using their own secret key (or passphrase), publicly releasing the encrypted documents is not a smart solution. This can allow attackers to perform brute-force attacks offline or, through social engineering attacks, obtain the keys (pass phrases) for the encrypted documents they have acquired.

In both of these cases, a sparse database can prevent *fishing* by adversaries. Loosely speaking, fishing means attempting to access the contents of the database in bulk by repeatedly querying it. With a sparse database, data is not stored in a contiguous block of memory. Instead it is stored at addresses known only to the owner of the data, making it difficult for an adversary to locate and retrieve data is does not own. In other words, this removes the requirement to explicitly grant access to each user for accessing their own document, whilst also allowing the user to privately and anonymously access the database.

5.2 Limitations

We know that computation in single-server solutions must be at least on the order of size of the database. In CPIR solutions using homomorphic encryption, another inherent lower bound is the number of plaintext multiplications. One plaintext multiplication must be performed for each row of the database. The *inner product* step in the constant-weight PIR protocol is an example of this. Line 2 of Algorithm 2 performs the same operation in MulPIR. This number of plaintexts multiplications is necessary to account for each row in the database. The number of homomorphic multiplications is sublinear in the number of database rows for solutions such as MulPIR. However, the computation time required to perform the required plaintext multiplications compares to the the runtime for the homomorphic multiplications. This is despite the fact that plaintext multiplications are much faster than homomorphic multiplications (Table 2.1).

One possible remedy is to perform all or some plaintext multiplications in an offline phase to reduce the latency in an online phase. We will investigate this further in future work.

5.3 Future Work

To examine the effect of parallelization on the PIR protocols, we require a multi-thread implementation of SEALPIR and MulPIR to compare with that of constant-weight PIR. We hypothesize that constant-weight PIR performs better when run in parallel compared

to related work due to the parallel nature of the protocol, but we leave a more detailed analysis for future work.

The binary equality operators mentioned in this work, while not applicable for PIR, are useful in practice. A detailed comparison of those operators is also required. For this, they must be implemented using a homomorphic encryption system that natively supports binary operations such as TFHE [14].

Chapter 6

Conclusion

In this thesis, we introduced *operators over constant-weight codewords* and their application in tasks such as PIR. We proposed the plain, arithmetic, and binary constant-weight equality operators for comparing constant-weight codewords. We also described efficient mappings from other domains to constant-weight codewords. We reiterated existing solutions for the task of private information retrieval and showed how equality operators are used for some solutions in the single-server setting. With our new equality operator, we designed a PIR protocol based on an approach previously deemed impractical with constant-weight equality operators at its core.

For our evaluation, we compared constant-weight equality operators with existing folklore operators. Our experiments showed that plain constant-weight operators are up to 10 times faster than the equivalent folklore operator. The arithmetic constant-weight equality operator is also up to 10 times faster than the arithmetic folklore operator when both operators are parallelized. We also compared constant-weight PIR, the PIR protocol using constant-weight equality operators with existing work such as SEALPIR and MulPIR. Constant-weight PIR was not advantageous in the case of index PIR, i.e. when the database is fully packed. However, it can outperform previous work in scenarios where the parameters of the database such as domain size and response size scale.

Finally, we discussed how the cases where constant-weight PIR is advantageous translate to real-world properties and applications. Constant-weight PIR is the first practical solution to PIR using an approach that was previously dismissed due to the high cost of equality operators. This work shows how equality operators can be used in a practical setting.

References

[1] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private Information Retrieval for Everyone. *Proceedings on Privacy Enhancing Technologies*, 2016(2):155–174, 2016.

[2] Adi Akavia, Dan Feldman, and Hayim Shaul. Secure Data Retrieval on the Cloud: Homomorphic Encryption meets Coresets. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):80–106, 2019.

[3] Adi Akavia, Craig Gentry, Shai Halevi, and Max Leibovich. Setup-Free Secure Search on Encrypted Data: Faster and Post-Processing Free. *Proceedings on Privacy Enhancing Technologies*, 2019(3):87–107, 2019.

[4] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–Computation Trade-offs in PIR. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1811–1828. USENIX Association, August 2021.

[5] Andris Ambainis. Upper Bound on the Communication Complexity of Private Information Retrieval. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming*, pages 401–407, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[6] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979, May 2018.

[7] Sebastian Angel and Srinath Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, Savannah, GA, November 2016. USENIX Association.

[8] A Beimel, Y Ishai, E Kushilevitz, and Jean-François Raymond. Breaking the $O(n^{1/(2k-1)})$ Barrier for Information-theoretic Private Information Retrieval. *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 261–270, 2002.

[9] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF Formulas on Ciphertexts. In Joe Kilian, editor, *Theory of Cryptography*, pages 325–341, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] Charlotte Bonte and Ilia Iliashenko. Homomorphic String Search with Constant Multiplicative Depth. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'20, pages 105–117, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme. In Martijn Stam, editor, *Cryptography and Coding*, pages 45–64, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. Association for Computing Machinery.

[13] Hao Chen, Kim Laine, and Peter Rindal. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1243–1255, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[15] B. Chor, N. Gilboa, and M. Naor. Private Information Retrieval by Keywords. *IACR Cryptol. ePrint Arch.*, 1998:3, 1998.

[16] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private Information Retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[17] Daniel Demmler, Amir Herzberg, and Thomas Schneider. RAID-PIR: Practical Multi-server PIR. In *Proceedings of the ACM Conference on Computer and Communications Security*, volume 2014, 2014.

[18] Changyu Dong and Liqun Chen. A Fast Single Server Private Information Retrieval Protocol with Low Communication Cost. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, pages 380–399, Cham, 2014. Springer International Publishing.

[19] Yarkın Doröz, Berk Sunar, and Ghaith Hammouri. Bandwidth Efficient PIR from NTRU. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *Financial Cryptography and Data Security*, pages 195–207, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[20] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. *Proceedings of the 15th international conference on Practice and Theory in Public Key Cryptography*, 2012:1–16, 2012.

[21] Craig Gentry. Fully Homomorphic Encryption using Ideal Lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[22] Craig Gentry and Shai Halevi. Compressible FHE with Applications to PIR. 11892:438–464, 2019.

[23] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully Homomorphic Encryption with Polylog Overhead". In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 465–482, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[24] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In Ran Canetti and Juan A Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[25] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[26] Bailey Kacsmar, Basit Khurram, Nils Lukas, Alexander Norton, Masoumeh Shafieinejad, Zhiwei Shang, Yaser Baseri, Maryam Sepehri, Simon Oya, and Florian Kerschbaum. Differentially Private Two-Party Set Operations. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 390–404. IEEE, 2020.

[27] 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*, pages 364–373, 1997.

[28] Travis Mayberry, Erik-Oliver Blass, and A. Chan. Efficient Private File Retrieval by Combining ORAM and PIR. In *NDSS*, 2014.

[29] A. A. Razborov. Lower bounds on the Size of Bounded Depth Circuits over a Complete Basis with Logical Addition. *Mathematical notes of the Academy of Sciences of the USSR*, 41(4):333–338, 1987.

[30] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On Data Banks and Privacy Homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[31] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

[32] Radu Sion and Bogdan Carbunar. On the Computational Practicality of Private Information Retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 2006–06. Internet Society, 2007.

[33] R Smolensky. Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 77–82, New York, NY, USA, 1987. Association for Computing Machinery.

[34] Julien P Stern. A New and Efficient All-Or-Nothing Disclosure of Secrets Protocol. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology — ASIACRYPT'98*, pages 357–371, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[35] Kurt Thomas, Jennifer Pullman, Kevin Yeo, A Raghunathan, Patrick Gage Kelley, L Invernizzi, B Benko, Tadek Pietraszek, S Patel, D Boneh, and Elie Bursztein. Protecting Accounts from Credential Stuffing with Password Breach Alerting. In *USENIX Security Symposium*, 2019.

[36] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully Homomorphic Encryption over the Integers. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 24–43, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[37] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Secure Pattern Matching using Somewhat Homomorphic Encryption. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, CCSW '13, page 65–76, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Xun Yi, Mohammed Golam Kaosar, Russell Paulet, and Elisa Bertino. Single-Database Private Information Retrieval from Fully Homomorphic Encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1125–1134, 2013.