Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

División de Estudios de Licenciatura

Applying homomorphic computing in the cloud

por

Jesús Antonio Soto Velázquez

en opción al grado de

Ingeniero en Tecnologías de Software

San Nicolás de los Garza, Nuevo León Fecha

Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

División de Estudios de Licenciatura

Applying homomorphic computing in the cloud

por

Jesús Antonio Soto Velázquez

en opción al grado de

Ingeniero en Tecnologías de Software

San Nicolás de los Garza, Nuevo León Fecha

Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

División de Estudios de Licenciatura

Los miembros del Comité de Tesis recomendamos que la Tesis «Applying homomorphic computing in the cloud», realizada por el alumno Jesús Antonio Soto Velázquez, con número de matrícula 1570031, sea aceptada para su defensa como opción al grado de Ingeniero en Mecatrónica.

El Comité de Tesis

Tu asesor
Asesor

Sinodal 1
Coasesor

Sinodal 2
Revisor

Vo. Bo.

Dr. Arnulfo Treviño Cubero
División de Estudios de Licenciatura

San Nicolás de los Garza, Nuevo León, Fecha

Agradecimientos

A Dios ...

Gracias a la Universidad Autónoma de Nuevo León y a la Facultad de Ingeniería Mecánica y Eléctrica por el apoyo prestado durante el transcurso de la carrera.

A mis padres ...

No olvidar al asesor, sinodales, instituciones que dieron apoyo.

Resumen

Jesús Antonio Soto Velázquez.

Candidato para el grado de [LLENAR].

Universidad Autónoma de Nuevo León.
Facultad de Ingeniería Mecánica y Eléctrica.

Título del estudio:

Applying homomorphic computing in the cloud

Número de páginas: **??**.

Objetivos y método de estudio: El objetivo de este trabajo de tesis es ...

Contribuciones y conclusiones: La principal contribución de esta tesis es ...

Firma del asesor: _____

Tu asesor

Contents

List of Figures

List of Tables

Abstract

Objetives and Study Method

This work proposes an implementation of a client-server architecture based software that uses HElib to enable homomorphic encryption and perform computations on the encrypted data. Cloud computing is a cost-efficient alternative to host data remotely and subsequently perform computations on it. Confidentiality becomes an issue when data is delegated to the cloud, as often the information hosted is considered to be sensitive. A typical approach is to make use of a cryptographic algorithm that encrypts the data with a secret key that only the owner has. However, as long as the owner of the data wishes to make changes to the encrypted data, he would have to download the encrypted data, decrypt it using his secret key, perform the desired computations or changes to it, and reencrypt it once again before uploading it to the server in the cloud. An alternative to this approach is to make use of *homomorphic encryption*, and advanced technique in cryptography that enables computations on encrypted data without needing the secret key.

Homomorphic encrypting is a well-sounding alternative over the typical approach of encrypting data every time it is send to the cloud; however, so far the schemes that make this possible have been thought of as being too slow and expensive, storage wise. A library that implements homomorphic encryption, HElib, is based off an homomorphic encryption scheme that seems very promising because of the optimizations and improvements over other past schemes. Using HELib, it would be possible to reevaluate the situation and show to what degree it would be feasible to make a solution based on homomorphic encryption.

The proposed solution consists in taking the functionalities provided by HElib, and build a client-server based software that aims to address the main scenarios found in the proposed case study. Even though the proposed implementation could in theory work for a number of examples, it is designed taking into account the case study; therefore, both are closely related.

Consider the scenario where a household has an expected pattern of activity, ie. empty during the day and non-empty at night, so that the householder seeks to ascertain the number of people inside at any point in time during the day. Assuming the householder has put in place certain sensors around the building so that it detects and counts who comes in and out, he would like to learn the value of the counter remotely. As the householder chooses to store the value in the cloud, he quickly realizes he does not want others to learn of this value, not even the cloud service itself, as to prevent potential burglars to break in when the household is empty. Additionally, he wishes that the counter gets updated at certain intervals, e.g. every 20 minutes.

The problem found in encryping and then uploading the data every time a change is made, is that the overhead costs might be too high to do this often. For one, bandwidth would be spent on each ocassion, and the time required for encryption and uploading might not be that short.

Contributions

In contrast with other works, the proposed solution tackles on the problem by employing homomorphic encryption. It is a simple client-server architecture which could be expanded to adapt with other scenarios and requirements. The proof of concept provides the means to operate on the encrypted data, ie. performing additions on the counter value, without compromising confidentiality and wasting resources caused by the overhead costs of reencrypting data at every change.

To show the feasibility of using HElib to do homomorphic encryption, experiments were designed by varying the value of a security parameter k. This value has a direct impact on the execution time of key generation, encryption, and decryption, as well as the size of the resulting public key. The findings show that even though key generation time is long, taking approximately between 9 and 13 seconds, and key size is large, being between 35MB and 165M, it is not considered to be an issue, as this step is done one time only. On the other hand, encryption, decryption, and

addition times are amazingly fast, none of them exceding over 1 second. However, the ciphertext raises some doubts over the magnitude of its size: during development, it was found that on average, ciphertext size was 74MB; meanwhile, during experimentation, it was found that ciphertext size rose no higher than 45kB. To the best of our knowledge, there is yet no way to predict the size of the ciphertext by using HElib. Therefore, if it were found to keep the size of the ciphertext below 45kB, it would certainly be feasible to take this solution to the cloud; otherwise, it would not be possible if each ciphertext had a size that exceded several MB.

Chapter 1

Introduction

Communication has always been a basic human need, and the means to do it is especially linked to the available technology. Thanks to the advances in techonology, it has become more common to exchange information with our peers remotely. Whether it is our personal information or current location, there are many scenarios where it is necessary to pass that information to another point. It becomes an issue when the information is considered to be *sensitive*, and thus, the confidentiality of it must be protected, so that no one other than the sender and the intended receiver get to learn about the sensitive information. The medium used by both the sender and receiver to transmit any piece of information is called a *channel*, which is often considered to be unsecure. It is assumed that apart from the sender and receiver, there might be a third party that tries to eavesdrop on the communication lines. In this case, it becomes important to protect the confidentiality of the information shared between the sender and receiver.

For the scenarios where confidentiality becomes important, *cryptography* becomes an attractive solution, so that encryption techniques are used to put the information in some kind of *safe*, preventing any eavesdropper to learn of its contents. The issue at hand becomes even more interesting when the safe containing the private information does not go directly to the intended party, but is rather stored somewhere in the cloud, ie. the Internet. As long as the proper encryption mechanisms are put in place, the confidentiality is not necessarily compromised if the data were stored in the cloud. However, merely storing static information is

not quite interesting: one would want to modify it as the need arises, such as when updating personal data. Such goal calls for *homomorphic encryption*, which is an advanced technique used to modify the already encrypted data without compromising its confidentiality.

There are many instances where it is required to send information to someone else. To sign up for a service, for instance, several pieces of information are required. Usually, name and email address are required as basic pieces of information. Depending on the service, telephone number and personal address might also be required. In this case, the subscriber might actually be worried about how safe his information is being kept. He would feel safer if he had some kind of proof that vouched for the confidentiality of his data. This also affects the service, as less people would sign up for the service if they had no means to protect the data. There are several approaches to protect sensitive data, and one of the most effective ones is based on cryptography.

Cryptography is the study of techniques that enable secret communication, such as ciphers, that is, encryption and decryption algorithms to be used on sensitive data. Commonly, these algorithms are used to protect the data from eavesdroppers that try to pry on it. Ciphers can be seen as the means of putting sensitive data into a box with a lock, and only those with the appropriate secret key are able to access the contents of the box. It becomes troublesome to manage the secret key, because once it has been found by someone, any piece of data encrypted with the same lock becomes vulnerable to unauthorized access. Often, there is also a secure channel that is mostly dedicated to the exchange of secret keys; however, it is often limited and has certain constraints that makes it unfeasible to use it for anything else.

It has become increasingly common to delegate computing tasks to cloud services in order to save resources. For instance, these services might be used when it cannot be afforded to build and maintain a data center for the storage and computing needs that an investor might have. This is especially the case when the level of

activity is seasonal: this is, requiring more processing and storage capabilities than usual during certain periods of the year. Therefore, a common solution is to make use of cloud storage and computing services, so that the costs only increase during these usage peaks. This way, the budget is spent as much as the resources are being employed, while saving configuration and maintenance costs. However, as Srinivasan (2012) [**?**] points out, allowing the cloud service to make use of the data has raised concerns on security, since it is hard to trust that the cloud provider will not look at the data and do something with it.

A typical solution to this scenario consists in encrypting the data before storing it in the cloud; therefore, assuring its confidentiality. It is a great solution when it is only needed to read the data, without actually making any changes on it. In order to view the encrypted data, it has to be downloaded and decrypted using the proper secret key. However, when it is required to modify the encrypted data in some way, it requires to be downloaded and decrypted back to its plaintext form, before modifying it. Once it has been modified, the data has to be re-encrypted and re-uploaded to the cloud, in a potentially slow manner, depending on the size of the data. Abadi (2009) [**?**] explains that this approach turns out to have high overhead costs caused by the transfer of the encrypted data back and forth between the user and the cloud. In other words, it is very bandwidth intensive to encrypt and decrypt complete tables or columns out of the cloud every time some kind of change or analysis must be performed.

In scenarios where the information is not considered to be sensitive, it would be appropriate to make use of cloud computing services directly, as it would cause no harm to grant access to the data that is to be used if it were to be considered public. However, real life applications often imply sensitive data, and it should not be handled trivially, as there is no telling what kind of use an eavesdropper might give to the data.

A simple sounding, yet complicated solution to this problem would be to manipulate the encrypted data in some way such that the contents are not revealed,

but on decryption, ends up being correct. In other words, to perform some kind of computation on the encrypted data, even if the private key is not known, so that when it is decrypted, the change is taken into account and the result is the same as if the operation had been performed on plaintext data. For some time, it had been thought that doing something like that was not possible, as ciphers often consisted of permutations and substitutions. However, Rivest et al. (1978) [**?**] noted an interesting property found in the RSA algorithm that proved otherwise. Such property is called "homomorphic", and has since then gone through a lot of research, resulting in many schemes such as ElGamal [**?**], Goldwasser—Micali [**?**], Benaloh [**?**], Paillier [**?**], among others.

1.1 Problem Definition

Even though there are many homomorphic encryption schemes that have been created to allow for computations on encrypted data, so far, they are not considered fast enough to build efficient secure cloud computing solutions. In recent years, a couple of homomorphic cryptosystems implementations have surfaced, namely: HElib [**?**] and FHEW [**?**]. However, they are best thought of low-level building blocks for homomorphic encryption. Even though both implementations are based off optimized versions of the proposed homomorphic schemes by Brakerski et al. (2011) [**?**] and Ducas and Micciancio (2014) [**?**] respectively, there have not been attempts to use these libraries to build solutions in cloud computing. As such, it has not been shown if with current implementations it is still unfeasible to build a software solution using a homomorphic scheme. In this case, feasibility is put in terms of processing time and required storage size.

1.2 Motivation

Pötzelsberger (2013) [**?**] reports that there are many areas in which homomorphic encryption could be used, such as the medical, marketing, and financial fields. Until now, there was no way to make a concrete implementation of a solution related to these areas, since available schemes were either too limited or too slow. Using HELib, a library that provides fully homomorphic encryption, it would be possible to reevaluate the situation and show to what degree it would be feasible to make a solution based on homomorphic encryption.

The proposed solution consists in taking the functionalities provided by HElib, and build a client-server based software that aims to address the main scenarios found in the proposed case study. It is better thought of as proof of concept that shows how homomorphic encryption can be used in an environment that emulates cloud computing. On this attempt, basic communication functionalities are explored, as well as the flow in which the tasks are divided between the client and the server. Particular attention is put on the performance of this application, notably execution time of the homomorphic evaluations, as well as the storage required for certain items.

The idea behind this attempt is that by showing a compelling example of how homomorphic encryption can be applied in a simple scenario, cloud service providers and developers might start considering how to apply homomorphic encryption in other ways and mediums, and thus, expanding on software that makes use of it.

1.3 Hypothesis

Building a client-server based solution using the homomorphic encryption functionalities provided by HElib is feasible in terms of processing time.

1.4 Objectives

General Objective To develop a client-server based solution using HElib to perform homomorphic evaluations on encrypted data.

Specific Objectives

1. To establish a client-server architecture where homomorphic encryption can be applied.
2. To identify which factors pose a challenge to deem applications of homomorphic encryption as inefficient.
3. To show the use of HElib to setup, encrypt, and decrypt as simple as can be.
4. To collect performance data on the use of homomorphic encryption.

1.5 Case study

Consider the scenario where a household has an expected pattern of activity, ie. empty during the day and non-empty at night, so that the householder seeks to ascertain the number of people inside at any point in time during the day. Assuming the householder has put in place certain sensors around the building so that it detects and counts who comes in and out, he would like to learn the value of the counter remotely. As the householder chooses to store the value in the cloud, he quickly realizes he does not want others to learn of this value, not even the cloud service itself, as to prevent potential burglars to break in when the household is empty. Additionally, he wishes that the counter gets updated at certain intervals, e.g. every 20 minutes.

A typical approach consists in encrypting the counter value every time it is recalculated, replacing the older value stored in the cloud server. Considering the

overhead costs of this process, he considers another potential approach that makes use of homomorphic encryption. The sought solution needs to eliminate the need to reupload the counter value every time it is updated by a change in the number of people in the household. Additionally, the private key should only be in the possession of the householder, so that even the cloud hosting service does not have the ability to learn the value of the counter. Finally, whenever he needs to know of the current value of the counter, he can download the encrypted value and, using the previously generated secret key, decrypt the data to obtain the value of the counter.

1.6 Structure

This thesis is made up of the following chapters: introduction, background, related works, methodology, experiments, experiments and results, and conclusions.

In this chapter, both the problem definition and motivation are posed to build a client-server architecture using HElib to address the scenario of the proposed case study.

In chapter 2, necessary basic notation and definition of concepts on cryptography are presented to acquaint the reader with the content presented in the thesis.

Chapter 3 presents some works that are related to solutions based on homomorphic encryptography. The characteristics of each work is briefly described and compared with the ones presented in this thesis.

Chapter 4 presents the implementation details of the proposed client-server architecture, describing each component and how the functionalities of HElib are put in use. Additionally, the case study is addressed within the scope of the client-server solution as well.

Chapter 5 presents the performed experiments to test certain parts of the client-server implementation related to the use of homomorphic encryption, as well as showing the results pertaining to the processing times and size. Afterwards, it

presents conclusions based on these results.

Finally, chapter 6 presents general conclusions obtained from the results during the experimentation phase. Aditionally, some recommendations regarding the implementation of the client-server architecture are presented to be considered as future work.

Chapter 2

Background

Information security is a broad topic that covers many different aspects, from which two of them are particularly important for this work: confidentiality and authentication. There are some aspects from cryptography that would be relevant to review, such as asymmetric and symmetric cryptography, as well as the concepts: *plaintext*, *ciphertext*, and *secret key*. In order to explain more advanced topics, number theory and abstract algebra are introduced. Homomorphic encryption is explained thoroughly, noting the mathematical properties and its categories, as well as some possible applications of this type of encryption. The first fully homomorphic encryption scheme is introduced, paving the road for the BGV scheme. Finally, a brief description of cloud computing and its relevant aspects are described, as well as the impact of secure computation.

2.1 Information Security and Data Confidentiality

Seen from a general perspective, Whitman and Herbert (2011) [**?**] describe security as "the quality or state of being secure—to be free from danger". Danger would refer to a potential harming action that an adversary can do, whether it is intentional or not. Computer security is defined by the NIST Computer Security Handbook (1995) [**?**] as follows:

"The protection afforded to an automated information system in order to attain the applicable objectives of preserving the *integrity*, *availability*, and *confidentiality* of information system resources (includes hardware, software, firmware, information/data, and telecommunications)".

This definition introduces two relevant concepts in the previous definition, namely: *information* and *confidentiality*. RFC 2828 (2000) [**?**] defines information as "facts and ideas, which can be represented (encoded) as various forms of data," and data as "information in a specific physical representation, usually a sequence of symbols that have meaning: especially a representation of information that can be processed or produced by a computer." Although both terms have different connotations depending on what it represents, both words are used interchangeably in this work. Now that information has been described as a concept, confidentiality can be introduced. More specifically, data confidentiality refers to the assurance that private or confidential information is not made available or disclosed to unauthorized individuals [**?**]. This term is closely related to *privacy*, which gives individuals the control of what information related to them may be accessed or stored by whom. Loss of confidentiality would imply unathorized access or disclosure of information.

2.2 Cryptography Concepts

Cryptography is the study of protecting data and communications. It involves communicating messages or information between two or more parties by changing the appearance of the messages, so that it becomes very difficult for unauthorized parties to intercept or interfere with the transmission of the information [**?**]. Cryptography should not be confused with cryptology, because even though they overlap with each other, cryptology includes *cryptanalysis* as well. Without going too deep in the subject, cryptanalysis is defined by Stallings (2010) [**?**] as the study of principles and methods of transforming an unintelligible message back into an intelligible message *without* knowledge of the secret key. Basically, it studies the a cipher to the point

where it can break it, and learn the message without having the secret key.

Other important aspects that have been linked to cryptography apart from confidentiality are *authentication* and *integrity*. Authentication gives the receiving parties the means to make sure that the source of the communication is really who it is thought to be, and not someone pretending to be the legitimate sender. On the other hand, integrity, among other things, ensures that the message received has not been tampered in any way. Different kind of cryptographic algorithms are required to satisfy these goals, and the same goes for confidentiality.

Ryabko and Boris (2005) [**?**] approach the study of cryptography by explaning the classical problem of transmitting secret messages from a sender A to a receiver B. Both the sender and receivers, A and B, can be thought of persons, organizations, or various technical systems. They are formally described as abstract "parties" or "entities", but it is often more convenient to identify both of them as human participants that go by the names of Alice and Bob, instead of using the letters A and B. For the messages to go from Alice to Bob, there has to be some kind of medium or channel where the information is transmitted. Generally, it is assumed that the channel can be potentially accessed by a third party that is neither the sender nor the receiver. On top of it, it is also assumed that either the receiver or sender has an adversary or enemy E, who is constantly trying to tamper or know the content of the messages passed between Alice and Bob. This attacker E who is constantly eavesdropping on the communication lines is usually called Eve, who is thought to have powerful computing facilities and is able to make use methods to learn of the message contents. It is clear that both Alice and Bob want to protect their messages so that their contents are unclear to Eve. This implies a confidentiality goal, and it can be reached by using encryption algorithms, or as they are commonly known: *ciphers*.

Before a message is sent from Alice to Bob or viceversa, the sender *encrypts* the message. In other words, a certain algorithm is applied on the message so that the content of it is unclear to a third party, such as Eve. Once the sent message has

been received by the other party, he *decrypts* it to recover the original content of the message before encryption, known as *plaintext*. In order for this to work, Alice and Bob must have agreed in advance about various of the details needed, such as the algorithm and parameters used by the sender, so that the receiver knows how to perform the decryption properly. Since it would be troublesome that Eve learned of those details, Alice and Bob would have to use some sort of secure channel, such as trusted messengers or couriers, or set up a private meeting where no one else can eavesdrop. Making use of this secure channel, however, is more costly than using the regular communication channel between Alice and Bob, and it might not be available at all times, which is why it is recommended to use it only to agree on some details in advance, like the parameters for the encryption algorithm, as well as a *private key*.

Stanoyevitch describes on his Introduction to Cryptography (2010) [**?**] book that the above scheme would make up a cryptosystem, since it makes use of cryptographic algorithms with the purpose of protecting confidentiality. As it has been hinted, such a cryptosystem has two parts: *encryption*, which is done by the sender as means of putting or transforming the actual **plaintext** into **ciphertext**, and *decryption*, which is done once the message has been received at the other end and has the purpose of translating the ciphertext back into the original plaintext message. Both encryption and decryption are usually done using a **key**, and it is intended that only the sender and receiver learn of the nature and value of the key.

Stinson (2005) [**?**] formally defines a cryptosystem as a five-tuple $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$, where the following conditions are satisfied:

1. $\mathcal{P}$ is a finite set of possible plaintexts;
2. $\mathcal{C}$ is a finite set of possible plaintexts;
3. $\mathcal{K}$, the keyspace, is a set of possible keys;
4. For each $k \in \mathcal{K}$ there is an *encryption rule* $e_k \in \mathcal{E}$ and a corresponding *decryp-*

tion rule $d_k \in \mathcal{D}$. Each $e_k : \mathcal{P} \to \mathcal{C}$ and $d_k : \mathcal{C} \to \mathcal{P}$ are functions such that $d_k(e_k(x))$ for every plaintext element $x \in \mathcal{P}$.

The following image depicts the process flow of a simple cryptosystem that is put in place. The plaintext and encryption key is used as input by an encryption algorithm, which results in a ciphertext that gets sent to the other end of the communication line. Then, the ciphertext is used as input by a decryption algorithm, which takes a decryption key H, finally resulting in the original plaintext. Depending on the type of cryptosystem (symmetric or asymmetric), the encryption keys can be considered to be the same or different, i.e. $K_A \neq K_B$ if asymmetric, or $K_A = K_B$ if symmetric.

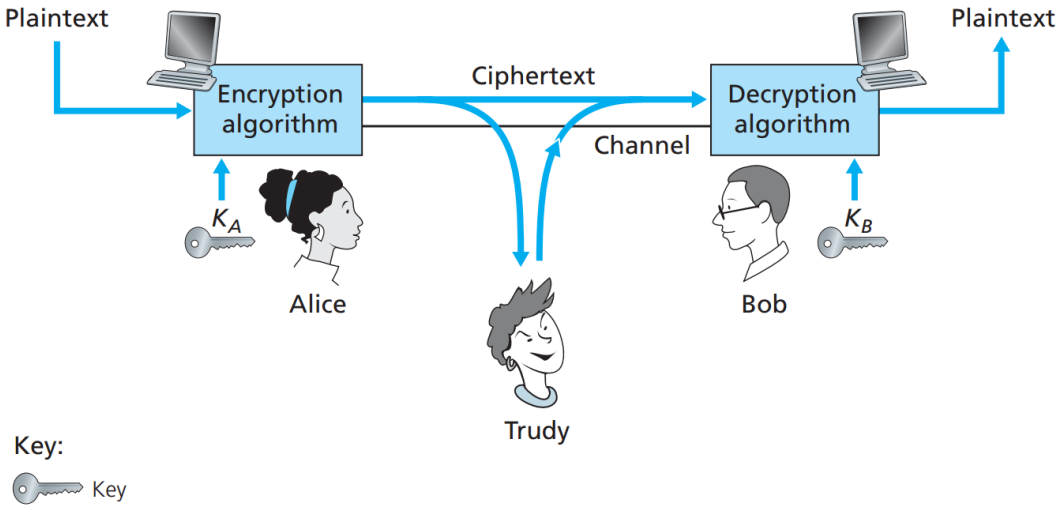

Figure 2.1 – Basic cryptosystem diagram. Source: Kurose and Ross (2010) [**?**].

2.2.1 The first fully homomorphic encryption scheme

The use of the plaintext, ciphertext, and key concepts are recurrent throughout this work, and given their relevance, they are explained as follows, along with other concepts, the same way Cameron (2003) [**?**] and Stallings (2010) [**?**] describe

them.

Plaintext.- It is the original message that Alice wants to send to Bob. Neither Bob nor Alice want the message to be known to Eve. This piece of data is fed into the encryption algorithm before it is sent to the other end.

Encryption algorithm.- This algorithm uses a key that performs several substitutions and transformations on the plaintext. The result is a ciphertext that cannot be translated back to its original plaintext form without the appropriate key.

Secret key.- It is a value that both Alice and Bob agree upon, it is usually kept secret, since it is used by the encryption algorithm to obtain the ciphertext. The encryption algorithm will produce a distinct output depending on the key being used. In other words, if the secret key is leaked to Eve, this would mean that Eve can now learn any messages that are being produced using that key.

Ciphertext.- This is the modified or scrambled message produced as output by the encryption algorithm. Its value depends on the algorithm and plaintext, as well as the chosen key.

Decryption algorithm.- It is like the encryption algorithm, except that its run in reverse. The decryption algorithm has two inputs: the ciphertext and its corresponding key. It then performs the necessary substitutions and transpositions so that the ciphertext is translated back into the original plaintext.

It is assumed at all times that Eve has access to the communication lines, and can always know the ciphertext that is being sent between Alice and Bob. It is also a good practice to assume that the encryption and decryption algorithms are known to Eve, which leaves the key as the most important part of the scheme. As mentioned previously, Alice and Bob need to establish the means to agree on a private key, without anyone else knowing what this key is. As soon as the key is known to the

attacker, i.e. Eve, all further communication is compromised. When that happens, another key has to be chosen and agreed upon once more by Alice and Bob.

The nature of the encryption and decryption can be classified in two broad categories: symmetric and asymmetric key encryption. Agrawal (2012) [**?**] describes both categories by looking at the keys used. In **symmetric encryption**, the key used for encryption is virtually the same used in decryption. Therefore, key distribution has to be done before the tranmission of the messages. The key length has a direct impact on the security offered by the encryption algorithm. Meanwhile, **asymmetric encryption** employs two different keys: public and private. The public key is closely tied to the receiver, and it is used for encryption of the data. It is called public because it is available for general use; therefore, Alice can use Bob's public key to send him messages. On the other hand, the private is kept secret from the outside world, and is only available by an authorized person. The public and private keys always come in pairs, and this is because while the public key is used to encrypt data; its counterpart, the private key, is used to decrypt the data.

2.3 Introduction to Number theory

In other to understand more topics pertaining cryptosystems, number theory and its notation, as well as some relevant theorems must be introduced. According to Stark (1970) [**?**], number theory consists of the study of the properties of whole numbers, and it is considered to be inextricably linked to cryptography. Goodrich introduces number theory and most of its relevant aspects in his Algorithm Design [**?**] book. The information is briefly summarized as follows.

2.3.1 Fundamental Theorems

Given positive integers a and b, we use the notation

$$a|b$$

to indicate that a **divides** b, that is, b is a multiple of a. If $a|b$, then we know that there is some integer k, such that $b = ak$. From this definition, the following properties are found:

Theorem on Divisibility: *Let a, b and c be arbitrary integers. Then*

- If $a|b$ and $b|c$, then $a|c$.
- If $a|b$ and $a|c$, then $a|(ib + jc)$, for all integers i and j.
- If $a|b$ and $b|a$, then $a = b$ or $a = -b$.

An integer p is said to be **prime** if $p \geq 2$ and its only divisors are the trivial divisors 1 and p. Therefore, in the case p is a prime, $p|q$ implies $d = 1$ or $d = p$. An integer greater than 2 that is not prime is said to be **composite**. Also, two integers which only share a common divisor of 1 are considered to be **relatively prime**.

Fundamental Theorem of Arithmetic: *Let $n > 1$ be an integer. Then there is a unique set of prime numbers $\{p_1, \ldots p_k\}$ and positive integer exponents $\{e_1, \ldots, e_k\}$, such that*

$$n = p_1^{e_1} \cdots p_k^{e_k}$$

The above product $p_1^{e_1} \cdots p_k^{e_k}$ is the prime decomposition of n. In other words, any positive integer is made up of the product between two prime numbers.

2.3.2 MODULAR ARITHMETIC

The remainder of a when divided by n is expressed as $a \mod n$, that is:

$$r = a \mod n$$

which means that there is some integer q, such that

$$a = qn + r$$

The remainder r is resulting from $a \mod n$ is always an integer in the set $\{0, 1, 2, \ldots, n-1\}$, even when a is negative. The integer n is called the **modulus**.

Congruence modulo n is a relevant concept to mention as well. If

$$a \mod n = b \mod n,$$

then we say that a is **congruent** to b modulo n, so it can bet written in this way:

$$a \mod n \equiv b \pmod{n}$$

Therefore, if $a \equiv b \mod n$, then $a - b = kn$ for some integer k.

Congruences have the following properties:

1. $a \equiv b \pmod{n}$ if $n|(a - b)$.
2. $a \equiv b \pmod{n}$ implies $b \equiv a \pmod{n}$.
3. $a \equiv b \pmod{n}$ and mob $b \equiv c \pmod{n}$ imply $a \equiv c \pmod{n}$.

Suppose that there is a mapping of all integers by the $\pmod{n}$ operator into the set of integers $\{0, 1, \ldots, (n-1)\}$. The technique used to perform arithmetic operations within the confines of this set is known as **modular arithmetic**. Modular arithmetic exhibits the following properties:

1. $[(a \mod n) + (b \mod n)] \mod n = (a + b) \mod n$
2. $[(a \mod n) - (b \mod n)] \mod n = (a - b) \mod n$
3. $[(a \mod n) \times (b \mod n)] \mod n = (a \times b) \mod n$

And so, the rules for ordinary arithmetic invoving addition, subtraction, and multiplication carry over into modular arithmetic, except that the result is always within the boundaries of the defined set, which is between 0 and $n - 1$.

Having explained modulo operation, as well as a having introduced modular arithmetic operations, a very convenient notation, called the set of residues, can be now described. Let Z_n denote the set of nonnegative integers less than n:

$$Z_n = \{0, 1, \cdots, (n-1)\}$$

The set Z_n is also called the set of **residues**, or **residue classes** modulo n, because if $b = a \mod n$, b is sometimes called the **residue** of a modulo n. The notation Z_n represents the integers between 0 and $n-1$. Thus, when it says that "Let a be a member of Z_n", it really means the same thing as "Let a be an integer between 0 and $n-1$."

Modular arithmetic in Z_n, where operations on the elements of Z_n are performed mod n, show properties similar to those in traditional arithmetic, such as the **associativity**, **commutativity**, **distributivity** of addition and multiplication, and the existence of **identity** elements 0 and 1 for addition and multiplication, respectively. However, other operations such as division and exponentiation, behave very differently than they do for normal arithmetic.

Every element x in Z_n has an **additive inverse**, that is, for each $x \in Z_n$, there is a $y \in Z_n$ such that $x + y \mod n = 0$. In other words, it means there is an element a' that multiplies a modulo n which equals 1. A **multiplicative inverse** of x is an element $z^{-1} \in Z_n$ such that $xx^{-}1 \equiv 1 \mod n$. As well as in regular arithmetic, 0 does not have a multiplicative inverse in Z_n. There are some nonzero elements that do not have a multiplicative inverse in Z_n. However, for every n that is prime, every element $x \neq 0$ of Z_n has a multiplicative inverse.

2.3.3 ORDER AND GENERATORS

Given a prime p and an integer a between 1 and $p-1$, the **order** of a is the smallest exponent $e > 1$ such that

$$a^e \equiv 1 \mod q$$

A **generator**, or primitive root, of Z_p is an element g of Z_p with order $p-1$. It is called a generator, because the repeated exponentiation of it can generate all the elements in Z_p^*. Z_p^* is a subset of Z_p, defined to be the set of integers between 1 and n that are relatively prime to n.

Theorem on generators: *If p is a prime, then set Z_p has $\phi(p-1)$ generators.*

2.3.4 OTHER RELEVANT POINTS ON MODULAR ARITHMETIC

Cliff and Ken [**?**] review a several important concepts regarding modular arithmetic, and are thus listed as follows:

- *Fermat's Little Theorem* Let p be a prime, and let x be an integer such that $x \mod p \neq 0$. In other words, that p does not divide the integer x. Then

$$x^{p-1} \equiv 1 \pmod{n}$$

- *Euclid's Division Theorem.* For every integer m and positive integer n, there exist unique integers q and r such that $m = nq + r$ and $0 \leq r < n$. By definition, r is equal to $m \mod n$.

- *Adding multiples of n does not change values mod n.* That is, $i \mod n = (i + kn) \mod n$ for any integer k.

- Mods (by n) can be taken anywhere in calculation, as long as $\mod n$ is taken from the final result.

- *Commutative, associative, and distributive laws.* Addition and multiplication mod n satisfy the commutative and associative laws, and multiplication distributes over addition.

- The expression "$x \in Z_n$" is used to mean that x is a variable that can take on any of the integral values between 0 and $n - 1$.

2.4 Abstract Algebra: Groups and Rings

Stallings (2010) [**?**] describes groups and rings as some of the fundamental elements of a branch of mathematics known as abstract algebra. In abstract algebra, the focus is mostly on those elements that can be operated algebraically; in other words, how two elements of a set can be combined to obtain a third element in the set. The nature of the set is defined by the operations that can be performed on the elements of the set. It has been chosen that, by convention, the two main classes of operations on set elements are usually the same as the notations for addition and multiplication on traditional arithmetic. However, there is not really a limitation on the kind of operations that can be defined using abstract algebra.

2.4.1 Groups

A **group** G, denoted by $\{G, \cdot\}$ is a set of elements with a binary operation, which is denoted by the operator $\cdot$. This operator is generic, and can refer not only to multiplication, but also to addition or some other mathematical operation. It associates to each ordered pair (a, b) of elements in G an element $(a \cdot b)$ in G, such that the following axioms are followed:

(A1) Closure: If a and b belong to G, then $a \cdot b$ is also in G.

(A2) Associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for all a, b, c in G.

(A3) Identity element: There is an element e in G such that $a \cdot e = e \cdot a = a$ for all a in G.

(A4) Inverse element: For each a in G there is an element a' in G such that $a \cdot a' = a' \cdot a = e$

It is worth noting that if a group has a finite number of elements, it is called a **finite group**, and the order of the group is equal to the number of elements in the group. Otherwise, the group is an **infinite group**. Moreover, a group is said to be **abelian** if it satifies the following condition, known as the commutative property:

(A5) Commutative $a \cdot b = b \cdot a$ for all a, b in G.

When the group operation is addition, the identity element is 0; the inverse element of a is $-a$; and subtraction is defined with the following rule: $a-b = a+(-b)$.

2.4.2 RINGS

A *ring* R, which is sometimes denoted by $\{R, +, \times\}$, is a set of elements with two binary operations, called *addition* and *multiplication*, such that for all a, b, c in R, there are some axioms that must be obeyed. It must be pointed out that a ring fulfills the same axioms as an abelian group with respect to addition; that is, axioms A1 through A5. The rest of the axioms obeyed by a ring are listed as follows:

(M1) Closure under multiplication: If a and b belong to R, then ab is also in R.

(M2) Associativity of multiplication: $a(bc) = (ab)c$ for all a, b, $c \in R$.

(M3) Distributive laws: $a(b + c) = ab + ac$ for all a, b, c in R.
$(a + b)c = ac + bc$ for all a, b, c in R.

Essentially, a ring is a set in which addition, subtraction $[a - b = a + (-b)]$, and multiplication can be done without leaving the set. Furthermore, a ring is said to be **commutative** if it satisfies the following additional condition:

(M4) Commutativity of multiplication: $ab = ba$ for all a, b, c in R.

Additionally, there are other axioms that, if obeyed, turn the commutative ring into an **integral domain**. The axioms are:

(M5) Multiplicative identity: There is an element 1 in R such that $a1 = 1a = a$ for all a in R.

(M6) No zero divisors: If a, b in R and $ab = 0$, then either $a = 0$ or $b = 0$.

2.4.3 Homomorphisms

As it has been previously noted, operations within a closed group always result in another element from the group. Only one operation is allowed in a group: addition or multiplication, for instance. There may also be two groups or sets that are made up of different elements, each with its own operation. A *homomorphism* consists of the construction of a function that *translates* elements from one group to another group with the same properties.

Group homomorphism

According to Beachy and Blair (2006) [**?**], a homomorphism is defined as follows: Let G_1 and G_2 be groups, and let $\phi : G_1 \to G_2$ be a function. Then ϕ is said to be a **group homomorphism** if

$$\phi(a * b) = \phi(a) *' \phi(b)$$

for all a, b in G_1.

Consider the example presented by Sorzano (2013) [**?**], where there are two sets: $S = \{a, b, c\}$ and $S' = \{A, B, C\}$ with the operations $* : S \times S \to S$ and $*' : S' \times S' \to S'$. The operations within each group map two elements to another element from the same group. For example, $b * c = a$ and $a * a = a$, and in the other group, $A * A = A$ and $B * C = A$. So it can be seen that there is some similarity

in the nature of the operations from both groups. Now consider the homomorphism between both groups, where:

$$\phi : S \to S'$$
$$\phi(a) = A$$
$$\phi(b) = B$$
$$\phi(c) = C$$

Therefore, the following holds true using the above homomorphism:

$$b * c = a \to \phi(b) *' \phi(c) = \phi(a) \to B *' C = A$$

What this means is that a homomorphism serves as a link to translate operations from one group to the other, keeping the same properties.

Ring Homomorphism

Similarly to groups, there are also homomorphisms for rings, the main difference being that, instead of a single operation, two operations are considered.

Let R and S be rings with addition and multiplication. The map $\phi : R \to S$ is a homomorphism if:

1. ϕ is a group homomorphism on the additive groups $(R, +)$ and $(S, +)$
2. $\phi(xy) = \phi(x)\phi(y)\ \forall x, y \in R$

2.5 Homomorphic Encryption

Homomorphic is a word that has its roots in Greek, and it means "the same shape". It is used in various areas, such as abstract algebra, which then inspired its use in

cryptography. This concept refers to the ability to make computations, i.e. operations such as addition and multiplication, on encrypted data without sharing the secret key to decrypt the data prior to the computations.

Before homomorphic encryption, any change to the secret value contained in a ciphertext required the corresponding secret key, so that it could be decrypted, applied the change, and re-encrypted. However, homomorphic encryption eliminates the need of sharing the secret key, especially with those that are not to be completely trusted, in the event that there is some change that must be applied to encrypted data. Lange (2011) [**?**] mentions several applications that would benefit from a homomorphic encryption function, given its ability to manipulate encrypted data. Such applications fall in the domain of: e-cash, e-voting, private information retrieval (encrypted databases), and cloud computing.

2.5.1 What is Homomorphic Encryption?

According to Pötzelsberger (2013) [**?**], a *homomorphic cryptosystem* is a cryptosystem which encryption function is a homomorphism, and thus preserves group operations performed on ciphertexts. Either arithmetic addition or multiplication is the group operation considered. A homomorphic encryption is additive if:

$$\mathcal{E}(x + y) = \mathcal{E}(x) \otimes \mathcal{E}(y)$$

where $\mathcal{E}$ denotes an encryption function, $\otimes$ denotes an operation depending the used cipher and x and y are plaintext messages. A homomorphic encryption is multiplicative if:

$$\mathcal{E}(x \cdot y) = \mathcal{E}(x) \otimes \mathcal{E}(y)$$

where again $\mathcal{E}$ denotes an encryption function, $\otimes$ denotes an operation depending on the used cipher and x and y are plaintext messages.

The above conditions denote that the result of the operations between both inputs, in this case x and y, is going to be the same whether the operations are performed after or before encryption. This property is what makes it possible to perform computations on an encrypted value which later can be decrypted to the correct result.

2.5.2 Classification of Homomorphic Encryption Schemes

Homomorphic encryption schemes are classified in two broad groups, depending on how they are defined, namely: partially homomorphic and fully homomorphic.

Partially homomorphic encryption schemes are defined over a group, and they support one operation at most: either addition or multiplication. On the other hand, *fully homomorphic encryption* schemes are defined over a ring, and can support up to two operations: addition and multiplication.

According to Pötzelsberger (2013) [**?**], even though somewhat homomorphic cryptosystems support a limited number of homomorphic operations, they are the building blocks for fully homomorphic encryption, and provide much more efficiency and shorter ciphertexts than their fully homomorphic counterparts.

2.5.3 Explaining Homomorphic Encryption

Craig Gentry (2009) [**?**] explains the concept behind homomorphic encryption in a very simple way. Imagine that Alice has a jewelry store and wants her workers to assemple raw materials (diamonds, gold, silver, etc.) into finished products (necklaces, rings, etc.), but does not trust them, thinking they might run off with the raw materials at their first chance. So instead of giving them direct access to the materials, Alice places the materials inside a transparent glove boxes, which are then promptly locked with a key that only Alice has. This way, workers can put their

hands inside the gloves that are connected to the box, and through them, work with the raw materials. Additionally, the workers are able to put in things inside the boxes, such as soldering iron to use on the raw materials, even though they cannot take anything out of it, since they do not have the key to open the box. Once the products are finished, Alice can then recover the products from the box using her key. The analogy is not perfect, because the workers are able to see the materials while working on them, but it is relevant because homomorphic encrytion means that they cannot take the materials with them.

Hayes (2012) [**?**] offers a proof of concept that accurately describes the very esssence of a homomorphic cryptosystem. It is shown as follows: consider that the plaintext consists of integers; to encrypt a number, double it; to decrypt it, divide by 2. Using this scheme, addition and a nonstandard version of multiplication can be done on the encrypted data. Given the plaintext inputs x and y, we can encrypt each of them separately, add the ciphertexts, and finally decrypt the result. This calculation would give the correct answer, expressed as $2x + 2y = 2(x + y)$. In other words, it is the same result as if each plaintext input were multiplied separately and then added with each other, because in the end, dividing it by two causes the same effect on both sides. Regarding multiplication, a little tweak has to be done for it to work. The product of the ciphertexts is defined as $\frac{(C_x C_y)}{2}$, while the plaintexts are multiplied by the regular formula xy. As an example, consider $x = 5$ and $y = 7$. The result of the multiplication would come as $x \times y = 35$. To perform the same evaluation between ciphertexts, both inputs would be encrypted following the rule previously defined:

$$C_x = 5 \times 2 = 10$$

$$C_y = 7 \times 2 = 14$$

The multiplication is performed as $\frac{10 \times 14}{2} = 70$, and it is divided by 2 to decrypt the ciphertext: $\frac{70}{2} = 35$, which turns out to be the correct result of the multiplication.

Certainly, the above cryptosystem is considered to be *fully homomorphic*, since it supports both addition and multiplication. However, it is definitely not a secure

one, since anyone that knows of the workings of the cipher can easily get to learn the secret message. This is also partly the reason of why the encryption and decryption algorithms are not usually kept as a secret, because it would be quite impractical to make a different encryption and decryption algorithm every time it were discovered.

Stuntz (2010) [**?**] provides an example of a cipher scheme that has an inherent homomorphic property. Consider the popular, yet insecure, ROT13 cipher, also known as the Caesar cipher. The ROT13 is a simple substitution cipher which changes each letter with the letter that is thirteen positions ahead on encryption. To do decryption, it does the opposite: each character goes back thirteen positions in the alphabet. ROT13 is partially homomorphic with respect to the concatenation operation, because it is possible to concatenate two different pieces of ciphertexts. At decryption, the concatenation of the two pieces of ciphertexts ends up to be same as the concatenation of the original messages, the plaintext. Therefore, it is partially homomorphic with respect to concatenation.

Consider, for instance, the plaintext pair $P_1 = HELLO$ and $P_2 = WORLD$, which both concatenate to "HELLOWORLD". If both plaintexts were encrypted using ROT13, the ciphertexts would be $C_1 = URYYB$ and $C_2 = JBEYQ$, and if concatenated, "URYYBJBEYQ". So if the result of the ciphertext concatenation were decrypted, it would result in "HELLOWORLD", which is exactly the same message obtained when the pair of plaintexts were concatenated originally.

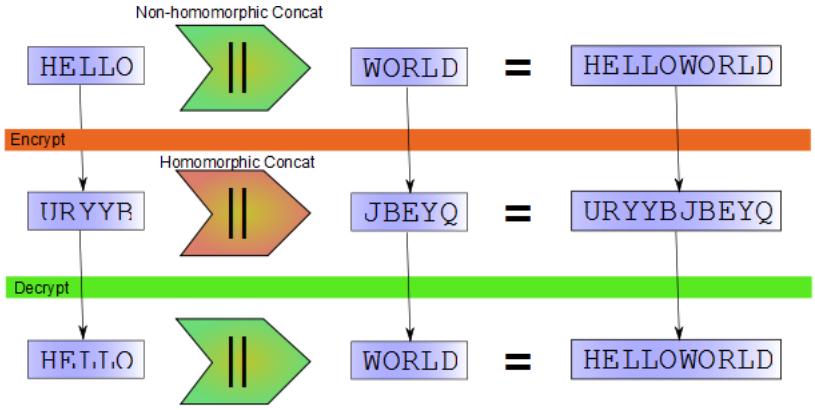

Figure 2.2 – Example of the homomorphic property in ROT13. Source: Stuntz (2010) [?].

2.5.4 The first fully homomorphic encryption scheme

In an article published by the research staff of the NSA [?], it is discussed how the term of fully homomorphic encryption has been around for about 30 years, and only recently, in 2009 to be exact, a very important breakthrough regarding fully homomorphic encryption was made by Craig Gentry. Thanks to this breakthrough, the first fullly homomorphic scheme was created, opening the doors of research to this topic that was thought to be at a dead end.

As Gentry [?] himself points out, by "fully" it means that there are no limitations on what manipulations can be performed on the ciphertext. Given ciphertexts $c_1, \ldots, c_t$ that encrypt $m_1, \ldots, m_t$ with a fully homomorphic encryption scheme under some key, and given any efficiently computable function f, anyone can efficiently compute a ciphertext (or set of ciphertexts) that encrypts $f(m_1, \ldots, m_t)$ under that key. This permits all kinds of computations on the encrypted data. Of course, no information about the plaintexts $m_1, \ldots, m_t$ or the value of $f(m_1, \ldots, m_t)$ is leaked.

The first step in Gentry's blueprint was to construct a *somewhat homomorphic encryption scheme* (SWHE), which was capable of evaluating "low-degree" polyno-

mials homomorphically. In order to obtain a fully homomorphic encryption scheme, Gentry proposed a *bootstrapping theorem* which made it possible to transform the original somewhat homomorphic encryption scheme into a *leveled* fully homomorphic encryption one.

One of the main problems that fully homomorphic encryption faces is the managing of "noise" in the ciphertext. In this case, noise refers to the gradual distortion of ciphertexts that is caused by doing operations (e.g., addition or multiplication) on them. As more operations are performed on the ciphertext, the larger the noise becomes, rendering the ciphertext undecipherable, as it would end up with the wrong result after attempting to decrypt. As Hayes [**?**] points out, each homomorphic addition doubles the noise, and each multiplication squares it. Thus, the number of operations must be limited, should one not wish to "corrupt" the ciphertext.

An obvious solution to the problem of accumulating noise is to decrypt the data and decrypt it whenever the noise begins to approach the critical threshold. Thus, the noise is resetted to its original level, allowing for more operations. The problem with this approach is that decryption needs the secret key, which is usually not going to be available to whoever is performing the operations on the ciphertext.

Instead, Gentry developed a process called *bootstrapping* to overcome the problem of noise accumulation. The *evaluate* function used in the cryptosystem can virtually perform any computation, as long as it has not reached the critical threshold of noise. Therefore, that very function is used to run the *decrypt* function, which takes as input the noise-heavy ciphertext that has gone through several operations, and the *encrypted secret key*. This makes sense because the decrypt function relies on the evaluate function, which was made to work on encrypted values. So when the *decrypt* function is run, the decrypted result is not the plaintext, but rather a new encryption, with reduced noise. This process of re-encrypting and refreshing the noisy ciphertext can be repeated as many times as needed, because then it technically has no limitations on the number of operations it can perform. However, extra layer of encryption will consequently increase the overall computational effort

required to complete the set of operations desired to be performed. As the research staff from the NSA exemplifies [**?**], if the process were to be used by Google to search the web homomorphically, the normal computing time required would be multiplied by about a trillion. This is the reason why this scheme is not practical enough as a solution for implementation.

2.5.5 Learning with Error Problems

Some problems are considered to be *hard* to solve, meaning that no polynomial time algorithm is known. However, this is a good thing in cryptography, since data encryption relies on the computational difficulty to solve problems. As the slides by Sahni (1999) [**?**] point out, many modern cryptosystems rely on the difficulty of these problems, such as RSA and Elliptic Curve Cryptography.

Blum (1993) [**?**] suggests that learning problems have certain properties which make them possible sources of cryptographic hardness. Two of these problems are the learning with errors problem, and its variation, the ring-learning with error problem.

The learning with errors (LWE) problem was introduced by Regev (2005) [**?**]; it is defined as follows:

For security parameter λ, let $n = n(\lambda)$ be an integer dimension, let $q = q(\lambda) \geq 2$ be an integer, and let $\chi = \chi(\lambda)$ be a distribution over $\mathbb{Z}$. The LWE (n, q, χ) problem is to distinguish the following two distributions: In the first distribution, one samples (a_i, b_i) uniformly from $\mathbb{Z}_q^{n+1}$. In the second distribution, one first draws $s \leftarrow \mathbb{Z}_q^n$ uniformly and then samples $(a_i, b_i) \in \mathbb{Z}_q^{n+1}$ by sampling $a_i \leftarrow \mathbb{Z}_q^n$ uniformly, $e_i \leftarrow \chi$, and setting $b_i = \langle a, s \rangle + e_i$. The LWE (n, q, χ) assumption is that the LWE (n, q, χ) problem is infeasible.

Regev proved that for certain moduli q and Gaussian error distributions χ, the LWE (n, q, χ) assumption is true as long as certain worst-case lattice problems are hard to solve using a quantum algorithm. Later, Peikert (2009) [**?**] de-quantized

Regev's results, proving that the LWE (n, q, χ) assumption is true as long as certain worst-case lattice problems are hard to solve using a *classical algorithm* as well.

At a later date, the ring learning with errors (RLWE) problem was introduced by Lyubaskevsky et al. (2010) [**?**]. The learning with errors (LWE) problem and the ring learning with errors (RLWE) problem are syntactically identical, aside from using different rings (the first one uses $\mathbb{Z}$ and the second one a polynomial ring).

Regev (2005) [**?**] gives a convenient definition of the ring-LWE problem. Let n be a power of two, and let q be a prime modulus satisfying $q = 1 \mod 2n$. Define R_q as the ring $\mathbb{Z}_q[X]/\langle x^n + 1\rangle$ containing all polynomials over the field $\mathbb{Z}_q$ in which x^n is identified with -1. In ring-LWE, we are given samples of the form $(a, b = a \cdot s + e) \in R_q \times R_q$, where $s \in R_q$ is a fixed secret, $a \in R_q$ is chosen uniformly, and e is an error term chosen independently from some error distribution over R_q. The goal is to recover the secret s from these samples (for all s, with high probability).

Because of the hardness and versatibility of both problems, they have become popular as the basis for cryptographic constructions. Regev (2005) [**?**] notes that LWE has been used as the basis of public-key encryption schemes that are considered secure under chosen-plaintext attacks, chosen ciphertext attacks, oblivious transfer protocols, and more.

To understand how a LWE can be applied in cryptography, consider the following simple cryptosystem described by Regev (2005) [**?**], which is parameterized by integers n (the security parameter), m (number of equations), q (modulus), and a real $\alpha > 0$ (noise parameter). One possible choice that guarantees both security and correctness is the following. Choose q to be a prime between n^2 and $2n^2$, $m = 1.1 \cdot n \log q$, and $\alpha = \dfrac{1}{\sqrt{n} \log^2 n}$. All additions are performed modulo q in the following description.

- **Private key**: The private key is a vector s chosen uniformly from $\mathbb{Z}_q^n$.

- **Public key**: The public key consists of m samples $(a_i, b_i)_{i=1}^m$ from LWE distribution with secret s, modulus q, and error parameter α.

- **Encryption**: For each bit of the message, choose a random set S uniformly among all 2^m subsets of $[m]$. The encryption is $(\sum_{i \in S} a_i, \sum_{i \in S} b_i)$ if the bit is 0 and $(\sum_{i \in S} a_i, \lfloor \frac{q}{2} \rfloor + \sum_{i \in S} b_i)$ if the bit is 1.

- **Decryption**: The decryption of a pair (a, b) is 0 if $b - \langle a, s \rangle$ is closer 0 than to $\lfloor \frac{q}{2} \rfloor$ modulo q, and 1 otherwise.

Such a cryptosystem is quite inefficient, but it serves its purpose to establish a link between the LWE problem and its use in cryptography.

Hayes (2012) [**?**] writes in his article that recently, Brakerski, Vaikuntanathan and Gentry have developed a variant of the learning-with-errors system that takes a different approach to noise management. Instead of stopping the computation at intervals to re-encrypt the data, they incrementally adjust parameters of the system after every computational step in a way that prevents the noise level from ever approaching the limit.

2.5.6 Brakerski-Gentry-Vaikuntanathan scheme

The Brakerski-Gentry-Vaikuntanathan (BGV) scheme (2011) [**?**] is a leveled fully homomorphic encryption that does not rely on bootstrapping, but rather employs it as an optimization. However, it still follows Gentry's blueprint to build a fully homomorphic cryptosystem, thus lattice-based cryptography is employed, albeit with certain improvements. It is described as *leveled* in the sense that the parameters of the scheme depend (polynomially) on the depth of the circuits that the scheme is capable of evaluating.

This scheme dramatically improves performance compared to previous attempts at fully homomorphic encryption, and also bases security on weaker as-

sumptions. The security of the BGV scheme is based on the ring-learning with error (RLWE) problem that has a 2^{λ} security against known attacks.

As it happens with the first fully homomorphic encryption scheme proposed by Gentry (2009) [**?**], making computations on the ciphertext creates a noise that gradually accumulates. Specifically, performing one addition between two ciphertexts roughly doubles the noise, while multiplication squares it. Decryption succeeds as long as the magnitude of the noise does not surpass a certain threshold. However, the BGV scheme proposes a *noise-management technique* that keeps the noise in check by reducing it after doing homomorphic operations, without depending on the bootstrapping technique previously mentioned. The noise management technique is called *modulus switching*, and it was developed by Brakerski and Vaikuntanathan (2011) [**?**], albeit with a modification.

The lemma that describes the original modulus-switching technique says that an evaluator who does not know the secret key s, but instead only knows a bound on its length, can transform a ciphertext c modulo q into a different ciphertext modulo p while preserving correctness. This is expressed as $[\langle c', s\rangle]_p = [\langle c, s\rangle]_q \mod 2$. The interesting property found here is that if s is short and p is sufficiently smaller than q, then the "noise" in the ciphertext actually decreases, meaning that $|[\langle c', s\rangle]_p| < |[\langle c, s\rangle]_q|$. This technique allows the evaluator to reduce the magnitude of the noise without knowing the secret key, and without depending on bootstrapping. The BGV scheme takes the aforementioned technique, and further improves noise management by making use of a *ladder of gradually decreasing moduli*.

This homomorphic scheme has been implemented as a software library in C++, called HElib [**?**]. The implementation focuses on the effective use of the Smart-Vercauteren ciphertext packing techniques and the Gentry-Halevi-Smart optmizations. Halevi and Shoup [**?**] describe in a paper the algorithms employed on the implementation, as well as some considerations to have in mind depending on the hardware used.

2.5.7 Applications

As it was previously mentioned, various individuals and corporations are hesitant to make use of cloud services, mainly because of the security risks it poses, such as privacy loss and tampering of the data. Traditional cryptosystems can protect confidentiality if the only goal is remote storage; but it is restricted to just that. However, if the encryption employed were homomorphic, the cloud server where the data is stored would be able to perform meaningful computations on the encrypted data. The scenenarios where homomorphic encryption can be applied are many, and some of the most important areas are: medical applications, financial applications, advertising and pricing, electronic voting, data mining, and biometric authentication.

Medical applications.- A straightforward application of homomorphic encryption in a medical context is the storage and processing of medical records of patients. Most of the time, these records are considered private, but a hospital would be greatly benefited if it could rely on a cloud service rather than setting up its own data center. If homomorphic encryption has been used as a cipher to protect the medical records, then there could be meaningful pieces of information that could be extracted from it. For example, if a doctor in another clinic was granted access to the patient's medical record, he could make use of a function that makes homomorphic evaluations on the encrypted record to obtain statistical data from the patient. And this would be possible without the cloud server from learning what information the medical record contained.

Financial applications.- Data about corporations, such as their stock price, performance or inventory is often considered to be private, and also very relevant towards making investment decisions. There are many kinds of computations that could be done on this private data, such as running predictive simulations on stock price performance.

Advertising and pricing.- Nowadaways, user activity is heavily monitored by the

websites that he browses, and uses this data as to build a profile and direct to him appropriate ads. Take Amazon as an example: amazon keeps a history on what items the user is looking at, even if they were not bought, for further advertising. The user might get emails advertising the same or similar products, or might even see the same ads on unrelated sites, such as Facebook [**?**]. The above example is just the tip of the iceberg. There is a lot of contextual information that can be retrieved from the user to gain insight on his interests and current status, without him having to be active. For example, he could be suscribed to a service that sends alerts to his smartphone whenever he is walking nearby a shop that could be of his interest. Geolocation and user profile are pieces of information that are considered to be private, the user might not subscribe to the service if it just shared around his information. If, however, he is guaranteed that his information is being kept encrypted, and that it stays that way whenever a computation is made to see if there is a nearby store of his interest, then he might not be as reluctant to make use of the service. The applications on advertising using homomorphic encryption are diverse, because there are pieces of data that can be obtained passively, such as the user's geolocation or browsing habits.

Electronic Voting.- Although not necessarily related to cloud computing, electronic voting is another area where homomorphic encryption can make such voting scheme feasible and convenient. Voting could be done from anywhere as long as the participant is provided with Internet access. Simply put, homomorphic encryption can be used to hide the content of a ballot by calculating the total number of votes, without having to decrypt any of the ballots. However, electronic voting is not as trivial as it seems, because apart of confidentiality, there are other security parameters that are equally important, such as: correctness of the tally, democracy, robustness, verifiability, fairness, and verifiable participation. Therefore, applying homomorphic encryption would only satisfy one of the many requirements for electronic voting to be feasible.

Data mining.- Closely related to advertisement, data mining has several issues on privacy. Using homomorphic encryption would mean that certain sensitive attributes of customers data can be protected from unauthorized parties. Yang et al. (2014) [**?**] have proposed a solution that allows a data miner to compute frequencies of values or tuples of values in the customers' data in a privacy preserving manner. The advantage of this solution is that, unlike other existing attempts to make data mining more private, is that this solution is both fully private and fully accurate, without compromising one in exchange of the other.

Biometric Authentication.- Biometric authentication is an area of application quite different from the rest. This is because biometric data is assumed to be public, since fingerprints virtually on every object touched by a person. Therefore, according to Pötzelsberger (2013) [**?**], what is considered to be private is the relationship between the biometric data and the person from whom it comes. It can be applied on the scenario where the user tries to authenticate using one of his biometric features: the biometric feature would be stored and encrypted before it looks for a match in the database. The homomorphic evaluation would consist in the comparison between the user's biometric feature and all of the existing rows in the database. Traditional cryptographic would not be appropriate for this task, since biometric data can vary slightly each time, so an exact match would not work out. Assuming that the biometric feature is stored as a binary string, the homomorphic evaluation would consist in computing the Hamming distance between two feature vectors, and a predefined threshold to achieve the comparison. This is how the biometric data itself can stay public, but the relationship of whom it belongs to is kept private.

2.6 Services in the cloud

The cloud, a trendy term used to describe a network of servers usually accessible through the Internet, currently provides two important services: storage and computation. As Xu et al. in (2011) [**?**] point out, cloud storage services, such as Dropbox, Skydrive, Google Drive, and Amazon S3 have become very popular in recent years. As the use of these services become more common, more sensitive information is kept in the cloud, such as emails, health records, private videos and photos, company finance data, government documents, etc.

Li et al. (2009) [**?**] introduce cloud computing as a new term that views computing as a utility which enables convenient, on-demand access to computing resources that can be rapidly deployed with minimal management overhead and great efficiency. Cloud computing has the potential to benefit its users in avoiding large capital expenditure used for deployment and management of both software and hardware, such as setting up a data center.

All of these services are opening up a new era powered by the software as a service (SaaS) computing architecture. One of the main advantages of storing this information in the cloud is that the owners are relieved from the burden of data storage and maintenance at all times. There are many plans that adjust to the on-demand needs of the owner of the data, which is way less costly than building and maintaining a whole data center. This implies that the clients of these services can expect to gain reliability and availability from having the data stored remotely. On the other hand, the data is necessarily brought out of their control and protection. This becomes a risk for the owner of the data, since the information stored in the cloud could potentially be accessed by unauthorized individuals, such as competitors or malicious attackers. There also exists the possibility that the information stored could be tampered in some way, thus compromising its integrity. Therefore, the computing and storage service providers are usually not trusted completely, which is why it has been difficult for the public and organizations to fully adopt the use of

these services.

The aforementioned issue brings up the need of having a mechanism that verifies that a cloud provider is storing the whole database intact, even the portions that are rarely accessed. Ateniese et al. (2014) [**?**] point out that, fortunately, a series of *proofs-of-storage* protocols have been proposed to solve this problem related to data integrity. Xu et al. (2014) [**?**] define a proof of storage (POR or PDP) as a cryptographic tool, which enables a data owner or third party auditor to audit integrity of data stored remotely in a cloud storage server, without keeping a local copy of data or downloading data back during auditing. In other words, applying a proof of storage in a cloud server can guarantee that the data has not been tampered with; and in the event that it is, most likely the service could be held liable for damages to the data. A huge advantage of using a proof of storage is that the data does not have to be downloaded to confirm its integrity, which makes it ideal to make use of it routinely.

One of the simplest use cases for cloud computing is when a user independent from the cloud wants to store some data in it for later retrieval. In this simple case, data confidentiality and integrity can be trivially ensured. For that, typical cryptography primitives can be usedby encrypting the user's data before it is sent for storage in the cloud. To ensure confidentiality, the key used for encryption is kept secret from the cloud provider. However, real life applications are not as simple. Damgard et al. (2013) [**?**] mentions that the cloud is more than a storage medium; in particular, computation over the stored data is outsourced to the cloud. Sometimes it might even be distributed among distinct cloud servers geographically apart, operated by different parties. Wang et al. (2009) [**?**] describe the nature of the problem at length, because usually a cloud service is not solely restricted as a data warehouse. The data stored in the cloud may be frequently updated by the users, including insertion, deletion, modification, appending, reordering, etc. This kind of complex scenario is what makes the use of traditional cryptographic primitives so difficult to adapt, especially because the data is not stored in a single cloud server,

but is instead redundantly stored in multiple physical locations to further reduce the data integrity threats. Therefore, making use of cloud services highly improve data avalability and storage flexibility, data confidentiality and integrity is put at risk.

In order to tackle these issues, there have been many attempts to make cloud services, especially cloud computing, more secure. These attempts are very different in nature, though very similar regarding its goals. These are *secure multi-party computation* and *homomorphic encryption*. David, Nishimaki, et al. (2015) [**?**] describes secure multiparty computation (MPC) as the means to allow mutually distrustful parties to compute functions on private data that they hold, without revealing data to each other. There are many methods and approaches proposed to perform secure multiparty computation, although only recent attempts are considered to be efficient enough for application. On the other hand, homomorphic encryption is based on the idea of performing computations on the ciphertext that has gone through an encryption algorithm, without having to decrypt it beforehand. This detail makes homomorphic encryption a good technique for cloud services, since only one authorized user is able to decrypt the ciphertext after the necessary computations have been performed on it.

Chapter 3

Related Works

Pending ....

Chapter 4

Methodology

The methodology in this work describes all the considerations and steps taken to build a solution to the problem analyzed from the case study. As such, the methodology has a very close relationship with the case study. The problem studied addresses a situation where a person stores sensitive data on the cloud, with the intention of accessing and modifying said data at a later time. The solution itself is an *implementation of a client-server based software* that makes use of a library that enables homomorphic encryption called HElib.

Even though the object of study is the application of homomorphic encryption in the cloud, it is not trivial to find a fitting situation where it can be applied. The search becomes more complex as the limitations of state-of-the-art homomorphic encryption tools are considered. The case study that was thought of is simple enough so that the currently available tools can be used effectively, and complex enough so that applying homomorphic encryption is compelling and better-suited than other alternatives.

This section discusses the case study and the means to offer a solution based on the cloud. Then, it describes a proof of concept in a general way as a stand-alone solution. The tasks required for the proof of concept to work are briefly described, and some considerations regaring key management are noted. Afterwards,it mentions the scenario where a client-server architecture is employed, discussing some decisions that were made regarding the communication between the client and the

server. Finally, it mentions how the aforementioned tasks are being split between each component.

4.1 Case Study

The case study considered for this work is explained as follows: commonly, a household has more than one member, e.g. partner and children. The responsible *householders* (i.e. the parents) might want to keep an eye on the house while being away, especially if their children are left behind. They would like to ensure that their children stay inside during this time, and that nobody else, except for a babysitter, enters the house. And if it were the case that somebody got in or out, they would probably desire to be promptly notified of it. This holds true especially when they are away for long periods of time, during a vacation, for example. A traditional solution consists in setting up a surveillance system throughout the house, which would certainly work to prevent robbery, but wouldn't necessarily work as a measure to know whether or not the children have left the house. Additionally, such a system usually requires a montly or yearly fee. Therefore, setting up a surveillance system might be too costly for some families, and not be quite adaptive to their needs. It might be more effective to deploy a system that detects when people enter and exit the building through the main doors, and keep a counter of it.

The main idea is that the *surveillance counter* initializes at some point in time, and, as people go in and out, the counter increases or decreases, respectively. Setting up the sensors and other pieces of hardware can be considered as a Do-It-Yourself project, as resources could easily be found. Wilson [**?**] mentions several considerations to take when tracking people inside a building using binary sensors. In this case, however, it is simpler to set it up, as it is not necessary to keep track of the people inside the building: only if they go in our out past the main doors.

The issue comes out when the recorded data, that is, the counter, is to be accessed or modified securely over the web.

4.2 Proposed Solution

The proposed solution addresses the issue hinted at the end of the study case description. The counter data that is recorded by the sensors is to be sent a server somewhere in the cloud, where it will be stored indefinitely, awaiting for the data to be accessed or modified in the future. It might not be appropriate to have a dedicated server at home for this functionality, which is why the computing task is instead delegated to the cloud. The data stored in the cloud server can be accessed from anywhere else; however, one of the most important aspects of the whole scheme remains unaddressed: confidentiality.

Usually, the householder would not allow others to know about the actual counter, including the cloud server itself, as this information is considered to be *sensitive*. The traditional approach is to make use of public-key cryptography to *encrypt* the data and prevent from anybody else to know the counter. Once the counter data has been encrypted successfully, it can then be safely uploaded to the server in the cloud. Then, when the householder desires to view the value of the counter, he would have to download the data and *decrypt* it in order to view it. This sounds reasonable until it is considered to apply changes to this value, since it would have to go again through the encryption process. Indeed, this would imply that every time there is a change in the counter, the whole ciphertext would have to be reuploaded to the server. Considering that person comes in and out of a building quite frequently, it would turn out to be a heavy burden that carries an overhead cost, computational and bandwidth-wise, that keeps accumulating every time there is an update.

An alternative approach is to simply notify the cloud service of any changes that occur, so that it performs the addition or subtraction itself instead of relying on receiving the whole encrypted result. Even though this would not be possible to do with traditional cryptography, it can be done using *homomorphic cryptography*. The concept behind this approach is that, while the initial counter value is being

stored on the server, the client part that resides on the household sends over any change of individuals that come in our outside the building. Thus, the cloud service performs the appropriate homomorphic computations on the currently stored value. This occurs without the need of reuploading the counter data, as the cloud service has no need of decrypting the data in the first place. Since the full ciphertext is not being re-sent, the overhead costs are reduced significantly.

Whenever the householder needs to know the value of the counter, he can authenticate with the cloud service to download the current ciphertext that represents the counter. The ciphertext can decrypted using the private key generated at the beginning of the process. This aspect remains the same whether homomorphic encryption is used or not, and there is no way around it without compromising confidentiality. Considering that a system that detects when people come in and out might not be perfect, and if somehow the counter gets to an incorrect value, it could be reset to the correct amount by the user as required. In this case, it would be unavoidable to send a newly encrypted counter value.

The solution itself consists in an implementation of a client-server software that aims to store and modify securely a counter by making use of homomorphic encryption. The various tasks that help reach this goal are distributed by the client and server components of the software. The client focuses on the initialization of the homomorphic encryption scheme, generation of public and private keys, encryption of initial counter data, among other things. On the other hand, the server is dedicated to store and manage the public key of the household and its current counter value. Both parts of the software utilize a library called HELib, which makes it possible to run homomorphic evaluations. In other words, this library is used perform basic operations on the data, such as addition, subtraction, and multiplication. Consequently, it is also used to generate the public and private keys, encrypt data, and decrypt it. Once the encrypted counter is stored in the server, a different kind of client, i.e. the householder, can then ask to download and decrypt it. It makes sense to say that it will be a different kind of client, since most likely the interested

person would not be at the household, i.e., place where the counter was initialized.

4.3 Details of the HELib library

An implementation of an homomorphic encryption scheme, the Brakerski-Gentry-Vaikuntanathan (BGV), is openly available as a C++ library, called HELib. Using this library, it is possible to encrypt integer values and perform operations on them while being encrypted. There are many parameters to be considered when using this implementation, and these usually define how the public and private keys are going to be like. For this particular application, most parameters can be left in their default values, as they seem to serve the intended purpose.

Part of the description found in the code repository of HElib states that the library is considered to be low-level, and given its difficulty and constant changes, it is not that appropriate to build big applications with it. 'At its present state, this library is mostly meant for researchers working on HE and its uses. Also currently it is fairly low-level, and is best thought of as "assembly language for HE". That is, it provides low-level routines (set, add, multiply, shift, etc.), with as much access to optimizations as we can give.' Brakerski et al. [**?**]

It is important to note that the library recently started to support multi-threading, which would considerably speed up the evaluation times of the homomorphic operations. Not too long ago it started to support bootstrapping as well, a technique that prevents the ciphertext from getting too much noise and that eventually results in the incorrect homomorphic evaluations.

4.4 Setting up the parameters and context

The HElib library is an implementation of a homomorphic encryption scheme, and as such, it has certain requirements before it can encrypt data or even generate keys. First of all, it has a list of parameters which must be manually set. Depending on

the parameters applied, some aspects that directly affect the keys and ciphertext are altered. The suggested values are used for the majority of the parameters. Altering the values of the parameters could be an interesting way to do experimentation with the key size and computation time required.

The following table briefly describes each parameter used in HElib. It does not go into much depth, as those details are mostly pertaining to the algorithms in the library.

R	number of rounds	default=1
p	plaintext base	default=2
r	lifting	default=1
d	degree of the field extension	default=1
c	number of columns in the key-switching matrices	default=2
k	security parameter	default=80
L	number of levels in the modulus chain	default=heuristic
s	minimum number of slots	default=0
repeat	number of times to repeat the test	default=1
m	use specified value as modulus	optional
mvec	use product of the integers as modulus	optional
gens	use specified vector of generators	optional
ords	use specified vector of orders	optional

Cuadro 4.1 – Parameters used in HElib

4.5 Key Generation and Serialization

As the scheme implemented in HElib is based on public-key cryptography, it requires the use of public and private keys. The owner of the data has to generate his own set of keys as the first step, and it is only the public key which he should be willing to share. The user then shares his public key with the cloud service by some means of

registration, so he does not have to share it every time he makes use of the service. The public key is used to encrypt the counter data for the first time before sending it over to the server, and the server itself requires this public key in order to perform any homomorphic operation. The generation of the keys is pseudorandom, so extra care should be taken as to not use a predictable seed, so that the public and private keys are not recreated by another party.

In order to generate a set of keys, a component called a *context* needs to be instantiated using certain parameters. Firstly, the public key is obtained by simply instantiating an object of the class *FHESecKey*. Then, a copy of the public key is made as the foundation for the private key. This exact copy applies a method called *GenSecKey(w)*, which takes w as a seed to create the real private key. Finally, it goes through a process that computes key-switching matrices, which are used in the internal algorithms of the library to create the secret key. The code for these actions are shown below:

```
FHEcontext* context;
FHESecKey* secretKey;
FHEPubKey* publicKey;

context = new FHEcontext(m, p, r);

publicKey = new FHESecKey(*context);
secretKey = publicKey;

secretKey->GenSecKey(w);
addSome1DMatrices(*secretKey);
```

Before proceeding to encrypt the data, it is important to have the public key serialized and ready to send it to the server. Serialization is a popular term used to describe the encoding of objects and the objects reachable from them, into a string of bytes. Serialization is a term introduced by Oracle [**?**], and it is commonly used

for lightweight persistence and communication via sockets. In this case, it is used as the means of temporarily storing the public key in a stream of bytes so it can be shared with a remote server that will be able to read such stream. Only after the contents of the saved object have been fully read, it will be possible to reconstruct the public key. Fortunately, the HElib provides a very simple way to do serialization, since the class *FHESecKey* supports the «operator which works quite nicely with a couple popular serialization classes in C++ called *istringstream* and *ostringstream*. The use of this class is showed as follows:

```
ostringstream pkstream;
pkstream << *publicKey;
```

This way, the data stored inside the public key object has been stored as a stream of bytes in a highly portable object from the *ostringstream* class. Using this newly populated object, the public key can be shared using sockets.

4.6 Encryption of the Counter Value

In order the protect the value of the counter from being known from other parties, including the server itself, it must be encrypted. This means that an encryption algorithm, provided by the HElib, is to be used on the plaintext that represents the counter. The result of this process is a *ciphertext*, which, unless provided with the corresponding private key, cannot be deciphered back to its plaintext form.

Following the conventions defined in the HElib, objects from the *EncryptedArray* and *PlaintextArray* are initialized using the *context* and G as parameters which were previously set. *PlaintextArray* can be seen as a container for the plaintext, whereas *EncryptedArray* is seen as a general container that works as an medium between the plaintext and ciphertext data. Once it has been done, a method called *encode* of the same class can be used on the PlaintextArray object to prepare the data for encryption. The ciphertext is stored in a different kind of container, which

comes from the class *Ctxt*. This container is initialized using the public key as argument. Afterwards, the encryption of the plaintext data can be done using a method of the *EncryptedArray* object simply called *encrypt*. This method receives as arguments the objects of the plaintext and ciphertext containers, and the public key. Therefore, even though *PlaintextArray* and *Ctxt* are containers for either the plaintext or ciphertext, respectively, the *EncryptedArray* is the medium that supports the translatation from one to the other.

```
EncryptedArray ea(*context, G);
PlaintextArray counter(ea);
counter.encode(people);
Ctxt& encryptedCounter = *(new Ctxt(*publicKey));
ea.encrypt(encryptedCounter, *publicKey, counter);
```

This ciphertext can be used in several ways: its value can be modified by adding, substracting or multiplying it by some arbitrary value, it could also be decrypted at any time using the private key, or it could be serialized as it was done with the public key, so another party, e.g. the server, can receive and store it.

4.7 Operation Flow: Single Component

Before attempting to break down the whole implementation in two parts for both the server and client, a proof of concept is implemented which combines the tasks of both parts into a single program. The flow of this program represents how the data gets transformed through several tasks.

1. Parameters to use with the HELib are arbitrarily chosen, while other required values are computed.
2. Context of the HELib library is set.
3. Public and private keys are created using the context.

4. Optionally, public key can be serialized into a file so it can be used later.
5. The structures to store plaintext and ciphertext are declared.
6. The initial counter value is set into the plaintext structure.
7. The plaintext is encrypted using the public key and stored into the ciphertext structure.
8. Values are arbitrarily added or substracted from the ciphertext.
9. Ciphertext is decrypted using the private key, and its value is stored in another plaintext structure.
10. Newly decrypted plaintext is printed to verify correct result from operations.

4.8 Software Architecture and Data transmission

Even though the case study started out considering cloud services, an actual implementation of the application can be addressed with a client-server architecture. The client-server architecture is a popular model that consists of a two parts: a client and a server. Usually, the server just waits for any kind of request from a client to do some kind of task. Meanwhile, the client usually starts some kind of task, but depends on a response from the server to complete. Most of the time, the client depends on the server because it might have something that the client does not, like a database.

In this case, the implementation of the counter application is approached using a client-server model. The server represents the cloud service that stores, serves, and modifies the encrypted data on request, while the client plays the part of obtaining and encrypting the sensitive data. The client also takes care of tasks such as initializing the components defined by the library, generating the public and private

keys, encrypting the data, and serializing the encrypted counter before sending it over to the server. Meanwhile, the server takes care of establishing the communication details via TCP sockets, reconstructing the received ciphertext, and performing operations on it. Although a more secure implementation would consider using the Secure Socket Layer (SSL), it is simplified to cover only the details pertaining the use of homomorphic encryption.

Once the user obtains the initial data, i.e. by counting the number of people currently in the house at a given time, he proceeds to feed the data to the client program, so it is encrypted using the public key. After the counter data has been encrypted successfully, the data is then sent to the server via sockets. There are two possible protocols that can be for this part: UDP and TCP. UDP stands for User Datagram Protocol and is used in conjunction with the Internet Protocol. UDP works best when the data units that are being sent are very small. It is also a little bit problematic considering that it does not reassembles data packets once they have arrived at the destination. The other option that was considered was TCP, which stands for Transmission Control Protocol. It is a little friendlier to use in the sense that it reassembles datagrams in the correct order once they arrive at their destination. However, it adds a little bit of overhead cost since it adds a header section per segment. For this scenario, it is preferred to use TCP instead of UDP, especially since it might become complicated to keep track of the order that the data packets arrive.

4.9 Sending and reading serialized data

The serialized data that is sent through sockets is sufficiently large so that the operation cannot be done with a single function call, which is why a few considerations have to be made in the implementation. It is recommended to have some kind of supporting function that continuously is attempting to send data to the server, until there are no bytes left.

The following snippet of code shows how the ciphertext is serialized using the *ostringstream* class, and then a function call is made to a supporting function which continuously attempts to send all the remaining bytes to the server. It is relatively simple to keep track of how many bytes are being sent and how many are left, which is why it stops sending data when there are no bytes left.

```
ostringstream oss;
oss << encryptedCounter;

if(sendalldata(sockfd, oss.str().c_str(),
  &msgsize) == -1) {
    printf("ERROR. Only \%d bytes sent! \n",
      msgsize);
}

int sendalldata(int s, char const* buffer, int *len)
{
  int total = 0; // bytes sent
  int bytesleft = *len; // bytes left to send
  int n;

  while(total < *len) {
    n = send(s, buffer+total, bytesleft, 0);
    if (n == -1) { break; }
    total += n;
    bytesleft -= n;
  }
  *len = total;
  if(n == -1)
    return -1;
  else
```

```
    return 0;
}
```

As it cannot be expected that the serialized ciphertext of great size will be succesfully sent with a single send function call, the same goes for the receiving end. Certain measures have to be taken so that all the data is received completely and in the appropriate order.

The following piece of code extracted from the server side where the ciphertext is being received. First of all, a buffer is declared and initialized with the size of the ciphertext, which was previously received. A function called *bzero()* is used to prepare and empty the buffer. Then, a *while* loop is done as long as there are bytes remaining to be received. As more iterations go by, more data is appended to the buffer. A couple of variables are used to keep in check how many bytes have been read and how many are still pending to be read.

```
char* responseBuffer = new char[responseBufferSize];
bzero(responseBuffer, responseBufferSize);
bytes_read = 0;
int bytes_remaining = responseBufferSize;
int this_recv;
while(bytes_remaining > 0) {
  this_recv = recv(newsockfd, responseBuffer+bytes_read,
              bytes_remaining, 0);
  if(this_recv <=0) error("error on receive");
  else {
    bytes_remaining -= this_recv;
    bytes_read += this_recv;
  }
}
```

4.10 Reconstructing the Ciphertext

It was mentioned previously that serialization enabled persistence of the public key and ciphertext and made it easier for the object data to be sent via sockets. However, the serialized objects cannot be used the way they are received. In this case, the ciphertext was serialized before being sent to the server, and once it is properly received, the data must be used to reconstruct the original ciphertext object. If this is not done, there is no way to perform operations on the ciphertext, or even decrypt it to know its value.

The following piece of code describes the steps to be taken in order to reconstruct the object that will contain the ciphertext. First of all, the buffer that was previously filled with data is copied into a string variable, then it goes through a couple of steps so it can be used as a *istringstream* object. The preparation seems complicated, but the reconstruction of the ciphertext object is actually quite simple. Since HElib is friendly with the »operators, it can be done in one step. This operator basically takes the data from one variable and puts it in another kind of variable, one being a *istringstream* object, and the other a *Ctxt* object.

```
string strBuffer((const char*) responseBuffer, bytes_read);
istringstream serialCipher;
serialCipher.str(strBuffer);

Ctxt receivedCipher(publicKey);
serialCipher >> receivedCipher;
```

4.11 Changes in the counter

Once the initial value of the counter is received by the server, there are two possible scenarios that could occur next: the value gets viewed by the householder, or a change in the counter occurs. Whenever a significant change is observed by the

client program, the change is promptly notified to the server, which could be either imply that the value goes up or down. When the cloud service receives the value of change, it either adds or subtracts it from the previously stored value. The change can then be known to the user by retrieving the encrypted counter.

Bootstrapping is an advanced technique in fully homomorphic encryption which allows more homomorphic evaluations to be run on a ciphertext. As it has been mentioned, whenever a homomorphic evaluation is performed on the ciphertext, a small amount of noise is added to it, and after a certain threshold, the ciphertext becomes unusable, in the sense that it no longer decrypts back to the correct value. As an alternative to bootstrapping, however, it is possible to reset the counter, i.e. upload a newly created ciphertext with an initial counter value, so that more homomorphic evaluations can be performed on it. Considering that the operations that will be done most of the time are addition and subtraction, this is not something that will occur often, but eventually will. As of December 2014, HElib included bootstrapping as an optimization of the scheme, which would increase the number of operations that can be done on a ciphertext before the noise makes it impossible to decipher the encrypted value. However, as the implementation of this counter system was started before December 2014, it was not considered to make use of bootstrapping.

4.12 Operation Flow: Client and Server

The proof of concept that makes use of homomorphic encryption ends up being completely linear in the sense that it has no communication at all with different parts of the outside world. The case study implies that there is a set location where the counter is initialized and uploaded to the cloud, and a variable location where the value of it is accessed. Therefore, the proof of concept is translated into a client-server architecture, where the tasks are delegated accordingly.

In the following figure, a breakdown of the client-model architecture employed is depicted. As previously mentioned, the solution is split into two parts: client and

server sides. Each side has specific tasks that must be performed independently. The figure was conceived as an attempt to organize the software structure and assign a functionality to one of the components. The client side is in charge of the tasks that are performed at the place where the counter is initialized, i.e. the household. Such tasks include: setting up the parameters and context required by the HElib, generating the set of public and private keys, registering the public key, encrypting the initial counter value, serializing the ciphertext, recording changes in the counter value, and decrypting the received ciphertext. The following illustration depicts the client-server architecture that was conceived.

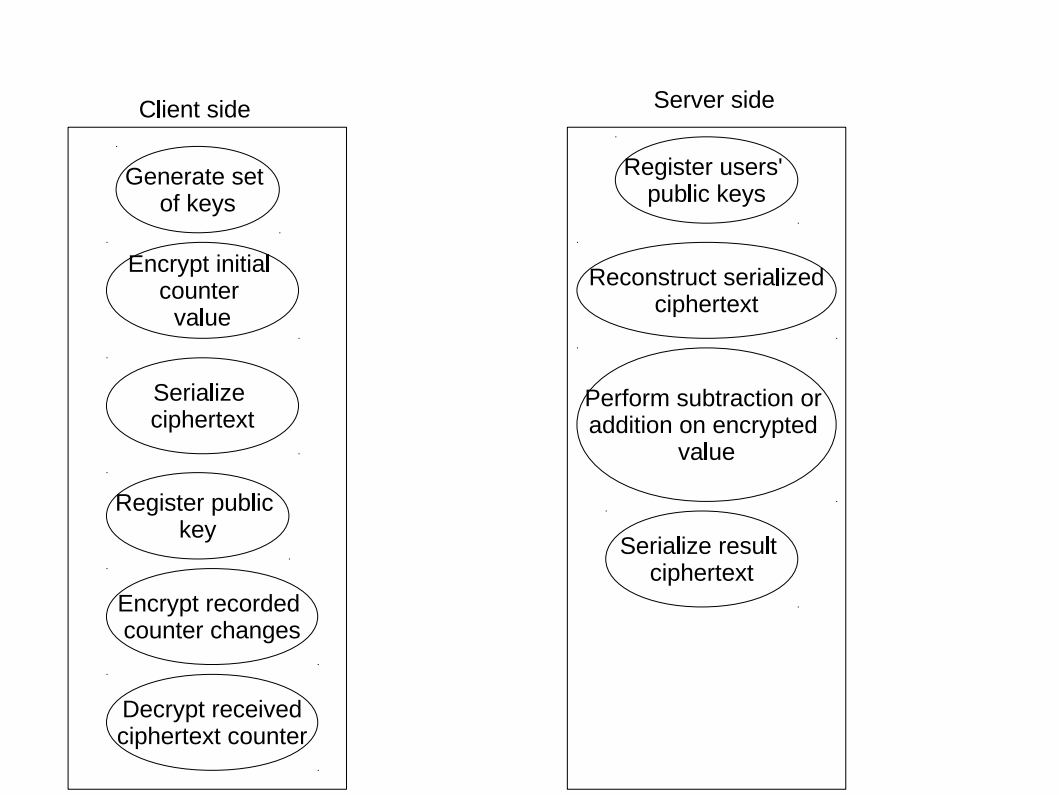

Figure 4.1 – Architecture breakdown into client and server.

In order to understand how the workflow occurs inside of the client-server architecture, it is better to think of two possible clients: one that is located at the household, and another one that requests access to the counter data. The sensor system is the former one, whereas the user is the latter. Usually, the client that

retrieves data from the sensor begins by generating a set of keys and registering it with the server, and then encrypting the value of the initial counter before sending it over. Whenever a change of the value is notified to the server, it performs the corresponding operations: addition if people go inside, or subtraction if people go outside the building. Finally, whenever the user wants to learn the value of the counter, it can request to download it from the server. It is also possible for the user to reset the value of the counter as he sees fit. This flow of operation is shown in the following sequence diagram.

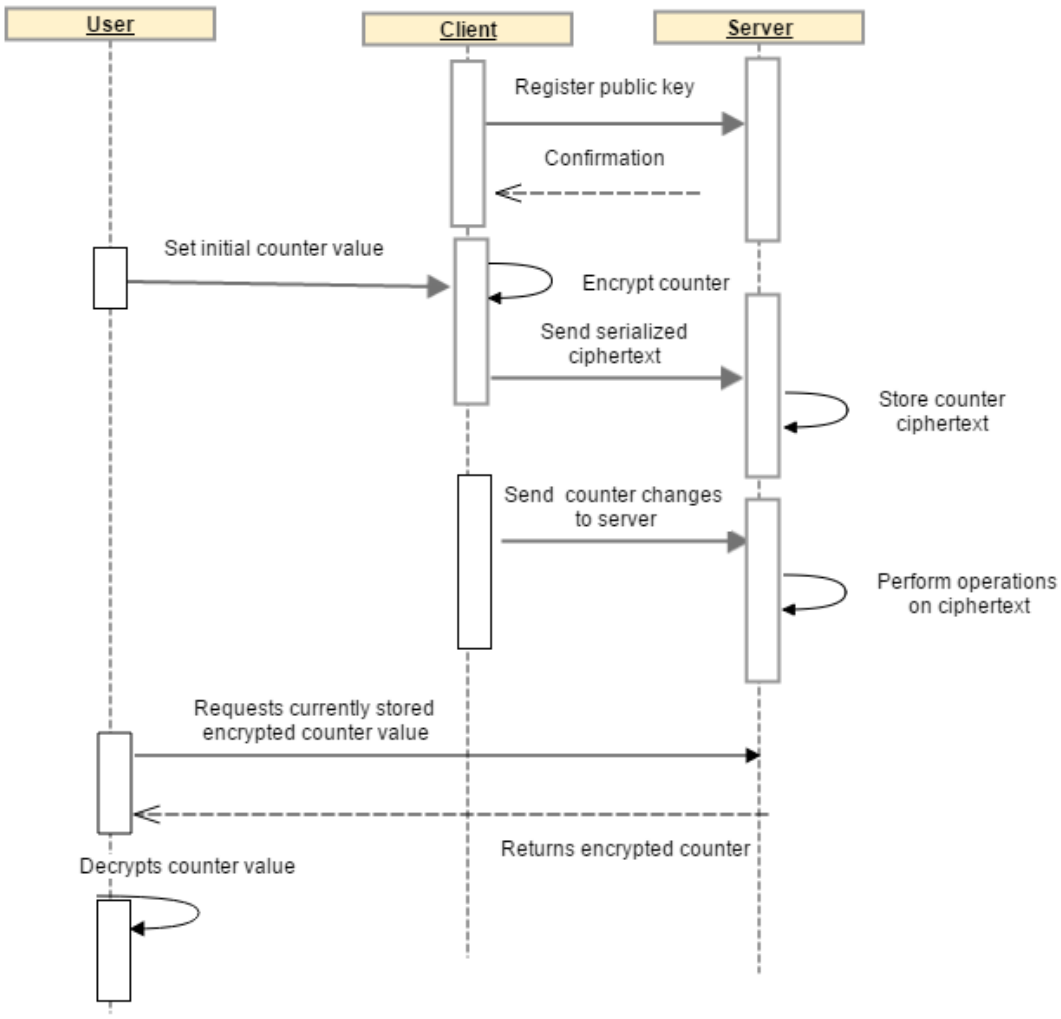

Figure 4.2 – Sequence diagram showing operation under normal conditions

4.13 Discussion

The aspect of user registration is not part of the planned implementation itself, but it is recommended to consider how to handle different users. There might many more than a couple of households that make use of such a counting service. This is why each householder would have to register his public key and go through some kind of authentication mechanism every time they restart or download the counter data. It is especially important to take the appropriate measures so that somebody else does not reset the counter value of a household that is not theirs, because it would be chaotic when the householder looks at the counter and finds a value that does not represent reality. To keep the registration under control, it might be ideal to keep a list of registered users either on a database or on a simple file for each household which would contain basic pieces of information such as an email address or username, along with the hash value of a password and the appropriate public key for the household that is being registered.

There were several problems at this stage that were directly related to the implementation. The transmission of serialized data between the client and server was especially problematic. The early attempts at sending serialized data to the server were not quite successful. Many different kind of errors were found, and the most common one was related to memory allocation. After going through several trial and error attempts, it was found the errors started appearing as the message got larger. This seemed to be inevitable as the default settings of a TCP blocking socket were not adequate to send large pieces of data.

After doing the right adjustments, the problem seemed to go away, except that *the result was not quite as expected.* It was known inmediately that something was up with the received ciphertext, as it could not decrypt properly. It helped to look at the data that the server received, and it was quite evident that it was not complete. Indeed, only certain fragments of the ciphertext were received successfully, which led to an incorrect decryption. Several adjustments were made to the process of

receiving data, so that a couple of control variables were used to keep in check how many bytes were being received. It was set so that it would only stop reading data from the client until all of the bytes had been read and assembled in a buffer. Both the functionalities of reading and writing the ciphertext on a socket were relatively more complicated considering that it was done without the support of a framework that specialized on it.

In summary, a description of the case study was given, a solution based on homomorphic encryption was proposed to tackle the problem found in the case study, details of the HElib were discussed, and all of the relevant considerations pertaining to the implementation were explained. These considerations took on points such as the parameters and context, key generation and serialization, software architecture, transmission of serialized data, reconstruction of the ciphertext, and applying changes to the counter. Finally, all of these tasks were addressed when considering a client-server implementation.

Chapter 5

Experiments and Results

This section describes the design of the experiments related to the use of fully homomorphic encryption in HELib, as well as a summary of the results obtained. The objective of the experimental design was to test whether or not homomorphic encryption were practical enough to be applied in the cloud as a web application. The feasibility of application in the web depends heavily on the time required to perform homomorphic evaluations, which is affected by several factors. One of these main factors is the *security parameter* k, which is a value closely related to the generation of public and private keys and how secure they are. The experimental design consists in changing the k parameter to different values, as to see to what degree it has impact on the key generation, encryption, decryption and addition *processing times*, as well as the *size* of the resulting public key and ciphertext. Finally, the results obtained are presented and briefly discussed.

5.1 Setup

The experiment is performed by running several iterations of the program that runs in the underlying client-server architecture. To evaluate how the security parameter k affects the time required for certain activities in the software, such as key generation, encryption, decryption, and homomorphic addition of ciphertext. The security parameter k is considered important because the value affects how secure the resulting keys and ciphertexts become. In the design document by Halevi and Shoup [**?**

], it does not specify to what degree changing the value of k would make the cipher vulnerable to cryptanalysis or a brute-force attack; however, it does specify that the technique they propose uses a *ciphertext packing* circuit that can be evaluated homomorphically in time $T \cdot \text{polylog}(k)$.

To show how the k security parameter affects the operation of the homomorphic encryption program, the experiment considers several values, namely: 20, 40, 80, 100; where 80 is the default value found in the examples of the HElib. Each value is seen as a *treatment* in the experimental design. For each treatment, the program is run through **20 iterations** of key generation, encryption, decryption and addition.

In order to perform the necessary iterations, certain parts of the code from both the client and server sides were taken and adjusted, so that all the operations were performed on the same program. Consequently, overhead operation costs such as sending data between the client and server are not considered for this experiment.

When generating the public and private keys, all of the other parameters are kept *intact*, so that it is only the k value that gets changed. Consequently, the other mechanisms slightly vary because they make use of the previously generated public key.

It is important to consider that the experimentation was done on a notebook which had an AMD Elite A4-5150M processor. The processor is dual core and it runs at 2.7 GHz. Therefore, the results might differ depending on the processor used.

5.2 Results

The results are presented in the following table, where the column represents the characteristic or attribute considered, such as the processing times and the size of the public key and ciphertext files measured in bytes. Consider that the values presented are the averages obtained from the set of iterations run.

k	Key Generation	Key Size	Encryption	Decryption	Addition	Ciphertext Size
40	13.13 s	165.92 MB	0.3559 s	0.0483 s	0.07406 s	30.99 kB
60	10.71 s	91.63 MB	0.4143 s	0.0566 s	0.07738 s	36.04 kB
80	8.380 s	57.08 MB	0.6424 s	0.0578 s	0.08260	37.73 kB
100	9.556 s	35.45 MB	0.5793 s	0.0642 s	0.08294	43.67 kB

5.3 Discussion

The first characteristic that is being evaluated is the time needed to generate the set of public and private keys. Usually, the generation of keys is performed only once per user. Only if the private key were compromised, would the user create a new set of keys. Even though 13 seconds, which was the highest average at $k = 40$, is relatively a short time. Directly linked to the generation of keys, the size of the key is considered as well. This is because the public key usually has to be shared with others. In this particular case, the averages range between 165.92 at $k = 40$ and 35.45 MB at $k = 100$. The difference between the sizes is quite large, and even smallest key is significantly large by itself. Additionally, it is puzzling as to why the size of the public key is decreasing as the k parameter gets larger. In cryptography, a larger k usually implies a slowdown in the whole process, but in this case, it does quite the opposite, at least regarding the key generation time.

Regarding the encryption, decryption, and addition times, the average amount of time required is quite low, and it should not become a bottleneck even if many homomorphic evaluations are performed. Even though the encryption times are longer than the decryption times, none of them surpassed 1 second, which makes it quite acceptable in application.

Perhaps the most important point to consider is the size of the ciphertexts generated. In the proposed architecture, a ciphertext is constantly sent to the server to perform operations and back to the client for decryption. In the iterations run, the max average of the ciphertext size was found to be 43.67 kB at $k = 100$, while

smallest average was 30.99 kB at $k = 40$. Looking at how the ciphertext sizes increases, it can be seen that it has a proportional relationship with the security parameter k. As k gets larger, so does the size of the ciphertext. This would mean that higher security sacrifices storage, and consequently takes more time to transfer between the client and server.

Even assumming that a security parameter of $k = 100$ is used, this shows that the size of the ciphertext is sufficiently small to be used in a web application hosted in the cloud. However, while the client-server architecture program was being developed, an average size of 74 MB per ciphertext was noted. At this time, it is not possible to recreate the scenario where ciphertexts of such a size were created, which leads to the impression that there is another factor not considered that accounts for the size of the ciphertext.

In conclusion, the aforementioned findings suggest that using HElib to apply homomorphic encryption on the cloud might indeed be feasible; however, as it has been noted, there are unknown circumstances that affect the size of the ciphertext. To the best of our knowledge, there is yet no way to predict the size of the ciphertext by using HElib. Therefore, if it were found to keep the size of the ciphertext below 45kB, it would certainly be feasible to take this solution to the cloud; otherwise, it would not be possible if each ciphertext had a size that exceded several MB. Until these circumstances are made clear and evaluated properly, it is not possible at this time to show that using HElib to make use of homomorphic encryption in the cloud is indeed feasible.

Chapter 6

Conclusions

Se presentó un enfoque para la detección de documentos duplicados el cual está basado en reconocimientos de entidades de un texto y aprendizaje computacional. Se trabajó con los reportes ciudadanos del Centro de Integración Ciudadana (CIC) como caso de estudio.

Otras cosas ...

6.1 Comentarios finales

Algo más que quieras decir ...

6.2 Contribuciones

Recordar cuál fue tu contribución.

6.3 Trabajo a futuro

Lo que faltó por hacer.