Practical use of Partially Homomorphic Cryptography

Eugénio Alves da Silva

Thesis to obtain the Master of Science Degree in

Mestrado Bolonha em Segurança de Informação e Direito no Ciberespaço

Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia

September 2016

TÉCNICO LISBOA

FACULDADE DE DIREITO
UNIVERSIDADE DE LISBOA

ESCOLA NAVAL

Acknowledgments

I want to express sincere appreciation to my Master's Thesis advisor, Prof. Miguel Pupo Correia, for his valuable guidance, encouragement and patience. Without his experienced and competent support, and without his continued availability, this work would not be possible.

I would like also thank the scientific coordination of the Course *Mestrado Bolonha em Segurança de Informação e Direito no Ciberespaço*, Prof. Carlos Caleiro from Instituto Superior Técnico, Prof. Eduardo Vera-Cruz Pinto from Faculdade de Direito de Lisboa and Prof. Fernando Ribeiro Correia from Escola Naval, for their rare initiative of cooperation between schools with so different scopes and traditions, and for the quality they could bring to the course.

This work was supported by the European Comimission through project H2020-653884 (SafeCloud) and by national funds through Fundacão para a Ciência e a Tecnologia (FCT) with reference:

UID/CEC/50021/2013 (INESC-ID).

Resumo

No passado, os investigadores e os profissionais de sistemas de informação assumiram sempre que os dados cifrados não podem ser processados. Mais recentemente o trabalho de Craig Gentry mostrou serem teoricamente possíveis sistemas criptográficos que permitem a execução de qualquer função computável sobre dados cifrados. Ao mesmo tempo, a explosão da computação em nuvem tornou esses sistemas ainda mais apetecíveis. Infelizmente, apesar de grandes avanços, a criptografia completamente homomórfica, ou seja, capaz de executar qualquer função sobre dados cifrados, não atingiu patamares de desempenho que permitam aplicações práticas. Por esse facto tem prosseguido a aposta na criptografia parcialmente homomórfica, ou seja, aquela que apresenta propriedades homomórficas apenas para algumas funções.

Este trabalho tem como objectivo facilitar a adopção da criptografia parcialmente homomomórfica. Para tanto foi desenvolvida uma biblioteca abrangente de esquemas criptográficos parcialmente homomórficos, única no seu género. A linguagem escolhida para a sua implementação foi o Java, a linguagem mais popular para o desenvolvimento de aplicações, tanto nos servidores, do lado na nuvem, como nos dispositivos terminais, como computadores pessoais e telemóveis inteligentes. A biblioteca inclui funções para cifra, decifra, geração de chaves, e execução de funções homomórficas.

Por forma a avaliar a sua aplicabilidade prática, foram desenvolvidos dois sistemas que a utilizam:

- o serviço de coordenação HomomorphicSpace, que é um espaço de tuplas que armazena tuplas cifradas, possuindo operações adicionais como retornar tuplas dentro um determinado intervalo de valores, e retornar resultados de operações de soma e multiplicação. Verifica-se experimentalmente que os tempos de execução obtidos são compatíveis com a utilização prática do sistema.
- um sistema de ficheiros remoto, para máquinas Linux, que guarda em nuvem dados cifrados por esquemas criptográficos com propriedades homomórficas. Graças a essas propriedades é possível realizar no servidor um conjunto de pesquisas de ficheiros, sem fazer qualquer tipo de descodificação prévia. É feita uma avaliação experimental que sugere que os tempos de execução são aceitáveis para uma utilização prática.

Palavras-chave: Criptografia Homomórfica, Biblioteca Java de Cifras Homomórficas, Espaços de Tuplas Cifrado, Sistemas de Ficheiros em Nuvem

Abstract

Researchers and practitioners for decades assumed that encrypted data cannot be processed. Most recently, Craig Gentry work shown to be theoretically possible to build cryptosystems that allow the evaluation of any computable function on encrypted data. In parallel, the explosion of cloud computing has made these systems even more desirable. Unfortunately, despite remarkable advances, fully homomorphic encryption systems, i.e., systems that allow the evaluation of any function on encrypted data, did not reach acceptable levels of performance enabling practical applications. For that reason, important research has been made on partially homomorphic encryption systems, ie, systems that have homomorphic properties for specific functions.

This work aims to facilitate the adoption of partially homomomorphic encryption. For that, a multiple scheme library of functions, needed to implement partially homomorphic cryptography, was developed. This library is unique in its kind. The language chosen for its implementation was Java, the most popular language for developing applications, both for server side, on the cloud, as for terminal devices, such as personal computers and smart phones. The library contains functions for encryption, decryption, key generation, and execution of homomorphic functions.

In order to assess the practical use of the library, we designed two services with it:

- The HomomorphicSpace coordination service that is a tuple space that stores encrypted tuples but still supports operations like returning tuples with values within a certain range, and the result of sum and multiplication operations. The document presents an experimental evaluation of the coordination service. We observed a negligible overhead when homomorphic operations like comparisons were used.
- A remote file system for a Linux machine, that keeps cloud data and metadata encrypted with cryptographic schemes with homomorphic properties. Thanks to these properties it is possible to perform a set of file search operations, without any prior decoding. An experimental evaluation was made in order to show that the final result have acceptable execution times.

Keywords: Homomorphic Cryptosystems, Homomorphic Encryption Java Library, Encrypted Tuple Spaces, Cloud File Systems

Contents

Acknowledgments . . . iii
Resumo . . . v
Abstract . . . vii
List of Tables . . . xi
List of Figures . . . xiii

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Topic Overview . . . 1
1.3 Objectives . . . 2
1.4 Thesis Outline . . . 2
1.5 Thesis Organization . . . 3

2 Background and Related Work **5**
2.1 Full Homomorphic Encryption . . . 5
2.2 Partially Homomorphic Encryption . . . 7
2.2.1 Comparison Operations . . . 7
2.2.2 Order Related Operations . . . 8
2.2.3 Word Search over Encrypted Text . . . 11
2.2.4 Add Operation . . . 12
2.2.5 Multiplication Operation . . . 12
2.2.6 Join Operation . . . 13

3 MorphicLib - An Homomorphic Encryption Library **15**

4 HomomorphicSpace - An Homomorphic Tuple Space **21**
4.1 DepSpace . . . 21
4.2 Threat Model . . . 22
4.3 Commands . . . 22
4.4 Architecture and Functioning . . . 24

5 HomoFuse - An Homomorphic File System **27**
5.1 Purpose 27
5.2 Threat Model 28
5.3 Design 29
5.4 Implementation 29

6 Experimental Evaluation **37**
6.1 MorphicLib 37
6.2 HomomorphicSpace 41
6.3 HomoFuse 44

7 Conclusions **49**

Bibliography **51**

List of Tables

2.1 Comparison Deterministic Encryption Schemes 8
2.2 Taking balls from a bag without replacement 10

3.1 MorphicLib's main classes 16

6.1 Random Encryption 38
6.2 Deterministic Encryption 38
6.3 Order Preserving Encryption 38
6.4 Searchable Text Encryption 39
6.5 Summable Encryption (Paillier) 39
6.6 Multiplicable Encryption (RSA) 40
6.7 Encryption/decryption execution times for a single operation in the Linux machine 40
6.8 Homomorphic operation execution times for a single operation in the Linux machine . . . 41
6.9 Exact match execution times (ms) 42
6.10 Ordered Operations Execution Times 43
6.11 Execution Times of the Command *Tree* (miliseconds) 44
6.12 Execution times of the copy into the folder (ms) 45
6.13 Execution times of the copy from the folder (ms) 45
6.14 Search execution times over 101 files 47
6.15 Search execution times over 1001 files 47

List of Figures

2.1 Homomorphic Principle 7

3.1 MorphicLib API (summary) 17

4.1 DepSpace architecture with 4 server replicas 21
4.2 HomomorphicSpace architecture 24

5.1 Basic Architecture of HomoFuse 28
5.2 HomoFuse Class (summary) 30
5.3 Virtual File System 34
5.4 Real Files on the Server 34
5.5 Root Folder 34
5.6 Server File corresponding to the root folder 35
5.7 Contents of a File in the Virtual File System 35
5.8 Content of a File on the Server 35

6.1 Search execution time versus number of distinct words 43
6.2 Execution times of the copy into the folder (ms) 45
6.3 Execution times of the copy from the folder (ms) 46

Chapter 1

Introduction

1.1 Motivation

Researchers and practitioners for decades assumed that encrypted data cannot be processed. Generally, it is necessary to decrypt the data before performing any operations over that data. This problem becomes especially important when large data resides in a public cloud [1, 2]. In this case, there is a dilemma between two alternatives:

- the data is decrypted in the server-side (in the cloud), which poses security issues, namely the need to pass the key to the server and have the information exposed to insider threats in the cloud [3, 4] at least during the operation;
- the data is decrypted in the client-side, which involves downloading the data from the cloud (typically expensive and slow) and prevents using the computational power of the cloud.

A good solution to this dilemma would be to perform the desired operations directly on the encrypted data, at the server-side, where it is stored. This would avoid the cost of moving the data, would allow leveraging the cloud's computation power, and would solve the security issues.

1.2 Topic Overview

The term *homomorphic encryption* designates forms of encryption that allow some operations to be performed over encrypted data, without decrypting it. With those forms of encryption, it is possible to perform resource-intensive computing tasks at server-side without having to decrypt the data first. Homomorphic encryption became popular with Gentry's work [5] [6], which was coincident with the emergence of cloud computing. Gentry's scheme provides fully homomorphic encryption (FHE), so it allows performing arbitrary computation on encrypted data. Other FHE schemes were presented in the following years [7, 8].

Although in theory FHE solves the problem of computing encrypted data outsourced to a cloud, the performance of these schemes is too poor for practical applications [9]. For that reason, much effort

has been placed in developing and using *partially homomorphic encryption* (PHE) schemes [10, 11, 12, 13, 14, 15, 16, 17]. PHE schemes allow performing some computation over encryption data, but not arbitrary computation like FHE.

One of the most remarkable works in the practical use of Partially Homomorphic Systems is CryptDB [16]. CryptDB is an important step towards the deployment of PHE in real systems. According to that system's website, it is now used, or has inspired software from companies such as SAP, Microsoft, and Google [18]. CryptDB is a relational database management system that stores encrypted data and allows doing SQL queries. The system combines a set of PHE schemes and has enough performance for many applications. The source code was implemented in C++ and is available.

Some authors restrict the concept of homomorphic operation to arithmetic operations, namely sums and multiplications. However, in this thesis we consider the wider notion of operation of CryptDB, that is "any processing activity that is useful for data management". That includes operations like *text search*, and comparisons of the types *equal*, *unequal*, *greater than*, *less than*, and *greater or equal*. In this paper we will use this comprehensive meaning of the concept of PHE, when referring to homomorphic operations and properties.

1.3 Objectives

This work aims to facilitate the adoption of partially homomomorphic encryption, making available a Library including a comprehensive set of Partially Homomorphic Encryption Schemes.

Besides the unit test of the library functions, two different applications, using the library, where programmed and tested. The objective is to evaluate the usability of the encryption methods in real world applications.

1.4 Thesis Outline

We start by presenting *MorphicLib*, a new partial homomorphic cryptography library that can be used to implement a wide-range of applications. The library contains functions (normally) executed at the client side and functions for the server-side. For the client-side there is the encryption scheme, i.e., functions for encryption, decryption, and key generation. For the server-side there are homomorphic equivalent operations (addition, multiplication, comparison, etc.). The library was programmed in Java in order to ensure portability, i.e., that it can be executed in different platforms, both client and server-side. Moreover, Java is arguably the most popular general purpose programming language today [19], with a large set of APIs, and a strong programming community.

As one of our main objectives is to show that the Library can be used in real applications with acceptable performance, we have built two real world applications using it.

The first application is one interesting in its own right, the *HomomorphicSpace coordination service*. Coordination services like Google Chubby [20], Google Megastore [21], Apache Zookeeper [22], and GigaSpace's tuple space (now part of XAP) [23] are important components in current cloud systems.

They are used for tasks such as synchronization, locking, orchestration, metadata storage, leader election, and replica failure detection.[1] *DepSpace* [24, 25] is a tuple space, i.e., a coordination service that follows Linda's associative memory paradigm [26], similarly to GigaSpace's tuple space. DepSpace is replicated, so it can tolerate arbitrary (Byzantine) faults in some of its replicas.

HomomorphicSpace is an extension of DepSpace with homomorphic encryption (MorphicLib), so that data (tuples) can be stored encrypted at the servers. DepSpace's commands to read and retrieve tuples were extended with operators for inequality, less/greater relations, and keyword search, all over encrypted data. Moreover, HomomorphicSpace supports addition and multiplication of tuples in the server. Data is never decrypted at the server, only at the client after retrieval. HomomorphicSpace is Byzantine fault-tolerant like DepSpace.

The second application we have built, is a *remote filesystem*. This document presents a secure file system with the characteristic of using a server side architecture available in any webhosting service: a simple web server running PHP. We took advantage of homomorphic properties to put on the server side the heaviest search tasks while maintaining a very simple architecture. At the client side, the implemented file system is based on FUSE - *filesystem in userspace* [27]. FUSE provides a POSIX interface, which gives it great potential. It allows to mount any data structure, as if it were a normal file system, when viewed through the operating system. Furthermore the application that implements the file system does not have to run in the kernel and can be supported by a user process.

The file system uses as the physical repository, a set of files with fully encrypted names, data and metadata, residing in the file space of a webserver. This repository is a flat folder, without any structure denouncing the hierarchy of the original folders. From the user side, it is seen as a tree of normal files, with the folder hierarchy defined by the user, correct metadata, and with the content unencrypted.

All encryption operations, and cipher key management is done on the client side, but transparent to the user. Moreover, the heavy search operations are done on the server side, without any decryption, thanks to the homomorphic capabilities of the cryptographic schemes used.

It is of great importance to demonstrate that the developed Library, makes possible to build homomorphic systems, that have performance levels suitable for real world applications. For that, we made a comprehensive experimental evaluation of the two applications, in order to show that the times obtained, are acceptable for a reasonable practical use.

All the developed software, the Library and the two Applications were made availlable publicly in Git Hub, under the link `https://github.com/euroegas/homorep`, for general use.

1.5 Thesis Organization

The thesis is organized in the following chapters:

- This introduction
- A Theoretical introduction of the Homomorphic Encryption and its state of the art

[1] A list of uses for Zookeeper is here: `https://cwiki.apache.org/confluence/display/zookeeper/poweredby`

- Description of the MorphicLib Library, with the justification of the choices made
- Description of the HomomorphicSpace coordination application
- Description of the HomoFuse file system in user space application
- The Experimental Evaluation that shows the usability of the Library
- The final conclusions reached

Chapter 2

Background and Related Work

In this chapter we will make a brief visit to the State of the Art of the *homomorphic encryption*. We will start by the *full* homomorphic encrytpion, the Gentry discoveries, the concepts of *somewhat homorphic schemes* and the need of the *bootstrapping* operation and the performance handicap. Then we will concentrate on the *partially homomorphic encryption*.

2.1 Full Homomorphic Encryption

As mentioned, a *full homomorphic encryption* cryptosystem supports arbitrary computation on ciphertexts. That is, it enables the evaluation of any desirable functionality, which can be run on encrypted inputs, and produces an encryption of the desired result.

Craig Gentry presented a solution in his PhD thesis [5] using a lattice-based cryptography scheme. The proposed scheme supports the basic logical operations on ciphertexts, from which it is possible to construct circuits to perform arbitrary computation.

He started with a so called *somewhat homomorphic encryption* scheme - *SHE*. For small polynomials, a SHE scheme works as a FHE scheme. However it is limited to simple operations, because each ciphertext produced has a noise component, and this noise grows as one chains intermediate operations, until ultimately the noise makes the resulting ciphertext indecipherable.

To understand better what a SHE scheme is, consider the toy Encryption Scheme that Gentry used as example in his thesis introduction [5]:

- Take an odd integer *p* that will be our *secret key*
- To encrypt a bit *b* to an integer *c*, we just make $c = b + 2x + kp$ where:
  - *x* is a random integer in (-(p-1)/2, (p-1)/2)
  - *k* is a random integer
- To decrypt just make $b = ((c \mod p) \mod 2)$, as $\mod p$ get rid of the *kp* term, and $\mod 2$ get rid of the *2x* term.

Please note that the decryption only works because we have imposed the condition that $x \in (-(p-1)/2, (p-1)/2)$, so that $(b+2x+kp) \mod p = b+2x$. $b+2x$ is the "noise component" of our encryption scheme. As we chain operations the noise cumulatively grows. The decryption will work while the noise is maintained lower than p.

We can now verify the homomorphic properties of our toy encryption scheme. If we add two ciphers, we get:

$$c = c_1 + c_2 = b_1 + b_2 + 2(x_1 + x_2) + (k_1 + k_2)p = b_1 \oplus b_2 + 2x + kp$$

That is, if one adds the two cryptograms, the result is the encryption of the xor (binary addition) of the two plaintexts. However decryption will only work if $(b1 + 2x1) + (b2 + 2x2)$ is in $[-(p-1), (p-1)]$.

The same way, if we multiply two ciphers:

$$c = c_1 c_2 = b_1 b_2 + 2(b_1 x_2 + b_2 x_1 + 2x_1 x_2) + kp = b_1 b_2 + 2x + kp$$

the result is the multiplication of the two ciphers

Decryption works whenever $(b_1 + 2x_1)(b_2 + 2x_2)$ is in $[-(p-1), (p-1)]$. In fact, the first operation for the decryption is the modular operation $\mod p$. If the noise is not in $[-(p-1), (p-1)]$, it will be altered and the next $\mod 2$ operation will not work.

The only problem with this Encryption Scheme is that the noise increases quickly when one execute a chain of operations. when the noise exceeds *p* the decryption is no longer possible. This is the reason why this kind of encryption schemes are called *somewhat homomorphic encryption schemes*.

The scheme used by Gentry [6] was not the toy system described above, but a more efficient and secure one, based on *Ideal Lattices*, that is an asymmetric encryption scheme. However, without modifications, the ideal lattices scheme is also a somewhat homomorphic scheme, as the problem of the noise increasing remains.

So, what must be done in addition, in order to transform a somewhat homomorphic scheme in a full homomorphic encryption?

The answer is a method called *bootstrapping*, which is the Craig Gentry's breakthrough discovery for the problem of the homomorphic encryption.

The idea is to refresh the encryption after each simple operation. This technique refreshes the ciphertexts by homomorphically computing the decryption function on encrypted secret key, and bringing the noise of the ciphertexts back to acceptable levels.

If one somewhat encryption scheme can support the bootstrapping operation plus a NAND operation, it is *bootstrappable*.

In fact, the bootstrapping operation guarantees that the NAND can be executed without any noise increment as the bootstrapping technique takes the noise level to the original level. Having the method of executing a NAND, one can execute any logical operation as a sequence of NANDs.

The problem is the huge number of operations needed to execute the NAND of 2 bits. Besides the operation, we need to encrypt the secret key [1] bit by bit, then execute the decryption operation bit by bit.

[1] note that decrypting with the encryption of the public key, is the homomorphic operation of the decryption of the plaintext, and thus leads to a new encryption of the input

This make the technique fundamentally impractical because of the excessive computational time it requires. After the initial Gentry work, various refinements were proposed, but the problem of performance remains.

A good state of the art document of the Full Homomorphic Encryption can be found in [28]. An excelent comparison of the different proposals can be found in [29]

2.2 Partially Homomorphic Encryption

As noted above, the attempts to achieve a Full Homomorphic scheme resulted in inefficient algorithms [9]. As a consequence, much effort has been put in developing of partial homomorphic encryption systems that have useful homomorphic properties.

In the literature the term partial homomorphic cryptography usually refers to homomorphic properties that allow arithmetic operations: sum, multiplication or both. However, in this document we follow the definition of homomorphism in [16] and use the term to mean any operation on encrypted data. In short, we use the term to designate any cryptographic operation that follows the scheme presented in Figure 2.1 applies.

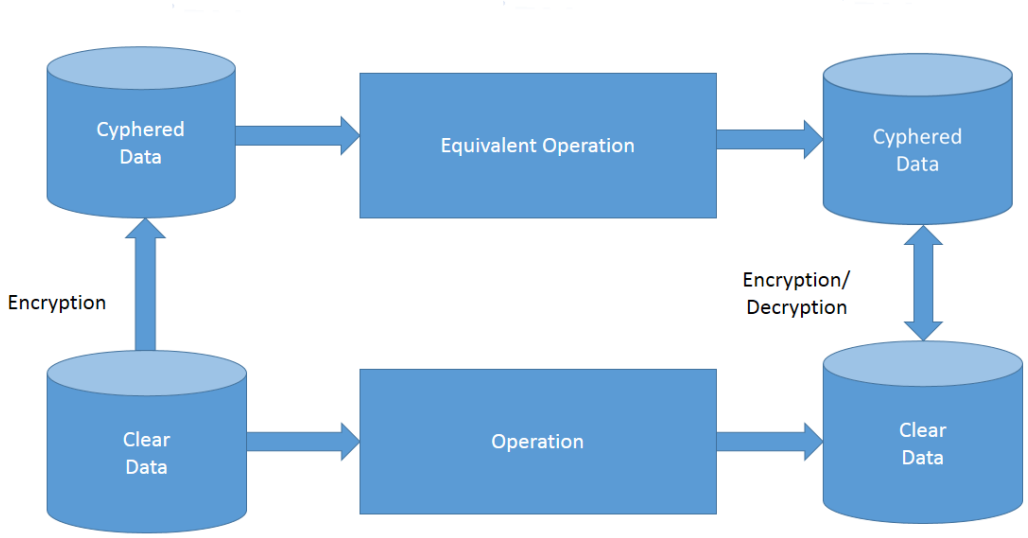

Figure 2.1: Homomorphic Principle

2.2.1 Comparison Operations

Next we start to describe the most important of such operations, and their state of the art on current days.

Description

In a client/server environment where the client is trustworthy and the server is not, the client sends an encrypted value to be compared. The server compares this value with the encrypted values stored, to determine equality as it was working with the plaintext.

In order to have this capacity, the encryption used must be *Deterministic*. That is, a given value in plaintext should give rise always to the same ciphertext, whenever it is encrypted. Of course, this creates a vulnerability to cryptanalysis, once the opponent, from the moment he can meet the meaning of a ciphertext, he can look for equal ciphertexts, knowing that they always have the same meaning.

More formally, a deterministic encryption have not the indistinguishability property. If I know that c1 = encryption of m1 and c1 = encryption of m2, I know that m1 = m2. It is worse if a public key cryptography scheme is used, as one can perform a chosen plain text attack “CPA” [30]. As the encryption key is public, the attacker can encrypt an arbitrary number of plaintexts, and have the possibility of building a huge database of paintext/ciphertext pairs to be used for cryptanalisys. For this reason, deterministic encryption should never use asymmetric encryption schemes.

Encryption Schemes

The encryption schemes used for this purpose are usually schemes based on pseudo-random permutations, such as Blowfish and AES. This pseudo-randomness is what allows creating determinism. A chaining operation mode, usually a variant of the CBC, ensures the applicability to messages of any length. To achieve a determinist encryption, the difference in relation to the normal use of the CBC is to use a fixed initialization vector.

In [16] the encryption schemes of Table 2.1 are used.

Table 2.1: Comparison Deterministic Encryption Schemes

$\leq$ 64 bits	Blowfish	If the value has less than 64 bits, padding is used (insertion on the left, or right, of zeros or ones)
$>$64 bits and $\leq$ 128 bits	AES	If the value has not exactly 128 bits, padding is used (insertion on the left, or right, of zeros or ones)
$>$128 bits	Variant of CBC with AES	To ensure determinism use allways the same Initialization Vector. To avoid equal prefix (two messages with the same initial part having Cryptograms with equal beggining) a double encryption of the blocks is made: back to front, then front to back.

2.2.2 Order Related Operations

Description

With this type of encryption we intend to address the operations of $>$ and $<$ inequality, as well as sorting (sort), and range queries.

To request a comparison, the client sends the encrypted parameters to compare to the server. For example, to obtain values greater than 10, the client send the encryption of "10" to the server, alongside with the indication of the operation "$>$". The server, without decrypting the parameter received, neither the content stored, returns the encrypted values that match the query. In the case of a sort operation, the server returns the encrypted values in an ordered list, following the order of the clear values that originated them.

To ensure these properties we must use an *order preserving encryption*. In this type of encryption, if one have a pair of values from the plaintext space, *a* and *b*, and *a* $>$*b*, then *encrypt(a)* $>$*encrypt(b)*. An Order Preserving Encryption is also Deterministic. If *a* = *b*, then *encrypt(a)* = *encrypt(b).* For that reason, an Order Preserving Encryption Scheme is, at least, as vulnerable as the Deterministic Encryption. In fact, this kind of encryption is more vulnerable than deterministic encryption [10], once it also reveals the order.

By its nature, it is inevitable that order preserving encryption reveals the order. But we need to ensure that we do not reveal more than that fact. For example, given two numbers c1 and c2, it may reveal whether *c1* $>$ *c2* or *c1* $<$ *c2*, but it must hide, as much as possible, if *c1* and *c2* are approximate values or distant ones.

Encryption Schemes

There are just a few theoretical cryptosystems with the order preserving property, and even less implementations. An example is the one implemented in the homomorphic database system CryptDB [16], based on the method described in [10].

The simplest Order Preserving Encryption (OPE) System consists of simply adding a key to the plaintext:

$$ciphertext = plaintext + Key$$

ex: if $key = 37$, $enc(7) = 7 + 37 = 44$; $dec(44) = 44 - 37 = 7$

In this case we can ensure that for any a, b, of the plaintext domain, if $a > b$, then $enc(a) > enc(b)$. Thus, we can compare encrypted values, order them, and make range queries with encrypted parameters.

The problem is that this naive encryption system, reveals more than the order. In particular it reveals the distance between two numbers: *enc(a) – enc(b) = a - b*. If one manages to know the plaintext of a single ciphertext, she can deduct the clear value of all the other ciphertexts, since the difference between the ciphertext and the corresponding plaintext is always the key.

A much smarter way to construct an OPE encryption system is the method described by Boldyreva et al. [10]. To understand the method consider that we want to encrypt a plaintext space of M elements, matching every plaintext a ciphertext extracted from a ciphertext space of N elements, where N $>$M. Let us call *domain* to the space of plaintext of size M, and *range* the ciphertext space of size N. In theory (forget for a moment the practicability) a way to perform the encryption would be taking at random M elements from the ciphertext space, of size N, and order them in a table of size M. This table, is an

injective function f: M $\rightarrow$ N. The encryption of the plaintext x, y, would be the value corresponding to the x-th position in that table. As we ordered the table, we had guarantee that the greater is x, the greater will be y, so we have an encryption that preserves the order.

The key of this Encryption System is the table itself. Thus, the possible number of different tables is the number of different keys. This number is the number of combinations of M elements in the N set, given by the well known combination formula: $\frac{N!}{M!(N-M)}$ where "!" represents the factorial of the number. In this scheme, the key is the table itself. The system is not practical, although working in theory, because it would be necessary to generate, and then share between the encryption and decryption functions, a table that could have billions of rows, in the case of a big M. For example, if one want to encrypt 32-bit integers, using 64-bit integers, one would need to generate, maintain and search a table of 2^{32} lines of 64-bit each (about 32 Gbytes).

What Boldyreva et al. did [10], was to find a system to generate only the portion of the table needed to encrypt and decrypt. In fact, if we can figure out a way to generate only one element of the ordered set, when we need it as a ciphertext, it would be great. Boldyreva et al. called this method *lazy sampling*.

To understand the method we have to consider first, how could we build the entire table. One way would be to have a hypothetical bag with N balls, M of which black, and N-M white. As it is supposed N being much bigger than M, it would be in the bag much more white than black balls. Now we start to take balls from the bag, without replacement. Most likely we would take several white balls, before appearing the first black ball. Then we proceed taking back more balls until we get the second black ball. And so on.

With our experiment we could obtain, for example the following table:

Table 2.2: Taking balls from a bag without replacement

Number of black balls obtained	Total number of balls taken
1	14
2	33
3	52
4	77
5	103
6	128
7	133
...	...

This table could then be used for the above mentioned encryption. If we need to encrypt the element x of the domain, we would go to the table and take the x-th element. In the example the encryption of 6 is 128. For decryption of y it is just necessary to determine in which row of the table, y is placed. The row number is the plaintext x. In the example, the decryption of 133 is 7.

In probability theory, the probability of taking x black balls, after having withdrawn y balls, follows a *hypergeometric* distribution, whose formula is as follows:

$$P_{HGD}(x, N, M, y) = \frac{(\frac{y}{x}).(\frac{N-y}{M-x})}{(\frac{N}{M})}$$

There is a connection between the order-preserving functions (one of possible tables of M elements taken from N) and the hypergeometric distribution. If we could write a function that, using a key k, could give, for every x, a pseudo-random y value following the hypergeometric distribution, we would have our problem solved. We would just use this function to calculate y, for every x, without needing to store the entire table. Unfortunately, it is not known any effective method to write such a function. On the other hand, there is a known method to calculate the inverse: given y, calculate x, according to a hypergeometric distribution. This method is explained in [31].

Of course, the variables are not generated randomly, since if it were, it would be impossible to replicate this randomness for decryption. The trick is to replace the random elements with values generated by a pseudorandom function, which elements actually are deterministic. Those pseudorandom values are generated having as seed, a key shared between the encryption and decryption operations, thus constituting a symmetric encryption system.

2.2.3 Word Search over Encrypted Text

Description

The goal is to conduct keyword searches in an encrypted text without having to decrypt it. Note that, word by word, this is a match operation.

Encryption Schemes

To achieve this objective, the question is more the mechanism to use than the kind of encryption used. If we simply make a deterministic encryption of the text, word by word, it would work. However, we would reveal unnecessary information about the text. As the encryption would have to be Deterministic, one opponent would know the number of times a word appears in the text, as well as their position.

So, the solution is to create an auxiliary encrypted text that will be used for the search, instead of searching in the complete text itself.

The method to create this auxiliary search text is:

1. split the original text into words using a given set of separators. For example; spaces, line changes, commas, etc.;
2. extract all distinct words, without repeating words;
3. rearrange the set of words randomly;
4. encrypt each word with a deterministic encryption scheme;
5. concatenate the encrypted words using a separator;
6. concatenate all this, with the actual text, after encrypting it with a strong random encryption scheme.

The attacker will know how many distinct words the plaintext have, but not the order or the frequency they appear on it. But with this method, one can made all the necessary keyword search.

The literature has several different searchable schemes, being this subject one of most dynamic fields in homomorphic encryption research [11, 13]. One of the innovative ideas in this area is image search [12].

2.2.4 Add Operation

Description

The aim is to apply an operation to two encrypted values, so that the result obtained is the encryption of the sum of the original values. In addition we do not want to decrypt anything to achieve the result.

Encryption Schemes

The cryptosystem used is the Paillier Cryptosystem. In this system, the operation that is performed againts the encrypted values to obtain the encryption of the sum of two clear values is the multiplication.

$$encryption(a+b) = encryption(a).encryption(b)$$ [32]

The Paillier cryptosystem also allows the multiplication by a nonencrypted constant:

$$encryption(k \times a) = encryption(a)^k$$

The multiplication by a constant in the plaintext, corresponds to the exponentiation of the ciphertext to that same constant.

Other schemes are known to insure the homomorphic property for additions and subtractions, like Benaloh [33] and Naccache-Stern [34], the latter allowing, like the Pallier cryptosystem, the multiplication by a plain constant.

2.2.5 Multiplication Operation

Description

The aim is to apply an operation to two encrypted values, so that the result obtained is the encryption of the multiplication of the original values, without decrypting any values.

Encryption Schemes

As shown bellow RSA encryption will work, as:

$$\varepsilon(x_1).\varepsilon(x_2) = x_1^e.x_2^e \mod m = (x_1.x_2)^e \mod m = \varepsilon(x_1.x_2 \mod m)$$

The same way, the homomorpic property for multiplication can be found in El Gamal cryptosystem [35]. This cryptosystem also allows exponentiation by a plain parameter.

2.2.6 Join Operation

Description

Suppose that a client asks the server to join two encrypted databases tables based on a common field. If the two tables were encrypted with the same key, a deterministic encryption of the common field would solve the requirement. The case becomes more complicated if each of the tables are encrypted with a different key. The match of the common field will not work and we need an additional work during the encryption.

Encryption Schemes

In this case it is common to use hashing methods:

1. it is created a helper field consisting of a collision-resistant hash, which will be used for the Join operation. This hash is deterministic, i.e. for the same key and the same value an equal hash will be produced.
2. use a hash construction that allows changing the key. If we have 2 different keys, say k and k', a parameter that allows the transformation of an hash produced with k' to an hash produced with k, and vice versa, is sent to the server.

The Join operation is particularly important for database applications. In this work we are not specially interested in the database case, as there is already CryptDB. For that reason, we will not implement this form of cryptography in our library.

Chapter 3

MorphicLib - An Homomorphic Encryption Library

MorphicLib is a novel library of partial homomorphic cryptographic functions written in Java and providing a Java API. MorphicLib was not developed from scratch, but based on existing source code whenever possible. The objective was both to simplify the task and to avoid introducing bugs, which tend to appear due to the complexity of cryptographic code. This library can be used both at the client-side to encrypt and decrypt data, and at the server-side to do operations over encrypted data.

The first question posed at the beginning of the implementation of the library, was: which programming language to use? For this decision, it matters to determine the location where the encryption and decryption operations are performed. The rational behind the use of homomorphic encryption is that the encryption and decryption must be made as close as possible to the user, in his local trusted systems. On the other hand, the resource intensive homomorphic operations, have to be done on the servers, which have greater processing power, and where the cryptograms reside. At the same time, we avoid massive data transfers.

So, at first sight, we should have two libraries:

1. library with the cryptographic scheme: key generation, encryption and decryption, written in a language appropriate to client applications. For example in Javascript, intended to run in a web browser;

2. library with the homomorphic operations, designed to run on the encrypted data of the server, and written in a language appropriate for servers, such as Java or PHP.

However, JavaScript is not considered by many to be a safe language for encryption systems programming, as can be seen in [36]. Moreover, the use of JavaScript would be restrictive for the use of the library, in a desktop or local application, since it is primarily a browser language.

From the point of view of the homomorphic operations at the server side, we considered the option of the PHP language. Once again this option does not fit the argument of versatility, since PHP is primarily a language to be executed at webservers, and not a general purpose language.

In this way, we decided to move towards a language the most versatile and universal as possible, allowing a single library with all functions:

- key generation;
- encrypt;
- decrypt;
- homomorphic operations.

Among all the possible options, Java is the most popular language, as it can be seen in the Tiobe index of popularity Tiobe [19].

Also important for the decision, was that Java has a huge set of libraries already available, particularly in the area of cryptography, that could be used as a good starting point for our work. That is the case of javax.crypto.* package familly, and the class java.util.Base64. In addition, one of the use cases extends the Tuple Space server, DepSpace [24], which is also written in Java.

The code of the library is organized in classes, one per *homomorphic property*. One crucial different between PHE and FHE is that in the former data has to be encrypted taking into account the kind of operation that will be supported over the encrypted data. With FHE, on the contrary, arbitrary computation is possible over encrypted data (at a cost, in terms of performance). As we opted for PHE, for each homomorphic operation we have four kinds of functions (or methods):

- key generation function, typically used at client-side;
- encryption function, typically used at client-side;
- decryption function, typically used at client-side;
- homomorphic operation functions, which allow doing operations over encrypted data, typically used at the server-side.

Next we explain the implementation of the functions for each homomorphic property. Information about the properties of the PHE algorithm, the operations supported, and the classes are in Table 3.1. Figure 3.1 shows a summary of the library API.

Table 3.1: MorphicLib's main classes

Property	**Homomorphic Operations**	**Class**	**Input Data Types**
Random	None (strong cryptanalisys resistance)	HomoRand	Strings, Byte Arrays
Deterministic	Equality an inequality comparisons	HomoDet	Strings, Byte Arrays
Searchable	Keyword search in text	HomoSearch	Strings
Order preserving	Less, greater, equality comparisons	HomoOpeInt	32 bit Integers
Sum	Add encrypted values	HomoAdd	BigInteger, String
Multiplication	Multiply encrypted values	HomoMult	BigInteger, String

```
public class HomoRand {
    public static SecretKey generateKey()
    public static byte[] encrypt(SecretKey key, byte[] IV, byte[] plaintext
        )
    public static byte[] decrypt(SecretKey key, byte[] IV, byte[]
        ciphertext)
}

public class HomoDet {
    public static SecretKey generateKey()
    public static byte[] encrypt(SecretKey key, byte[] plaintext)
    public static byte[] decrypt(SecretKey key, byte[] ciphertext)
    public static boolean compare(byte[] op1, byte[] op2)
        throws UnsupportedEncodingException
}

public class HomoOpeInt {
    public static SecretKey generateKey()
    public long encrypt(SecretKey key, int plaintext)
    public int decrypt(SecretKey key, long ciphertext)
}

public class HomoSearch {
    public static byte[] wordDigest(SecretKey key, String word)
    public static SecretKey generateKey()
    public static String encrypt(SecretKey key, String plaintext)
    public static String decrypt(SecretKey key, String ciphertext)
    public static boolean searchAll(String words, String ciphertext)
}

public class HomoAdd {
    public static PaillierKey generateKey()
    public static BigInteger encrypt(BigInteger m, PaillierKey pk)
        throws Exception
    public static BigInteger decrypt(BigInteger c, PaillierKey pk)
    public static BigInteger sum(BigInteger a, BigInteger b, BigInteger
        nsquare)
    public static BigInteger dif(BigInteger a, BigInteger b, BigInteger
        nsquare)
    public static BigInteger mult(BigInteger a, int prod, BigInteger
        nsquare)

}

public class HomoMult {
    public static KeyPair generateKey()
    public static BigInteger encrypt(RSAKey key, BigInteger value)
    public static BigInteger decrypt(RSAKey key, BigInteger ciphertext)
    public static BigInteger multiply(BigInteger op1, BigInteger op2,
        RSAPublicKey publicKey)
}
```

Figure 3.1: MorphicLib API (summary)

Random – Class HomoRand The cryptographic Random scheme is not homomorphic, but was included in the library for completeness. This scheme, is called Random because every time that a given value is encrypted, it gives a different cryptogram. In fact, it is not an homomorphic encryption system, but can be used in a general homomorphic aware application precisely when no homomorphic property is required for certain data. In this case, Random is more secure than any of the homomorphic encryption schemes as it is not vulnerable to a chosen plaintext attack [37].

For this scheme we have used the Advanced Encryption Standard (AES) implementation of the javax.crypto package with CBC mode and PKCS #5 padding. This algorithm is recommended for legacy and future use by ENISA [38].

What gives this scheme the randomness property (same cleartext producing different ciphertexts) is the use of a random Initialization Vector (IV).

Deterministic – Class HomoDet In order to make possible equality comparison operations we need deterministic encryption, i.e., encryption in which the same plaintext originates always the same ciphertext. The deterministic scheme is essentially the same as the random encryption scheme, except that the IV takes a fixed value. In order to avoid that plaintexts with the same beginning have the same beginning on the correspondent ciphertext, we make a second encryption with the blocks in the reverse order. This form of encryption is weaker than the random scheme, but necessary for equality and inequality determinations [38, 16]. Needless to say, in this encryption system an attacker will be able to notice if two equal ciphertexts correspond to the same plaintext. Otherwise, this encryption scheme is as strong as AES encryption.

Searchable – Class HomoSearch The searchable scheme aims to produce a ciphertext that allows searching for words within it, without having to decrypt it. The trivial option would be to encrypt the text word by word with a deterministic encryption system. However, this approach would provide too much information to an attacker: frequency of words, position of the words in the text, and size of the words. To avoid those drawbacks we have built a scheme closely following the solution in CryptDB [16]. The encryption for this scheme was implemented with the following sequence of steps:

1. it builds a list of distinct words found in the text (hides the frequency);
2. it encrypts each word with deterministic encryption;
3. it obtains a sha-256 (also recommended by ENISA [38]) hash of each encrypted word (hides the size of words);
4. it orders the obtained list randomly (hides the position in the text)
5. the text to be searched is encrypted with the random scheme and the list of hashes is attached.

Searching for keywords in text consists in:

- the client encrypts and hashes the keyword(s) to be searched;

- the server searches for these hashes in the list and returns the encrypted text if there is a match.

To decrypt the text the list of hashes is not necessary.

Order Preserving – Class HomoOpeInt Order preserving encryption aims to allow comparisons of encrypted values such as *greater than*, *less than*, and *greater or equal to*. We implemented this scheme by supporting the encryption of 32-bit signed integers (Java's *int* primitive type). Encryption maps each value into a positive number in the range [0, MaxLong/2]. The algorithm implemented was the one described by Boldyreva et al. [10]. The implementation was based on CryptDB's C++ implementation obtained in GitHub [18]. A challenge of the implementation was to find a reverse hypergeometric pseudo-random variate generator method, as CryptDB's code was too complex. Instead we used a Java implementation of the algorithm described in [31] available at GitHub [39].

Sum – Class HomoAdd As partial homomorphic scheme for the sum operation, we used the Paillier cryptosystem [37]. In order to be able to work with numbers as large as necessary, we decided to use as inputs big integers, namely Java's *BigInteger* class. For the implementation of Paillier we have adapted the Java code authored by Hassan found in the web [40].

The Paillier cryptosystem is an asymmetric scheme with the following two keys: public key – the pair (*n*, *g*); private key – the pair (λ, μ). The parameters *n*, *g*, λ, and μ are generated from two big prime numbers *p* and *q*. The parameter *n* = *p.q*, is part of the public key. So, the security of the system is based on the fact that an attacker cannot find *p* and *q* factorizing *n*. This is the same problem used by RSA, so the length of *n*, two times the length of *p* and *q*, should follow the recommendations for RSA, and have at least 2048 bits [38].

This scheme also supports *multiplication* of encrypted values by constants. For that purpose, we raise the encrypted value to the constant (for a sufficiently large n):

$$Enc(a + b \, mod \, n) = Enc(a).Enc(b) \; mod \, n^2$$
$$Enc(k.m \, mod \, n) = Enc(m)^k \; mod \, n^2$$

Note that in PHE the operations performed with the encrypted data do not have to be the same that would be executed with plaintext. Those operations just need to produce the desired result, i.e., the result obtained must be the encryption of the result that would be obtained executing the original operation over the plaintext. This is the case with Paillier, in which to obtain the encryption of a sum, a product is made. The same way, the multiplication by a constant is determined by rising the encypted value to that constant.

Multiplication – Class HomoMult For multiplication we used RSA, again with big integers. We used the standard Java functions in *javax.crypto* for encryption, decryption, and key generation. No padding is used to guarantee the homomorphic property.

We implemented encryption functions accepting inputs of the types *BigInteger* or *String* (containing an integer).

Two aspects should be noted:

1. in this way of using RSA both keys must be kept secret, otherwise chosen plaintext attacks would be possible;
2. the partial homomorphism for multiplication is valid for the modular multiplication, being the module the same used in the encryption scheme. As the RSA keys have more than one thousand bits, that means that we can confortably work with 32 bit integers or even 64 bit long integers. Actually we can work with BigIntegers of hundreds of bits provided that the multiplications do not exceed the value of the module used in the encryption.

This chapter presented MorphicLib, a library intended to facilitate the development of applications using partially homomorphic encryption. The library includes a set of encryption schemes with different homomorphic properties, with functions that can be used both on client and server sides.

Chapter 4

HomomorphicSpace - An Homomorphic Tuple Space

This chapter presents HomomorphicSpace, a coordination service that leverages MorphicLib to handle encrypted data at the server.

HomomorphicSpace is an extension of DepSpace, so we start by presenting the latter.

4.1 DepSpace

DepSpace (Dependable Tuple Space) is a fault- and intrusion-tolerant *tuple space* [24]. Architecturally it is client-server system implemented in Java (see Figure 4.1). The server-side is replicated in order to tolerate arbitrary faults. The client-side is a library that can be called by applications that use the service. Clients communicate with the servers using a Byzantine fault-tolerant total order broadcast protocol called BFT-Smart. The most recent version supports extensions to the service [25]. A stable prototype is available online.[1]

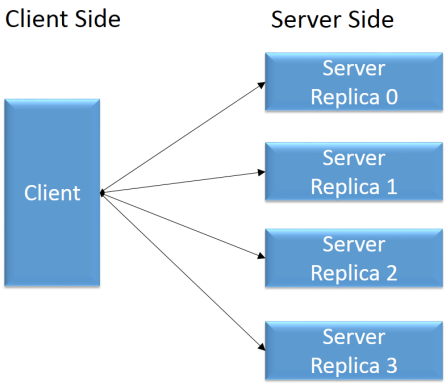

Figure 4.1: DepSpace architecture with 4 server replicas

[1] https://github.com/bft-smart/depspace

The service provides the abstraction of tuple spaces. A tuple space can be understood as a shared memory that stores *tuples*, i.e., sequences of *fields* (data items) such as (1, 2, a). Tuples are accessed using *templates*. Templates are special tuples in which some fields have values and others have undefined values, e.g., wildcards meaning any value ("*"). A template *matches* any tuple of the space that has the same number of fields, in which the values in the same position are identical, and the undefined values match in some sense. For example, the template (1, *, a, *), matches the tuples (1, 2, a, b) and (1, 7, a, 14), but neither (1, 2, b, 4), where the 3rd field does not match, or (1, 2, a, b, 5), where the number of fields are different.

DepSpace supports a set of commands, issued by clients and executed by the servers. Here we consider the following commands:

- *out tuple* – inserts a tuple in the space;
- *inp template* – reads and removes from the space a tuple that matches the template;
- *rdp template* – reads but does not remove from the space a tuple that matches the template;
- *inAll template* – reads and removes from the space all tuples that match the template;
- *rdAll template* – reads but does not remove from the space all tuples that match the template.

DepSpace does not support homomorphic operations. However, it allows fields to be encrypted and basic equality matching by storing a hash jointly with the encrypted field. This solution however is vulnerable to trivial brute force and dictionary attacks. It does support the definition of access control policies using its policy-enforcement mechanism.

4.2 Threat Model

The threat model we consider for HomomorphicSpace is similar to the threat model for DepSpace except for one, crucial difference: we consider that any server may be adversarial and try to read the content of the tuples it stores. We consider that all tuples of their fields for which confidentiality has to be preserved are encrypted using homomorphic encryption, preventing malicious servers from doing such an attack.

Similarly to DepSpace, adversaries may compromise up to f out of $3f + 1$ servers and stop them or modify their behavior arbitrarily. This is tolerated using replication and the BFT-Smart protocol. Network messages may also be tampered with by the adversary, but the system uses this using secure channels.

4.3 Commands

HomomorphicSpace extends DepSpace to allow commands over tuples with encrypted data items. More precisely in comparison with DepSpace, HomomorphicSpace:

- supports the original match operations over encrypted data;

- extend matching beyond the equality and wildcards with more complex matches, i.e., inequality, order comparisons (lower, greater), and keyword presence in a text, all over encrypted data;
- allow addition and multiplication off encrypted fields.

Besides values and wildcards ("*"), HomomorphicSpace's *templates* can include the following fields:

- *% word*$_1$ *... word*$_n$ – matches a textual field containing all the words indicated;
- $>$ *val* – matches a numeric field containing a value greater than *val*;
- $>=$ *val* – matches a numeric field containing a value greater or equal to *val*;
- $<$ *val* – matches a numeric field containing a value lower than *val*;
- $<=$ *val* – matches a numeric field containing a value lower or equal to *val*.

HomomorphicSpace adds three commands to those provided by DepSpace (Section 4.1).

The first is *crypt id template* and aims to define a *tuple encryption type*. The command takes as input an identifier (id) for the type it will create, and a template with the homomorphic operation desired for each of the fields, which will determine the homomorphic property. For example, if the template contains for a given field the operation "=", the system infers that the encryption to be used for that field is deterministic, which is the strongest that allows that operation. If no operation is indicated, the field will not be encrypted. The complete list of interpreted operations is:

- $=, <>$ – determinist encryption
- $>, >=, <, <=$ – order preserving encryption
- $\%$ - searchable encryption
- $+$ – Paillier
- $\&$ – RSA
- . – random encryption
- other value – no encryption

The second command is *rdSum template*. This command starts by collecting all the tuples that match the template similarly to *rdAll*, then sums the (encrypted) fields with $+$ in the template. The function returns a single tuple with the result.

The third command is *rdProd template*, which works similarly to *rdSum* but does multiplications instead of sum.

This scheme allows a single type of encryption per field (unlike, e.g., CryptDB). However, with the tuple data structure this is not a restriction. For instance, for tuples with a single numeric field, two operations like equality and sum can be supported by transforming that field in two and using the tuple encryption type ($=$, $+$).

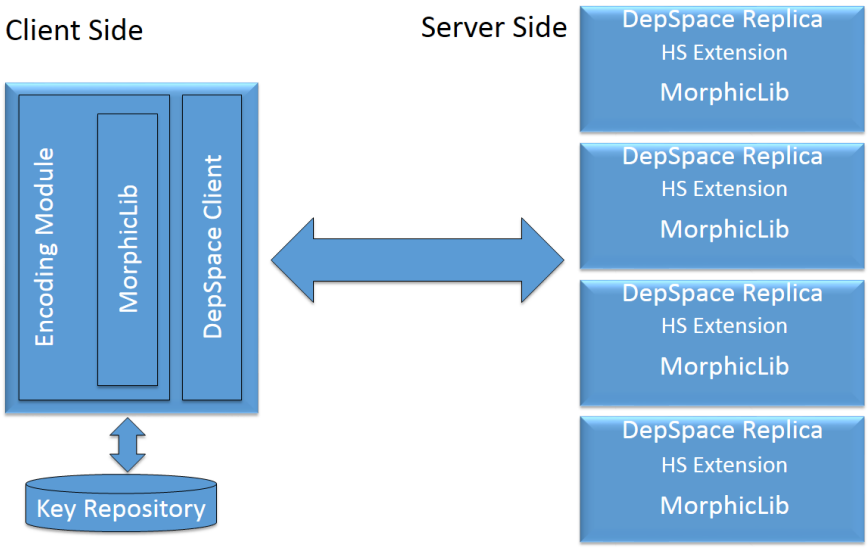

Figure 4.2: HomomorphicSpace architecture

4.4 Architecture and Functioning

Architecturaly the HomomorphicSpace is similar to DepSpace, with a client-side and a server-side. Figure 4.2 represents the system with 4 replicas, i.e., with $f = 1$. From the confidentiality point of view, the server-side is untrusted and the client-side trusted.

The server-side of the system is mostly DepSpace code with the server-side of the MorphicLib and with extensions to process the homomorphic operations. The client-side includes MorphicLib's and DepSpace's client-side libraries. The main functions of the client is to encrypt tuples and send them to the tuple space, and to decrypt them before they are delivered to the application. When a tuple is encrypted, the encryption keys are stored in a *key repository* (a folder with one file per key). Next we describe both sides in more detail.

Client side. When the *crypt* command is issued (i.e., that method is called), the library generates keys for every field of the tuple for which homomorphic properties are desired. These keys are stored jointly with the tuple encryption type (id and template) in the key repository.

All the other commands (*out*, *inp*, etc.) include an id that the library uses to retrieve the corresponding tuple encryption type and keys from the repository. If the operation indicated in a field is not compatible with the encryption defined with the *crypt* command, the command returns an error.

The library uses the DepSpace client library to send to the servers the command and the fields. If the command is an *out*, the fields are encrypted with the scheme defined in the tuple encryption type and the keys previously stored. If the command involves reading tuples it contains the operation and encrypted values. Note that each field of each id has its own key (or key pair for RSA), but the same field for the same id is always encrypted with the same key.

When the library receives a reply from the servers, it does the opposite, i.e., it decrypts the encrypted fields using the corresponding schemes and keys.

Server side. The server-side handles different commands in different ways. The *out* command is executed the same way as in DepSpace. The fields may be encrypted but they come encrypted from the client so the tuple is stored unmodified. The *inp* and *rdp* commands were modified using DepSpace's extension mechanism in order to support the $=$, $<>$, $>$, $>=$, $<$, $<=$, and text search operations over encrypted data, returning one of the matching tuples. The *rdall* and *inall* commands work similarly, as rdp and inp, but return all matching tuples. The *rdSum* and *rdProd* commands are implemented as a modification of the original *rdAll* command that returns a single tuple with the relevant fields respectively added or multiplied.

This chapter presented HomomorphicSpace, a coordination service capable of storing and processing encrypted data. The service can search encrypted data with matching operators like $=, <>, >, >=, <, <=$, find text based on keywords, and execute sums and multiplications. All those functions are performed without any decryption of the encrypted data.

Chapter 5

HomoFuse - An Homomorphic File System

This chapter presents HomoFuse, a cloud file system that leverages the functions provided by the MorphicLib library. HomoFuse provides a POSIX-like interface, so it can be used in Linux similarly to other file systems. However, the files are stored encrypted in a server, accessed using a web interface.

5.1 Purpose

The objective is to provide a remote file system with the following properties:

1. The server must store all the files in a standard *POSIX file system* where:
   (a) The file content shall not be disclosed;
   (b) The file name must be encrypted;
   (c) Nothing should be revealed, on server-side, about the files modification date;
   (d) The folder structure shall not be disclosed
2. The server must present to the outside world a REST interface, what allows the use of a standard web server and HTTP or HTTPS;
3. The server cannot have access to any key;
4. The communication between the client and the server can use a secure channel (HTTPS), including with client side authentication;
5. At client side, the file system must be mounted with a *mount* Linux command, being visible to the user as an operating system unencrypted folder;
6. It should be possible to list all files created on a certain date range, and the selection of these files should be made on the server side;

7. It should be possible to list all files containing a particular keyword, and the selection of these files should be made on the server side.

The necessary homomorphic properties to achieve these goals are:

- determinist - to search an encrypted file name;
- order preserving - to search for date range;
- searchable - to search text by keyword.

In addition to the homomorphic encryption, random encryption is used to encrypt whenever the data just needs to be hiden, without the need to perform any operation on them, at the server side. As we have seen the random encryptions are not vulnerable to CPA attacks, and therefore should be used whenever homomorphic properties are not justified.

The basic architecture for the implementation of the System, shown in Figure 5.1, has very simple requirements: The client side is composed by a Linux machine running Java. The server side is a web server running PHP and communicating over the HTTP or HTTPS protocol. This is a configuration almost universally available in common webhosting services,

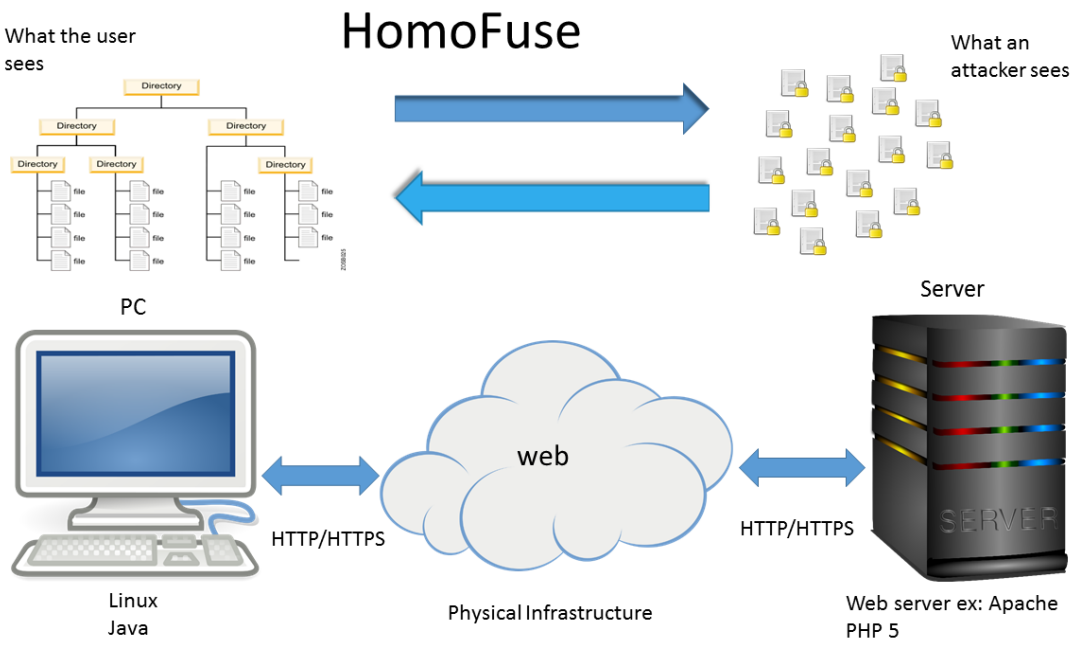

Figure 5.1: Basic Architecture of HomoFuse

5.2 Threat Model

For this service it is considered that the server side data can be read or altered by an attacker.

The system does not address the problem of data loss, what could be achieved by other methods, like redundancy. The system also do not address client side security, which is considered a trustworthy system.

The system guarantees that, if an attacker reads the data in the server side will not obtain any information about the content information, its structure, file and folder names and date of creation and modification.

If the attacker manages to get write or update privileges, he can not create understandable tampered information: meaningful files inserted correctly in a folder.

5.3 Design

The implementation of the client side is based on the FUSE library, specifically on its implementation in Java, *jnr-fuse* [41] .

The FUSE platform, is an interface for programs that run in userspace of a Linux system, and can export a file system to the kernel. FUSE provides a POSIX interface, which gives it great potential as it allows to mount the file system as if it were a normal file system. It was originally written in C++ by Tejun Heo and has since been at a constant evolution with many contributions. The code is available on GitHub [27].

jnr-fuse is an implementation of Java, using Java Native Runtime (JNR), written by Sergey Tselovalnikov, and available on GitHub [41]. The Java Runtime Native (JNR) is a Java API to integrate native libraries and memory.

A FUSE based project has three main components:

1. the interface module to the Linux kernel;
2. the library that can be invoked by a program running in userspace without any privilege;
3. the implementation of the functions to be called by the kernel. Those functions are intended to be programmed by the final implementor. This is where we have worked.

In this thesis we name *virtual file system* the folder as viewed by the user, and *real file system* the files as actually stored on disk on the server file system.

5.4 Implementation

In our implementation we have created a Java class, HomoFuse, which extends the class FuseStubFS of the jnr-fuse library: *public class HomoFuse extends FuseStubFS*

The *main* method of the class is very simple. It just creates an object of itself and invokes the mount method, to which it passes the *mount point*.

When the system is initialized (in the HomoFuse class constructor) it checks if a file with the cryptographic keys for the file system already exists. If not, this file is created containing all the keys, of all the

encryption schemes necessary to the system. The key generation methods of the MorphicLib library are used. From there on, these keys are always used, in order to maintain the consistency of the encryption and decryption operations.

The next step was to implement the relevant methods invoked by the kernel. In our case we have implemented (overriding) the methods of the Figure 5.2.

```
public class HomoFuse {

    public int getattr(String path, FileStat stat)

    public int mkdir(String path, @mode_t long mode)

    public int readdir(String path, Pointer buf, FuseFillDir filter,
        @off_t long offset, FuseFileInfo fi)

    public int create(String path, @mode_t long mode, FuseFileInfo fi)

    public int write(String path, Pointer buf, @size_t long size
        @off_t long offset, FuseFileInfo fi)

    public int read(String path, Pointer buf, @size_t long size,
        @off_t long offset,  FuseFileInfo fi)

    public int rename(String path, String newName)

    public int rmdir(String path)

    public int truncate(String path, long offset)

    public int unlink(String path)
}
```

Figure 5.2: HomoFuse Class (summary)

This set of overrided methods allows to present to the operating system a virtual file structure in a transparent way. The user can create, modify or read the files as if they contained no encrypted information, using the normal operating system primitives.

All encryption operations, as well as decryptions and folder structure management, are made without the user noticing the complexity behind. On the other hand, at the server side, there is a set of files with names and contents unreadable. In no case is made any encryption or decryption operation at the server side. No key is sent to the server. Nevertheless, the server is able to perform all the required operations on the files, including searches by date range or by keywords.

The server repository is a flat folder without any hierarchical structure, where there are two types of files:

1. files that represent folders on the virtual file system, including one for the root folder;
2. files that represent files on the virtual file system.

The name of the files that represent folders consists of the letter *d*, followed by a sha-256 hash of the

deterministic encryption of the full pathname.

The name of the files that represent files consists of the letter *f*, followed by a sha-256 hash of the deterministic encryption of the full pathname.

Note that the use of deterministic encryption allows one to search files by name, despite this information being encrypted at the server. The sha-256, in turn, ensures that the file names (encoding the entire pathname) are not too long, and are independent of the length of the actual pathname.

Each of these files of the server side, representing folders or files of the virtual file system, contains at the begining one metadata line, with the following content:

1. The original name of the file or the folder, encrypted with the random encryption scheme of the Library;
2. The user ID and the group ID of the file or folder owner, encrypted with the deterministic scheme. With this scheme, we can look by file or folder owner at the server side, passing to it the user and group ID;
3. The creation date, counted as the number of days after January first of 1970. The number is encrypted with the Order Preserving Encryption scheme, allowing search by date range at server side.
4. The second inside the day, is also encrypted with an Order Preserving Encryption.

In addition to the metadata line, files that represent folders contain a line with the name of each file or subfolder inside it. These names are encrypted with a random encryption. However the *key* and the *initial vector* used is not the same as the one used to encrypt the name in the file or subfolder metadata line. Thus, it is not possible to know the contents of the folders.

In addition to the metadata line, the files that represent files, include the net content (the payload), encrypted with the scheme *Searchable*. The choice of this type of encryption will allow us to search, on the server side, the files containing certain keywords.

The manipulation of the physical folder on the server, is made by a single PHP script that handles HTTP requests using the POST method of this protocol.

The PHP script has the following characteristics:

- it is strictly *stateless*;
- it never handles any key;
- it sends to the client the minimum required information;
- it uses the native file system of the server;
- It does not invoke any executable other than the standard shell commands.

In our case, the invocation of the script is performed by an Apache web server, without any pluggin or additional functionality beyond the PHP 5 execution capability. Thus, another type of web server could

be used, provided it can run PHP 5, for example, NGINX (`https://www.nginx.com/`), or Microsoft IIS (`http://www.iis.net/`). According to a recent survey, [42] these three platforms equip about 75 % of the active sites.

At client side, the application HomoFuse after mounting the virtual file system, checks whether the file corresponding to the root folder is already created in the server. If not, it creates one. This is the only initialization that is made on the server for a new file system. In the case of a new file system, the keys required for encryption are generated, and are stored in a file on the client. This is the only initialization at the client side. Thus, the creation of a new file system does not need any user action.

From there on, the HomoFuse client application waits for an invocation from the operating system. That invocation is made through one of the overridden methods listed above, and whose behavior we describe:

getattr - request of folder or file attributes - After the encryption of the name of the folder or file, the line of its metadata line is read from the server. The user, the group and the true creation date is returned to the operating system.

mkdir - request to create a folder - The metadata line is generated and a file is created on the server with this line. A line with the encrypted name of the new folder is added to the file representing the parent folder. The name is encrypted with a deterministic scheme, as it is in the metadata line. However the encryptions are made with different keys and Initial Vectors, in order to unable an attacker to cross the two pieces of information.

readdir - request for a folder content listing - The corresponding file to the folder is read. Its content is decoded and delivered to the operating system.

create - request to create a file - The file is created on the server, containing the metadata line described above. It is added to the parent folder a line with the name of the encrypted file. As in the case of folder creation, the encryptions are made with different keys and Initialization Vectors, in order to unable to cross the two pieces of information.

write - write request to a file - The content is written in the corresponding file on server. It is encrypted with the searchable encryption scheme, to allow future keyword search.

read - read request of a file - The content is read from the corresponding server file, decrypted, and delivered to the operating system.

rename - a file name change - not implemented in our prototype.

rmdir - delete a folder - The corresponding file is deleted from the server, as well as all sub folders and lower hierarchy files. The parent folder is updated by removing the line corresponding to the deleted folder.

truncate - truncation of a file - not implemented in our prototype.

unlink - delete a file - The corresponding file is deleted from the server. The parent folder is updated by removing the line corresponding to the deleted file.

In order to obtain a use case with greater use of homomorphic encrypton schemes, we have created two special cases:

- File Search, with creation date between two dates;
- File Search of files containing a given word.

The searches are done using the command " ls' on pseudo-folders, as explained bellow:

Search by date - The files searched by dates can be listed with the following commands:

- $ls \ -options \ date\_YYYYMMDD$ - Lists all files created between the date indicated and the present;
- $ls \ -options \ date\_YYYYMMDD\_YYYYMMDD$ Lists all files created between the two dates indicated.

The pattern is detected by the function *readdir*, which encrypts with an OPE scheme the date or the dates indicated.

As the date of file creation is contained in the line of metadata, encrypted with a OPE scheme, the server can select the desired files without any decoding.

Search keywords - In this case one needs to give a command: $ls \ -options \ word\_keyword$

The pattern is detected by the function *readdir*. The word given is encrypted in the same way that the words are encrypted by searchable encryption scheme (deterministic encryption followed by an hash), and is sent to the server for this to search.

As the file contents are encrypted with searchable encryption, which contains each distinct word encrypted in the same way, the server can select the files that contain the word, without requiring any decryption.

Figure 5.3 shows a folder tree of a virtual file system, obtained with the *tree* command.

```
euroegas@euroegas-HP-EliteBook-2530p:~$ tree mnt
mnt
├── hello.txt
├── meuteste.txt
├── sub
│   └── ola.txt
├── testatexto.txt
├── teste2.txt
├── teste3.txt
├── teste4.txt
├── teste5.txt
├── texto.txt
├── xoti.txt
└── yy.txt

1 directory, 11 files
```

Figure 5.3: Virtual File System

Figure 5.4 shows the corresponding folder list on the server.

```
euroegas@euroegas-HP-EliteBook-2530p:/var/www/html/folder$ ls -l
total 52
-rw-r--r-- 1 www-data www-data 160 Jan  1  1990 dc33_bMqlB-LtzqKLRpCgmwLwHFPlsb0M1IvGUzZraSk=
-rw-r--r-- 1 www-data www-data 410 Jan  1  1990 dh9-Nwn2e8R5ctBnry15_WYOTD3umcozCKoqbeY6Od14=
-rw-r--r-- 1 www-data www-data 385 Jan  1  1990 f0tqGaepB2vSyhawRuwZmUJODgCsaDJEs19nSGdxIQgk=
-rw-r--r-- 1 www-data www-data 230 Jan  1  1990 f4ISGtcYRmvJbNn9WWkcmxk2Ebtnlb5-_RqF4wcVB78M=
-rw-r--r-- 1 www-data www-data 230 Jan  1  1990 f6eQ_xRY_yuE_4U2qXxSdLF94mamkSOzP5E8kSYFskk8=
-rw-r--r-- 1 www-data www-data 230 Jan  1  1990 fik4BEDRIYcaSPnj8GXdlPbPekKD-MZTA83iR_jj9NFU=
-rw-r--r-- 1 www-data www-data 230 Jan  1  1990 fKbIP7Asspy1CjDyXmrg4IXKE7nCmDUpeGF1ezXUTUAk=
-rw-r--r-- 1 www-data www-data 230 Jan  1  1990 fKilWyIuGCJ7AHXrt_SopPpogjk2s9F4y6O2cq-b7S0g=
-rw-r--r-- 1 www-data www-data 275 Jan  1  1990 fQGKWSfQndDICcdk1eKfWXOOsdtlD0Dqv-7grMCywKeM=
-rw-r--r-- 1 www-data www-data 340 Jan  1  1990 fqSca_DcFuByxeHwjg9jficPPCnMLw7dOIzjCSozMFUA=
-rw-r--r-- 1 www-data www-data 230 Jan  1  1990 fsmjwhF__JXWORBqM9EzPkYI2JBPBlHASduEs_tZDiOc=
-rw-r--r-- 1 www-data www-data 385 Jan  1  1990 fuzHxr9_5eruUM0SMi2kRjGqOokS0xuIw1HgztIaaomQ=
-rw-r--r-- 1 www-data www-data 230 Jan  1  1990 fvR5ufmD2tMMsaOwZ6brQib8OtmjXSKNCpaxInBbkHPs=
euroegas@euroegas-HP-EliteBook-2530p:/var/www/html/folder$
```

Figure 5.4: Real Files on the Server

Note that the names of the actual files on the server are unintelligible and that the creation date is always January 1, 1990.

The listing of the virtual file system root folder produces the result of the Figure 5.5.

```
euroegas@euroegas-HP-EliteBook-2530p:~/mnt$ ls -l
total 0
-rwxrwxrwx 0 euroegas euroegas    6 Mai  9 17:12 hello.txt
-rwxrwxrwx 0 euroegas euroegas   16 Mai  9 19:18 meuteste.txt
drwxrwxrwx 0 euroegas euroegas 4096 Mai  6 15:46 sub
-rwxrwxrwx 0 euroegas euroegas   17 Mai  9 15:15 testatexto.txt
-rwxrwxrwx 0 euroegas euroegas    6 Mai  9 19:35 teste2.txt
-rwxrwxrwx 0 euroegas euroegas   12 Mai  9 19:36 teste3.txt
-rwxrwxrwx 0 euroegas euroegas   19 Mai 10 15:08 teste4.txt
-rwxrwxrwx 0 euroegas euroegas    6 Mai 13 16:34 teste5.txt
-rwxrwxrwx 0 euroegas euroegas   11 Mai 27 14:37 texto.txt
-rwxrwxrwx 0 euroegas euroegas    3 Mai  6 15:46 xoti.txt
-rwxrwxrwx 0 euroegas euroegas    3 Mai  6 16:06 yy.txt
euroegas@euroegas-HP-EliteBook-2530p:~/mnt$
```

Figure 5.5: Root Folder

If we look at the content of the server file corresponding to the root folder, we see the result of the Figure 5.6. The long line is the line of metadata, and the rest contains the encrypted names of files and subfolders.

```
euroegas@euroegas-HP-EliteBook-2530p:/var/www/html/folder$ cat dh9-Nwn2e8R5ctBnry15_WYOTD3umcozCKoqbeY6Od14\=
tChvk6cw3IqHFlJeDRHMFQ== AxESwv3n9AxC3j86gEcZ6pe4bClR3KWNrCT/q3ghipc= 5LwX84zmWCW5tZhMwM8l8A== 2305865552423288832 2305904392886288384
in+5AoRVK4bL7jdKfQzByg==
qUx7zf5Ppa21aoSn5S5Abg==
BKNB2gBw0nr1p5FO7v68ag==
3Ghr9iJUDeCiwBalTFL3tw==
/nM9XAIKdr9Jx7jymM7w0w==
+xVvfcc0k9ricYyh4cEMKg==
zVxMLC/5w3XLgD8sMb1sgQ==
lIAi6w7H50MGYy5T70CEZQ==
kcrP1bWV19L7JbJfaw/pLQ==
RCY7GM/jovSdTLp5yrUE/Q==
tMr0ppUFsNIEbuaA5U/dag==
euroegas@euroegas-HP-EliteBook-2530p:/var/www/html/folder$
```

Figure 5.6: Server File corresponding to the root folder

On the other hand, a file whose content is clear on the virtual file system as shown in the Figure 5.7 corresponds, on the server, to the content shown on Figure 5.8.

```
euroegas@euroegas-HP-EliteBook-2530p:~/mnt$
euroegas@euroegas-HP-EliteBook-2530p:~/mnt$ more texto.txt
novo texto
euroegas@euroegas-HP-EliteBook-2530p:~/mnt$
```

Figure 5.7: Contents of a File in the Virtual File System

```
euroegas@euroegas-HP-EliteBook-2530p:/var/www/html/folder$ more fvR5ufmD2tMMsaOwZ6brQib8OtmjXSKNCpaxInBbkHPs\=
4y4TVcNTx4JtW1xhWLZXIA== AxESwv3n9AxC3j86gEcZ6pe4bClR3KWNrCT/q3ghipc= 5LwX84zmWCW5tZhMwM8l8A== 2305865552423288832 2305905713588731904
CAsN0T4eGes7knWf9Mq3AA==:Xh5NXfZeyJwk1eqW5GCiLw==:3M4NwqezLWJUHrTUPb91JPJTfxS8140PnwrhSODBTw4=
euroegas@euroegas-HP-EliteBook-2530p:/var/www/html/folder$
```

Figure 5.8: Content of a File on the Server

As shown, the observation of the server content, discloses virtually no information about the virtual file system represented.

This chapter presented HomoFuse, a filesystem that stores the file contents on a remote webservice using a REST interface. Despite the file search operations, by content and by date, being made at the server side, no decryption is needed.

Chapter 6

Experimental Evaluation

We did a set of experiments to evaluate the performance of the MorphicLib library, and then of both services: HomomorphicSpace and HomoFuse. The experiments were executed in two personal computers. The first had an Intel(R) Core(TM) i7-3537U CPU @ 2.00 GHz, 4 GB RAM, and Windows 8.1 (64 bits). The second had an Intel(R) Core(TM)2 Duo CPU U9400 @ 1.40 GHz, 3,5 GB RAM, and Ubuntu 15.10 (64 bits). The Linux machine is 5 years older than the Windows machine, so it has worse performance. The software was executed using Java 1.8 with Oracle JDK in the Windows Machine and OpenJDK in the Linux Machine. The 2 machines were connected by an IEEE 802.11b/g/n switch (up to 54 Mbps).

6.1 MorphicLib

For each method of the library that as been tested, we obtained the system time using Java's *System.currentTimeMillis* method, just before and immediately after the call to the code to be measured, then we subtracted both. As the granularity of that method is 1 millisecond and most of the operations have a shorter duration, we have executed many (n) operations between readings of time, in order to avoid rounding errors.

Random. For the random scheme we used as plaintext a 64-byte long string containing digits from 0 to 9. The execution blocks had $n = 100,000$ operations and each round was repeated 30 times. The average and standard deviation of these experiments for the two machines are shown in Table 6.1. Recall that the random scheme is essentially AES with CBC mode. We used 128-bit keys. Notice that these are values for 100 thousand operations, so the encryption and decryption times in the Windows machine were around 7.4 μs. Note also that the standard deviation is for the 30 rounds, so it does not express rigorously the deviation of one execution of one operation. The Linux machine is much older so the performance is much worse as expected.

Table 6.1: Random Encryption

Execution Time (100.000 operations)	*Windows* (ms)	*Linux* (ms)
Encryption	740 ± 7	1745 ± 98
Decryption	737 ± 5	1837 ± 112

Deterministic. The deterministic scheme was evaluated similarly to the previous one: same plaintext, same value of n and of rounds. The results are in Table 6.2. The values observed are a little more than the double of those obtained for random. This is not surprising as this scheme first encrypts the plaintext with AES, similarly to random, next it does the same with the ciphertext in reverser order of the blocks.

Table 6.2: Deterministic Encryption

Execution Time (100.000 operations)	*Windows* (ms)	*Linux* (ms)
Encryption	1640 ± 8	3845 ± 217
Decryption	1589 ± 15	3682 ± 41

Order Preserving Encryption. For this benchmark, we started to generate a key, then encrypted and decrypted the numbers from 0 to 99999. Each execution was repeated 30 times and n was again 100,000. What is peculiar in this case is the high standard deviation observed (Table 6.3). In all the 30 rounds we encrypted exactly the same values and used the same n. What differed was the key, that was generated at the beginning of each round. We conclude that the performance of order preserving encryption depends on the key used. The reason seems to be that the Lazy Sample algorithm we used [10] is based on a binary search, where the number of steps is pseudo-random and key-dependent.

Table 6.3: Order Preserving Encryption

Execution Time (100.000 operations)	*Windows* (ms)	*Linux* (ms)
Encryption	437 ± 346	1640 ± 1040
Decryption	296 ± 255	1166 ± 779

Searchable Encryption. This evaluation was similar to the previous. The plaintext was the phrase: “one two three four five six seven eight nine ten eleven twelve thirteen fourteen fifteen sixteen”. Then we evaluated one homomorphic operation of search over encrypted text (we sought for the word ”ten”). In this case there is an asymmetry between the encryption and the decryption times (see Table 6.4). This is due to the fact that the encryption step involves also making a list of hashes of all distinct words, whereas decryption simply disregards this data.

Table 6.4: Searchable Text Encryption

Execution Time (100.000 operations)	*Windows* (ms)	*Linux* (ms)
Encryption	30572 ± 659	63364 ± 163
Decryption	1036 ± 115	2467 ± 14
Word Search	115 ± 13	326 ± 4

Summable Encryption. This test consisted in generating two big integers of 128 bits and encrypting them. The operations were executed in rounds of only 100 operations, as they are considerably slower than the previous functions. Each round was executed 30 times. Recall that Paillier can also be used to multiply an encrypted value by a clear number (Section 3).

Table 6.5 shows the results for 5 operations: encryption (of one of the two numbers), decryption (of one of the two numbers), sum (of the two encrypted numbers), subtraction (of the two encrypted numbers) and multiplication (of one of the numbers by a clear number). The subtraction is made by multiplying the second term by -1, then summing the two, so it is slower than addition and multiplication.

Table 6.5: Summable Encryption (Paillier)

Execution Time (100 operations)	*Windows* (ms)	*Linux* (ms)
Encryption	8966 ± 87	22191 ± 150
Decryption	8428 ± 62	20716 ± 97
Sum	5 ± 1	13 ± 1
Subtraction	150 ± 5	323 ± 11
Multiplication by clear number	21 ± 1	52 ± 3

Multiplicable Encryption. This test also involved generating two big integers of 128 bits and encrypting them. This time each round had 10,000 operations and the number of rounds was 30. The operations were: encryption (of one of the numbers) decryption (of one of the numbers), and multiplication (of two encrypted numbers).

Note in Table 6.6 the difference of execution times between encryption and decryption. Decryption is about twenty times slower than encryption, although both operations are the same, an exponentiation. The reason is that encryption uses the RSA public key, and decryption the RSA private key. By exchanging the two keys we obtained inverse results: slower encryption, faster decryption. As both keys are supposed to be kept secret, it is optional to use one or another for encryption or decryption. All keys were obtained with class *java.security.KeyPairGenerator*.

Table 6.6: Multiplicable Encryption (RSA)

Execution Time (10.000 operations)	*Windows* (ms)	*Linux* (ms)
Encryption	526 ± 15	1689 ± 54
Decryption	10242 ± 26	30911 ± 106
Multiplication	45 ± 2	127 ± 5

Discussion. Let us now compare the values obtained for a single encryption/decryption with all schemes in the Linux machine (see Table 6.7). Clearly the slowest scheme is Paillier (sum), with times of hundreds of milliseconds. The rest of the times are all 4 orders of magnitude better except for decryption of the multiplication scheme (2 orders) and the encryption for the searchable scheme (3 orders). This means that for the HomomorphicSpace the insertion and retrieval of summable tuples will be much slower than other operations.

Table 6.7: Encryption/decryption execution times for a single operation in the Linux machine

Encryption Scheme	*Encryption Time* (ms)	*Decryption Time* (ms)
Random	0.017	0.018
Deterministic	0.038	0.037
Order Preserving	0.016	0.012
Searchable	0.633	0.024
Summable	222	207
Multiplicable	0.169	3

Table 6.8 shows the time for individual homomorphic operations. Next, text searches are the faster as they are essentially hash comparisons. Sums and multiplications are orders of magnitude slower. We also present values for equality and less/greater relations that are not really homomorphic operations, but simple number comparisons (in italics in the table).

Table 6.8: Homomorphic operation execution times for a single operation in the Linux machine

Encryption Scheme	*Operation*	*Time (ms)*
Deterministic, order preserving	Exact match	*0.003*
Order preserving	Lesser, greater	*<0.001*
Searchable	Word search	0.003
Summable	Sum	0.132
Summable	Subtraction	3.233
Summable	Multiply by constant	0.518
Multiplicable	Multiplication	3.091

6.2 HomomorphicSpace

To deploy the architecture depicted in Figure 4.2, we used the Linux machine to run the client (left-hand side of the figure), and the Windows machine to run the servers (right-hand side of the figure). Although we used a single machine for the server-side, it contained 4 server replicas. The client-side application was a *command line user interface* that allows inserting commands to be executed by the tuple space, for example: *out (a,b,1,2)*. The application also includes special commands to execute and time sequences of insert and query commands for performance evaluation purposes.

The performance of a tuple space, especially of the read and retrieval commands, depends strongly on the load of the space (the commands are slower if the tuple space has many tuples, as the space has to be searched). As our focus is on the homomorpic operations, we started all the experiments with an empty tuple space. For each experiment we explain how it was loaded.

Performance of tuple exact matching with encrypted fields. In order to evaluate the performance of exact matching (equality) with encrypted fields for the relevant encryption schemes (and no encryption), we made the following test:

1. insert (*out*) 100 tuples with a single field in the tuple space. The encryption method used were: Determinist, Order Preserving and no encryption at all;
2. execute an exact match with *rdp value* and decrypt the tuple retrieved (if encrypted);
3. retrieve all tuples from the space with *inAll ** and decrypt the 100 tuples (if encrypted).

Each test was executed 30 times and the times for the three steps were measured. The results are in Table 6.9.

A first conclusion from the table is that encryption has no impact in the match operations, as the comparisons without encryption (2nd row) and with encryption (3rd and 4th rows) take very similar times (column for command *rdp*).

A second conclusion is that the use of encryption (3rd/4th rows versus 2nd row) did not cause observable delay in the experiments. This result is consistent with the values obtained for the library tests, with encryption/decryption times that are fractions of a millisecond. Furthermore, the encryption/decryption load is *at the client*, not at the server side, so it has no impact in the capacity of the servers to process requests.

Table 6.9: Exact match execution times (ms)

Encryption used	*out 100 tuples*	*rdp 1 tuple*	*inAll*
No encryption	3659 ± 465	30 ± 5	235 ± 39
Deterministic	3747 ± 724	35 ± 5	342 ± 62
Order Preserving	3771 ± 580	32 ± 8	312 ± 75

Performance of the sum and multiplication operations. The test for sum started with the insertion of 10 tuples with a single field. For each of the tuples the field contained the value 1000, 2000, . . . , 10000, encrypted with the Paillier scheme. The performance test itself consisted in executing the command *rdSum **, which sums all the (encrypted) tuples and returns a single tuple with the encrypted result. This value (55000) is then decrypted.

The test was executed 10 times and the average time was 391 milliseconds, with a standard deviation of 27 milliseconds. This confirms that addition is much slower than other operations. This kind of execution times may be acceptable in some applications with a small number of commands, but probably not in a situation with thousands of commands.

The test for the multiplication operation was similar to the sum test, except that the encryption scheme used was RSA and the command executed *rdProd **. The result of the operation was 36288×10^{32}. The average time was 97 milliseconds, with a standard deviation of 19 milliseconds.

Performance of text search. The performance of text search was evaluated with the following experiment:

1. generate a random string containing a variable number of distinct 5-character words separated by spaces;
2. insert (*out*) 10 tuples with this string;
3. insert one tuple with the same string with the word “hello” appended;
4. send 10 *rdp* commands searching for the “hello” word;
5. calculate the average execution time of the 10 *rdp* commands.

The number of distinct 5-character words was varied from 100 to 1000 in steps of 100. The process was repeated 10 times. The average time varied from 332 ms for 100 words to 749 ms for 1000 words, with an average of 554 ms. Figure 6.1 shows a graph of the variation. Note that when the number of

distinct words is multiplied by 10, the execution time is multiplied only by a factor of 2.1. (the search time does not vary with the number of identical words, as only the distinct words are inserted in the encrypted search string)

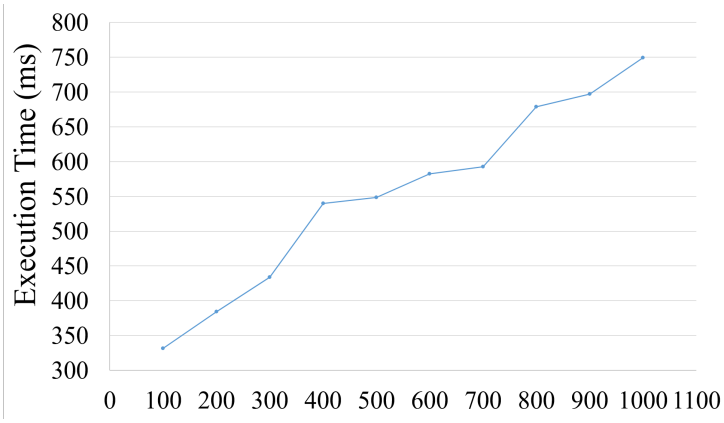

Figure 6.1: Search execution time versus number of distinct words

Performance of the ordered operations. In order to evaluate the performance of the ordered operations we have made the following test:

1. insert (*out*) 100 tuples with a single field in the tuple space, with values from 0 to 99, encrypted with the Order Preserving scheme;
2. execute an *rdp* (read one matched tuple), with the parameters indicated in the first column of Table 6.10 and decrypt it;
3. execute an *inAll* (read and delete all matched tuples), with the parameters indicated in the first column of Table 6.10 and decrypt.

The test was repeated 30 times, in order to calculate the average and the standard deviation of the execution times of the operations *rdp* and *inAll*.

Table 6.10: Ordered Operations Execution Times

Condition	*rdp (ms)*	*inAll (ms)*	*Tuples selected*
$*$ (match all)	35 ± 8	257 ± 27	100
$=$ (match)	29 ± 9	33 ± 22	1
$<> 50$	28 ± 6	209 ± 7	99
< 50	31 ± 7	195 ± 46	50
$<= 50$	30 ± 8	174 ± 21	51
> 50	29 ± 7	181 ± 26	49
$>= 50$	34 ± 8	204 ± 53	50

We can observe in the table that the execution times of the *rdp* command are all inside the deviation intervals of each other, meaning that the type of match does not affect the execution times.

For the *inAll* operation we can see that the slower operation in the one that reads all the tuples (*), the second slower is the one that reads all minus one (<>), and the faster operation is the one that reads just one tuple (=). The other operations have execution times somewhere in the middle. This allow us to conclude that the execution time of the *inAll* operation depends not on the type of comparison, but on the number of tuples retrieved. This is caused by the communication delay caused by more data.

6.3 HomoFuse

This section aims to assess the usefulness, in terms of execution times, of the HomoFuse filesystem. Recall that it was based on the homomorphic encryption schemes implemented in MorphicLib, and it is capable of executing server side search operations, without any decryption.

The machine used was an HP EliteBook 2530p, running Ubuntu 10.15. Both the server side, and the client side were implemented on the same machine in order to get rid of the latency introduced by the communications.

The Java version used on the client side was openJDK 1.8. On the server side, the web server was an Apache server, version 2.4.12.

In order to have a term of comparison for the measurements made, a full copy of the virtual file system was created in a folder of native machine file system, using the command *cp -a*. It thus becomes possible to compare the execution times of the operations over a local file system, and over the virtual file system we have built.

command *tree* - Reading folders and attributes - To assess how the system behaves when reading folders and file metadata, the command *tree* (Figure 5.3) was executed both on the virtual system and on the copy created on the physical filesystem. The command was executed one hundred times in each filesystem, in order to determine the average and the standard deviation.

The times, in milliseconds, are shown in table 6.11. It is possible to observe that there is a significant difference in the execution of the command in the two filesystems. However, the virtual file system still has acceptable times, of the order of 200 ms.

Table 6.11: Execution Times of the Command *Tree* (miliseconds)

Command	*Average Virtual FS*	*STD Virtual FS*	*Average Physical FS*	*STD Physical FS*
tree	181.46	100.86	2.88	1.31

Write in the Virtual Filesystem - In order to assess the writing on the Virtual File System, we generated files with filler text of the type “Lorem Ipsum”, using the application available at `http://pt.lipsum.com`, with a content of 100, 500, 1000, 5000 and 10000 words.

We tested the copy of these files from a normal folder of the machine to *the virtual file system* and to *the local file system*. The command (*cp*) was repeated 10 times in order to obtain the average.

The results were those shown in the Table 6.12 and represented in the Figure 6.2.

Table 6.12: Execution times of the copy into the folder (ms)

Number of words	*Virtual FS*	*physical FS*
100	162.9	3.8
500	265.7	3.9
1000	337.7	4.0
5000	1733.4	4.1
10000	4206.6	4.3

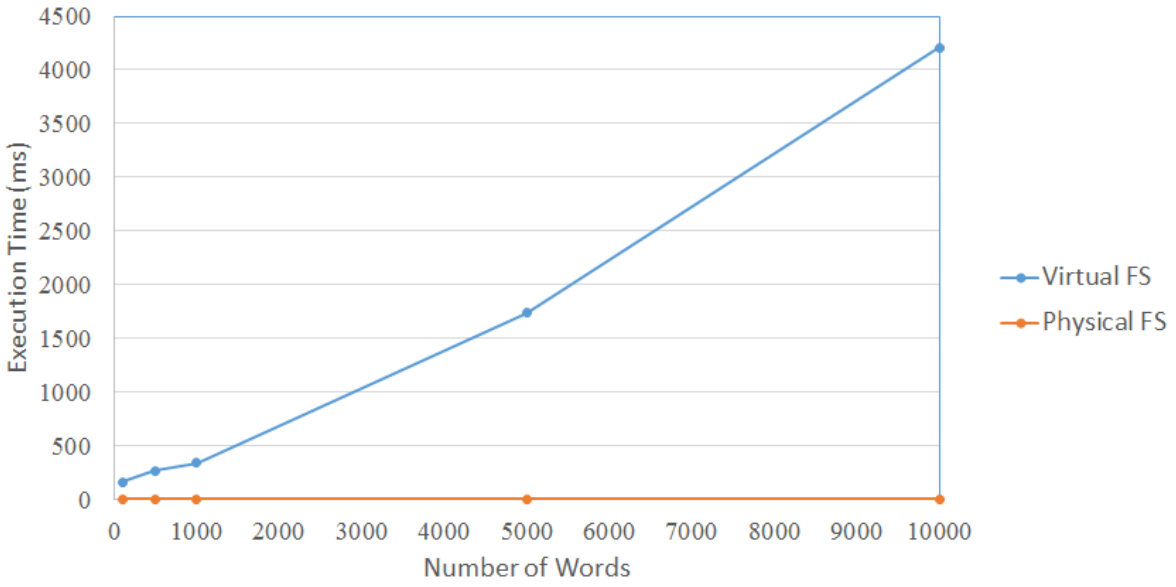

Figure 6.2: Execution times of the copy into the folder (ms)

As it can be seen, the execution time increases linearly with the size of the files, and quite strongly. In fact, to copy a file to the virtual folder, implies all the encryption operatios.

Reading from the Virtual File System - Proceeding in the same way to copies from the file system, to a normal folder on the machine, in order to test the reading, we got the values shown in Table 6.13 represented in the graph of the figure 6.3.

Table 6.13: Execution times of the copy from the folder (ms)

Number of Words	*Virtual FS*	*Physical FS*
100	80.9	4.0
500	81.3	4.0
1000	86.9	4.0
5000	87.4	4.2
10000	94.1	4.8

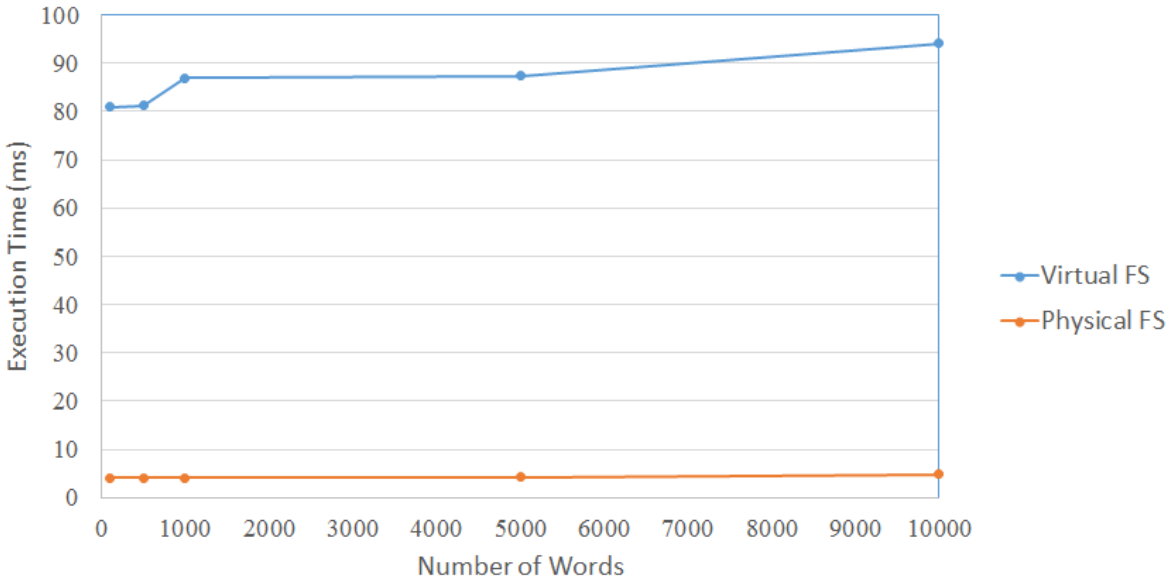

Figure 6.3: Execution times of the copy from the folder (ms)

Two facts are evident from these numbers:

1. The copy runtimes out of the folder, which correspond to read the repository are much smaller than those of the writing to the repository;
2. The growth with the size of the files, is much slower.

This is due to the choice of cryptographic scheme *Searchable*, for the file contents encryption. The more words a file contains, the higher the operations required. For reading there is no need to do word by word decryption. Only the file net content, encrypted with *Random*, has to be decrypted. So, as expected the execution time grows more slowly with increasing numbers of words.

Search Files by dates and words - In order to test the execution times of the search by date (ls YYYYMMDD), and keyword search (ls word_yyyyyy), we have proceeded as follows:

1. we copied 100 ” Lorem Ipsum ” files, filled with 100 words, into the virtual file system. An intentional program error was introduced, so that the creation date was registered 10 days before the actual test date, in order to create a file 10 days on the past;
2. a new ” Lorem Ipsum ” file of 100 words, was copied to the same folder, this time with the correct date of the test;
3. in the latter file, the word “ xpto ”, which does not exist in the remaining 100 files, was included;
4. then, we have executed ten times the command *ls date_YYYYMMDD*, where YYYYMMDD was the test date, and obtained as expected the last copied file (the only one with a date greater than or equal to the test date);

5. the same way, we have executed ten times the command *ls word_xpto*, having obtained, as expected, the last copied file (the one with the word " xpto ").

The resulted execution times are reported in the Table 6.14.

Table 6.14: Search execution times over 101 files

(miliseconds)	*Average*	*STD*
Commans ls date_YYYYMMDD	51.6	3.1
Command ls word_xpto	50.5	3.8

Then we repeated the test, but this time with 1000 files, in order to determine the impact of searches by date and word, in a folder ten times larger.

The results are reported in the Table 6.15.

Table 6.15: Search execution times over 1001 files

(miliseconds)	*Average*	*STD*
Command ls date_YYYYMMDD	190.2	4.2
Command ls word_xpto	189.9	4.5

As can be seen, the obtained times are still very low, on the order of 200 milliseconds, being imperceptible to the human user. This is due to the fact that these searches are done on encrypted data without any encryption or decryption operation.

In a sufficiently fast web server, operations times would approach to network latency times, in the order of tens of milliseconds.

This chapter presented an evaluation of the three components developed: MorphicLib, HomomorphicSpace and HomoFuse. It was shown that partially homomorphic encryption schemes allows the construction of viable applications, with an acceptable performance.

Chapter 7

Conclusions

The objective of the dissertation was to present MorphicLib, a Java Library intended to be used in applications that use homomorphic encryption for protecting data confidentiality. MorphicLib provides a set of useful homomorphic properties: equality comparison, order preserving, keyword searchable text, addition and multiplication. We believe this library has many different applications.

We designed and implemented the HomomorphicSpace based on MorphicLib. HomomorphicSpace is a tuple space, that allows storing and retrieving tuples. This service permits doing coordination operations such as synchronization, locking and metadata storage. On the contrary of the tuple space in which it is based, DepSpace, the HomomorphicSpace allows matching tuples with conditions over encrypted fields, such as inequalities and text search. It also allows consolidating read results performing sums and multiplications over encrypted data.

We have shown that the MorphicLib library can be used both at the client and the server side of a system like the HomomorphicSpace, with good performance, and with a negligible overhead, especially for equality and inequality comparisons. The sum and multiplications are the exception, which should be used with care.

In addition, in this thesis we have shown that is possible to create a cloud file system, based on a webserver running PHP without any call to executables, and where the data is encrypted with homomorphic encryption schemes, allowing heavy search operations running at the server side, without conducting any decryption of the data.

The use of MorphicLib library in a virtual file system, although leading to higher run times that an operation on a local file system not encrypted, it is still viable for common uses, since it has the runtimes in the order of100 milliseconds, for reading and for searches by date and by word.

We hope to have contributed to the idea that the partially homomorphic encryption schemes allows the construction of viable applications, with an acceptable performance.

The library built for this work, MorphicLib, is, to the best of our knowledge, the most comprehensive of partially homomorphic schemes libraries publicly available. This library can, in the future, be used in various applications of homomorphic cryptography.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53 (4):50–58, Apr. 2010.

[2] P. Mell and T. Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 2011.

[3] Cloud Security Alliance. The notorious nine: Cloud computing top threats in 2013, Feb. 2013.

[4] F. Rocha and M. Correia. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Proceedings of the 1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments*, 2011.

[5] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[6] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[7] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology – EUROCRYPT 2010*, pages 24–43. Springer, 2010.

[8] M. Yagisawa. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2015/474, 2015. `http://eprint.iacr.org/`.

[9] K. Lauter, M. Naehrig, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security*, pages 113–124, 2011.

[10] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, pages 224–241, 2009.

[11] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):222–233, Jan 2014.

[12] B. Ferreira, J. Rodrigues, J. Leitão, and H. Domingos. Privacy-preserving content-based image retrieval in the cloud. *CoRR*, abs/1411.4862, 2014. URL `http://arxiv.org/abs/1411.4862`.

[13] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security*, pages 123–145, 2015.

[14] Y. Gahi, M. Guennoun, and K. El-Khatib. A secure database system using homomorphic encryption schemes. *CoRR*, abs/1512.03498, 2015. URL `http://arxiv.org/abs/1512.03498`.

[15] D. Liu and S. Wang. Query encrypted databases practically. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 1049–1051, 2012.

[16] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[17] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 463–477, 2013.

[18] R. A. Popa et al. CryptDB webpage. `https://css.csail.mit.edu/cryptdb/`, 2015.

[19] TIOBE. Tiobe index. `http://www.tiobe.com/tiobe_index`, 2016.

[20] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.

[21] J. Baker, C. Bond, J. Corbett, and J. Furman. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Biennal Conference on Innovative Data Systems Research*, pages 223–234, 2011.

[22] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 145–158, 2010.

[23] GigaSpaces. XAP 9.0 documentation – product overview – concepts. `http://wiki.gigaspaces.com/wiki/display/XAP9/Concepts`, 2011.

[24] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference*, pages 163–176, Apr. 2008.

[25] T. Distler, C. Bahn, A. Bessani, F. Fischer, and F. Junqueira. Extensible distributed coordination. In *Proceedings of the 10th ACM SIGOPS/EuroSys European Systems Conference*, pages 10:1–10:16, 2015.

[26] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, Jan. 1985.

[27] T. Heo. Filesystem in userspace. https://github.com/libfuse/libfuse, 2001.

[28] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand. A guide to fully homomorphic encryption. Cryptology ePrint Archive, Report 2015/1192, 2015. http://eprint.iacr.org/2015/1192.

[29] A. El-Yahyaoui and M. D. Elketanni. Fully homomorphic encryption: state of art and comparison. International Journal of Computer Science and Information Security (Vol. 14, No. 4, April 2016, 2016.

[30] R. Anderson. *Security Engineering - A Guide To Build Dependable Distribution Systems*. Wiley, 2001.

[31] V. Kachitvichyanukul and B. Schmeiser. Computer generation of hypergeometric random variates. *Statistical Computation and Simulation*, 22:127–145, 1985.

[32] S. Choinyambuu. Homomorphic tallying with paillier cryptosystem, 2009. E-Voting Seminar 2009.

[33] L. Fousse, P. Lafourcade, and M. Alnuaimi. Benaloh's dense probabilistic encryption revisited. *CoRR*, abs/1008.2991, 2010. URL http://arxiv.org/abs/1008.2991.

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

[35] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, pages 10–18. Springer-Verlag, 1985.

[36] T. Ptacek. Javascript cryptography considered harmful. https://www.nccgroup.trust/us/aboutus/newsroom-and-events/blog/2011/august/javascript-cryptographyconsidered-harmful/, 2011.

[37] J. Katz and Y. Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC, 2007.

[38] ENISA. Algorithms, key size and parameters report – 2014, Nov. 2014.

[39] M. H. Derkani. Hypergeometric.java. https://github.com/masih/sina/blob/master/src/main/java/DistLib/hypergeometric.java, 2013.

[40] O. Hasan. Paillier.java. http://liris.cnrs.fr/~ohasan/pprs/paillierdemo/Paillier.java, 2009.

[41] S. Tselovalnikov. Jnr-fuse. https://github.com/SerCeMan/jnr-fuse, 2015.

[42] Netcraft. February 2016 web server survey. https://news.netcraft.com/archives/2016/02/22/february-2016-web-server-survey.html, 2016.