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

# Índice general

# Índice de figuras

# Índice de cuadros

Capítulo 1

# Introduction

Because of the increasing people's needs, it has become more common to exchange information with our peers. Whether it is our e-mail address 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 must be protected, so that no one other than the sender and receiver get to know about it. In order to protect the confidentiality of the information, cryptography could be used via encryption and decryption algorithms. The issue at hand becomes even more interesting when the information that needs to be passed somewhere does not go directly to another person, but is rather stored somewhere by someone in the cloud. The confidentiality is not broken if this scenario were to happen; however, it becomes more complicated if it were wished for the information to be modified while being stored in the cloud. Such goal calls for homomorphic encryption, which is an advanced technique able to modify the already encrypted data without losing its confidentiality.

There are many instances where it is required to send information to someone else. To sign up for a service, for instance. 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. And in such 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.

Cryptography is the study of ciphers, that is, encryption and decryption algorithms to be used on 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 correct key are able to access the contents of the box.

It has become increasingly common to delegate computing tasks to cloud services in order to save resources. For instance, it might be employed when the *owner/company* cannot afford its own data center for its storage and computing needs. This is especially the case when the level of activity is seasonal: this is, during certain times of the year that require more processing and storage than usual. Therefore, a common solution is to make use of cloud storage and computing services. This way, the budget is spent as much as the resources are being employed, while saving configuration and maintenance costs. However, allowing the cloud service to make use of the data has raised concerns on security [reference], 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 it in some way, it still has to be downloaded and recovered to its plaintext form, before modifying it. Once it has been modified, it has to be re-encrypted and re-uploaded to the cloud, in a potentially slow manner. This approach turns out to have high overhead costs [reference] caused by the transfer of the encrypted data back and forth between the *owner/company* and the cloud.

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 whether or not there is an eavesdropper listening on the communication lines.

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. It had been thought that doing something like that was not possible, as ciphers often consisted of permutations and substitutions. However, an interesting property found in the RSA algorithm proved otherwise [reference]. Such property is called homomorphic, and has since then gone through a lot of research, resulting in many schemes such as [...].

Homomorphic encryption is a cipher that makes use of an homomorphic property in order to alter the data while it is encrypted, by using some operation like addition or multiplication. It has been found that homomorphic encryption is not yet truly usable in real environments, mainly for two reasons: the size of the keys and the time it takes to decrypt or encrypt [reference]. An implementation of fully homomorphic encryption in the C++ language, HELib, has made an effort to make homomorphic evaluation runs faster.

Motivation

## 1.1 Problem Definition

## 1.2 Motivation

There are many areas in which homomorphic cryptography could be used, such as the medical, marketing, and financial fields [reference]. Until now, there was no way to make a concrete implementation related to these areas, since most available schemes where either too limited or too slow. Using HELib, it would be possible to reevaluate and show how feasible an implementation of a real life use case would be.

## 1.3 Hipothesis

## 1.4 Objectives

### General Objective

### Specific Objectives

1. Uno
2. Dos
3. Tres

## 1.5 Study Case

## 1.6 Estructura de la tesis

Esta tesis se compone de los siguientes capítulos: introducción, marco teórico, trabajos relacionados, metodología, caso de estudio, experimentos y resultados y conclusiones.

En este capítulo se planteó la definición del problema y la motivación para trabajar con los reportes del CIC para implementar un sistema de detección de duplicados.

El capítulo **??** presenta la notación y las definiciones de conceptos necesarios para familiarizar el lector con el contenido técnico presentado en la tesis.

En el capítulo **??** se presentan algunos trabajos que abordan el tema de la detección de documentos duplicados y se describe brevemente que métodos y que documentos se utilizan para cada trabajo.

El capítulo **??** presenta el sistema de detección de duplicados de forma general; se explican cuáles son los distintos procesos que forman al sistema independientemente del contexto en el que se apliquen.

En el capítulo **??** el sistema propuesto se aplica a los reportes ciudadanos del CIC. Aquí se describen las implementaciones de cada uno de los procesos que forman el sistema de detección de duplicados.

El capítulo **??** presenta los experimentos realizados para probar el funcionamiento del sistema de detección de duplicados, muestra los resultados de desempeño obtenidos y presenta conclusiones en base a esos resultados.

Finalmente en el capítulo **??** se presentan las conclusiones generales obtenidas a partir de los resultados del sistema de detección de duplicados durante las pruebas y se presentan las posibles mejoras que serán incorporadas como trabajos a futuro.

Capítulo 2

# Background

### Information Security and Data Confidentiality

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 key. A brief description of cloud computing and its relevant aspects are to be described, and how secure computing has an impact on it. Finally, the concept of homomorphic encryption will be explained, noting the mathematical properties and its categories of somewhat homomorphic encryption and fully homomorphic encryption.

Seen from a general perspective, Whitman and Herbert 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 intentionally or not. Computer security is defined by the NIST Computer Security Handbook 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 defines information as “facts and ideas, whichcan 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.

### 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.

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.

[? ] approaches 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 messngers 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.* As Stanoyevitch [**?** ] describes it on his Introduction to Cryptography book, the above scheme would make up a cryptosystem, since it makes of cryptographic algorithms. 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.

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 [**?** ] and Stallings [**?** ] 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 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 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. [**?** ] 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.

# Capítulo 3

# Trabajos Relacionados

En este capítulo se presentan cuatro trabajos relacionados con el sistema de detección de reportes duplicados desarrollado en esta tesis. Se realiza una breve descripción de cada uno de los trabajos y se explican las similitudes y diferencias entre esos trabajos y este trabajo de tesis.

## Capítulo 4

# Methodology

This work looks at a case study which presents an issue directly related to the confidentiality of information when it is being stored in the cloud. It discusses the situation where it is important to be able securely store a secret value, as well as make modifications on it. A solution to the problem is proposed by employing homomorphic encryption. Then, it describes in a general way a proof of concept as a stand-alone solution. Afterwards, the scenario of using a client-server architecture is considered, discussing some decisions that were made regarding the communication needs. Relevant points on key management are mentioned, and also, how the tasks previously mentioned are being split by each component.

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. 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 than other alternatives.

### Case study

The case study is explained as follows: a household commonly has more than one member, e.g. partner and children. The responsible members of the household (i.e. the parents) might want to keep an eye on the house while being away, especially if 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 wish to be promptly notified of it. This is especially true for the case 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. Setting a surveillance system might be too costly for some families, and not be quite adaptive to their needs. It might be more efficient to deploy a system that detects when people enter and exit the building through the main door, and keep a counter of it. The main idea is that the *surveillance count* gets initialized 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. The issue comes out when the recorded data, that is, the counter, is to be accessed remotely via some kind of web application.

[INSERT SAMPLE FIGURE]

### Proposed solution

The proposed solution addresses the issue hinted at the end of the study case description. This refers to making available the counter data to the householder without compromising confidentiality. Assuming that there might be more than one person who would have access to this data per household, it might not be appropriate to set up a Peer to Peer communication scheme. It makes more sense to have a server in the cloud that stores the counter data, waiting for requests from the clients to deliver the information. The data stored in the server is to 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 server, 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. The counter value would then have to be encrypted using a certain cipher before it is uploaded to the cloud. However, this would imply that every time that 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 which has an overhead cost 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 initial counter value is being stored on the server, the client part that resides on the household sends over any change of individuals that are inside the building, and so the cloud service, i.e. the server, performs the appropriate homomorphic computations on the currently stored value. This occurs without the need of anybody reuploading the counter data again, and the server has no need of decrypting the data in the first place. Since the full ciphertext is not being resent, the overhead cost is 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 in the beginning. 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. 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 a secure 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 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 user and his 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 it, 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 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 place where the counter was initialized.

**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 were left in their default values, as they seemed to serve the purpose.

Part of the description in the code repository of HELib, it is stated that the library is considered to be low-level, and given its difficulty and constant changes, not that appropriate to use to build big applications with it. [quote] '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 .$^{a}$ssembly 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.'

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 results in the incorrect evaluation of the operations.

### 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 |

### Key generation and serialization

As this library requires the use of public and private keys, the user is required to create his own set of keys, 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 the homomorphic operations. The generation of the keys is pseudorandom, so extra care should be taken as to not to use a predictable seed, so that the public and private keys are not recreated by someone else.

After instantiating context with all the necessary parameters, the set of keys can be generated. 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 copy makes use of 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.

```
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. It is commonly used for lightweight persistence and for communication via sockets [reference: http://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html ]. In this case, it is used as a means of temporarily saving 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 is 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 class in C++ called *istringstream* and *ostringstream*. The use of this class is showed as follows:

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

This way, the data of the attributes found in 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.

### 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 brought back to its plaintext form.

Following the conventions defined in the HElib, an object from the *EncryptedArray* is initialized using the *context* and *G* which were previously set. *EncryptedArray* can be seen as a container for the plaintext, where the value of the counter is placed. Once it has been done, a method called *encode* of the same class can be used 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.

```
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.

### Single component operation flow

Before attempting to break down the whole implementation in two parts: the server and client side, a proof of concept was implemented which combined 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.

### 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 the server to do it completely. 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 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 homomorphic encryption context, 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) over TCP, it was simplified to cover only the details pertaining the use of homomorphic encryption.

Once the user has obtained 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 previously generated public key. After the counter data has been encrypted successfully, the data is then sent to the server via sockets. There were two possible protocols that could have been used 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. In the end, it was chosen to use TCP over UDP, especially since it might become complicated to keep track of the order of the data packets.

#### Sending and reading serialized data

The serialized data that is sent through sockets was very large to be done with a single function call, which is why a few considerations have to be made on 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 attemps 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 at the appropriate time.

```
  ostringstream oss;
  oss << encryptedCounter;

  if(sendalldata(sockfd, oss.str().c_str(), &msgsize) == -1) {
    printf("ERROR. Only \%d bytes sent!\textbackslash 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 complete and in the appropriate order.

The following piece of code extracted from the server side of the software goes through the part where the ciphertext is being received after it has been serialized. 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;
  }
}
```

### Reconstructing the ciphertext

It was mentioned before that serialization allowed persistence and made it easier for object data to be sent via sockets. However, the serialzied 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 can contain the ciphertext. First of all, the buffer that was previously filled with data is then copied into a string variable, then it goes through a couple of steps so it can be used as a *istringstream* object. The preparation is certainly more complicated, but the reconstruction of the object is quite simple itself. 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;
```

### Changes in the counter

Once the initial value of the counter has been 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 positive or negative. When the cloud service receives the value of change, it either adds or subtracts it from the previously stored value. The change is immediately visible to the householder who might wish to know the current value of the counter.

As the implementation of the counter was started before December 2014, bootstrapping was not considered to use. 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.

### Architecture breakdown

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 considered case study implies that there is both a household where the counter is initialized and another location where the current counter value can be accessed. Considering this aspect, it makes sense to have a server in the cloud that can be accessed securely which will contain the current value of the counter for each household. This requires the server to store the public key as well, since to be able to modify the ciphertext of the counter data, such a key is needed.

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.

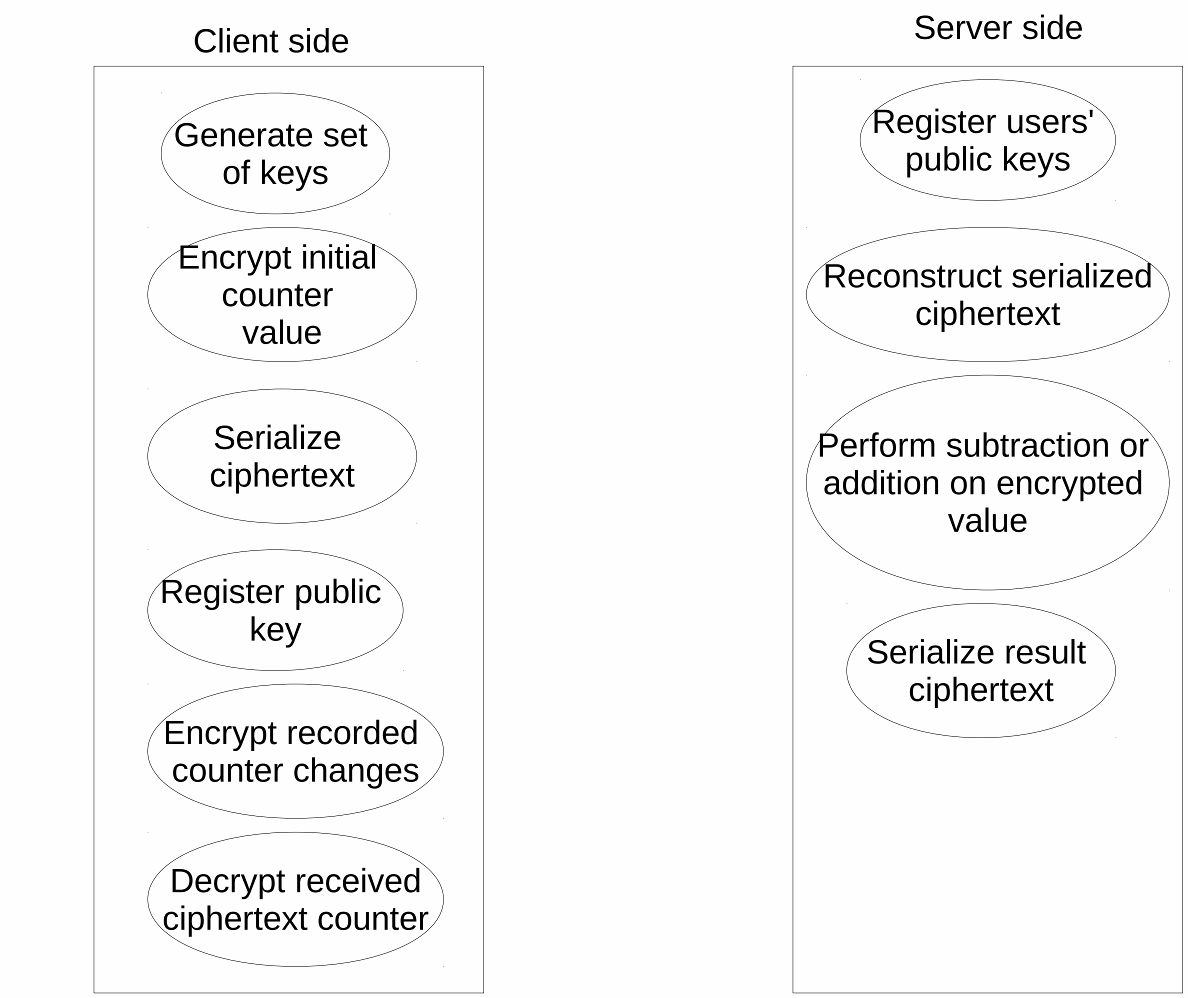
Client side
Generate set of keys
Encrypt initial counter value
Serialize ciphertext
Register public key
Encrypt recorded counter changes
Decrypt received ciphertext counter
Server side
Register users' public keys
Reconstruct serialized ciphertext
Perform subtraction or addition on encrypted value
Serialize result ciphertext

**Figura 4.1** – Architecture breakdown into client and server.

[ADD SEQUENCE DIAGRAM AND DESCRIPTION]

**Discussion**

The part 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 just a 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. Th e 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 was done without the support of a framework that specialized on it.

[TALK ABOUT DECRYPTION]

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 dealing with changes of the counter.

Capítulo 5

# Experimentos y Resultados

En esta sección se explica el proceso de selección de los datos utilizados para el etiquetado, el método de detección de duplicados híbrido y el método de detección de duplicados supervisado. También se muestran y discuten los resultados obtenidos de los experimentos realizados para el etiquetado y ambos métodos de detección de duplicados y se presentan conclusiones sobre los resultados obtenidos.

## 5.1 Configuración experimental

### 5.1.1 Resultados

### 5.1.2 Discusión

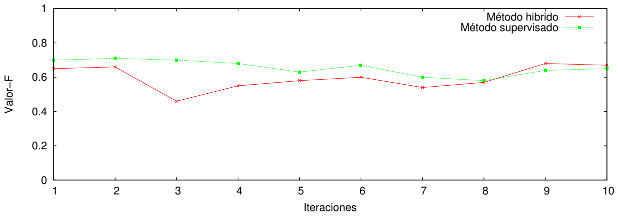

**Figura 5.1** – Porcentajes de valor-F obtenidos durante las pruebas de desempeño de los sistemas de detección de duplicados híbrido y supervisado.

# Capítulo 6

# Conclusiones

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.

# Ficha autobiográfica

Jesús Antonio Soto Velázquez

Candidato para el grado de [LLENAR]

Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

Tesis:

Applying homomorphic computing in the cloud

Nací el . . .