U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

MSc

2.º CICLO

FCUP
ANO

U.PORTO

Fully Homomorphic Encryption and its application to Private Search

Ivan de Vasconcelos Costa Pêgo da Silva

FC

Fully Homomorphic Encryption and its application to Private Search

Ivan de Vasconcelos Costa Pêgo da Silva

Dissertação de Mestrado apresentada à
Faculdade de Ciências da Universidade do Porto em
Matemática

2022

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

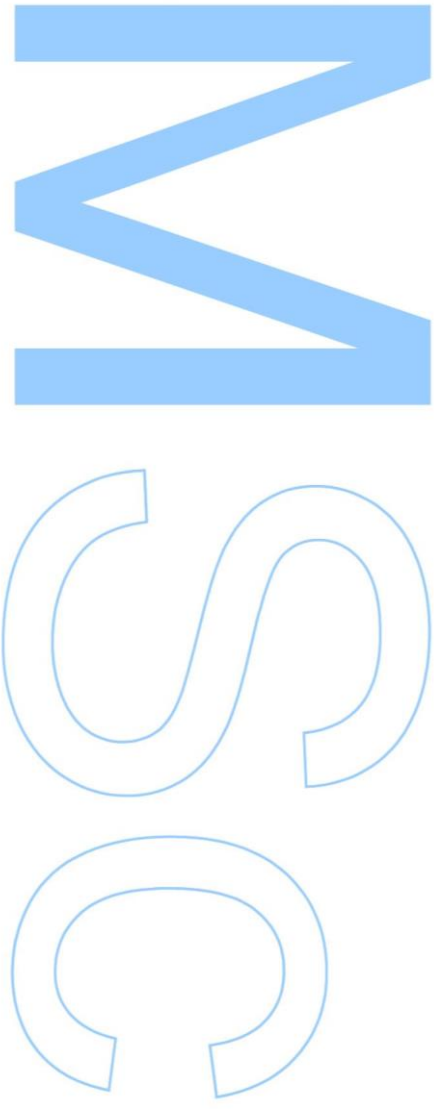
MSc

Fully Homomorphic Encryption and its application to Private Search

Ivan de Vasconcelos Costa Pêgo da Silva
Master's in Mathematical Engineering
Departamento de Matemática
2022

Orientador
António Machiavelo, Faculdade de Ciências da Universidade do Porto

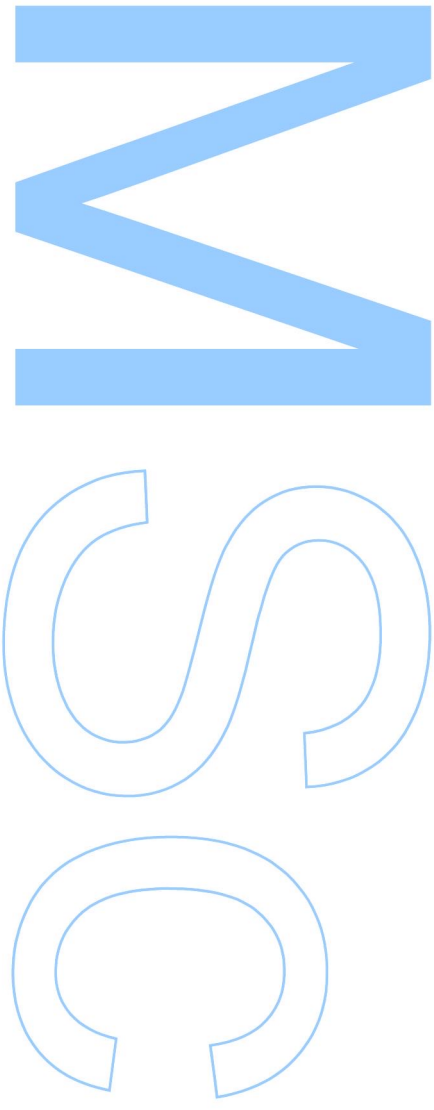
Coorientador
Rogério Reis, Faculdade de Ciências da Universidade do Porto



Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Acknowledgements

First of all, I would like to thank António Machiavelo and Rogério Reis for all the support, availability and knowledge shared during this last step of my academic journey. I also want to thank Ivone, although she is not mentioned as a supervisor, she was present throughout the whole process and was always willing to help me. It was a pleasure to have the opportunity to work with people that value knowledge and know how to share it in the most appealing way. All that, and more, made this an experience I will never forget.

I would also like to thank my grandparents, João and Natalia, for being there for me since day one, you are the pillars of who I am today. To my parents, Sandra and Duarte, thank you not only for the patience throughout the years but also for all the wisdom and encouragement that you gave me. You are the ones I am most thankful to, and I hope to have made you proud.

To Pakita and Gika, and their respective families, thank you for being such a crucial part in raising me as well as in my everyday life, both of you mean a lot to me.

To my partner, Bruna, I would like to thank for your support, companionship and love. I am glad to share this and much more milestones with you. I would also like to mention Bruna's parents and brother for being incredible kind people and even without realising it, being an important factor in my journey.

A thank you to Gringo, for all the moments that made this journey in FCUP memorable. To Defcon a thank you for everything, you are the definition of brothers from other mothers.

Last but not least, a thank you to all my friends and family that I was not able to mention. For the ones that know the role they played, my biggest and most sincere thank you!

UNIVERSIDADE DO PORTO

Abstract

Due to the technological advancements over the years, security of data is, nowadays, more important than ever. Therefore, better encryption schemes are necessary to maintain data confidential and at the same time take full advantage of the several new features that this new digital world brings, such as Artificial Intelligence. Homomorphic encryption schemes come as a solution to this problem, since these type of schemes allow us to perform computations on encrypted data without ever having the necessity of decrypting it, ensuring confidentiality at all times.

The idea of computing over encrypted data was first introduced in 1978 as a "privacy homomorphism " by Rivest et al. [1], evolving then to what we now know as homomorphic encryption. This schemes can be split into three different categories, with the Fully Homomorphic Encryption (FHE) schemes being the ones with more potential, since they allow to perform computations in encrypted data with more than one operation and an arbitrary number of times. Gentry's breakthrough [2], presented in 2009, was the first feasible FHE scheme to be published, which was also a huge leap in the quest for a "complete" FHE scheme, since some of the foundations he presented in his work can be found throughout several FHE schemes that were then published.

In this dissertation, it is done a in-depth analysis of a FHE scheme in rings, which was proposed by Gribov et al. [3], in 2018. Firstly, the encryption process as a whole is studied and described in detail. An analysis of its security liabilities is also presented, where some interesting conclusions are raised which may help a user to prevent possible attacks. Finally, it is done an implementation of this scheme, in Python, applied to third-party private search, as well as a performance analysis.

Keywords: Fully Homomorphic Encryption, Third-Party Private Search, Public-Key Encryption, Rings, Idempotents

UNIVERSIDADE DO PORTO

Resumo

Devido aos grandes avanços tecnológicos dos últimos anos, a segurança de dados é, hoje, mais importante do que nunca. Como tal, é necessário desenvolver novas e melhores cifras de modo a que a confidencialidade dos dados possa ser mantida enquanto se aproveita ao máximo todas as vantagens que este novo mundo digital nos traz, como por exemplo, o uso de Inteligência Artificial. As Cifras Homomórficas surgem como uma solução para este problema, visto que este tipo de cifras permite efetuar cálculos com dados cifrados sem haver nunca a necessidade de os decifrar, mantendo assim a confidencialidade dos mesmos durante todo o processo.

A ideia de efetuar cálculos sobre dados cifrados foi introduzida pela primeira vez em 1978 como um "privacy homomorphism " por Rivest et al. [1], evoluindo até ao que hoje se conhece como Cifras Homomórficas. Este tipo de cifras pode ser dividido em três categorias diferentes, sendo as Cifras Totalmente Homomórficas (FHE) aquelas que têm mais pontencial, uma vez que permitem efetuar cálculos sobre dados cifrados com mais de uma operação e um número arbitrário de vezes. A descoberta de Gentry [2], em 2009, foi a primeira Cifra Totalmente Homomórfica viável, o que foi também um avanço muito significativo na procura de uma Cifra Totalmente Homomórfica "completa", visto que muitos dos fundamentos apresentados no trabalho dele podem ser encontrados em várias cifras publicadas posteriormente.

Nesta dissertação, faz-se uma análise detalhada de uma Cifra Totalmente Homomórfica em anéis, publicada em 2018 por Gribov et al. [3]. Primeiramente, é feito um estudo detalhado de todo o processo de cifragem. Exploram-se também alguns problemas de segurança, onde alguns resultados interessantes são apresentados e que poderão ajudar um utilizador a evitar possíveis ataques. Por fim, apresenta-se uma implementação desta cifra, em Python, com o objetivo de que a mesma seja usada para pesquisa privada de terceiros, assim como uma análise do seu desempenho.

Palavras-chave: Cifras Totalmente Homomórficas, Pesquisa privada de terceiros, Cifras de Chave Pública, Anéis, Idempotentes

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The purpose of encryption is to ensure confidentiality of data and, nowadays, this is more important than ever. The technological evolution and the increasing digitisation of services make inevitable the sharing of sensitive data through public networks. In this context, better encryption tools are needed to ensure the confidentiality of data while also allowing to take advantage of this new digital world. For example, the use of sensitive data to train machine learning models, by Third-Parties, for diseases diagnosis is constrained by the General Data Protection Regulation and, as a consequence, the true potential of Artificial Intelligence (AI) cannot be completely explored.

In order to freely use data and, at the same time, guarantee privacy, encryption schemes should be used to encrypt the data before sharing it and, ideally, those schemes should allow computations on encrypted data, i.e., if the result of a computation between encrypted elements is decrypted, then the result obtained must be the same as of computing such elements unencrypted.

The idea of computing over encrypted data was first introduced in 1978 as a "privacy homomorphism" by Rivest et al. [1]. This catered the attention of researchers in the field of cryptography due to its potential for being a possible solution to the computing without decryption problem, and it then developed into the study of homomorphic encryption today.

After the work from Rivest et al., researchers in the field of cryptography began to seek for a Homomorphic Encryption (HE) scheme that allowed for computations on encrypted data using more than one operation. However, in the following twenty years, the many attempts of an homomorphic encryption scheme resulted in schemes that allowed for

computations on encrypted data with only one operation. This kind of schemes are known as Partially Homomorphic Encryption (PHE) schemes.

RSA, a well known public key cryptosystem also introduced in 1978 by Rivest et al. [4], is an example of a PHE scheme with the usual product. However, this scheme is deterministic, meaning that, with a given key, the encryption of the same plaintext will always result in the same ciphertext, and, consequently, it does not achieve the highest level of security possible for an HE scheme.

In 1985, El Gamal introduced a PHE [5] which allows only one operation (the usual product) but it has the maximum level of security for an HE scheme. A few years before, in 1982, Goldwasser-Micali published the first probabilistic (or non determininstic) PHE scheme [6], and most schemes published in the following couple of decades were strongly inspired by this one, such as Benaloh's [7], in 1994, which was a generalization of Goldwasser-Micali, and Naccache-Stern's [8], which was an improvement on Benaloh's scheme. More schemes followed this pattern, however, this all led to Paillier's encryption scheme [9], which is a PHE for the usual sum. What makes this scheme, and its variants, so special is their efficiency but also, as El Gamal, they achieve the highest level of security for an homomorphic encryption scheme.

Throughout the years, there was also another category of homomorphic encryption schemes being published, which allow for more than one operation to be computed on encrypted data. However, in these schemes, only a limited amount of such computations are allowed. This type of schemes are known as Somewhat Homomorphic Encryption (SWHE) schemes. Nonetheless, towards the end of the century, they became much more complete, serving as a stepping stone to achieve a Fully Homomorphic Encryption (FHE) scheme, i.e., a scheme where more than one operation is allowed, and computations can be performed an unlimited number of times.

An example of such a more complete scheme is BGN [10], published in 2005, which allows for arbitrary additions and only one product, and is considered by many to be an important stepping stone towards a FHE scheme. In fact, not many years later, in 2009, Gentry published the first feasible FHE scheme [2]. Until Gentry's publishing, most attempts of a FHE scheme were proved to be unsecure or there was no computational capacity, at the time, to implement such schemes.

A security liability that is common among homomorphic encryption schemes is accumulating encryptions of 0, i.e., if an attacker is able to accumulate enough encryptions of 0

then the security of the scheme might be at risk. To overcome this problem many schemes disguise encryptions of 0 with what is designated by *noise*. However, this noise tends to increase with the number of computations performed, which may lead to an incorrect decryption in the end. The main idea introduced by Gentry was bootstrapping, which helps to solve this problem. On the other hand, bootstrapping is computationally expensive, and this along with its complex mathematical foundations made Gentry's scheme not exactly applicable in real-life scenarios.

Nonetheless, its foundations were very solid, which led to a huge leap in the quest of a "complete" FHE scheme and most schemes published since then are either optimizations of this scheme or its foundations are similar. In fact, after Gentry's work, the research of FHE schemes was mainly split into the four following categories:

- Ideal Lattices - Schemes in this branch are, in a sense, optimizations of Gentry's, some examples are the work of Gentry and Haveli [11], in 2011, and also in the same year, the work of Scholl and Smart [12];
- Integers - First appearance in 2010, by Van Djik et al. [13]. The main motivation behind these schemes its the simplicity of the concept, however they are not practical, which makes this the least favorite category for researchers;
- (Rings) Learning With Error, (R)LWE - Originated in 2011, by Brakerski and Vaikuntanathan [14], and these schemes are based on the LWE problem, which is considered one of the hardest problems to solve in practical time, even for post-quantum algorithms. RLWE is an algebraic variant of LWE, which is more efficient for applications;
- N^{th} degree-truncated polynomial ring unit (NTRU)-Like FHE schemes - Mainly know for their efficiency and for their use of Multi-Key FHE, which as the name may suggest, allows for computations between data encrypted with distinct keys. First proposed by López-Alt et al. [15], in 2012.

Another note to make, regarding SWHE schemes, is that after Gentry's breakthrough, the perspective on such schemes changed, i.e., some attempts of FHE schemes were simply converted to a more efficient SWHE scheme.

In 2018, Alex Gribov et. al [3] introduced a fully homomorphic encryption scheme in rings, that does not rely on bootstrapping. Therefore, it may be more suited for practical usage than previous schemes, namely its application to private search. In this

thesis, it is done an in-depth study of this scheme, which includes the description of the core encryption process, an analysis of its security, its application to private search, and an implementation of this application, in Python and using some Sage modules [16], together with a performance analysis. Moreover, some new results are also presented which act as recommendations for a more secure usage of this scheme.

The remaining of this document is organized as follows.

In Chapter 2, some preliminary concepts and results, from different areas of Mathematics, are introduced, including some basic notions from Cryptography. We also introduce some convenient notation.

The analysis of the whole scheme is discussed in Chapter 3, which consists of first explaining the core encryption process, followed by a brief explanation of how to perform an embedding of our data, which is a concern one must have when applying such scheme in a real life scenario.

This analysis is followed by a discussion on its security, in Chapter 4, which revolves mainly around the embeddings mentioned above, since a poor choice of these might lead to plaintexts being exposed. While discussing this topic we also present some results proved by us while working on this project, and that are of great importance for a better usage of the scheme presented.

In Chapter 5, we introduce private search, which consists in privately checking if the encryption of a certain element belongs to an encrypted database. We also present an overview of our implementation of this scheme, in Python, as well as a performance analysis. Notice that all the code wrote is shared in the Appendix.

Finally, in Chapter 6 we summarize our main contributions and propose some future research topics.

Chapter 2

Preliminaries

In this chapter, we give some introductory concepts from abstract algebra, as well as from cryptography, that are considered to be relevant going forward.

The first section is dedicated to mathematical preliminaries, and the following to the preliminaries related with Cryptography, where a few definitions are shared, that hopefully, will help the reader to better understand where these encryption schemes fit within the world of Cryptography.

2.1 Mathematical Preliminaries

We begin this section with a revision of some group theory concepts that will aid the reader in the remaining of these mathematical preliminaries. We then proceed to present results that are more directly related to our work. Note that we do not include proofs in these preliminaries unless they are somewhat relevant to this project, nonetheless, for the most curious readers we recommend the work of T.-Y. Lam [17] and T.W. Judson [18] for a more complete introduction on such topics.

Brief group theory revision

Let A and B be two sets. A relation $\sim$ from A to B is a subset of the cartesian product $A \times B$. For any $a \in A$ and $b \in B$, we write $a \sim b$ to denote that (a, b) is in the relation $\sim$, otherwise we write $a \nsim b$. When $A = B$, $\sim$ is also called a binary relation on A.

A binary relation $\sim$ on a set A is said to be an *equivalence relation* if and only if it is reflexive, symmetric and transitive, i.e., for all a,b and c in A:

1. $a \sim a$ (reflexivity);

2. $a \sim b$ if and only if $b \sim a$ (symmetry);

3. If $a \sim b$ and $b \sim c$, then $a \sim c$ (transitivity).

Let $\sim$ be an *equivalence relation* on A. For any $a \in A$, the set $[a]_\sim = \{b \in A \mid a \sim b\}$ is called the *equivalence class* containing a, while the set of all equivalence classes, $A/\sim = \{[a]_\sim \mid a \in A\}$, is called the *quotient* of A by $\sim$.

One of the most common examples of an equivalence relation is a *congruence modulo* n on the set of integers $\mathbb{Z}$. For a positive integer n, two integers a and b are said to be congruent modulo n if their difference $a - b$ is a multiple of n. The notation for a congruent with b modulo n is the following:

$$a \equiv_n b \text{ or } a \equiv b \ (mod\ n).$$

In this context an equivalence class consists of those integers which have the same remainder on division by n. The set of integers modulo n, which is denoted by $\mathbb{Z}_n$, is the set of all congruence classes of the integers for the modulus n.

A *n-ary operation* in A is a mapping from A^n to A, where $n \in \mathbb{N} = \{1, 2, 3, ...\}$. Then, a binary operation is a mapping $\star : A^2 \to A$, which means that if (a, b) is an ordered pair of elements from A, then $a \star b$ is a unique element of A.

Definition 2.1 (Idempotent)**.** An element x of a set S equipped with a binary operator $\cdot$ is said to be idempotent under $\cdot$ if,

$$x \cdot x = x.$$

The binary operation $\cdot$ is said to be idempotent if every element of S is an idempotent under $\cdot$.

That said, a *group* is an ordered pair $(G, \star)$, where G is a non-empty set and $\star$ is a binary operation on G (called the group operation), satisfying the following properties, for any $x,\ y,\ z\ \in G$:

1. $x \star (y \star z) = (x \star y) \star z$, i.e., $\star$ is associative;

2. There is an element $e \in G$ such that for all $x \in G$, $x \star e = e \star x = x$. This element is unique and is called the *identity element*;

3. For any x in G there is an element x' in G such that $x \star x' = x' \star x = e$. The element x' is called the *inverse* of x in G.

Moreover, if the group operation is commutative we say that G is an *Abelian group* or *commutative group*.

Proposition 2.2. *Let $\mathbb{Z}_n$ be the set of equivalence classes of the integers mod n and $a,\ b,\ c \in \mathbb{Z}_n$. Then,*

1. *Addition and multiplication are commutative:*

$$a + b \equiv b + a \ (mod\ n),$$
$$a \cdot b \equiv b \cdot a \ (mod\ n);$$

2. *Addition and multiplication are associative:*

$$a + (b + c) \equiv (a + b) + c \ (mod\ n),$$
$$a \cdot (b \cdot c) \equiv (a \cdot b) \cdot c \ (mod\ n);$$

3. *There are both an additive and a multiplicative identity:*

$$a + 0 \equiv a \ (mod\ n),$$
$$a \cdot 1 \equiv a \ (mod\ n);$$

4. *Multiplication distributes over addition:*

$$a \cdot (b + c) \equiv a \cdot b + a \cdot c \ (mod\ n);$$

5. *For every integer a there is an additive inverse $-a$:*

$$a + (-a) \equiv 0 \ (mod\ n);$$

6. *Let a be a nonzero integer. Then $gcd(a, n) = 1$ if and only if there exists a multiplicative inverse b for a, modn, i.e., a non-zero integer b such that,*

$$a \cdot b \equiv 1 \ (mod\ n).$$

Note that from this proposition we can conclude that $\mathbb{Z}_n$ form a group with addition modulo n. On the other hand, the same is not true when the group operation is the product modulo n, since there is no element a in $\mathbb{Z}_n$ such that $a \cdot 0 \equiv 1 \ (mod\ n)$. Nonetheless, if n is a prime number, then every element in $\mathbb{Z}_n \setminus \{0\}$ has a multiplicative inverse modulo n.

We define a *subgroup* H of a group G to be a subset H of G such that when the group operation of G is restricted to H, H is a group in its own right. Notice that every group G with at least two elements will always have at least two subgroups: the subgroup consisting of the identity element alone and the entire group itself, where the former is called the *trivial subgroup*. The following proposition states a more direct method of verifying if a subset H of G is a subgroup.

Proposition 2.3. *Let H be a subset of a group G. Then H is a subgroup of G if and only if $H \neq \emptyset$, and whenever $a, b \in H$ then $a \star b^{-1} \in H$, where $\star$ is the group operation for G.*

One of the most important definitions in this thesis is the concept of *homomorphism*, namely *group homomorphism* and *ring homomorphism*.

Definition 2.4 (Group Homomorphism)**.** A homomorphism between groups $(G, \star)$ and $(H, \diamond)$ is a map $\phi : G \to H$ such that $\phi(a \star b) = \phi(a) \diamond \phi(b)$ for $a, b \in G$.

If this map is bijective, then its called an *isomorphism*, and we denote such relation between G and H as $G \cong H$.

Given two groups $(G, \star)$ and $(H, \diamond)$, one can define a new group, $G \times H$, as follows.

Proposition 2.5. *Let $(G, \star)$ and $(H, \diamond)$ be groups. The set $G \times H$ is a group under the operation $(a_1, b_1) * (a_2, b_2) = (a_1 \star a_2, b_1 \diamond b_2)$ where $a_1, a_2 \in G$ and $b_1, b_2 \in H$.*

Following Proposition 2.2, and as referred before, $\mathbb{Z}_n$ is a group for the addition modulo n and, by the above result, $\mathbb{Z}_n \times \mathbb{Z}_n$ is also a group with the following operation

$$(a_1, b_1) + (a_2, b_2) = (a_1 + a_2 \ (mod\ n), b_1 + b_2 \ (mod\ n)),$$

for any $a_1, a_2, b_1, b_2 \in \mathbb{Z}_n$. This is of great use throughout this project, more specifically the fact that this can be generalized to a direct product of k groups. Therefore, if the groups are all the same, in this specific case, $\mathbb{Z}_n$, the resulting group is denoted by $\mathbb{Z}_n^k = \mathbb{Z}_n \times ... \times \mathbb{Z}_n$ (k times).

Rings, Fields and Ideals

Definition 2.6 (Ring)**.** A *ring* is a triple $(R, +, \cdot)$ such that $(R, +)$ is an Abelian Group, and the following conditions are satisfied:

- $(a \cdot b) \cdot c = a \cdot (b \cdot c)$, for $a, b, c \in R$ (multiplication is associative);

- There is an element $1 \in R$ such that $1 \neq 0$ and $1a = a1 = a$ for each element $a \in R$;
- For any $a, b, c \in R$ one has,

$$a \cdot (b + c) = a \cdot b + a \cdot c,$$
$$(a + b) \cdot c = a \cdot c + b \cdot c,$$

  i.e., multiplication distributes over addition.

We can easily observe that $\mathbb{Z}_n$ is a ring, since $(\mathbb{Z}_n, +_n)$ is an Abelian group and by Proposition 2.2 we confirm the remaining conditions.

Note also that just as we have subgroups of groups, we have an analogous class of substructures for rings. A subring S of a ring R is a subset S of R such that S is also a ring under the inherited operations from R.

A very common type of rings are polynomial rings, which also play a huge role in this thesis. A polynomial ring in the variable x with coefficients in a ring R is denoted by $R[x]$ and is formed by the set of polynomials in x and the usual operations of polynomial addition and multiplication. A polynomial in $R[x]$ is therefore an expression of the form:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ + a_{n-1} x^{n+1} + a_n x^n,$$

for some $n \in \mathbb{N}_0$, and where $a_i \in R$, for all $0 \leq i \leq n$. In this work, we use a multivariate polynomial ring, meaning that instead of having only one variable x, it has a set of variables $x_1, ..., x_k$, and is denoted by $R[x_1, ..., x_k]$. Notice that if R is a commutative ring, then $R[x_1, ..., x_k]$ is also commutative. Therefore, for a commutative R, the elements of $R[x_1, ..., x_k]$ are linear combinations of monomials of the form $x_1^{\alpha_1} x_2^{\alpha_2} ... x_k^{\alpha_k}$ with coefficients in R, where $\alpha_i \in \mathbb{N}_0$ for every $i \in \{1, ..., k\}$. Notice also that $1 = x_1^0 x_2^0 ... x_k^0$.

After the previous definition we can now formally define a *field*.

Definition 2.7 (Field)**.** A ring $(R, +, \cdot)$ is called a field if

1. The product is commutative ($(R, +, \cdot)$ is a commutative ring);
2. There are no *zero divisors*, i.e., for any $a \neq 0$ in R there is no nonzero element b in R such that $ab = 0$ (if R verifies this along with the above condition then its called an *integral domain*);

3. For any $a \in R$, $a \neq 0$, exists a unique element a^{-1} such that $aa^{-1} = a^{-1}a = 1$ (R is a *division ring*).

Proposition 2.8. *$\mathbb{Z}_n$ is a field if and only if n is prime.*

In this thesis, we work with a particular type of multivariate polynomial rings, where the coefficients are elements of $\mathbb{Z}_n$, as will be shown in Chapter 3. However, we want these coefficients to be elements of a field, therefore, the above proposition is fundamental.

Recall that a group homomorphism is a map that preserves the operation of the group. Similarly, a homomorphism between rings preserves the operations of addition and multiplication in the ring. More specifically, if $(R, +_R, \cdot_R)$ and $(S, +_S, \cdot_S)$ are rings, then a ring homomorphism is a map $\phi : R \to S$ such that:

$$\phi(a +_R b) = \phi(a) +_S \phi(b)$$
$$\phi(a \cdot_R b) = \phi(a) \cdot_S \phi(b),$$

for any $a, b \in R$. As for groups, ϕ is an isomorphism if it is bijective. The following proposition states some important properties of a ring homomorphism.

Proposition 2.9. *Let $\phi : R \to S$ be a ring homomorphism.*

1. *If $(R, +, \cdot)$ is a commutative ring, then $(\phi(R), +_S, \cdot_S))$ is a commutative ring;*
2. *$\phi(0_R) = 0_S$;*
3. *Let 1_R and 1_S be the identities for R and S, respectively. If ϕ is surjective, then $\phi(1_R) = 1_S$;*
4. *If $(R, +, \cdot)$ is a field and $(\phi(R), +_S, \cdot_S)) \neq \{0_S\}$, then $(\phi(R), +_S, \cdot_S))$ is a field.*

In ring theory, there is a special class of subrings called *ideals*. An *ideal* in a ring $(R, +, \cdot)$ is a subring I of $(R, +, \cdot)$ such that, for any $a \in I$ and $r \in R$,

- $a \cdot r \in I$, i.e., $I \cdot r \subset I$ (I is a left-ideal);
- $r \cdot a \in I$, i.e., $r \cdot I \subset I$ (I is a right-ideal).

Note that every ring $(R, +, \cdot)$ has at least two ideals, $\{0_R\}$ and $(R, +, \cdot)$, which are called the *trivial ideals*.

Recall that the *kernel* of a ring homomorphism $\phi : R \to S$ is given by,

$$ker\ \phi = \{r \in R \mid \phi(r) = 0_S\},$$

and notice that the following proposition holds.

Proposition 2.10. *The kernel of any ring homomorphism $\phi : R \to S$ is an ideal in R.*

Given an ideal I of R, we can define a equivalence relation in R as follows:

$$a \sim b \text{ if and only if } a - b \in I.$$

It is easy to see that this is a congruence relation due to the properties of I. The equivalence class of an element $a \in R$ is given by,

$$[a] = a + I = \{a + r \mid r \in I\}.$$

This equivalence class is sometimes referred to as *a modulo I*. The set of all equivalence classes is denoted by R/I, which together with the following operations,

$$(a + I) + (b + I) = (a + b) + I$$

and,

$$(a + I)(b + I) = (ab) + I$$

forms a new ring, called a *factor or quotient ring*. We can easily see that addition is well defined, however, note the following theorem that guarantees that R/I is a ring with multiplication defined as above.

Theorem 2.11. *Let I be an ideal of $(R, +, \cdot)$. The factor group R/I is a ring with multiplication defined by*

$$(r + I) \cdot (s + I) = r \cdot s + I,$$

for any $r, s \in R$.

Proof. We already know that R/I is an Abelian group under addition. Let $r + I$ and $s + I$ be in R/I. We must show that the product $(r + I) \cdot (s + I) = r \cdot s + I$ is independent of the choice of coset, i.e., if $r' \in r + I$ and $s' \in s + I$ then $r's'$ must be in $rs + I$. Since $r' \in r + I$, there exists an element a in I such that $r' = r + a$. Similarly, there exists $b \in I$ such that $s' = s + b$. Notice that $r's' = (r + a) \cdot (s + b) = r \cdot s + a \cdot s + r \cdot b + a \cdot b$ and $a \cdot s + r \cdot b + a \cdot b \in I$ since I is an ideal, consequently, $r' \cdot s' \in r \cdot s + I$. Finally we can easily verify the associative law for multiplication and the distributive laws. □

This results ends this section of mathematical preliminaries since it is very important within the encryption scheme explored in this thesis, as will be explained in the following chapter.

2.2 Cryptography Preliminaries

Encryption is the method by which information is made private, i.e., information is converted to code, such that, without performing decryption one is unaware of the original piece of information. In computing, unencrypted data is known as *plaintext*, and encrypted data is called *ciphertext*. The methods used for an encryption/decryption process are called *encryption schemes*, or *ciphers*.

Alice and Bob are usually used in the literature to refer to two entities that wish to exchange messages in a secret way.

Symmetric and Public-Key Encryption schemes

Encryption schemes can be either symmetric or public-key. *Symmetric encryption schemes*, also known as *private-key encryption schemes*, rely on the same key to perform encryption and decryption, or one can easily obtain the key to decrypt from the one used to encrypt. That is, when Alice wants to send a message to Bob, she has to use the same key that Bob will use to decrypt the corresponding ciphertext. Consequently, they have to agree on a key beforehand, which is the main downside of this type of encryption scheme. A famous example of this type of schemes is AES [19].

In contrast, on *public-key encryption schemes* a pair of keys are generated, where one is made public, named *public-key* while the other remains private, denoted *private-key*. This type of schemes allows for two entities that never met, Alice and Bob, to exchange messages. When Bob wants to send an encrypted message to Alice, he first encrypts the message using Alice's public key, then, Alice decrypts the message using her private key. The most famous example of this type of scheme is RSA [4].

Deterministic and Probabilistic Encryption schemes

An important property of encryption schemes, that was briefly mentioned in the introduction, is having deterministic or probabilistic encryption. In a deterministic encryption

scheme, for a given encryption key, the same plaintext will always produce the same ciphertext. A famous scheme that has deterministic encryption is RSA, as shown next.

In RSA, the public-key is a pair (e, n) where n is the product of two large primes, p and q, and e is chosen such that $gcd(e, \phi) = 1$, where $\phi = (p-1)(q-1)$. The secret-key is another pair (d, n), where d is chosen such that $ed \equiv 1 \ mod \ \phi$.

To encrypt a message, it has first to be converted into a plaintext $0 \leq m < n$ and, then, its encryption is computed as follows,

$$E(m) = m^e \ \ (mod \ n),$$

where E denotes the encryption function. It is clear that the encryption of m does not change unless the key used for encryption changes.

On the other hand, with a probabilistic encryption, under the same conditions, this does not happen, as a matter of fact, we would often get a different ciphertext. A well-known example of a scheme with probabilistic encryption is the Paillier encryption scheme as we show next.

To perform key-generation we first have to choose a pair of large primes p and q such that $gcd(pq, (p-1)(q-1)) = 1$, then let $n = pq$ and $\lambda = lcm(p-1, q-1)$. We then select a random element $g \in \mathbb{Z}^*_{n^2}$, where $\mathbb{Z}^*_{n^2}$ is the set of invertible elements of $\mathbb{Z}_{n^2}$. We do this by checking whether $gcd(n, L(g^\lambda \ mod \ n^2)) = 1$, where L is defined as $L(u) = \frac{(u-1)}{n}$, for every $u \in \mathbb{Z}^*_{n^2}$. Then we define the public-key as (n, g) and the secret-key as (p, q). For each message $m \in \mathbb{Z}_n$ a number $r \in \mathbb{Z}_n$, is randomly chosen, and encryption is performed as follows,

$$E(m) = g^m r^n \ \ \left(mod \ n^2\right).$$

This means that with the same key, two distinct messages would likely produce two distinct ciphertexts, due to the randomness of r.

Homomorphic Encryption

Let M denote the set of plaintexts and C the set of ciphertexts. Let $\odot_M$ and $\odot_C$ be operations in M and C, respectively. An encryption scheme is said to be Homomorphic if for any encryption key k, the encryption function E satisfies the property below,

$$\forall m_1, m_2 \in M \quad E(m_1 \odot_M m_2) \leftarrow E(m_1) \odot_C E(m_2), \tag{2.1}$$

where $\leftarrow$ means that can be directly obtained from. That is, in (2.1), $E(m_1 \odot_M m_2)$ can be directly obtained from $E(m_1) \odot_C E(m_2)$, i.e., without having to perform any decryption. And, as mentioned in the introduction, this is exactly the purpose of homomorphic encryption schemes. Notice, however, that only fully homomorphic encryption schemes give us total freedom when computing on encrypted data.

To conclude this section, we recall the three different types of homomorphic encryption schemes mentioned in the introduction.

- **Partially Homomorphic Encryption (PHE)** - The Homomorphic property is satisfied by one operation, unlimited times.

  Two very well known examples of PHE schemes are the RSA encryption algorithm (see [4]) and the Paillier encryption scheme (see [9]) based on the composite resiuosity problem, where the first verifies the homomorphic property with the usual product and the latter with the usual sum, as we show next.

  In RSA, after key generation we have a private-key (d, n) and a public-key (e, n) as we saw above. Recall that to perform encryption on a message, it has first to be converted into a plaintext $0 \leq m < n$ and, then, its encryption is computed as follows,

  $$E(m) = m^e \pmod{n}.$$

  That said, given $m_1, m_2 \in M$, where M is the set of plaintexts, we have,

  $$E(m_1) \cdot E(m_2) = \left(m_1^e \pmod{n}\right) \cdot \left(m_2^e \pmod{n}\right) = (m_1 \cdot m_2)^e \pmod{n} = E(m_1 \cdot m_2).$$

  Therefore, the result of encrypting the product of two plaintexts is equal to the product of their respective encryptions.

  As shown before, for the Paillier encryption scheme, after key-generation, we have a public-key (n, g) and a secret-key (p, q), and we perform encryption on a plaintext $m \in \mathbb{Z}_n$ as follows,

  $$E(m) = g^m r^n \left(\mod n^2\right),$$

where $r \in \mathbb{Z}_n$ is randomly chosen for each message. Therefore, given two plaintexts m_1 and m_2 we have,

$$\begin{aligned} E(m_1) \cdot E(m_2) &= \left(g^{m_1} r_1^n \pmod{n^2}\right) \cdot \left(g^{m_2} r_2^n \pmod{n^2}\right) \\ &= g^{m_1+m_2}(r_1 \cdot r_2)^n \pmod{n^2} \\ &= E(m_1 + m_2). \end{aligned}$$

This means that the encryption of a sum of plaintexts is equal to the product of their respective encryptions.

- **Somewhat Homomorphic Encryption (SWHE)** - The Homomorphic property is satisfied by a set of operations, a limited amount of times.
  Although these schemes have been around almost since the beginning of homomorphic encryption, it was near the year 2000 where they evolved into being considered a evolution to PHE. A very well known example of such schemes is BGN [10], that is considered to play a major role in the pursue of a feasible fully homomorphic encryption scheme. In this scheme although we can perform unlimited additions on encrypted data, we are only allowed one multiplication.

- **Fully Homomorphic Encryption (FHE)** - The Homomorphic property is satisfied by a set of operation an unlimited amount of times.

The latter type of schemes are the focus of this project, in particular the work of Gribov et al. [3] as mentioned in the introduction. In the following chapter it will be shown that this scheme has probabilistic encryption, since every time a plaintext is encrypted a random element is added to it. Therefore, the ciphertexts obtained from encrypting the same plaintext tend to be different each time.

Chapter 3

The Encryption Scheme

In this section, we present the encryption scheme in rings suggested by Gribov et. al in 2018 [3]. This scheme is private-key, which means that its range of applications might be more limited compared to a public-key encryption scheme. A difference to have into account, between a private-key encryption scheme and a public-key encryption scheme, its their respective security analysis. In a private-key encryption scheme one does not have to worry with some brute force attacks that are usually a concern regarding the security of a public-key encryption scheme [3]. However, with fully homomorphic encryption schemes (FHE), even private-key encryption schemes have to share some information with the public so that computations on encrypted data are allowed, otherwise it would defy the whole purpose of a FHE scheme. Therefore, even though the scheme is private-key, its security analysis is slightly different from a non-homomorphic private-key encryption scheme, as it will be seen in the next chapter.

In this scheme, plaintexts are elements of a ring that can be either private or public (i.e, it does not really matter if it is public or private, since this does not make any difference either in terms of security or for the scheme to work) and ciphertexts are elements of a public ring, such that the first is a subset of the latter. By public ring we mean that a set of rules for adding and multiplying elements is shared with the public, i.e., it is shared just enough information so that it allows for the public to compute on encrypted data.

In the next section, an overview of the encryption process is provided without giving details on how to define the needed rings and other required mathematical structures. These will then be introduced in the following sections, before presenting the detailed encryption process in Section 3.5.

3.1 Encryption Overview

As said above, in this scheme, plaintexts are elements of a ring R and ciphertexts are elements of a ring S, where $R \subset S$. Let E be the encryption function defined as

$$E(u) = u + i, \forall u \in R$$

where i is a random element of an ideal I of S [1]. That is, the encryption process consists of adding a random element of an ideal to the ciphertext, and it can be represented in a simple way by the following diagram

$$R \xrightarrow{E} S.$$

To perform decryption correctly, the goal is to return to our original plaintext, which is an element of R. Notice that, when encrypting, we add to our plaintext a element from I. Therefore, the first step when decrypting is to somehow reverse that procedure. To do this, we apply a map ρ from S to $R' = S/I$ that annihilates every element of I. However, these are not exactly the elements of R. Nonetheless, there is an isomorphism $\varphi : R' \to R$, which allows to finish the decryption process. This process can be represented by the following diagram

$$S \xrightarrow{\rho} R' \xrightarrow{\varphi} R.$$

Remark 3.1. When studying this encryption scheme, we realized that φ has indeed to be an isomorphism; otherwise, it would result in incorrect decryptions. Therefore, to avoid that, the generators of the ideal I of S have to be chosen carefully, as we explain further in the following subsections.

In most real life applications, plaintexts are usually not elements of a ring R. Therefore, in such cases, before performing encryption, an embedding from our set of original plaintexts to R has to be performed, which means that after decryption, we also need to return to this original set of plaintexts.

For example, suppose the set of data we wish to encrypt is a collection of names. In order to perform encryption, we first need to do an embedding of this set of names into R, i.e., represent each name with a distinct element of R. Similarly, when decrypting a ciphertext, the result is an element of R. Consequently, in order to recover the original name, we have to perform the inverse of our original embedding.

[1] Gribov et al. [3] refer to i as $E(0)$, i.e., encryptions of 0.

Let D denote the set of original data we wish to encrypt, then, the overall process can be represented by the following diagram

$$D \xrightarrow{\alpha} R \xrightarrow{E} S \xrightarrow{\rho} R' \xrightarrow{\varphi} R \xrightarrow{\beta} D,$$

where $\beta(\alpha(x)) = x, \forall x \in D$.

Notice that the encryption function E is an homomorphism modulo I, since for any $u, v \in R$, one has

$$E(u) + E(v) = u + j_1 + v + j_2 = u + v + j_3 = E(u + v) \text{ modulo } I,$$

and,

$$E(u)E(v) = (u + j_1)(v + j_2) = uv + j_3 = E(uv) \text{ modulo } I,$$

for $j_1, j_2, j_3 \in I$.

We can easily see, by the above equalities, that E, by itself, is not an homomorphism. However, in the context of this work, being an homomorphism modulo I is what is required.

3.2 The ring

Consider the following family of multivariate polynomial rings with coefficients in $\mathbb{Z}_p$,

$$S_n = \mathbb{Z}_p[x_1, ..., x_n]/\langle x_1^2 - x_1, ..., x_n^2 - x_n \rangle,$$

where $n \in \mathbb{N}$ and p is a prime number.

First of all, notice that S_n is a linear vector space over $\mathbb{Z}_p$. The so-called standard basis of this vector space consists of all distinct monomials of S_n with coefficient 1. We can easily see that there are 2^n of them, which means that S_n has p^{2^n} elements. For the most part, due to practical reasons, we think of elements of S_n as vectors, instead of polynomials. We do this by fixing a order among the elements of the standard basis and, given a polynomial, we simply compute its coordinate vector in relation to our ordered standard basis.

Example 3.1. Let $n = 3$ and $p = 5$, then we have,

$$S_3 = \mathbb{Z}_5[x_1, x_2, x_3]/\langle x_1^2 - x_1, x_2^2 - x_2, x_3^2 - x_3 \rangle.$$

First we choose an order among the elements of the standard basis of S_3, e.g.,

$$1, x_1, x_2, x_1x_2, x_3, x_1x_3, x_2x_3, x_1x_2x_3.$$

Then, for an arbitrary element of S_3, e.g.,

$$1 + 4x_1x_2 + 2x_3 + x_2x_3,$$

we have the corresponding coordinate vector,

$$(1, 0, 0, 4, 2, 0, 1, 0).$$

It is easy to see that the result of adding two of these coordinate vectors, with the usual sum of vectors, is equal to the vector obtained from the result of adding the respective polynomials. On the other hand, it is not obvious how one can define a product between those vectors that represents the product of their respective polynomials.

Nonetheless, in Section 3.4, a technique will be introduced which allows to define such a product.

3.3 The ideal

In Section 3.1 was mentioned that the generators of the ideal I, used for encryption, had to be chosen carefully, otherwise it could lead to incorrect decryptions. Before we explicitly introduce it, notice that, in this scheme, ciphertexts are elements of

$$S_r = \mathbb{Z}_p[x_1, ..., x_r]/\langle x_1^2 - x_1, ..., x_r^2 - x_r\rangle,$$

where $r > n$. The ideal I, used for encryption, is then defined over S_r in the following way

$$I = \langle x_{n+1} - w_n, ..., x_r - w_{r-1}\rangle,$$

where w_m is a random idempotent of S_m for $m \in \{n, ..., r-1\}$. Later in this section, it will be explained why choosing the generators of the ideal I as above guarantees that the decryption process works properly.

Recall that, now, the decryption process can be represented by the following diagram

$$S_r \xrightarrow{\rho} S_r/I \xrightarrow{\varphi} S_n.$$

Remark 3.2. In our opinion, Gribov et. al chose the generators of the ideal I to have the form $x_m - w_{m-1}$, for $m \in \{n+1, ..., r\}$, in order to easily define the map ρ, as we simply map every x_m to w_{m-1}. In this way, it is assured that any element from S_r is equivalent, in S_r/I, to an element of S_n, i.e., every equivalence class in S_r/I has at least one element of S_n.

Ideally, every equivalence class in S_r/I should have one and only one representative of S_n, such that, one can easily define an isomorphism φ between S_r/I and S_n, and therefore guarantee that decryption works correctly every time. For this to happen, there cannot be any elements of S_n in I, except 0, since this would result on two distinct elements of S_n in the same equivalence class of S_r/I.

We now proceed to prove that if the ideal I is generated as defined above, then, the only polynomial in the intersection of S_n with I is the null polynomial.

For the remaining of this section we use the following notation,

$$f(x_1, ..., x_r)$$

to denote a polynomial f from S_r, i.e., a polynomial which involves, at most, the generators $x_1, ..., x_r$.

Let $h(x_1, ...x_n) \in S_n \cap I$, then, there exists polynomials $f_1(x_1, ..., x_r), ..., f_{r-n}(x_1, ..., x_r) \in S_r$ such that,

$$h(x_1, ..., x_n) = f_1(x_1, ..., x_r) \cdot (x_{n+1} - w_n(x_1, ..., x_n)) + ... + f_{r-n}(x_1, ..., x_r) \cdot (x_r - w_{r-1}(x_1, ..., x_{r-1})).$$

We can easily see that the application $\mathbb{Z}_p[x_1, ..., x_r] \to \mathbb{Z}_p$ that evaluates a polynomial $g(x_1, ..., x_r)$ in a point $(a_1, ..., a_r)$ is well defined. However, the corresponding application between $\mathbb{Z}_p[x_1, ..., x_r]/\langle x_1^2 - x_1, ..., x_r^2 - x_r\rangle$ and $\mathbb{Z}_p$ is well defined only if $(a_1, ..., a_r)$ is a zero of every polynomial in $\langle x_1^2 - x_1, ..., x_r^2 - x_r\rangle$. This means that such applications is well defined only when every a_i, for $i \in \{1, ..., r\}$, is an idempotent. Therefore, we are only allowed to substitute $x_m = w_{m-1}(x_1, ..., x_{m-1})$ in $h(x_1, ..., x_n)$ when $w_{m-1}(x_1, ..., x_{m-1})$ is an idempotent, which is the case for $m \in \{n + 1, ..., r\}$. Performing these substitutions we obtain $h(x_1, ..., x_n) = 0$ which proves that the only element in the intersection of S_n with I is the null polynomial.
That said, if I is generated as above, then every equivalence class in S_r/I has only one element of S_n, and this allows us to easily define the isomorphism φ.

Notice that, if the elements w_m are not necessarily idempotents, then its possible that an element different from the null polynomial belongs to the intersection of S_n with I, and this means that φ would not be an isomorphism since $|S_r/I| < |S_n|$.

In the following example, we illustrate this argument with a simple case.

Example 3.2. Let $n = 2$, $p = 5$ and $r = 3$. Consider the ideal I of S_3 generated by $\langle x_3 - 2 \rangle$, noting that 2 is not an idempotent in S_2. We have that,

$$2 = (x_3 - 2) \cdot (-x_3 - 1).$$

Therefore $2 \in S_2 \cap I$, which means that $3 \cdot 2 \equiv 1 \pmod 5 \in I$. This lets us conclude that $I = S_3$, so we have that $|S_3/I| = 1 \neq |S_2|$, therefore there is no isomorphism between this two rings.

3.4 Idempotents

Notice that in S_n every x_i is an idempotent, and since a product of idempotents is again an idempotent, every monomial with coefficient 1 is also an idempotent of S_n. In fact, these are not the only idempotents that exist in S_n. For example, $1 - x_j$ is also an idempotent in this ring, for any $j \in \{1, ..., n\}$.

Consider the following elements

$$e_F = \prod_{i \in F} x_i \cdot \prod_{j \notin F} (1 - x_j), \tag{3.1}$$

where $F \subseteq \{1, ..., n\}$. Since every x_i and every $1 - x_j$ are idempotents of S_n, we can easily see that every e_F is also an idempotent of S_n, and there are 2^n of them, which is the number of subsets of $\{1, ..., n\}$. Notice also that for any two elements e_F, e_G such that $F \neq G$, we have that $e_F e_G = 0$, therefore these elements are pairwise orthogonal. This means that any sum of different e_F is still an idempotent, and there are 2^{2^n} of such sums. Moreover, these elements constitute an idempotent orthogonal basis of S_n and we take advantage of this when we have to generate random idempotents, i.e., our random idempotents are simply a random sum of elements from this idempotent orthogonal basis.

Recall that we could represent elements of S_n as vectors, however it was not easy to define a product between vectors that would represent the product of the corresponding polynomials of S_n. This is no longer the case if we consider the coordinate vectors relative to the orthogonal basis and define the product between any two vectors as follows

$$(a_1, a_2, ..., a_n) \cdot (b_1, b_2, ..., b_n) = (a_1 b_1, a_2 b_2, ..., a_n b_n).$$

Multiplying vectors as above guarantees that the result obtained will be equal to the vector representing the product of the corresponding polynomials. This is true because, $e_F e_G = 0$ for any $F \neq G$, i.e., the product of any two coordinates in distinct positions is 0. Therefore,

we only have to multiply coordinates in the same position in its respective vectors. Notice that the usual sum of vectors still satisfies this same property regarding the sum of polynomials.

From what was said above, we can conclude that S_n is isomorphic to a direct sum of 2^n copies of $\mathbb{Z}_p$.

Example 3.3. Considering Example 3.2, let us now build the respective idempotent orthogonal basis, where $F \subseteq \{1,2,3\}$, i.e., $F \in \{\{\},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$. Then, the elements of the orthogonal basis are:

$$\begin{aligned} e_0 = e_{\{\}} &= 1 + 4x_1 + 4x_2 + 4x_3 + x_1x_2 + x_1x_3 + x_2x_3 + 4x_1x_2x_3 \\ e_1 = e_{\{1\}} &= x_1 + 4x_1x_2 + 4x_1x_3 + x_1x_2x_3 \\ e_2 = e_{\{2\}} &= x_2 + 4x_1x_2 + 4x_2x_3 + x_1x_2x_3 \\ e_3 = e_{\{3\}} &= x_3 + 4x_1x_3 + 4x_2x_3 + x_1x_2x_3 \\ e_4 = e_{\{1,2\}} &= x_1x_2 + 4x_1x_2x_3 \\ e_5 = e_{\{1,3\}} &= x_1x_3 + 4x_1x_2x_3 \\ e_6 = e_{\{2,3\}} &= x_2x_3 + 4x_1x_2x_3 \\ e_7 = e_{\{1,2,3\}} &= x_1x_2x_3. \end{aligned}$$

Now, let, a and b be two elements of S_3. Take a to be the element defined in the previous section, $a = 1 + 4x_1x_2 + 2x_3 + x_2x_3$, and let $b = 3x_1 + x_1x_2x_3$. For further comparison, notice that,

$$\begin{aligned} a + b &= 1 + 3x_1 + 4x_1x_2 + 2x_3 + x_2x_3 + x_1x_2x_3 \\ ab &= 3x_1 + x_1x_3 + 2x_1x_2 + x_1x_2x_3. \end{aligned}$$

To compute the coordinates of a and b in the orthogonal basis we used a built-in function in Python, obtaining:

$$\begin{aligned} a &= e_0 + e_1 + e_2 + 3e_3 + 3e_5 + 4e_6 + 3e_7 \\ b &= 3e_1 + 3e_4 + 3e_5 + 4e_7, \end{aligned}$$

then we can represent a and b as a vector as follows,

$$\begin{aligned} a &\longrightarrow a' = (1,1,1,3,0,3,4,3) \\ b &\longrightarrow b' = (0,3,0,0,3,3,0,4). \end{aligned}$$

Applying the usual sum between vectors and the product defined as above, we get,

$$a' + b' = (1,4,1,3,3,1,4,2) = e_0 + 4e_1 + e_2 + 3e_3 + 3e_4 + e_5 + 2e_6 + 2e_7 =$$
$$1 + 3x_1 + 4x_1x_2 + 2x_3 + x_3x_2 + x_1x_2x_3 = a + b,$$

and,

$$a'b' = (0,3,0,0,0,4,0,2) = 3e_1 + 4e_5 + 2e_7 = 3x_1 + x_1x_3 + 2x_1x_2 + x_1x_2x_3 = ab.$$

This is of great use when we introduce private search in Chapter 5, where we have to perform several multiplications of this kind, allowing us to save a significant amount of processing time.

3.5 Detailed Encryption Process

After discussing an overview of the whole encryption process as well as some useful information moving forward, such as the form of the rings and the ability to build an idempotent orthogonal basis, we are now in the right conditions to present a more in depth description on how every step of the encryption process actually works.

The first step is key-generation, where we basically setup our encryption scheme so that the following steps are as objective as possible. Then we move onto the encryption and decryption parts, concluding this section with a simple example of an encryption/decryption procedure.

3.5.1 Key-Generation

First of all, Alice (the owner of a private database of plaintexts) creates the ring S_n:

$$S_n = \mathbb{Z}_p[x_1, ..., x_n]/\langle x_1^2 - x_1, ..., x_n^2 - x_n\rangle,$$

where $n \in \mathbb{N}$ is private and p is a prime number.

In Section 3.3, we mentioned that ciphertexts are elements of a ring S_r for $r > n$, and that the ideal used for encryption is defined by

$$I = \langle x_{n+1} - w_n, ..., x_r - w_{r-1}\rangle,$$

where each w_m represents a random idempotent from S_m.

The first step in key generation process is to build these two structures, i.e., to choose a value for r and then generate the random idempotents that are used for the generators of

the ideal I. As mention before, we compute these random idempotents, of S_m, as a random sum of elements from the respective orthogonal basis, i.e.,

$$w_m = \sum_{i=0}^{2^m} \alpha_i e_i,$$

where $m \in \{n+1, ..., r\}$, $\alpha_i \in \{0, 1\}$ and $(e_0, ..., e_{2^m})$ is the orthogonal basis of S_m.

The next step is to perform a change of basis from the standard basis of S_r to the orthogonal basis, followed by a random permutation on the generators of this basis, which is part of the private-key.

Finally, Alice publishes the ring P:

$$P = \mathbb{Z}_p[e_1, ..., e_{2^r}]/\langle e_1^2 - e_1, ..., e_{2^r}^2 - e_{2^r} \rangle,$$

where $e_i e_j = 0, \ \forall i \neq j$.

Remark 3.3. In our code, this procedure is done through a function called `prik`, shared in Appendix A.1, that given inputs n, p and r, it outputs, among other things, the idempotent orthogonal basis and the set of random idempotents selected for the generators of the ideal.

Notice that it is possible that certain elements from the orthogonal basis have mutual null coordinates across all ideal's generators. As a matter of fact, during this project we were able to prove that this number of elements is constant given a fixed value for n, as we discuss further in Section 4.2.3.

3.5.2 Encryption

As mentioned in Section 3.1, to encrypt a plaintext $u \in S_n$, we simply sum a random element i of the ideal I, to it, i.e., $E(u) = u + i$. These random elements of I are of the form

$$\sum_{j=n+1}^{r} (x_j - w_{j-1}) \cdot h_j,$$

where h_j is a random element of S_r. We then take the result of this sum, which is an element of S_r, and convert it to the orthogonal basis of S_r, obtaining, at last, our ciphertext as an element of the published ring P.

Remark 3.4. In our code, the encryption process is slightly different. In order to optimize the code as much as possible, we represent elements of S_r as coordinate vectors regarding

the respective orthogonal basis as soon as possible. This means that, by the time we perform encryption, we no longer generate a random element of our ideal, exactly as said above.

The first thing to note is that, since h_j is a random element of S_r, then this means that, in vector form, regarding the orthogonal basis, h_j is a random vector with coefficients in $\mathbb{Z}_p$. As for I, at this point we already have every of its generators in the orthogonal basis, in vector form, so we apply the product defined above in Section 3.4 for every pair of generator and respective random vector, h_j. This then gives us a vector with the coordinates of a random element of I in the orthogonal basis.

Finally, all we have to do is to turn u into a vector regarding the orthogonal basis, as we saw in Example 3.3, and add the random element obtained.

3.5.3 Decryption

When decrypting we no longer feel the need to work with vectors, so the first step is always to return to the standard basis. Then, all we have to do is to replace each x_j by w_{j-1}, starting with $j = r$ and ending with $j = n + 1$. When we conclude this procedure we will have our original plaintext.

Remark 3.5. Regarding our code, this step is done exactly as described. As mentioned in the key-generation step, the function `prik` outputs us the set of random idempotents w_j, used to generate the ideal, so we just need to use a simple substitution command.

To summarize, the private key consists of:

- The generators of the ideal I of S_r;
- The permutation π used on the elements of the orthogonal basis;
- The embedding from the database of plaintexts onto S_n.

3.5.4 Example

For a better understanding of everything that has been said in this chapter, lets take a look at an example of an encryption/decryption process. We chose to leave out the random permutation because it is irrelevant in this example and we also leave out the embedding process since it will be discussed in the following subsections.

The first thing to do is to set the parameters. Let $n = 2$, $p = 5$ and $r = n + 2$. Then we proceed to key-generation, where, with the aid of our function `prik` we obtain,

$$w_2 = 4x_1x_2 + x_1$$
$$w_3 = 4x_1x_2 + x_1 + x_2.$$

With this set of random idempotents, respectively from S_2 and S_3, we can now build our ideal I as follows,

$$I = \langle x_3 - w_2, x_4 - w_3 \rangle.$$

Then, choosing three random elements from S_4, h_1, h_2 and h_3 we have everything necessary to perform encryption on a plaintext.
Recall that plaintexts are elements of

$$S_2 = \mathbb{Z}_5[x_1, x_2]/\langle x_1^2 - x_1, x_2^2 - x_2 \rangle.$$

Then, choosing a plaintext at random, for example, $m = x_1 + 4x_1x_2$, we have that

$$E(m) = m + E(0),$$

where

$$E(0) = (x_3 - w_2) \cdot h_1 + (x_4 - w_3) \cdot h_2.$$

Therefore, we obtain,

$$c = E(m) = 2x_1x_2x_3x_4 + x_1x_2x_3 + 4x_1x_2 + 2x_1x_3 + 2x_2x_3 + 2x_2x_4 + x_3x_4 + x_1 + 3x_2 + 2x_3,$$

where $h1$ and h_2 are the following

$$h_1 = 3x_1x_2x_3x_4 + x_1x_2x_3 + 4x_1x_2x_4 + 2x_1x_3x_4 + x_2x_3x_4 + 2x_1x_2 + x_2x_3 + 4x_1x_4 + 4x_2x_4 + 4x_3x_4 + 4x_1 + 2x_2 + x_3 + x_4 + 1$$
$$h_2 = x_1x_2x_3x_4 + x_1x_2x_3 + 3x_1x_2x_4 + 2x_2x_3x_4 + x_1x_2 + 3x_1x_3 + 3x_2x_3 + 3x_1x_4 + 3x_2x_4 + 3x_3x_4 + 4x_1 + x_2 + 3x_3 + 4x_4 + 1.$$

Finally we just have to convert c to a linear combination of e_i,

$$c = e_1 + 3e_2 + 2e_3 + 3e_5 + e_7 + 2e_8 + 3e_{10} + e_{13}.$$

To perform decryption on c, we first have to return to the standard basis and then substitute every x_m for w_{m-1}, starting with $m = 4$ and ending with $m = 3$, i.e., there are, in this case, two steps of substitution:

- 1^{st} Step ($x_4 \longrightarrow w_3$)

$$c_1 = 2x_1x_2x_3 + 4x_1x_2 + 3x_1x_3 + 3x_2x_3 + x_1 + 2x_3$$

- 2^{nd} Step ($x_3 \longrightarrow w_2$)

$$c_2 = 4x_1x_2 + x_1$$

After performing decryption we got c_2, which is equal to m, our original plaintext.

Consider now another plaintext $m' = 2x_2$, and its respective ciphertext

$$c' = E(m') =$$
$$4x_1x_2x_3x_4 + 2x_1x_2x_3 + 3x_1x_3x_4 + 4x_1x_3 + 3x_2x_3 + x_1x_4 + 2x_2x_4 + 4x_1 + 3x_2 + x_3 + 2x_4 =$$
$$= 4e_1 + 3e_2 + e_3 + 2e_4 + 2e_5 + 4e_6 + 2e_7 + 2e_8 + 2e_9 + 3e_{10} + 2e_{11} + 2e_{12} + e_{14} + 4e_{15},$$

where h_1 and h_2 are the following,

$$h_1 = 4x_1x_2x_3x_4 + 2x_1x_2x_3 + 4x_1x_2x_4 + 4x_1x_3x_4 + 2x_1x_2 + 2x_1x_3 + 3x_2x_3 + x_1x_4 + 4x_1 + x_2 + 2x_3 + 4x_4 + 4$$
$$h_2 = x_1x_2x_3x_4 + 4x_1x_2x_3 + x_1x_2x_4 + 4x_1x_3x_4 + 2x_1x_2 + 3x_2x_3 + 2x_1x_4 + 2x_2x_4 + 3x_3x_4 + 4x_1 + 3x_3 + 3x_4 + 4$$

We have that,

$$\pi = c \cdot c' = 4e_1 + 4e_2 + 2e_3 + e_5 + 2e_7 + 4e_8 + 4e_{10},$$

and,

$$\sigma = c + c' = e_2 + 3e_3 + 2e_4 + 4e_6 + 3e_7 + 4e_8 + 2e_9 + e_{10} + 2e_{11} + 2e_{12} + e_{13} + e_{14} + 4e_{15}.$$

When performing decryption on π and σ we obtain,

$$Dec(\pi) = 0 = 2x_2 \cdot (x_1 + 4x_1x_2) = m \cdot m',$$

and,

$$Dec(\sigma) = 2x_2 + x_1 + 4x_1x_2 = m + m'.$$

As expected, the result of computations performed between ciphertexts, when decrypted, is equal to the result of this same operations performed on their respective plaintexts.

3.6 Embeddings

As mentioned before, in practical applications we often need to perform an embedding from our database (set of real-life plaintexts) onto S_n. There are two types of embedding, depending on whether our database has ring structure or not.

3.6.1 Embedding without ring structure

This is the simplest case, since our database does not have a ring structure, we simply do a one-to-one embedding without any sort of restrictions, which means that the number of possible embeddings of a database with k elements onto S_n is $\frac{p^{2^n}!}{(p^{2^n}-k)!}$, where $k \leq p^{2^n}$.

This type of embedding assures us that if any parts of a plaintext got exposed, it would not necessarily mean that elements of our original database would be exposed, since an attacker would have to know the embedding used. However, this might be the case when our database does have ring structure, as we will see in the next subsection as well as in the next chapter, when discussing security.

3.6.2 Embedding with ring structure

If our database has ring structure it is possible to perform an homomorphic embedding α, which means that from the embedding of the element 1, $\alpha(1)$, we can completely determine α through homomorphic properties. Another aspect to have into account is the fact that 1 is an idempotent of our ring, therefore $\alpha(1)$ must also be an idempotent of S_n, therefore there are 2^{2^n} possible homomorphic embeddings, i.e., the number of idempotents in S_n.

As we will explain further when discussing security, if our embedding is homomorphic, there are more concerns regarding security compared to a one-to-one embedding. However, we only need to know the embedding of 1 to determine the whole embedding, which is not the case with an one-to-one embedding. This is because, in the latter case, we have to know the embedding of every single element of our database.

To conclude this subsection, consider the following example of an homomorphic embedding between a database with ring structure and S_n.

Example 3.4. Suppose our database is the ring $\mathbb{Z}_5$ and we wish to perform an embedding into S_2, with $p = 5$, which has $5^4 = 625$ elements. As mentioned above, the first step is to define $\alpha(1)$, where $\alpha(1)$ is an idempotent of S_2. Recall that S_2 has $2^4 = 16$ idempotents, therefore, this is the amount of distinct homomorphic embeddings one can perform.

Choosing $\alpha(1) = x_1x_2$, one gets that α is completely determined as follows

$$\alpha(j) = jx_1x_2,$$

for $j \in \mathbb{Z}_5$.

Note that, since this embedding is fully homomorphic, it preserves every computation performed, and this is the main reason why there are more security issues, as we explain in the next chapter.

Chapter 4

Security

In this chapter, we discuss some security related topics regarding Gribov's encryption scheme.

First of all, notice that since this scheme is private-key (even though some information has to be public, specially to allow third-party private search), discussing security is slightly different compared to a scheme that is public-key.
Nonetheless, we are going to check why this scheme is secure against ciphertext only attacks, as well as why accumulating encryptions of 0 will not necessarily endanger original plaintexts. Then we move to another topic, aimed at the case where the database of plaintexts has ring structure and therefore we can perform a fully homomorphic embedding onto S_n. Within this discussion, we share some experiments as well as an analysis of a couple of enhancements suggested by the authors of the original scheme.

4.1 Security against ciphertext-only attacks

For an homomorphic encryption scheme, security against ciphertext-only attacks, i.e., the attacker only has access to a set of ciphertexts, seems to be difficult to achieve since if an attacker has access to enough ciphertexts he/she may be able to obtain several (plaintext, ciphertext) pairs using the homomorphic properties of the encryption function. Then, the attacker is able to perform a known-plaintext attack.

If original plaintexts come from $\mathbb{Z}_p$, then one can easily identify three ways for an attacker to obtain a pair (plaintext, ciphertext) from ciphertexts only:

1. Since $\mathbb{Z}_p$ is commutative, for any $a, b \in \mathbb{Z}_p$, we have $ab - ba = 0$. Then, by the homomorphic property of the encryption function, $xy - yx = E(0)$, for any x, y in the encrypted database of plaintexts.

2. For any $a \in \mathbb{Z}_p$, $a + ... + a = 0$ (p times), then, $x + ... + x = E(0)$ (p times) for any x in the encrypted database.

3. For any $a \in \mathbb{Z}_p$, by Fermat's little theorem we have $a^{p-1} = 1$, then, $x^{p-1} = E(1)$ for x in the encrypted database.

In this scheme, S_n is isomorphic to a direct sum of copies of $\mathbb{Z}_p$, as mentioned before. Therefore, we have that $ab - ba = 0$, $a + ... + a = 0$ (p times) and $a^{p-1} = 1$. This lets us conclude that from ciphertexts only, the attacker cannot obtain any non-trivial pair (plaintext, ciphertext).
This, however, means that an attacker might be able to accumulate encryptions of 0 through several queries. In the next subsection, we will explain why this is not necessarily an issue.

4.1.1 Accumulating encryptions of zero

In the previous subsection we showed that an attacker cannot obtain any non trivial (plaintext, ciphertext) pairs. However, he/she can indeed obtain several different encryptions of 0, i.e., elements of our ideal I, through several queries to the encrypted database. Our goal is to show that this is not necessarily an issue with this encryption scheme.

Note that the ideal I has finite dimension, which means that, in theory, with enough queries against the encrypted database, the attacker can eventually collect every element of this ideal. The obvious solution would be to increase the dimension of I such that it would be harder to gather every single one of its elements, however this would make the encryption scheme less practical which is not what we seek.

Even if we do not make any sort of enhancement, and supposing that the attacker is able to recover the ideal I, this just means that he can, from now on, recognize encryptions of 0. Moreover, this will not affect, in any way, the security of non-zero elements since in order for the attacker to be able to decrypt non-zero elements correctly he needs to have knowledge not only of the ideal I but also the specific mapping ρ mentioned in Section 3.1.

In the following example, we simulate a scenario in which we attempt to perform decryption on a ciphertext using a different mapping ρ.

4.1.1.1 Example

In this example we are going to be able to see what happens if we, as an attacker that has no knowledge of the mapping ρ, attempted to perform decryption on a ciphertext.

In the first part of this example we will be doing key generation and then encrypting a random plaintext as usual. For the second part we assume the role of an attacker that has already gathered every element of our ideal I. However, a different mapping will be chosen, i.e., we will find a different basis for the ideal I, and then attempt to perform decryption, only to come to the conclusion that the result is not our initial plaintext.

As always, the very first step is to set the parameters,

$$n = 2, \qquad r = n + 2 = 4, \qquad p = 3.$$

Then, we proceed to key generation, and, once again with the aid of our function `prik` we get

$$w_2 = x_1$$
$$w_3 = x_2 x_3,$$

which leads us to our ideal I:

$$I = \langle x_3 - x_1, x_4 - x_2 x_3 \rangle.$$

Finally we choose a plaintext at random, e.g., $x_2 - x_1$, and encrypt it, obtaining

$$c = E(x_2 - x_1) = x_2 - x_3 + x_4 x_3 x_1 - x_2 x_3 x_1,$$

where $h_1 = -1$ and $h_2 = x_1 x_3$.

Recall that we perform decryption by substituting each x_i by w_{i-1}, starting with $i = r$ and ending with $i = n + 1$, therefore our map from S_2 to S_4/I is given by $x_i \to w_{i-1}$.

We now take on the role of the attacker, which already knows every element of I. However, we wrongly assume that I has been generated as follows

$$I = \langle x_3 - x_1 + x_4 - x_2 x_3, x_4 - x_2 x_3 \rangle.$$

Then, we consider the following map:

$$1 \to 1 \qquad x_3 \to x_3 \qquad x_2 \to x_2$$

$$x_1x_2 \to x_1x_2 \qquad x_2x_3 \to x_2x_3 \qquad x_1x_3 \to x_1x_3$$

$$x_1x_2x_3 \to x_1x_2x_3 \qquad x_4 \to x_2x_3$$

$$x_1 \to x_3 + x_4 - x_2x_3.$$

If we attempt to decrypt c using this mapping, we get $x_2 - x_3$, which is not the original plaintext.

The previous example shows that accumulating encryptions of 0, even to the extent where we get every element of the ideal I, would not endanger non-zero elements, as long as the attacker is not aware of our chosen map ρ.

If the embedding from our database onto S_n is one-to-one, and not homomorphic, then, even if an attacker was able to obtain a non-trivial pair plaintext / ciphertext, this would not help him recover any other pair of such kind, since there is no algebraic correlation between encryptions of plaintexts. Concluding that with such embedding, is not enough for the attacker to correctly decrypt an element of S_r, since he still has to figure the correspondent element in our database, which is very unlikely with a one-to-one embedding.

On the other hand, if the embedding is homomorphic, correctly decrypting a ciphertext, might be enough to expose the original element of our database.

4.2 Security issues with an homomorphic embedding

4.2.1 Encryptions of the unity

As we mentioned towards the end of the previous subsection, when the embedding from our original database to S_n is an homomorphism, there are some security liabilities, since an attacker can use homomorphic properties to recover the original plaintext or even use certain known encryptions to aid him/her recover more pairs (plaintext, ciphertext).

In the previous subsection, three main ways an attacker can recover a pair (plaintext, ciphertext) were mentioned, even though he/she could not obtain any non-trivial pairs. This, however, can still be a problem, since with an homomorphic embedding, obtaining an encryption of 1 is much more significant due to the homomorphic properties of the embedding function.

Let us consider a computational unbounded attacker with access to the encrypted database, and which has accumulated enough distinct encryptions of 0 such that he/she is

able to recognize future encryptions of 0. This attacker can just go over every ciphertext and check which ones are idempotents modulo I, noting that with p prime the only idempotents of $\mathbb{Z}_p$ are 0 and 1. Therefore, with an homomorphic embedding, the only ciphertexts that are idempotents modulo I would be the respective encryptions of the idempotents of $\mathbb{Z}_p$.

Then, having the pair $(1, E(1))$, the attacker can use a brute force attack in the following way: for a certain ciphertext w, he/she goes over every $m \in \mathbb{Z}_p$ and checks if $w - E(1)m \in I$, where due to the homomorphic properties we have $E(1)m = E(m)$. Therefore, if the attacker verifies this for any pair (m, w), it means that w is an encryption of m.

There is also another way, although not as practical, for an attacker to obtain $E(1)$. Knowing that the encryption of an element $k \in \mathbb{Z}_p$ is of the form $ku + e$, where $e \in I$ and u is the idempotent of S_n in which the element 1 has been embedded to, he/she can just go over every $m \in \mathbb{Z}_p$ and divide the ciphertext $ku + e$ by m until the result is an idempotent modulo I, because if it is, then it must be of the form $u + e'$ where $e' \in I$. Having this encryption of 1 he/she can now replicate the procedure above.

4.2.2 Mutual null coordinates across ideal generators

First of all, note the following result.

Proposition 4.1. *Let $\mathbb{K}$ be a field and $R = \mathbb{K}^n$. The ideals of R are exactly its subsets of the form $\bigoplus_{i=1}^{n} A_i$ where for each i, $A_i = 0$ or $A_i = \mathbb{K}$.*

Proof. Let I be an ideal of R and let F be the set of all $i \in \{1, ..., n\}$ such that the *i-th* coordinate of all elements of I is null. It is clear that $I \subseteq \bigoplus_{i=1}^{n} A_i$ where $A_i = 0$ if and only if $i \in F$.

For every $j \notin F$, exists u in I such that $u_j \neq 0$. Let $u^{(j)}$ denote an element of I where $u_j \neq 0$. Since $\mathbb{K}$ is a field, u_j has an inverse u_j^{-1}, and taking $y = (0, ..., u_j^{-1}, ..., 0)$ one has $(0, ..., 1, ..., 0) = u^{(j)}y \in I$.

Finally, the sum of all these elements is equal to $u' = (..., 1, ..., 0, ..., 1, ...) \in I$, where the null coordinates are given by the set of indices F. Then, for any $u \in R$ we have that $uu' = u \in I$. Therefore, $\bigoplus_{i=1}^{n} A_i \subseteq I$, where $A_i = 0$ iff $i \in F$. □

We have already seen that S_r can be published explicitly as a direct sum of copies of $\mathbb{Z}_p$. Then, by the above proposition, one can conclude that every element of an ideal of S_r would have some coordinates with random values from $\mathbb{Z}_p$, while others would just

be equal to 0 (in vector form regarding the orthogonal basis), which means that some elements of the original plaintext could be exposed, since the encryption process is a simple addition. In other words, given a plaintext as a vector, we perform encryption by adding a random element of I to the original plaintext, however we now know that elements of I might have some coordinates null, which means that the vector resulting from adding our original plaintext with a random element of I would have some of its coordinates unchanged, which are exactly the exposed elements of the original plaintext.

Remark 4.2. This proposition was also very important on the optimization of our code, because now we can generate a random element of I with one computation only, which was not the case until this point. The way we do this is by first checking what are the mutual null coordinates among the generators of the ideal, as vectors, in the orthogonal basis of S_r. Let the set of these coordinates be N, we then generate a random vector of size 2^r with coefficients in $\mathbb{Z}_p$ where we annihilate every coordinate in N.

Recall the example in Subsection 4.1.1.1, where we had:

$$w_2 = x_1$$
$$w_3 = x_2 x_3,$$

which leads us to (as described in Example 3.3),

$$w_2 = (0,1,0,0,0,1,1,1,0,0,0,1,1,1,0,1)$$
$$w_3 = (0,0,0,0,0,0,0,0,1,0,0,1,0,0,1,1).$$

These are the coordinates of w_2 and w_3 in the orthogonal basis of S_4. If we compute the coordinates of x_3 and x_4 on this basis, we get

$$x_3 - w_2 = (0,4,0,1,0,4,0,4,1,0,1,0,4,0,1,0)$$
$$x_4 - w_3 = (0,0,0,0,1,0,0,1,4,1,1,4,1,1,0,0).$$

Then, $N = \{0,2,6,15\}$ is the set of mutual null coordinates between these two generators of our ideal I. Because of the above proposition, we know that any element of I will have every coordinate with random values, except the coordinates in N which will be always equal to 0. This means that we can compute $E(0)$ as

$$E(0) = (0, a_1, 0, a_3, a_4, a_5, 0, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, 0),$$

where $a_i \in \mathbb{Z}_5$, randomly chosen.

Suppose that now we wish to encrypt a plaintext $m = (m_0, ..., m_{15})$. Recall that $E(m) = m + E(0)$, therefore

$$E(m) = (m_0, a_1 + m_1, m_2, a_3 + m_3, a_4 + m_4, a_5 + m_5, m_6, a_7 + m_7, a_8 + m_8, a_9 + m_9, a_{10} + m_{10}, a_{11} + m_{11}, a_{12} + m_{12}, a_{13} + m_{13}, a_{14} + m_{14}, m_{15}).$$

We can easily see that some coordinates of $E(m)$ are just the coordinates of the original plaintext, which means that certain parts of the original plaintext might be exposed if the attacker is aware of which are the mutual null coordinates across the ideal generators. Even though this is most likely not the case, since the generators of the ideal I are part of the private-key, we decided to perform an experiment where we assume that an attacker has access to the encrypted database and attempts to recover these mutual null coordinates. The description of this experiment, as well as the obtained results are presented in the following subsection.

4.2.2.1 Mutual null coordinates experiment

In this section, we perform an experiment to show that one might be able to identify the mutual null coordinates across the ideal generators only by having access to the encrypted database and, therefore, be able to potentially recover certain elements of the plaintexts.

Notice that, with an homomorphic embedding, our database will be represented in S_n by multiples of our initial idempotent, i.e., the one chosen as the embedding of 1.

This example will be divided in two parts. In the first part, we use an homomorphic embedding to show that an attacker can easily identify the mutual null coordinates, and therefore can expose parts of a plaintext. Since it is not easy to build an example where we can choose an homomorphic embedding such that this scenario does not happen, in the second part of this example we use a one-to-one embedding to show the difference between a poorly chosen embedding and a good embedding.

Similar to Example 3.4, suppose our database is $\mathbb{Z}_{10007}$ and consider the following parameters:

$$n = 4, \quad p = 10007, \quad r = 7.$$

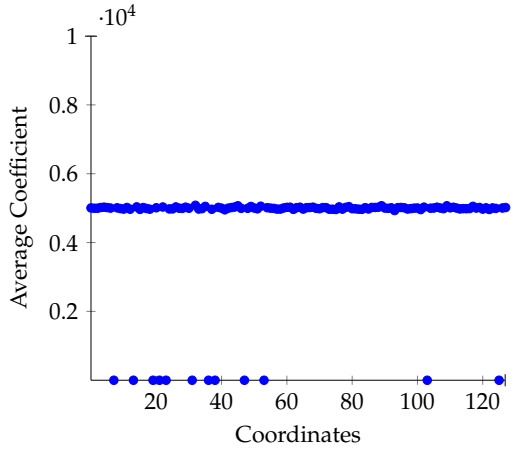
Let $\alpha(1) = m$, where m is a random monomial of S_4. Then, α can be completely determined as follows:

$$\alpha(j) = jm,$$

for $j \in \mathbb{Z}_{10007}$.

After performing this embedding, we now have an unencrypted database with elements of S_4. The next step is to encrypt this database, i.e., encrypt every single element of our database using the method described before.

A set of functions was developed, in Python, to illustrate that if an attacker has access to this encrypted database he/she can then compute the average value in every coordinate across every element. Therefore, in this way he/she is able to observe a different behaviour from the mutual null coordinates across the generator of the ideal I used for encryption. The developed functions are presented in the Appendix A.2, and the following figure shows the results of this experiment.



(A) n = 4, p = 10007, DB size = 10007

FIGURE 4.1: Mutual null coordinates - Homomorphic Embedding

As expected, we can clearly observe a different behaviour from some coordinates, that are exactly the mutual null coordinates across the generators of our ideal I used for encryption.

As mentioned, since it is not easy to build a good homomorphic embedding, in the second part of this experiment we use a one-to-one embedding to demonstrate that with an adequate embedding it is possible to diffuse these coordinates among the rest.

Consider our database to be the first k prime numbers, where k is a parameter of our choice. Since this set does not have a ring structure we perform a one-to-one embedding of each prime number into an element of S_n.

Given a prime number q, there are $\alpha_i \in \{0, 1, ..., p-1\}$ such that,

$$q = \sum_{i=0}^{2^n} \alpha_i p^i = \alpha_i + \alpha_i p + \alpha_i p^2 + ... + \alpha_{2^n} p^{2^n},$$

then, the embedding is performed as follows:

$$q \longrightarrow \alpha_0 + \alpha_1 x_0 + ... + \alpha_i m_i + ... + \alpha_{2^n} x_0 ... x_n,$$

where m_i is the *i-th* monomial of S_n.

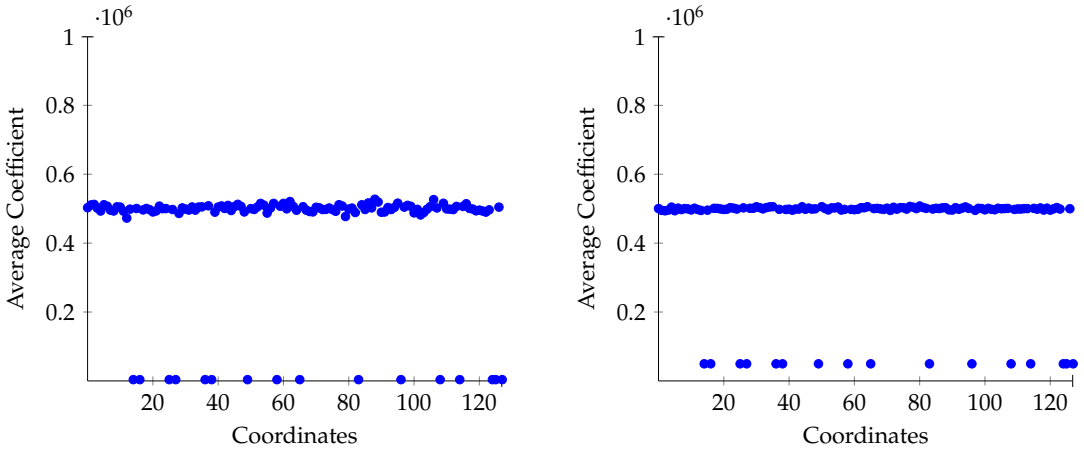
After performing the previous embedding for every element of the original database, we now have an unencrypted database with elements of S_n. The next step is to encrypt this set.

From now on, we will specify our parameters and present the results obtained for each of them, in plot form.

Consider the following parameters:

$$n = 4, \quad p = 1000003, \quad r = 7$$

Recall that the following figures represent the average value of every coordinate, throughout all elements of our encrypted database, with the parameters mentioned above:



(A) n = 4, p = 1000003, DB size = 1000

(B) n = 4, p = 1000003, DB size = 10000

Figure 4.2: Mutual null coordinates - One-to-one "poor" Embedding

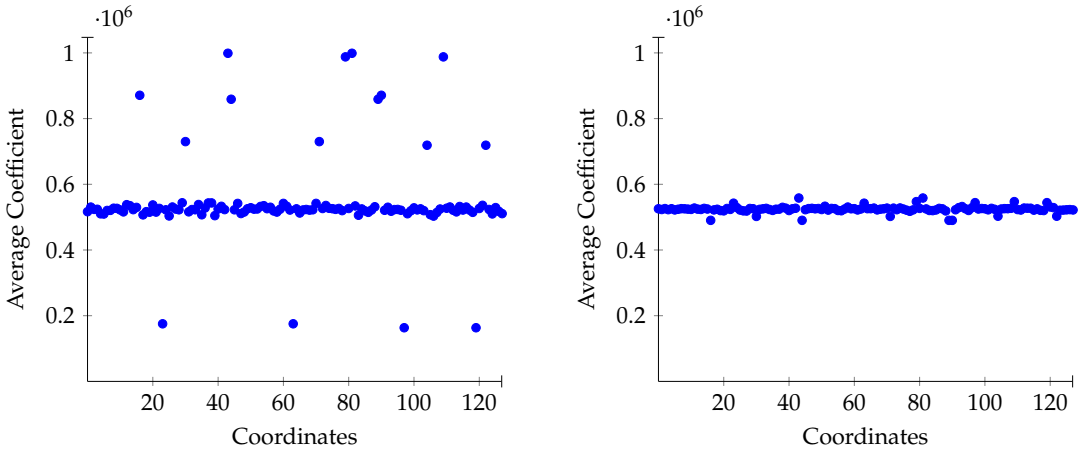
In both figures, we can clearly observe a different behavior in some coordinates, that correspond exactly to the mutual null coordinates among the ideal generators. However, this is an example of a "poor" embedding from our elements in S_n, since almost every coordinate of every tuple in our database is null. Due to computer limitations, we were not able to build a database as big as we wanted. However, in theory, if the last element of our database was a prime number with 140 bits, then this behavior would not be as obvious.

To try to illustrate this argument, let us slightly change the previous conditions. In the following scenario p is a prime number with 20 bits, and our database is the set of k

consecutive prime numbers such that the latest is the greatest prime number with $20 \cdot r$ bits. Consider, then, the following parameters:

$$n = 4, \quad p = 1048583, \quad r = 7, \quad \#Database = 1000/10000.$$

The results obtained are presented in Figure 4.3.



(A) n = 4, p = 1048583, DB size = 1000

(B) n = 4, p = 1048583, DB size = 10000

FIGURE 4.3: Mutual null coordinates - One-to-one"good" Embedding

We can immediately notice a different behavior in both plots, specially on the one on the right, i.e., as we increase the size of our database, the coordinates where we could previously observe a different behavior begin to blend in with the rest.

In conclusion, in order to omit, as much as possible, these mutual null coordinates across the ideal generators, we need to have in consideration the embedding used, i.e., we wish to have an embedding that guarantees that most elements in S_n are going to be used, the more the better. Recall once again that the second part of this example was to show the reader the difference between choosing a bad embedding instead of a good one. Nonetheless, the results are similar with an homomorphic embedding. However, as mentioned, it is not trivial to find an example of a good homomorphic embedding, which is an interesting future research topic.

4.2.3 2^n mutual null coordinates

When performing the previous experiment we noticed a interesting behavior regarding the mutual null coordinates among the ideal generator, this being the fact that in every test performed, the number of mutual null coordinates was always 2^n. Therefore, we decided

to look deeper into this question trying to prove that this does in fact happen every time, since we consider this to be a relevant aspect regarding the security of the encryption scheme.

We wish to show that among the generators $x_m - w_{m-1}$ for every $m \in \{n+1, ..., r\}$, where w_{m-1} is a random idempotent from S_{m-1}, there is always 2^n mutual null coordinates. In order to make this argument as intuitive as possible, note the following claims:

1. Each element of the orthogonal basis of S_n has exactly 2^{r-n} non-null coordinates in the orthogonal basis of S_r;

2. Given an arbitrary element of the orthogonal basis of S_n, half of its non-null coordinates are also non-null coordinates of $x_i, \forall i \in \{n+1, ..., r\}$.

Before getting into each claim, recall that the elements of the orthogonal basis of S_n are of the form:

$$e_F = \prod_{i \in F} x_i \cdot \prod_{j \notin F} (1 - x_j),$$

for $F \subseteq \{1, ..., n\}$. From this expression we know that each e_F represents a sum of monomials with coefficients alternating between 1 and -1.

Proposition 4.3. *Let $\overline{F} = \{1, ..., n\} \setminus F$, then, the monomials of e_F are of the form,*

$$m_J = (-1)^{|J|} \cdot \prod_{i \in F} x_i \cdot \prod_{j \in J} x_j,$$

for $J \subseteq \overline{F}$.

Proof. To prove that all monomials of e_F have this form, we need to prove that their sum, for every $J \subseteq \overline{F}$ is equal to e_F, i.e.,

$$\prod_{i \in F} x_i \cdot \prod_{j \in \overline{F}} (1 - x_j) = \sum_{J \subseteq \overline{F}} \left((-1)^{|J|} \cdot \prod_{i \in F} x_i \cdot \prod_{j \in J} x_j \right),$$

which means that is enough to prove that,

$$\prod_{j \in \overline{F}} (1 - x_j) = \sum_{J \subseteq \overline{F}} (-1)^{|J|} \cdot \prod_{j \in J} x_j.$$

Let us proceed by induction over the size of $\overline{F}$. If $|\overline{F}| = 0$, then $J = \{\}$, nd its obvious that we verify the expression above. Now, let $\overline{F} = \{b\}$ for $b \in \{1, ..., n\}$, then $J \in \{\{\}, \{b\}\}$, so we

have,

$$1 \cdot (1 - x_b) = (-1)^0 \prod_{j \in \{\}} x_j + (-1)^1 \prod_{j \in \{b\}} x_j = \sum_{J \subseteq \overline{F}} (-1)^{|J|} \cdot \prod_{j \in J} x_j.$$

Suppose that this is true when $|\overline{F}| = k - 1$. Now, consider $|\overline{F}| = k$ and let G be a set such that $G \subseteq \overline{F}$ and $|G| = k - 1$, i.e., $G = \overline{F} \setminus \{a\}$ for an element $a \in \overline{F}$.
We have,

$$\prod_{j \in \overline{F}} (1 - x_j) = \prod_{j \in G} (1 - x_j) \cdot \prod_{j \in \{a\}} (1 - x_j) = \left(\prod_{j \in G} (1 - x_j) \right) \cdot (1 - x_a).$$

By the induction hypothesis,

$$= \left(\sum_{J \subseteq G} (-1)^{|J|} \cdot \prod_{j \in J} x_j) \right) \cdot (1 - x_a) =$$

$$= \sum_{J \subseteq G} (-1)^{|J|} \cdot \prod_{j \in J} x_j - \sum_{J \subseteq G} (-1)^{|J|} \cdot x_a \prod_{j \in J} x_j =$$

$$= \sum_{J \subseteq G} (-1)^{|J|} \cdot \prod_{j \in J} x_j + \sum_{J \subseteq G} (-1)^{|J \cup \{a\}|} \cdot \prod_{j \in \{a\} \cup J} x_j.$$

Note that in the left sum J can be every subset of $\overline{F}$ that does not contain $\{a\}$, and in the right sum J can be every subset of $\overline{F}$ that contains $\{a\}$. Therefore, we have,

$$\sum_{J \subseteq G} (-1)^{|J|} \cdot \prod_{j \in J} x_j + \sum_{J \subseteq G} (-1)^{|J \cup \{a\}|} \cdot \prod_{j \in \{a\} \cup J} x_j = \sum_{J \subseteq \overline{F}} (-1)^{|J|} \cdot \prod_{j \in J} x_j$$

□

Now that we know the form of any monomial of an idempotent of a certain orthogonal basis, we can begin to prove the above claims.

For the first claim, consider,

$$A_F = \sum_{T \subseteq \{n+1, \ldots, r\}} e'_{F \cup T},$$

for $F \subseteq \{1, \ldots, n\}$, where e' denotes elements from the orthogonal basis of S_r.

Our intention is to show that $A_F = e_F$, where e_F is an element from the orthogonal basis of S_n, noting that A_F is a sum of 2^{r-n} parcels. Firstly, let $F' \subseteq \{1, \ldots, n\} \setminus F$ and $T \subseteq \{n+1, \ldots, r\}$. Suppose m is a monomial of A_F, such that m is not a monomial of e_F, i.e., m is of the form,

$$\prod_{i \in F} x_i \cdot \prod_{j \in F' \cup T} x_j,$$

where $T \neq \emptyset$. We can then rearrange m as follows,

$$\prod_{i \in F} x_i \cdot \prod_{j \in T} x_j \cdot \prod_{k \in F'} x_k,$$

and, for every $T' \subseteq T$, we have,

$$\prod_{i \in F} x_i \cdot \prod_{j \in T'} x_j \cdot \prod_{k \in T \setminus T'} x_k \cdot \prod_{l \in F'} x_l =$$

$$= \prod_{i \in F \cup T'} x_i \cdot \prod_{j \in F' \cup (T \setminus T')} x_j.$$

Firstly note that we consider the complement sets relative to $\{1, ..., r\}$. That said, if $F' \cup (T \setminus T') \subseteq \overline{F \cup T'}$ then m is a monomial of $e'_{F \cup T'}$, for every $T' \subseteq T$.

Proposition 4.4. *Let $F' \subseteq \{1, ..., n\} \setminus F$, $T \subseteq \{n+1, ..., r\}$ and $T' \subseteq T$, then*

$$F' \cup (T \setminus T') \subseteq \overline{F \cup T'}.$$

Proof. First, note that $\overline{F \cup T'} = \overline{F} \cap \overline{T'}$. Let $x \in F' \cup (T \setminus T')$, then

$$x \in F' \text{ or } x \in T \setminus T'.$$

Suppose $x \in F'$, then it is obvious that $x \in \overline{F}$ and since $T' \cap F' = \emptyset$ we have that $x \in \overline{T'}$.
On the other hand, suppose $x \in T \setminus T'$, i.e., x is an element of T and not an element of T', then it is obvious that $x \in \overline{T'}$, and since $F \cap T = \emptyset$ we have that $x \in \overline{F}$. □

We now know that the coefficient of a monomial m of $e'_{F \cup T}$, with $T \neq \emptyset$, in A_F, is given by the sum of its coefficients in each $e'_{F \cup T'}$, where T' goes through all the subsets of T. Let $|T| = k$ and $|F'| = l$, by Proposition 4.3 we know that the coefficient of m in each $e'_{F \cup T'}$ is given by

$$(-1)^{|F' \cup (T \setminus T')|} = (-1)^{l+k-|T'|}.$$

Noting that there are $\binom{a}{b}$ subsets of size b in a set of size a, which means that there are $\binom{k}{s}$ subsets of T with size s. Concluding that the coefficient of m in A_F is equal to

$$\sum_{i=0}^{k} (-1)^{l+k-i} \binom{k}{i}.$$

This expression is equal to 0 for any values of l and k, so we come to the conclusion that any term of A_F that is not a term of e_F has coefficient zero. On the other hand, it is easy to see that every term of e_F is a term of A_F, therefore $A_F = e_F$. Since the coordinates of e_F in the orthogonal basis of S_r are unique, the first claim is proved.

To prove the second claim, lets first prove that any x_i, for $n+1 \leq i \leq r$ has exactly 2^{r-1} non-null coordinates in the orthogonal basis of S_r.

We can prove this with a similar argument to the one used for the first claim. Let

$$B = \sum_{T \subseteq \overline{\{i\}}} e'_{\{i\} \cup T}.$$

Suppose that there is a monomial m in B different from x_i, but that contains x_i, i.e.,

$$m = x_i \cdot \prod_{j \in T} x_j,$$

for $T \subseteq \overline{\{i\}}$ such that $T \neq \emptyset$. Then, for any $T' \subseteq T$, we can rearrange m as follows

$$m = x_i \cdot \prod_{j \in T'} x_j \cdot \prod_{k \in T \setminus T'} x_k,$$

and since it is obvious that $T \setminus T' \subseteq \overline{\{i\}} \cap \overline{T'}$, we have that m is a monomial of $e'_{\{i\} \cup T'}$, for every $T' \subseteq T$. Following the argument used in the first claim, we have that the coefficient of m in B is equal to

$$\sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i},$$

which is equal to 0. Therefore, $B = x_i$.

To prove the claim, let e_F be an arbitrary element of the orthogonal basis of S_n, then, by the first claim we know that

$$A = \bigcup_{T \subseteq \{n+1,\ldots,r\}} \{F \cup T\},$$

represents the set of its non-null coordinates in the orthogonal basis of S_r. Now consider the following set

$$A' = \bigcup_{T \subseteq \{n+1,\ldots,r\} \setminus \{i\}} \{F \cup \{i\} \cup T\}.$$

for some $i \in \{n+1, \ldots, r\}$. We can easily see that $A' \subseteq A$ and A' is also a subset of the set of non-null coordinates of x_i, for $n+1 \leq i \leq r$. Therefore, A' represents 2^{r-n-1} common non-null coordinates between e_F and some x_i. To conclude the proof, we just have to note that if there were more common coordinates between an e_F and x_i, this would be a contradiction with the above result regarding the number of total non-null coordinates of x_i.

Now that we have proved the two previous claims, we can move onto the main proof of this subsection.

Proposition 4.5. *Let I be the ideal of S_r generated by $\langle x_{n+1} - w_n, ..., x_k - w_{k-1} \rangle$, where w_m are random idempotents of S_m, for $m \in \{n, ..., r-1\}$. Then, these generators have exactly 2^{r-k} mutual null coordinates in the orthogonal basis of S_r.*

Proof. First, notice that any idempotent of S_n can be written as a linear combination of elements from its own orthogonal basis and each of the elements from the orthogonal basis of S_n can be written as a linear combination of elements from the orthogonal basis of S_r as follows:

$$w_n = \sum_{F \subseteq \{1,...,n\}} \alpha_F e_F,$$

where $\alpha_F \in \{0, 1\}$. By the first claim, we know that

$$\alpha_F e_F = \alpha_F \sum_{G \subseteq \{n+1,...,r\}} e'_{F \cup G},$$

where $e'_{F \cup G}$ are elements from the orthogonal basis of S_r. We can easily conclude that for any two distinct idempotents, e_{F_1} and e_{F_2}, their set of non null coordinates in the orthogonal basis of S_r is disjoint, which means that every coordinate correspondent to the elements $e'_{F \cup G}$, has one and only one coefficient, α_F.

By our second claim, we also know that for this set of coordinates, half of its elements are also non null coordinates of x_{n+1}. Notice that every non null coordinate of any x_i has coefficient 1 (since every x_i is an idempotent). Then, in $x_{n+1} - w_n$, half of the set of coordinates of an idempotent e_F of S_n has coefficient $-\alpha_F$ while the other half has coefficient $1 - \alpha_F$, which means that independently of the value of α_F, half of these coordinates in $x_{n+1} - w_n$ will be equal to 0. Across 2^n idempotents, we would have exactly $2^{r-n-1}2^n = 2^{r-1}$ null coordinates.

Without loss of generality, suppose that $\alpha_F = 0$. Then, for the set of non null coordinates of e_F, the ones that are not mutual with x_{n+1} will be null coordinates. Recall from our second claim that these coordinates correspond to

$$A = \bigcup_{T \subseteq \{n+2,...,r\}} \{F \cup T\},$$

which are also the set of coordinates of an idempotent from S_{n+1} which takes part in the linear combination used to generate w_{n+1}, with a coefficient β_j. Similarly to what happens with $x_{n+1} - w_n$, the generator $x_{n+2} - w_{n+1}$ also splits A in two sets of coordinates:

- A_1 - Mutual coordinates with x_{n+2};
- $A_2 = A \setminus A_1$;

where, for any β_j, the null coordinates of $x_{n+2} - w_{n+1}$ are either the coordinates in A_1 or A_2, which will also represent the subset of every mutual null coordinates between $x_{n+1} - w_n$ and $x_{n+2} - w_{n+1}$, since $A_1, A_2 \subseteq A$. This subset will have exactly 2^{r-n-2} elements, therefore, across 2^n possible e_F, we have a total of 2^{r-2} mutual null coordinates, i.e., $|A|/2$.

The rest of the proof will be done by induction over the size of the set of generators of our ideal I.

For a set of size 2 we saw above that it is true. Suppose that it is true for a set of size $k-1$. Now consider a set of k generators. By the inductive hypothesis among the first $k-1$ generators there are 2^{r-k+1} mutual null coordinates. On the other hand, using the same argument as above for the $k-1$-*th* and the *k-th* generator we can easily conclude that in this set of k generators we would have $2^{r-k+1}/2 = 2^{r-k}$ mutual null coordinates among them. □

Since in our scheme the set of generators has always size $r-n$, we conclude that there will always be 2^n mutual null coordinates between the generators in this set, i.e., every element of our ideal I would have at least 2^n null coordinates.

4.2.3.1 Enhancements

Even though the authors of the original scheme did not explicitly explore this previous result, they did, however, briefly present a couple enhancements that the user can perform in order to avoid this problem. Thanks to the process we had to go through to be able to prove the above proposition, we are now capable of doing a more complete analysis on the enhancements presented.
These enhancements are the following:

1. Change the form of the generators of our ideal I used for encryptions of 0;
2. Apply a linear transformation on the orthogonal basis of S_r.

The first enhancement consists of changing the form of the generators of our ideal from $x_m - w_{m-1}$ to $x_{i_1}...x_{i_k}x_m - w_{m-1}$ where all $i_j < m$. The protocol is similar, but we have to take into account that everything we did before with x_m, we now have to do with the new

monomial, in particular, the order in the substitution step is defined similarly to what we have seen before, i.e., starting with the monomial that involves x_r and ending with the one that involves s_{n+1}. We can easily see that decryption is still unique.

Lets now check the maximum and minimum number of mutual null coordinates that one can obtain when using this enhancement.

Let $H = \{i_1, ..., i_k\}$ and $M_t = x_t x_{i_1} ... x_{i_k}$. By the previous subsection, we know that the non-null coordinates of M_{n+1} are given by

$$A = \bigcup_{T \subseteq \overline{H \cup \{n+1\}}} \{\{n+1\} \cup H \cup T\}.$$

Notice that given an arbitrary idempotent from the orthogonal basis of S_n, e_F, it has mutual non-null coordinates with M_{n+1} if and only if $H \subseteq F$. There are 2^{n-k} subsets $F \subseteq \{1, ..., n\}$ where $H \subseteq F$, i.e., this is the number of elements in the orthogonal basis of S_n that have mutual non-null coordinates with M_{n+1}.

We can then split the coordinates of $M_{n+1} - w_n$ into two sets:

(a) Union of the sets of coordinates of e_F, for every F such that $H \nsubseteq F$,

(b) Union of the sets of coordinates of e_F, for every F such that $H \subseteq F$.

Recall that,

$$w_n = \sum_{F \subseteq \{1,...,n\}} \alpha_F e_F,$$

where $\alpha_F \in \{0, 1\}$. Which means that the set of coordinates in (a) would have coefficients $-\alpha_F$, for every F where $H \nsubseteq F$. To explore the worst case scenario, i.e., the case where we would get the most amount of mutual null coordinates possible, we want every coordinate in (a) to have coefficient 0, meaning that we have $(2^n - 2^{n-k}) \cdot 2^{r-n}$ null coordinates already, so, the total number of null coordinates in the first generator is equal to

$$(2^n - 2^{n-k}) \cdot 2^{r-n} + 2^{n-k} 2^{r-n-1} = 2^r \cdot (1 - 2^{-k-1}),$$

since the set of coordinates in (b) have the same behaviour as the cases explored in the proof of Proposition 4.5, which means that no matter what, half of these coordinates have coefficient 0 in $M_{n+1} - w_n$. We can easily see that the number of null coordinates increases as k increases. Therefore, we can also conclude that the maximum number of mutual null coordinates across all generators is at most equal to the number of null coordinates in the first generator, since every following generator, in the worst case scenario, has a greater number of null coordinates than the first.

Let us then consider $k = n$ in order to guarantee the most amount of null coordinates in the first generator, then $M_{n+1} = x_{n+1}x_n...x_1$. Our goal now is to prove that the maximum number of mutual null coordinates is exactly the maximum number of null coordinates in the first generator. For $k = n$, we have $H = \{1, ..., n\}$, i.e., there is only one idempotent, e_F, that has mutual coordinates with M_{n+1}, this being when $F = \{1, ..., n\}$. We know that the set of non-null coordinates of $e_{\{1,...,n\}}$ is

$$B = \bigcup_{T \subseteq \{n+1,...,r\}} \{\{1, ..., n\} \cup T\}.$$

However, these are not all common with M_{n+1}. Splitting B into the following two sets of coordinates:

$$B_1 = \bigcup_{T \subseteq \{n+2,...,r\}} \{\{1, ..., n\} \cup T\},$$

and

$$B_2 = \bigcup_{T \subseteq \{n+2,...,r\}} \{\{1, ..., n, n+1\} \cup T\},$$

we can easily see that B_2 is the set of non-null coordinates of M_{n+1}, which means that it is also the set of mutual non null coordinates between M_{n+1} and $e_{\{1,...,n\}}$. Then, in the first generator, the coordinates in B_1 will have coefficient $-\alpha_{\{1,...,n\}}$, while the coordinates in B_2 will have coefficient $1 - \alpha_{\{1,...,n\}}$, where $\alpha_{\{1,...,n\}} \in \{0, 1\}$.

Suppose that $\alpha_{\{1,...,n\}} = 0$. Then, in the first generator, the coordinates in B_1 are null, which means that the set of non-null coordinates in the first generator is exactly B_2.

To conclude our argument we want to show that it is possible that every non-null coordinate in the remaining generators is in B_2. Suppose they are all of the form $M_l - w_{l-1}$, where

$$M_l = x_l x_{l-1}...x_1, \text{ for } n + 1 < l \leq r.$$

Then, with a similar argument used for the first generator, i.e., considering every coefficient in the linear combination of w_{l-1} correspondent to the idempotents that do not share non-null coordinates with M_l, to be null, we know that the non-null coordinates of a generator $M_l - w_{l-1}$, belong to the set,

$$C_l = \bigcup_{T \subseteq \{l+1,...,r\}} \{\{1, ..., n, n+1, ..., l\} \cup T\}.$$

Finally, since $C_l \subseteq B_2$ for every l we can then conclude that it is indeed possible for the total amount of mutual null coordinates among the ideal generators to be equal to the maximum amount of null coordinates in the first generator, i.e., $2^r \cdot (1 - 2^{-n-1})$.

With a similar but simpler argument, we can show that in the best case scenario there are no mutual null coordinates. Consider the first generator as we did above, however, in this case, instead of choosing every coordinate in (a) to have coefficient 0, we choose it to be 1. Consider also the same sets of coordinates B_1 and B_2 and recall that in the first generator their coordinates have coefficients $-\alpha_{\{1,...,n\}}$ and $1 - \alpha_{\{1,...,n\}}$, respectively.

Once again, suppose $\alpha_{\{1,...,n\}} = 0$, then the set of null coordinates in the first generator is exactly B_1. Now, to conclude our argument, we just have to show that it is possible the second generator not to have any null coordinates in B_1, meaning that there would not be any mutual null coordinates across all generators, independently of the remaining.

Consider the second generator as above and let, however, every coefficient in its set (a) be equal to 1 (instead of 0). One can easily see that the coordinates in B_1 are also in (a). That said, we conclude that in the best case scenario, the minimum number of mutual null coordinates is zero.

The second enhancement consists in applying a random linear transformation after applying the random permutation on the orthogonal basis.

Before we define this linear transformation, consider the following proposition.

Proposition 4.6. *Let $w_j = \sum_{i=1}^{n} a_{i,j} b_i$, where $B = (b_1, ..., b_n)$ is a basis of a vector space V and $A = (a_{i,j})_{i,j}$. Then, $B' = (w_1, ..., w_n)$ is a basis of V if and only if A is invertible.*

Proof. Suppose B' is a basis of V, then w_1,...,w_n are linearly independent, i.e.,

$$\sum_{j=1}^{n} \alpha_j w_j = 0,$$

if and only if $\alpha_1 = ... = \alpha_n = 0$. It is also true that

$$\sum_{j=1}^{n} \alpha_j w_j = \sum_{j=1}^{n} \left(\alpha_j \sum_{i=1}^{n} a_{i,j} b_i \right) = \sum_{j=1}^{n} \left(b_j \sum_{i=1}^{n} \alpha_i a_{j,i} \right).$$

Since B is a basis of V, this latter sum is equal to 0 if and only if $\sum_{i=1}^{n} \alpha_i a_{j,i} = 0$ for every $i \in \{1, ..., n\}$. However, the sum is equal to 0 if and only if $\alpha_1 = ... = \alpha_n = 0$, therefore for every j, $a_{j,i}$ are linearly independent. This is equivalent to say that $(a_{j,i})_j$ (the columns

of A) are linearly independent. Finally, we use the fact that the columns of A are linearly independent if and only if A is invertible to conclude our proof. □

The linear transformation is defined as follows

$$f_j = e_j + \sum_{i<j} c_i e_i, \tag{4.1}$$

where $c_i \in \{0, 1\}$ and e_i are the elements of the orthogonal basis of S_r. By the above proposition, we know that f_j form a new basis if and only if the matrix A, which coefficients we obtain from the above expression, is invertible.

In this case, the matrix A obtained is the following,

$$A = \begin{pmatrix} 1 & c_1 & c_1 & \dots & c_1 \\ 0 & 1 & c_2 & \dots & c_2 \\ 0 & 0 & 1 & \dots & c_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}.$$

We can easily see that A is invertible, since it is an upper triangular matrix and there are no null entries in its main diagonal, therefore we conclude that f_j form a new basis of S_r.

This new basis is no longer orthogonal, which means that multiplications will not be as effective. On the other hand, note that, if we compute $f_a f_a$, we obtain

$$f_a f_a = \left(e_a + \sum_{i<a} c_i e_i \right) \cdot \left(e_a + \sum_{i<a} c_i e_i \right) =$$

$$= e_a + 2e_a \sum_{i<a} c_i e_i + \sum_{i<a} c_i e_i = e_a + \sum_{i<a} c_i e_i = f_a.$$

Then every f_j is an idempotent, which means that f_j form a new idempotent basis of S_r.

With this enhancement it is possible to give an example of the best case scenario, however it is not trivial to analyze the worst case, since it depends on the shape of the generators in the orthogonal basis as well as on the parameter p.

In the best case it is possible to have no mutual null coordinates as we will show with the following example.

Example 4.1. Considering our parameters as $n = 2$, $p = 5$ and $r = 4$, with the aid of our software we obtain a pair of generators of the ideal I, where one of them is the following

$$x_3 + x_1 x_2 - 1 \to (-1, -1, -1, 0, -1, 0, 0, -1, 0, -1, 0, 1, 0, 0, 0, 1),$$

then, considering $c_3, c_{10} = 1$ and the remaining to be null, we obtain (applying 4.1),

$$f_j = -1 \text{ for } j \in \{1, 2, 3, 4, 6, 7, 9, 12, 16\},$$

and,

$$f_j = -2 \text{ for } j \in \{5, 8, 10, 11, 13, 14, 15\}.$$

Therefore, we do not have any mutual null coordinates, in this basis, among our generators.

On the other hand, for the worst case scenario, there is not much we can conclude, besides the fact that there is a possibility of obtaining more than 2^n mutual null coordinates.

Example 4.2. Consider the example above,but now assume that $c_i = 0$ for $i \in \{1, ..., 16\}$. This means that the first generator would have null coefficients in f_j for $j \in \{4, 6, 7, 9, 11, 13, 14, 15\}$. The second generator of the ideal I is

$$x_4 + 2x_1x_2x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_1 + x_3 + x_4 - 1 \rightarrow$$
$$(-1, 0, -1, 0, 0, -1, 0, 1, -1, 0, 1, 0, 0, 1, 0, 1).$$

Then, considering $c_8, c_9 = 1$ and the remaining equal to 0, we obtain,

$$f_j = 0 \text{ for } j \in \{2, 4, 5, 7, 9, 10, 12, 13, 15\},$$

which means that among these generators there are 5 mutual null coordinates in this new basis. Recall that in the orthogonal basis of S_r, without any enhancements, there were only $2^n = 4$.

This is, however, not a complete analysis, since we do not know how likely to happen are each of these scenarios, in either enhancement. So, in order to have a better idea of what to expect when using these enhancements, a program was developed (available at Appendix A.3) that tests key-generation with each of these enhancements with the purpose of checking how many mutual null coordinates one would obtain.

In Table 4.1, are presented the results of our main experiments.

The first conclusion we can take from these results is that the second enhancement is much more effective than the first one. In fact, the first enhancement strategy rarely shows improvements, and if it does, it is a very slight difference to the number of mutual null coordinates we obtain without any enhancement.

We can also observe that with the second enhancement, the average number of mutual null coordinates decreases when we increase the number of generators involved. The

variation of the other two parameters, n and p, does not show obvious patterns, however we can notice that when n increases. we begin to see that changing the value of p has more effect in the results, specially between $p = 11$ and the other values of p.

Table 4.1: Results of Enhancement Testing

			N° of mutual null coordinates (mean across 100 runs)		
n	p	$r-n$	**Enhancement 1 (Generators)**	**Enhancement 2 (New Basis)**	**Standard**
3	11	1	8.14	3.63	8
		2	7.87	1.22	
		3	7.89	0.26	
	10007	1	7.83	3.69	
		2	7.88	1.36	
		3	7.91	0.24	
	1073741827	1	7.91	4.19	
		2	8.23	1.31	
		3	7.77	0.21	
5	11	1	33.03	7.95	32
		2	32.10	1.03	
		3	33.41	0.31	
	10007	1	32.70	7.06	
		2	32.95	1.05	
		3	33.47	0.30	
	1073741827	1	31.74	6.84	
		2	31.92	0.96	
		3	32.48	0.37	
7	11	1	129.11	21.32	128
		2	129.87	4.31	
		3	127.72	1.05	
	10007	1	127.76	6.93	
		2	129.91	0.66	
		3	128.57	0.16	
	1073741827	1	129.50	6.90	
		2	129.77	0.42	
		3	128.53	0.19	

Chapter 5

Private Search

A possible application for the encryption scheme presented is what the authors of the original scheme call *private search*, i.e., the ability to verify if a certain element belongs to an encrypted database. In order to perform private search, we need to introduce a new step in the general process, that is going to be performed by the database keeper (e.g. the cloud).

What makes this application a non-trivial problem is the fact that this scheme has probabilistic encryption, which means that every time Alice (the owner of the database of plaintexts) encrypts the same x, $E(x)$ is likely to be different, therefore Carl (the database keeper) has to do something more than just trying to match our $E(x)$ to the encrypted elements on the database. What Carl actually has to do is to take advantage of the fact that $E(x) - E(y)$ is equal to $E(x - y)$ modulo I (same ideal used when encrypting) and then he should compute the following product:

$$P(x) = \prod_{E(y) \in E(D)} (E(x) - E(y)) = \prod_{E(y) \in E(D)} E(x - y), \tag{5.1}$$

where $E(D)$ represents the encrypted database. Since the encryption function is also an homomorphism for the usual product, we have that $P(x) = E(\prod_{E(y) \in E(D)}(x - y))$ modulo I, and this allows Alice to decrypt $P(x)$ in order to obtain $\prod_{E(y) \in E(D)}(x - y))$, meaning that, in general, $E(x)$ is an element of the encrypted database if the result of Alice's decryption is 0. However, this might not be the case every time, i.e., there is a possibility of obtaining a false positive as we will explain in Subsection 5.1.2).

Finally, the authors of the scheme present the idea of extending this concept to a third-party to allow it to perform private search in our encrypted database. In the next section, we will discuss this further, starting with a brief explanation of the adjustments that we

need to make in the encryption process, and then moving to a description of our design thinking to build a program, with the goal of testing a third-party private search scenario.

5.1 Third-Party Private Search

The general idea of this application is to allow for a third-party (Bob) to check, privately, if the encryption of certain elements belongs to the encrypted database.

Even though this scheme is private-key, some information has to be shared with Bob, so he is able to perform private search. The first thing that Bob has to know is how to embed his real life plaintext into an element of S_n. Consequently, Alice has to share the way to do this with Bob. Bob also has to know how to encrypt an element of S_n such that Alice (or anyone for that matter) is not capable of knowing which element Bob is planing to use private search on.

Then, Alice has to publish $E(D)$ using the orthogonal basis of S_{r+k} for some $k \geq 1$, noting that these additional k generators would allow Bob to encrypt his plaintexts (as an elements of S_n).

In the next subsection, we explain, in more detail, the protocol of the encryption process within the context of third-party private search.

5.1.1 Encryption Process

The overall encryption process is similar to what we have seen on Chapter 3. However, in this context, we also need to have into account that Bob needs to perform encryption and decryption. Therefore, we need to generate a private-key for Bob in the key-generation step.

5.1.1.1 Key-Generation

As always, the first step is key-generation, and, in this context, we generate a key for Bob and one for Alice. In order to generate a private-key for Bob we will need a new input, k. Then, we proceed to compute the generators of Alice's ideal I, as well as the generators of an ideal J that will be used by Bob, similarly to how Alice uses I. Note also that J is an ideal of S_{r+k} generated by $\langle x_{r+1} - w_n^{(1)}, ..., x_{r+k} - w_n^{(k)} \rangle$, where $w_n^{(i)}$, for $i \in \{1, ..., k\}$, are random idempotents of S_n.

Another new step within the key-generation process is the encryption of Alice's database, that consists of encrypting each element with Alice's encryption key.

5.1.1.2 Encryption

As mentioned in the beginning of this section, this step as well as decryption, within the context of third-party private search, have a slight difference compared to the standard encryption process.

We assume that Bob is already aware on how to do the embedding from the original database to S_n. Let x be the element of S_n that Bob wants to perform private search on. Then, the first step is to encrypt x, and Bob does this as follows,

$$E_B(x) = x + g,$$

where g is a random element of Bob's ideal J. Bob, then, sends $E_B(x)$ to Alice, expressed in the standard basis of S_{r+k}, since the conversion from the standard basis to the orthogonal basis is part of Alice's private-key.

After receiving $E_B(x)$ from Bob, Alice encrypts this element, i.e., Alice computes

$$E_A\left(E_B(x)\right) = E_B(x) + h,$$

where h is a random element of the ideal I.

Finally, before Alice sends this to Carl, she converts this element to the orthogonal basis of S_{r+k}.

5.1.1.3 Cloud Computations

As was refereed in the beginning of this chapter, Carl is going to be responsible for returning the product $P(x)$ to Alice. However, in the context of third-party private search, $P(x)$ is slightly different, since Alice receives $P(x)$ from Carl, she will not be able to know if the product is 0 or not, due to the fact the she is oblivious about the ideal J, which is part of Bob's private-key.

That said, Carl returns to Alice the following product

$$P(x) = \prod_{E_A(y) \in E_A(D)} \left(E_A\left(E_B(x)\right) - E_A(y)\right),$$

then, modulo the ideal $I + J$, we verify the following

$$P(x) = \prod_{y \in D} \left(E_A \left(E_B(x) - y \right) \right) = E_A \left(\prod_{y \in D} \left(E_B(x) - y \right) \right).$$

Note that the fact that every element is the orthogonal basis of S_{r+k} makes the computation of such product way more effective, which was the goal of the orthogonal basis all along. Since when performing private search, the number of multiplications is going to be equal to the number of elements in the database, for a large database, the computation of this product may be computationally very expensive, even with the aid of the orthogonal basis properties. In Subsection 5.1.3, this will be seen in more detail.

5.1.1.4 Decryption

Finally we have the decryption part,which includes two decryptions processes, one that has to be done by Alice and the other one by Bob.

When receiving $P(x)$ from Carl, Alice performs her decryption as usual, obtaining

$$Q(x) = \prod_{y \in D} \left(E_B(x) - y \right).$$

Alice then sends $Q(x)$ to Bob, that performs decryption with his own private-key, by the standard process, i.e., by substituting every x_m by $w_n^{(m-r)}$, starting with $m = r + k$ and ending in $m = r + 1$.

Since $E_B(x) = x + g$, and neither x or y have any terms involving x_m for $m \in \{r + 1, ..., r + k\}$, they are not affected by Bob's decryption. On the other hand, $g \in J$, and since the purpose of Bob's decryption is to annihilate the ideal J, we can then conclude that the results of his decryption would be

$$\prod_{y \in D} (x - y).$$

We mentioned in the beginning of this chapter that usually, $E(x)$ is an element of the encrypted database if the returned product is 0, noting that there is a slight possibility of false positives. In the next subsection, we explain how can these false positives happen as well as how one can previously choose the parameters in order to have the probability of a false positive as low as possible.

5.1.2 False Positives

If our plaintexts are elements of a ring with zero divisors, then there are non-zero elements a, b such that $ab = 0$. In our scheme, plaintexts are elements of S_n which is isomorphic to a direct sum of 2^n copies of $\mathbb{Z}_p$. For p prime, we know that $\mathbb{Z}_p$ has no zero divisors, meaning that for any $a, b \in \mathbb{Z}_p$, if $ab = 0$ then $a = 0$ or $b = 0$. This means that for any two non-zero elements u_1, u_2 in S_n, to have $u_1 u_2 = 0$, one has to have 0 in every coordinate in the orthogonal basis, either in u_1 or u_2 (or both). That is, for any two elements of S_n

$$u_1 = (a_1, ..., a_{2^n})$$
$$u_2 = (b_1, ..., b_{2^n}),$$

if $u_1 u_2 = 0$, then for every $i \in \{1, ..., 2^n\}$, one has

$$a_i b_i = 0,$$

i.e., $a_i = 0$ or $b_i = 0$.

However, the product that Carl computes involves N elements, which is the size of our database. Therefore, to obtain a false positive, we need to have 0 in every coordinate of at least one of the N elements involved in the product.

To be able to give an estimate probability, we consider that $\mathbb{Z}_p$ has a uniform distribution, therefore the probability of having 0 in a certain coordinate would be equal to $\frac{1}{p}$, which allows us to compute an upper bound for the probability of a false positive as follows.

The probability of having a certain coordinate null in at least one of the elements in the product is equal to

$$1 - \left(\frac{p-1}{p}\right)^N, \tag{5.2}$$

where $\left(\frac{p-1}{p}\right)^N$ is the probability of not existing any elements with 0 in a given coordinate. We can rewrite expression 5.2 as

$$\frac{p^N - (p-1)^N}{p^N}, \tag{5.3}$$

and, since $(p-1)^N$ is equal to

$$\sum_{i=0}^{N} \binom{N}{i} p^{N-i}(-1)^i = p^N - Np^{N-1} + \sum_{i=2}^{N} \binom{N}{i} p^{N-i}(-1)^{i-1},$$

we can substitute in our expression to get

$$\frac{Np^{N-1} + \sum_{i=2}^{N} \binom{N}{i} p^{N-i}(-1)^{i-1}}{p^N}. \tag{5.4}$$

Let us now show that Np^{N-1} is an upper bound of the numerator of the expression above, i.e.,

$$\sum_{i=2}^{N} \binom{N}{i} p^{N-i}(-1)^{i-1} \leq 0,$$

which is equivalent to prove that

$$\sum_{i=2}^{N} \binom{N}{i} p^{N-i}(-1)^{i} \geq 0.$$

Simplifying this sum, we get

$$\sum_{i=2}^{N} \binom{N}{i} p^{N-i}(-1)^{i} = (p-1)^N - p^N + Np^{N-1} = (p-1)^N - p^{N-1}(p-N).$$

That said, we can simply prove the following,

$$\left(1 - \frac{1}{p}\right)^{N-1} \geq \frac{p-N}{p-1}. \tag{5.5}$$

Let us prove this by by induction over N. For $N = 1$, we have

$$\left(1 - \frac{1}{p}\right)^{0} = 1 \geq 1 = \frac{p-1}{p-1}.$$

Suppose this is true for $N = k$, where $k \in \mathbb{N}$, i.e.,

$$\left(1 - \frac{1}{p}\right)^{k-1} \geq \frac{p-k}{p-1}.$$

Now, let us prove that it holds for $N = k+1$. We have,

$$\begin{aligned}
\left(1 - \frac{1}{p}\right)^{k} &= \left(1 - \frac{1}{p}\right) \cdot \left(1 - \frac{1}{p}\right)^{k-1} \\
&\geq \left(1 - \frac{1}{p}\right) \cdot \left(\frac{p-k}{p-1}\right) \\
&= \frac{p-k}{p} = 1 - \frac{k}{p} \\
&\geq 1 - \frac{k}{p-1} = \frac{p-k-1}{p-1},
\end{aligned}$$

Where the first inequality comes from our induction hypothesis. We concluded that Np^{N-1}

is an upper bound for the numerator in expression 5.4. Then, we obtain the following upper bound for the probability

$$\frac{Np^{N-1}}{p^N} = \frac{N}{p}. \tag{5.6}$$

This is, however, not enough to have a false positive, since this needs to happen across all coordinates. Therefore, an upper bound for the probability of a false positive is $\left(\frac{N}{p}\right)^{2^n}$.

Now that we have an estimate for the probability of a false positive, that depends on the size of the database and on the value of p, we are able to choose these parameters such that we minimize as much as possible the probability of a false positive. In Section 5.2, we share an example where we illustrate a poor choice of such parameters leading to false positives.

Notice that with an adequate choice of parameters, we did not obtain any false positives during our tests.

5.1.3 Code Walkthrough

In this section, we share the thinking process we used to design our program as well as some of the results obtained.

We start with a few observations regarding the implementation, then, we move to a more detailed walk-through of our code, not only discussing its final version, but also explaining the optimizations that were made along the way.

5.1.3.1 Design Thinking

The main goal of our implementation was to build a program that allowed us to replicate a third-party private search scenario in order to analyze the following two aspects:

- Reliability of this process regarding the amount of false positives;
- Processing time required to perform private search.

Since we were the only users of this program, we came to the conclusion that applying a permutation on the orthogonal basis is not a relevant step. We also omitted the embedding part, since we already mentioned that Bob, in order to perform third-party private search, has to have knowledge of the embedding. Therefore, our input is the element of S_n that we want to do private search on the encrypted database.

5.1.3.2 Key-Generation and Database

Recall that the function `prik`, as mentioned before in Subsection 3.5, was used for key-generation when using the standard encryption process. Now, we introduce the function `Tprik` that is used for key-generation within the context of third-party private search, and which is available at Appendix A.1.

This function has suffered several adjustments throughout this work. First, we attempted to replicate the encryption process step by step, which meant that we worked with polynomials in every step except during Carl's computations. However, this proved to be very time-consuming because of our computational resources. Then, the code was improved to take advantage of the efficiency on multiplying elements in the orthogonal basis. This was used to improve almost every step in the whole encryption process. Therefore, in the final version of our code, shared in Appendix A.1, we have an optimized function `Tprik` which, given inputs n, p, r and k, returns the following:

- `e`: Orthogonal basis of S_{r+k}

  To generate this basis we apply the expression introduced in Section 3.4, and then store the results in a list;

- `m`: Monomials in S_{r+k}

  Given the generators of S_{r+k}, i.e., $\{1, x_0, ..., x_{r+k-1}\}$, we create a list with the element 1 and then add to this list the product between the next generator and all elements already in the list, this results in something similar to

  $$[1, x_0, x_1, x_0x_1, x_2, x_0x_2, x_1x_2, x_0x_1x_2, ..., x_0x_1...x_{r+k-1}];$$

- `t`: Coordinates of the generators of S_{r+k} in the orthogonal basis of S_{r+k} (in vector form)

  We write all the generators of S_{r+k}, $\{1, x_0, ..., x_{r+k-1}\}$, in the orthogonal basis of this same ring;

- `E`: Coordinates of the monomials of S_{r+k} in the orthogonal basis of S_{r+k} (in vector form)

  In order to ease the conversion from the standard basis to the orthogonal basis, we use t and a similar process to the one used to create `m`. However, in this case, we begin with the vector $(1, ..., 1)$, of length 2^{r+k};

- `Edb`: The first 2^n elements of `E`;

  Since `t` is built similarly to `m`, we guarantee that the first 2^n of `E` are the monomials of S_n. We then take advantage of this to ease the process of building the database.

- `rr`: Random idempotents for Alice

  In the first versions of the code, in order to obtain an idempotent of S_n, we used to generate a random element of this ring and then check if it was an idempotent or not. However, we now use the fact that any idempotent of S_n can be written as a linear combination of elements from the orthogonal basis of S_n with coefficients 0 and 1. Therefore, in order to generate $r-n$ random idempotents, for Alice's encryption, from S_n, S_{n+1},...,S_{r-1}, respectively, we simply generate random vectors with coefficients 0 and 1 with length 2^n,2^{n+1},...,2^{r-1}. Then, we compute

  $$\sum_{j=0}^{len(a_m)} a_m[j] \cdot e_m[j], \tag{5.7}$$

  where a_m represents the random vector generated with length 2^m and e_m is the orthogonal basis of S_m. Finally, with the aid of `E`, we can easily convert the result of this sum to vector form.

- `rk`: Random idempotents for Bob

  Although Bob does not have knowledge of the orthogonal basis, we still use it in order to generate idempotents faster. However, in this case, we do not convert into vector form, i.e., the result remains a polynomial.

- `ac`: Generator of the ideal I used for Alice's encryptions

  This was one of the major differences when attempting to optimize our code. In the early stages we followed the protocol strictly, which means that to generate a random element of the ideal I we had to generate $r-n$ random elements of S_{r+k} and, then, compute a linear combination of these elements and the generators of I. However, when analyzing other issues, we came across a new point of view regarding all ideals of S_n, that allowed us to generate a random element of I much faster. We already showed a brief example of how we do this, but notice that, in our code, *ac* is a vector of 1's except in the coordinates that are mutually null across the generators of the ideal, meaning that all we have to do is generate a random element of S_{r+n+k} and then compute the product between that element and *ac* (the product defined in Section 3.4) and we obtain a random element of I.

Although we consider the construction of the database as well as its encryption to be part of the key-generation process, we decided to build separate functions to perform such tasks.

Our initial goal was to replicate the experiment mentioned by the authors of the original scheme, where they claim to have tested with a database that has 10^6 elements. However, along the way, we realised that we could not generate 10^6 random elements in a reasonable period of time. As a matter of fact, even 10^4 elements was already very time consuming, when using the suggested parameters by the authors of the original scheme. One step of our optimization process was to build the database in a way that its elements are already in the orthogonal basis, saving a great amount of time when compared to previous versions where we had to convert these elements to the orthogonal basis.

To encrypt the database we simply apply Alice's encryption method to every element in the database of plaintexts (explained in the following subsection).

In the pursue of an optimized code, one of the question discussed was whether it would be better to work with polynomials or converting every element to the orthogonal basis as soon as possible. A benefit from working with polynomials is how much easier it is to generate a random element of an ideal I, due to built-in functions of Sage. On the other hand, every time we need to perform big computations, working with polynomials was not the optimal option.

Comparative Analysis

We now describe the tests that were performed to compare the three main versions of our program over time. Consider that:

- **OB 1st** is our first version of the program considering the optimization related to the orthogonal basis;
- **Polynomial** refers to a version of our code where we worked with elements in polynomial form as much as possible;
- **OB Final** refers to the latest version of our code, which we share in Appendix A.1.

In our experiments we had measured the processing times for three different processes (key generation, database construction and database encryption) and for different sizes of the database (1, 1000, 10000 and 100000, whenever possible).

In Table 5.1, are presented the processing times of our first experiment, were we considered the following parameters:

$$n = 3, \quad p = 1031, \quad r = 6, \quad k = 1.$$

Table 5.1: First Comparison of Code Performance

	N° of elements	**OB 1st**	**Polynomial**	**OB Final**
Key Generation	——-	*0.39* s	*0.12 s*	*0.2 s*
DB Build	1	*0.0003 s*	*0.0007 s*	*0.0003 s*
	1000	*0.006 s*	*0.02 s*	*0.18 s*
	10000	*0.06 s*	*0.2 s*	*1.66 s*
	100000	*0.6 s*	*1.7 s*	*13.9 s*
DB Encryption	1	*0.023 s*	*0.02 s*	*0.003 s*
	1000	*15 s*	*15.3 s*	*2.68 s*
	10000	*143.25 s*	*129.82 s*	*26.5s*
	100000	*1463.28 s*	*1308.28 s*	*382.97 s*

Notice that the first version of the code, as well as the Polynomial one, take a longer time to encrypt when compared with OB Final. This is due to the fact that, somewhere within the encryption process, a conversion to the orthogonal basis has to be done, which is not the case in the final version of our code. Therefore, we were able to save a significant amount of time in the encryption process, even though there is a small increase in the processing time required to build the database.

In the second experiment, whose results are shown in Table 5.2, the parameters n and p were increased and a clear difference can now be seen between the performance of the different code versions. The parameters considered in this case were:

$$n = 5, \quad p = 1048583, \quad r = 8, \quad k = 1.$$

TABLE 5.2: Second Comparison of Code Performance

	Nº of elements	**OB 1st**	**Polynomial**	**OB Final**
Key Generation	—	*28.67 s*	*3.58 s*	*1.56 s*
DB Build	1	*0.0002 s*	*0.0001 s*	*0.0006 s*
	1000	*0.015 s*	*0.03 s*	*0.19 s*
	10000	*0.14 s*	*0.33 s*	*4.37 s*
DB Encryption	1	*0.10 s*	*0.18 s*	*0.016 s*
	1000	*85.15 s*	*181.04 s*	*9.52 s*
	10000	*866.29 s*	*1745.71 s*	*108.42 s*

Finally, in our last experiment, the parameters considered were the ones suggested by the authors of this scheme, which are:

$$n = 7, \quad p = 1073741827, \quad r = 10, \quad k = 1.$$

In Table 5.3, the results are shown for databases of size 1 and of size 1000. Due to limitations of time and computation power, we were not able perform the comparison for databases of sizes 10000 or 100000. However, the results obtained already show a huge difference between the processing times of the different versions of the code.

TABLE 5.3: Third Comparison of Code Performance

	Nº of elements	**OB 1st**	**Polynomial**	**OB Final**
Key Generation	—	*585 s*	*14.5 s*	*23.65 s*
DB Build	1	*0.0002 s*	*0.0002 s*	*0.0009 s*
	1000	*0.035 s*	*0.037 s*	*1.50 s*
DB Encryption	1	*41.8 s*	*3.18 s*	*0.063 s*
	1000	*>2000 s*	*>2000 s*	*49.15 s*

Notice that, in a real-life application, the key-generation process is performed only once and we do not have to build the database of plaintexts. Therefore, the elapsed times obtained for the encryption process are the ones that are most important. Given this, it is clear from the results in Table 5.7 that OB Final has a much better performance than the previous versions of the code.

5.1.3.3 Bob and Alice Encryption

In this section, we discuss the encryption process. Recall that Bob is the one performing private search. Therefore, we start by explaining how the Bob encryption process works within our program.

The first thing we do is to generate a random element from the ideal J. The reason for us to not use a similar optimization to the one used for the ideal I is due to the fact that usually J has one generator (according to the suggested parameters), which means that the whole process is still quite fast even without optimization.

After generating this element we simply add it to the plaintext that was given as input, by Bob, which is an element of S_n. Finally, we convert the result to vector form, where each coordinate is the coefficient regarding a specific monomial, according to the list `m` mentioned in Subsection 5.1.3.2.

Alice performs encryption similarly to what we have seen in Chapter 3, having `ac` generated in key-generation, we simply generate a random element, in this case, from S_{r+k}. We then compute the product of this element with `ac`, obtaining a random element of I. Then we just have to convert the output of Bob's encryption into the orthogonal basis of S_{r+k}, and we can simply add the random element from I previously computed. Note that every other element involved is already in the orthogonal basis of S_{r+k}.

5.1.3.4 Carl's Computation

Carl's task is to compute the product

$$P(x) = \prod_{E_A(y) \in E_A(D)} \left(E_A\left(E_B(x)\right) - E_A(y) \right),$$

and send it back to Alice.

Without any optimization, this task would be significantly harder. However, thanks to the fact that every element is in the orthogonal basis, Carl is able to compute this product quite fast.

5.1.3.5 Bob and Alice Decryption

The decryption from Alice and Bob is similar, therefore we only explain Alice's decryption since Bob's just differs in certain parameters.

After obtaining the product from Carl, we first convert it back to the standard basis, such that we are able to perform decryption, i.e., substitute every x_m by `rr[`$m-n-1$`]` (also converted to the standard basis), starting with $m = n + r$ and ending with $m = n + 1$. To do this we use the built-in function of Sage named `.subs()`.

After decrypting, Alice sends the result to Bob, where he perform his own decryption, obtaining either something different from 0 (meaning that the element does not belong to the database) or obtaining 0 (which either means that the element belongs to the database or is a false positive).

5.1.3.6 Testing function

Finally, we built a function called `Testing`, presented in Appendix A.1, where given an input $k \in \mathbb{N}$, it runs a third-party private search scenario as described above, k times, where each time the input is a random element of S_n, simulating a query from Bob. At the end we get the following output:

- Correct

  This happens when the element tested does not belong to the database and the final result is different from 0, or when the element tested belongs to the database and the final result is 0;

- False Positives

  As we mentioned in its respective section, there is a slight probability of false positives if we do not choose our parameters carefully. Obviously, false positives happen when the final result is 0 even though the initial element does not belong to the database;

- False Negatives

  Although it is not possible to have a false negative in this scheme, we decided to have this counter, which was very useful in the early stages of our program, to help us notice any irregular behaviour;

- Error (%);

- Elapsed Time (seconds).

5.2 Experimental Results

In this section we share the results of some experiments using the function `Testing` mentioned in the previous subsection. First, we present the results of some tests where we intentionally selected not so good parameters, leading to false positives.

In order to make this a simple example we consider our database to have size 1000, $k = 1$ and $r = n + 3$. Recall that an upper bound for the probability of a false positive (FP), in this example, is given by $\left(\frac{1000}{p}\right)^{2^{r+k}}$. The results obtained are presented in Table 5.4.

Table 5.4: Third-party private search with "poor" parameters

			Results over 500 runs		
p	r	Majorant for FP prob.	FP obtained	Error %	Elapsed Time
11	5	100%	500	100%	14.11 s
	6	100%	500	100%	27.72
	7	100%	500	100%	65.32 s
1009	5	56%	30	6.0%	22.85 s
	6	32%	1	0.2%	48.80 s
	7	11%	0	0.0%	124.57 s

As we can see, if we do not choose our parameters carefully, we may increase our chances of obtaining a false positive whenever a private search is performed, which is not what is desired.

Finally, we present some results obtained for several combinations of parameters, this time chosen properly, so that, ideally, we would not get any false positives.

Notice that since we could not replicate the execution times mentioned by the authors of the original scheme, it is difficult for us to show results using the suggested parameters, however we will attempt to present at least one run of a third-party private search scenario using such parameters.

In Table 5.5, are shown the results obtained when 1000 runs were performed for each combination of parameters, where we simply vary the values of $r + k$, considering $r = n + 3$ and $k = 1$, i.e., only the value of n varies, which is enough since what matters is the sum $r + k$. We also vary the value of p, choosing it adequately to prevent false positives.

Table 5.5: Third-party private search results over 1000 runs

			Results over 1000 runs
DB Size	p	r+k	Elapsed Time
1000	5003	7	95.16 s
		8	256.67 s
		9	824.58 s
	10007	7	94.51 s
		8	260.54 s
		9	778.80 s
10000	50021	6	264.54 s
		7	508.04 s
		8	920.64 s
	100003	6	249.42 s
		7	474.20 s
		8	889.71 s

As expected we did not obtain false positives in any of the tests performed. We can also observe that for a given database size, the difference in elapsed time between the different values of p is not significant. Also, as mentioned above, the upper bound for the probability of a false positive is $\left(\frac{DBsize}{p}\right)^{2^{r+k}}$, and since this value is always very close to 0, on this last experiments, we decided not to have a column with such data in the last and next tables.

Table 5.6 presents the results obtained for a database with size 100000. Even though we wanted to have results for bigger parameters, the elapsed time started to be impractical. Consequently, we have limited this experiment to 100 runs.

Table 5.6: Third-party private search results over 100 runs

		Results over 100 runs
p	r+k	Elapsed Time
500009	6	222.75 s
	7	411.79 s
1000003	6	213.40 s
	7	462.51 s

Once again, the upper bound for the probability of a false positive is very close to zero. Therefore, we did not obtain false positives in any of the these tests.

Finally, we attempted to perform this experiment with the parameters suggested by the authors of the original scheme, i.e.,

$$n = 7, \quad p = 1073741827, \quad r = 10, \quad k = 1 \quad \#Database = 10000.$$

It is clear that the upper bound for the probability of a false positive is very small, since p is much greater than the size of our database. The function `Tprik` was used to perform 10 runs with these parameters, obtaining the results presented in Table 5.7.

Table 5.7: Third-party private search results over 10 runs

		Results over 10 runs		
p	r+k	FP obtained	Error %	Elapsed Time
1073741827	11	0	0%	262.49 s

We did not obtain any false positives, however, we can observe that the elapsed time has increased significantly when compared to what we saw in the previous tables, where the number of runs performed was much greater. Therefore, we can conclude that we are limited not only in the number of runs we can perform but also in the database size we choose, since that every attempt to perform this experiment with a database of 100000 was unsuccessful.

Chapter 6

Conclusion

In this project we analyzed, with detail, a suggested fully homomorphic encryption scheme that does not use bootsrapping, as mentioned in the introduction. We were able to present some results we consider to be relevant, that were not discussed in the original article, specially in Section 4.2.2.1 where we show how an attacker ca use a simple method in order to recover parts of a plaintext, having only access to an encrypted database, and in Section 4.2.3 where we prove that without any type of enhancement, the number of mutual null coordinates across the generators of the ideal I used for encryption, is always equal to 2^n.

To conclude this project we explored the suggested application, for this scheme, which is private search. More specifically, the application suggested is for a third-party that wishes to perform private search. We then built a program in which we could simulate this scenario, i.e., we generate a random element, that is to be interpreted as a legit query from a third-party and then follow the protocol mentioned in Section 5.1.

Even though we were not able to exactly replicate the tests performed by the authors, we performed a variety of tests for several combinations of parameters, obtaining positive results in most, i.e., if we chose our parameters carefully then we could almost guarantee a satisfactory result, or in other words, no false positives.

This project also raised some questions that we believe to be worth of future research, such as, a possible enhancement in the encryption part, where we rely on a different ideal J in order to give the attacker a harder time when attempting to recover our ideal I used for encryption. Finally, another question that we consider to be worth of further exploring is the quest to build a good homomorphic embedding between our database and S_n. In this project, we were able to present a simple homomorphic embedding that,

as we saw, was not good enough to prevent an attacker from uncovering the mutual null coordinates among the generators of the ideal I and, therefore, possibly expose certain parts of plaintexts.

Bibliography

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[3] A. Gribov, D. Kahrobaei, and V. Shpilrain, "Practical private-key fully homomorphic encryption in rings," *Groups Complexity Cryptology*, vol. 10, no. 1, pp. 17–27, 2018.

[4] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[5] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.

[6] S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 173–201.

[7] J. Benaloh, "Simple verifiable elections," in *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop*, ser. EVT'06. USENIX Association, 2006, p. 5.

[8] D. Naccache and J. Stern, "A new public key cryptosystem based on higher residues," in *Proceedings of the 5th ACM Conference on Computer and Communications Security*, 1998, pp. 59–66.

[9] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[10] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *Theory of cryptography conference*. Springer, 2005, pp. 325–341.

[11] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2011, pp. 129–148.

[12] P. Scholl and N. P. Smart, "Improved key generation for gentry's fully homomorphic encryption scheme," in *IMA International Conference on Cryptography and Coding*. Springer, 2011, pp. 10–22.

[13] M. v. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2010, pp. 24–43.

[14] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Annual cryptology conference*. Springer, 2011, pp. 505–524.

[15] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, 2012, pp. 1219–1234.

[16] P. Zimmermann, A. Casamayou, N. Cohen, G. Connan, T. Dumont, L. Fousse, F. Maltey, M. Meulien, M. Mezzarobba, C. Pernet *et al.*, *Computational mathematics with SageMath*. SIAM, 2018.

[17] T.-Y. Lam, *A first course in noncommutative rings*. Springer, 1991, vol. 131.

[18] T. W. Judson, *Abstract Algebra: Theory and Applications*. Orthogonal Publishing, 2013.

[19] J. Daemen and V. Rijmen, "The block cipher rijndael," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 1998, pp. 277–284.

[20] C. Fontaine and F. Galand, "A survey of homomorphic encryption for nonspecialists," *EURASIP Journal on Information Security*, vol. 2007, pp. 1–10, 2007.

[21] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Computing Surveys (Csur)*, vol. 51, no. 4, pp. 1–35, 2018.

Appendix A

Appendix

A.1 Main Code

First we share the code used for standard encryption and decryption as well as for Third-Party Private Search. Note that we start by defining some functions that make the remaining of our code much simpler.

```
import time
import random
import numpy
#------------------------------
# FULLY HOMOMORPHIC ENCRYPTION
#------------------------------
#--------------------
# HELPFUL FUNCTIONS
#--------------------
# CREATES POLYNOMIAL RING
def ring(n,p):
    if n == 1:
        R = PolynomialRing(Zmod(p),'x0',1);
        R.inject_variables(None, None);
    else:
        R = PolynomialRing(Zmod(p),'x',n);
        R.inject_variables(None, None);

    return R
```

```
# CREATES QUOTIENT RING
def Qring(n,p):
    if n == 1:
        R = PolynomialRing(Zmod(p),'x0',1);
        R.inject_variables(None, None);
        S = R.quotient([p**2-p for p in R.gens()]);
        S.inject_variables(None, None);
        L = S.lifting_map();
    else:
        R = PolynomialRing(Zmod(p),'x',n);
        R.inject_variables(None, None);
        S = R.quotient([p**2-p for p in R.gens()]);
        S.inject_variables(None, None);
        L = S.lifting_map();

    return S, L

# RETURNS COORDINATES OF f IN m (BASIS)
def vec(n,p,m,f):
    R = ring(n,p);
    f = R(f);
    q = [];
    for p in m:
        q.append(f.monomial_coefficient(p))

    q = vector(q);

    return q

# BUILDS ORTHOGONAL BASIS OF Sn OVER Zp, ALL ELEMENTS ARE IDEMPOTENTS
def ideb(n,p):
    S, L = Qring(n,p);
    c = Subsets(S.gens(),submultiset=True);
    d = c.list();
    e = [];
    e.append(prod((1-d[c.cardinality()-1][i]) for i in
        range(len(d[c.cardinality()-1]))));

    for j in range(1,c.cardinality()-1):
        e.append((prod((d[j][i]) for i in
            range(len(d[j])))*(prod((1-d[c.cardinality()-1-j][i])
                for i in range(len(d[c.cardinality()-1-j]))))));

    e.append(prod((d[c.cardinality()-1][i]) for i in
        range(len(d[c.cardinality()-1]))));
```

```
69
70      e = [L(e[i]) for i in range(len(e))];
71
72      return e
73
74  # RETURNS RANDOM IDEMPOTENT FOR ALICE
75  def ridea(n,p):
76      V = VectorSpace(Zmod(2),2**n);
77      v = V.random_element();
78      h = v.list();
79
80      return h
81
82  # RETURNS RANDOM IDEMPOTENT FOR BOB
83  def rideb(n,p):
84      V = VectorSpace(Zmod(2),2**n);
85      v = V.random_element();
86      h = [];
87      for i in range(len(v)):
88          if v[i] != 0 :
89              h.append(i);
90          else:
91              i = i+1;
92
93      e = ideb(n,p);
94      r = sum(e[i] for i in h);
95
96      return r
97
98  # GIVEN b AS A LIN. COMB. OF ei RETURNS b EXPRESSED IN THE BASIS xi
99  def unorto(b,e):
100     uno = sum(int(b[i])*e[i] for i in range(len(b)));
101
102     return uno
103
104 # CREATES MONOMIALS LIST (ORDER)
105 def mon(n,p):
106     R = ring(n,p);
107     m = [R(1)];
108     for l in R.gens():
109         for i in range(len(m)):
110             m.append(l*m[i])
111
112     return m
113
```

```
114 # INCREASES VECTOR LENGHT (WITH ZEROS)
115 def exvec(v,n):
116     v1 = list(v);
117     vl = v1 + [0]*(2**(n)-len(v));
118     vl = vector(vl, immutable = True);
119 
120     return vl
121 
122 # BUILD T
123 def ti(n,p):
124     V = VectorSpace(Zmod(int(2)),n);
125     F = Zmod(p);
126     k = 0
127     l = len(V.basis());
128     v1 = [V[0]]+ [x for x in V.basis()];
129     while len(v1) != 2**l :
130         v2 = [];
131         for v in V.basis():
132             for i in range(k,len(v1)):
133                 s = v + v1[i];
134                 if s not in v2 and s not in v1:
135                     v2.append(s)
136         k = len(v1);
137         v1 = v1 + v2;
138 
139     A = column_matrix([x for x in v1]);
140     t = [a.list() for a in A];
141     for i in range(len(t)):
142         for j in range(len(t[i])):
143             t[i][j] = F(t[i][j])
144 
145     t = numpy.array(t);
146 
147     return t
148 
149 # E BUILD
150 def Ebld(n,t):
151     m = [numpy.array([1 for _ in range(2**n)])];
152     for l in t:
153         for i in range(len(m)):
154             p = l*m[i]
155             m.append(p)
156 
157     m = numpy.array(m);
158 
```

```
159     return m
160
161 #Check null entries in common (ideal generators)
162 def listint(a,b):
163     l = [x for x in a if x in b];
164
165     return l
166
167 def checknull(ac):
168     ind = [x for x in range(len(ac[0]))];
169     for k in ac:
170         ind1 = [];
171         for i in range(len(k)):
172             if k[i] == 0:
173                 ind1.append(i)
174         ind = listint(ind,ind1)
175
176     return ind
177
178 #---------------------------------
179 # STANDARD ENCRYPTION/DECRYPTION
180 #---------------------------------
181 #-----------------
182 # KEY GENERATING
183 #-----------------
184 def prik(n,p,r):
185     R = ring(n+r,p);
186     m = mon(n+r,p);
187
188     # Build Vector Space
189     e = ideb(n+r,p);
190     e_ = [];
191     for i in range(len(e)):
192         e_.append(vec(n+r, p, m, e[i]));
193
194     V = VectorSpace(Zmod(p),2**(n+r))
195     W = V.subspace_with_basis(e_);
196     t = [W.coordinate_vector(vec(n+r, p, m, k)) for k in R.gens()];
197
198     # Generate r random idempotents
199     rf = [];
200     for i in range(r):
201         rf.append(W.coordinate_vector(vec(n+r, p, m,ride(n+i,p))));
202
203     return e, W, rf, m, t
```

```
204
205 #-------------
206 # ENCRYPTION
207 #-------------
208 def Enc(mc):
209     mf = W.coordinate_vector(vec(n+r, p, m, mc));
210
211     # Generate r random elements of Sr
212     hi = [];
213     for i in range(r):
214         hi.append(W.random_element())
215
216     # Encryption
217     O = sum(W([x*y for (x,y) in zip((t[n+i]-rf[i]),hi[i])])
218         for i in range(r));
219     cr = mf+O ;
220
221     return cr
222
223 #-------------
224 # DECRYPTION
225 #-------------
226 def Dec(c):
227     Sr, L = Qring(n+r,p);
228
229     # Return to the standard basis
230     c1 = unorto(c,e);
231     c1 = L(Sr(c1));
232     rd = [unorto(k,e) for k in rf];
233
234     # Decryption
235     for i in range(r):
236         c1 = c1.subs({L(Sr('x%d'%(n+r-1-i))):rd[r-1-i]});
237         c1 = L(Sr(c1));
238
239     cf = L(Sr(c1));
240
241     return cf
242
243 #-----------------------------
244 # THIRD PARTY PRIVATE SEARCH
245 #-----------------------------
246 #----------------
247 # KEY GENERATING
248 #----------------
```

```
249 def Tprik(n,p,r,k):
250     tii = time.time()
251     R = ring(n+r+k,p);
252     m = mon(n+r+k,p);
253 
254     # Build Vector Space
255     e = ideb(n+r+k,p);
256 
257     # Build t
258     t = ti(n+r+k,p);
259 
260     # Build E
261     E = Ebld(n+r+k,t);
262 
263     Edb = E[:2**n];
264 
265     rr = [];
266     for i in range(r):
267         ra = rideb(n+i,p);
268         ve = vec(n+r+k,p,m,ra);
269         s = sum(ve[j]*E[j] for j in range(len(ve)));
270         rr.append(s);
271 
272     # Generate k random idempotents for BEnc
273     rk = [];
274     for i in range(k):
275         rk.append(rideb(n,p))
276 
277     ac1 = [(t[n+i]-rr[i]) for i in range(r)];
278     ac2 = checknull(ac1);
279     ac = numpy.array([1 for _ in range(2**(n+r+k))]);
280     for l in ac2:
281         ac[l] = 0;
282 
283     tf = time.time()
284     print(tf-tii)
285 
286     return e, rr, rk, m, t, E, Edb, ac
287 
288 #-----------------
289 # DATA BASE BUILD
290 #-----------------
291 def DBb(size):
292     ini = time.time();
293     Fa = Zmod(p);
```

```
294     a = Fa.random_element();
295     Fb = Zmod(2**n);
296     b = Fb.random_element();
297     DB = [(a+i)*Edb[(b+i)] for i in range(size)];
298     end = time.time()
299     print(end-ini)
300 
301     return DB
302 
303 #---------------------
304 # DATA BASE ENCRYPTION
305 #---------------------
306 def DBE():
307     ini = time.time();
308     eDB = [];
309 
310     # Encryption
311     for l in DB:
312         ha = numpy.array([Zmod(p).random_element() for
313             _ in range(2**(n+r+k))]);
314         a = l + ac*ha;
315         eDB.append(a)
316 
317     end = time.time()
318     print(end-ini)
319 
320     return eDB
321 
322 #-----------------
323 # BOB ENCRYPTION
324 #-----------------
325 def BEnc(bm):
326     Sr1, L = Qring(n+r+k,p);
327     V = VectorSpace(Zmod(p),2**(n+r+k));
328     bm = Sr1(bm);
329 
330     # Generate k random elements of Sr
331     hbv = [V.random_element() for _ in range(k)]
332     hb = [];
333     for l in hbv:
334         hb.append(Sr1(sum(m[i]*l[i] for i in range(len(l)))))
335 
336     tb = [];
337     for l in t:
338         tb.append(Sr1(unorto(l,e)))
```

```
339
340     # Encryption
341     O = sum((tb[n+r+i]-rk[i])*hb[i] for i in range(k));
342     cr = L(Sr1(bm+O));
343     cr1 = vec(n+r+k,p,m,cr);
344
345     return cr1
346
347 #-----------------
348 # ALICE ENCRYPTION
349 #-----------------
350 def AEnc(bc):
351     ba = sum(bc[i]*E[i] for i in range(len(bc)));
352
353     ha = numpy.array([Zmod(p).random_element() for _
354         in range(2**(n+r+k))]);
355     cr = ba + ac*ha;
356
357     return cr
358
359 #-----------------
360 # CLOUD COMPUTATION
361 #-----------------
362 def Carl(some):
363
364     # Product
365     pr = prod(k-some for k in eDB);
366
367     return pr
368
369 #-----------------
370 # ALICE DECRYPTION
371 #-----------------
372 def ADec(pr):
373     Sr1, L = Qring(n+r+k,p);
374     R = ring(n+r+k,p);
375
376     # Return to the standard basis
377     pr = unorto(pr, e);
378     c1 = L(Sr1(pr));
379
380     # Decryption
381     rda = [unorto(l,e) for l in rr];
382     for i in range(r):
383         c1 = c1.subs({L(Sr1('x%d'%(n+r-1-i))):rda[r-1-i]});
```

```
384         c1 = L(Sr1(c1));
385
386     cf = L(Sr1(c1));
387
388     return cf
389
390 #-----------------
391 # BOB DECRYPTION
392 #-----------------
393 def BDec(cf):
394     Sr1, L = Qring(n+r+k,p);
395     R = ring(n+r+k,p);
396     c1 = L(Sr1(cf));
397
398     # Decryption
399     for i in range(k):
400         c1 = c1.subs({L(Sr1('x%d'%(n+r+k-1-i))):rk[k-1-i]});
401         c1 = L(Sr1(c1));
402
403     ct = L(Sr1(c1));
404
405     return ct
406
407 #---------
408 # GLOBAL
409 #---------
410 def tpps(bm):
411     eb = BEnc(bm);
412     ac = AEnc(eb);
413     pt = Carl(ac);
414     cf = ADec(pt);
415     ct = BDec(cf);
416
417     return ct
418
419 #----------
420 # TESTING
421 #----------
422 def Testing(tries):
423     ini = time.time();
424     Sr1, Lr1 = Qring(n+r+k,p);
425     m1 = mon(n+r,p);
426     V1 = VectorSpace(Zmod(p),2**(n+r));
427     po = 0;
428     fp = 0;
```

```
429     fn = 0;
430     for i in range(tries):
431         v = V1.random_element();
432         vte = exvec(v,n+r+k);
433         te = sum(m1[i]*v[i] for i in range(len(m1)));
434 
435         te = Lr1(Sr1(te));
436         ct = tpps(te);
437 
438         if (ct == 0 and te in DBu) or (ct != 0 and te not in DBu):
439             po = po + 1;
440         else:
441             if ct == 0 and te not in DBu:
442                 fp = fp + 1;
443             else:
444                 fn = fn + 1;
445 
446     end = time.time();
447     print('Results with: n = ',n,', p = ',p,', r = ',r,', k = ',k)
448     print('  ')
449     print('  ')
450     print('Corretos: ', po)
451     print('  ')
452     print('Falsos Positivos: ',fp)
453     print('  ')
454     print('Falsos negativos: ',fn)
455 
456     tf = fp + fn;
457     per = (round(tf/tries, ndigits = 3))*100
458 
459     print('  ')
460     print('Erro: ',per, '%')
461     print('  ')
462     print('Time elapsed: ', round(end-ini, ndigits = 5), ' secs')
```

A.2 Mutual null coordinates experiment

The following code was used to generate the plots 4.1, 4.2 and 4.3 in section 4.2.2.1. Some functions used in this code were introduced in A.1.

```
#----------------------------------------
#BUILD DATABASE WITH GOOD 1 TO 1 EMBEDDING
#----------------------------------------
def DBb(size):
    ini = time.time();

    DBi = [];
    DBi.append(previous_prime(2**(20*(n+r))));
    for _ in range(size-1):
        DBi.append(previous_prime(DBi[-1]))

    DBif = [k.digits(p) for k in DBi];

    DB = [];
    for l in DBif:
        a = sum(l[i]*Edb[i] for i in range(len(l)))
        DB.append(a)

    end = time.time()
    print(end-ini)

    return DB

#----------------------------------------
#BUILD DATABASE WITH BAD 1 TO 1 EMBEDDING
#----------------------------------------
def DBb(size):
    ini = time.time();

    DBi = [];
    DBi.append(2);
    for _ in range(size-1):
        DBi.append(next_prime(DBi[-1]))

    DBif = [k.digits(p) for k in DBi];
    DB = [];
    for l in DBif:
        a = sum(l[i]*Edb[i] for i in range(len(l)))
        DB.append(a)

```

```
41      end = time.time()
42      print(end-ini)
43
44      return DB
45
46  #-----------------------------------------------
47  #BUILD DATABASE WITH BAD HOMOMORPHIC EMBEDDING
48  #-----------------------------------------------
49  def Dbhomo(size):
50
51      ini = time.time();
52      F = Zmod(2**n);
53      a = F.random_element();
54      ae = E[a];
55      DB = []
56      DB.append(ae);
57      for i in range(2,size):
58          DB.append(i*ae);
59
60      end = time.time()
61      print(end-ini)
62
63      return DB
64
65  #-------------------------------------------------------
66  #COMPUTE MEAN OF EVERY COORDINATE ACROSS ALL GENERATORS
67  #-------------------------------------------------------
68  def analy(eDB):
69      Xs = [];
70      t = len(eDB);
71      for i in range(len(eDB[0])):
72          a = sum(int(eDB[j][i]) for j in range(t));
73          Xs.append(a/t)
74
75      return Xs
```

A.3 Enhancements testing

To conclude this appendix we share the code used to compute the data in table 4.1. We only had interest in the number of mutual null coordinates among the ideal generators, therefore all we had to do was to slightly change our key-generating function defined in A.1.

```
#---------------------
# CHANGE TO fj BASIS
#--------------------
def change_basis(elm):
    ef = [];
    ef.append(elm[0]);
    c = [randint(0,1) for _ in range(2**(n+r)-1)];
    for i in range(1,2**(n+r)):
        f = elm[i]+sum(c[l]*elm[l] for l in range(0,i));
        ef.append(f)

    eff = numpy.array(ef);

    return eff

#-----------------------------------------------------
# KEY GENERATING (Optimization in ideal generators)
#-----------------------------------------------------
def prikg(n,p,r):
    R = ring(n+r,p);
    m = mon(n+r,p);

    # Build Vector Space
    e = ideb(n+r,p);

    # Build t
    t = ti(n+r,p);

    # Build E
    E = Ebld(n+r,t);
    Edb = E[:2**n];

    # Generate r random  idempotents for Enc
    rr = [];
    for i in range(r):
        ra = rideb(n+i,p);
        ve = vec(n+r,p,m,ra);
        s = sum(ve[j]*E[j] for j in range(len(ve)));
```

```
39          rr.append(s);
40
41      #Generate r random monomials
42      Ep = [];
43      for i in range(r):
44          a = randint(0,2**(n+i)-1);
45          Ep.append(E[a]);
46
47      ac1 = [(t[n+i]*Ep[i]-rr[i]) for i in range(r)];
48      ac2 = checknull(ac1);
49      leac = len(ac2);
50
51      return leac
52
53  #-----------------------------------------------
54  # KEY GENERATING (Optimization in basis change)
55  #-----------------------------------------------
56  def prikb(n,p,r):
57      R = ring(n+r,p);
58      m = mon(n+r,p);
59
60      # Build Vector Space
61      e = ideb(n+r,p);
62
63      # Build t
64      t = ti(n+r,p);
65
66      # Build E
67      E = Ebld(n+r,t);
68      Edb = E[:2**n];
69
70      # Generate r random idempotents for Enc
71      rr = [];
72      for i in range(r):
73          ra = rideb(n+i,p);
74          ve = vec(n+r,p,m,ra);
75          s = sum(ve[j]*E[j] for j in range(len(ve)));
76          rr.append(s);
77
78      ac1 = [(t[n+i]-rr[i]) for i in range(r)];
79      ac1f = [change\_basis(k) for k in ac1];
80      ac2 = checknull(ac1f);
81      leac = len(ac2);
82
83      return leac
```

```
84
85 #----------------------
86 #TESTING ENHANCEMENTS
87 #----------------------
88 def testoti(oti,reps,n,p,r):
89     ini = time.time();
90     s = 0;
91     for i in range(reps):
92         l = oti(n,p,r);
93         s = s + l;
94
95     sf = round(s/reps, ndigits = 3);
96
97     end = time.time();
98     print(end-ini)
99
100    return sf
101
102 #Notes:
103 #oti is either prikb or prig
104 #reps is the number of repetitions we wish to perform
```