Practical use of Partially Homomorphic Cryptography

Eugenio A. Silva
Instituto Superior Tecnico, Faculdade de Direito, Escola Naval

***Abstract*—Researchers and practitioners for decades assumed that encrypted data cannot be processed. However, recently with the emergence of cloud computing and Gentry's work on homomorphic encryption a great interest appeared on the use of partial homomorphic functions to process encrypted data. The paper presents MorphicLib, a new partial homomorphic cryptography library written in Java that can be used to implement a wide-range of applications. The paper shows the usefulness of the library with two services. HomomorphicSpace is coordination service, that is a tuple space that stores encrypted tuples but still supports operations like returning tuples with values withing a certain range. HomoFuse is 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, at server side, without any prior decoding. The paper presents an experimental evaluation of the library, the coordination service, and the remote filesystem service. We observed a negligible overhead when homomorphic operations were used.**

I. Introduction

Generally, it is necessary to decrypt the data before performing any operations over that data. This problem becomes specially important when large data resides in a public cloud [1], [20]. In this case, there is a dilemma between two alternatives: (1) either 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 to have the information exposed to insider threats in the cloud [5], [25] at least during the operation; or (2) the data is decrypted in the client-side, which involves downloading the data from the cloud (typically expensive and slow) and prevents using the computation 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. The term *homomorphic encryption* designates forms of encryption that allow operations to be performed over encrypted data, without decrypting it. Homomorphic encryption became popular with Gentry's work [13], 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 [28], [29]. Although in theory FHE solves the problem of computing encrypted data, the performance of these schemes is too poor for practical applications [18].

For that reason, much effort has been placed in developing and using *partial homomorphic encryption* (PHE) schemes [3], [4], [10], [9], [11], [19], [24], [23]. PHE schemes allow performing some computation over encrypted data, but not arbitrary computation like FHE. CryptDB is an important step towards the deployment of PHE in real systems [24]. CryptDB is a relational database management system that stores encrypted data and allows doing SQL queries. The source code is available in C++.

This paper presents *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 [26], with a large set of APIs, and a strong programming community.

The paper shows the usefulness of the library with two services. The first one, that is interesting in its own right, is the *HomomorphicSpace coordination service* which is based on *DepSpace* [2], [7]. This is a tuple space, i.e., a coordination service that follows Linda's associative memory paradigm [12]. 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. All these operations are done without decrypting the data.

The second application we have built, is a *remote filesystem*. 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.

The paper presents an experimental evaluation of the library and the two services. We observed no significant overhead for the encrypted operations.

The main contributions of the paper are: (1) MorphicLib, a library of PHE functions and operations in Java; (2) HomomorphicSpace, a tuple space that leverages the library to support coordination of distributed applications having the data always encrypted at the servers; (3) HomoFuse, a remote file system capable of executing complex searches on encrypted files, without decrypt them.

II. MorphicLib 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 code of the library is organized in classes, one per *encryption scheme*. Each scheme has 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.

Information about the properties of the PHE algorithm, the operations supported, and the classes are in Table I.

TABLE I
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

a) Random – Class `HomoRand`*:* 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. Random scheme is more secure than any of the homomorphic encryption schemes as it is not vulnerable to chosen plaintext attacks [17].

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

b) 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 predicates [8], [24]. Needless to say, in this encryption system an attacker will be able to notice that two equal ciphertexts correspond to the same plaintext. Otherwise, this encryption scheme is as strong as AES encryption.

c) 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 [24]. Encryption for this scheme is implemented as the following sequence of steps:

1) it builds a list of unique 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 [8]) 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 text if there is a match.

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

The literature has several different searchable schemes, being this subject one of most dynamic fields in homomorphic encryption research [4], [9]. One of the innovative ideas in this area is image search [10].

d) 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. [3]. The implementation was based on CryptDB's C++ implementation obtained in GitHub [22]. 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 [16], available at GitHub [6].

e) Sum – Class `HomoAdd`: As partial homomorphic scheme for the sum operation, we used the Paillier cryptosystem [17]. 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 [14].

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. 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 [8].

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.

f) 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.

III. HomomorphicSpace Coordination Service

This section 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.

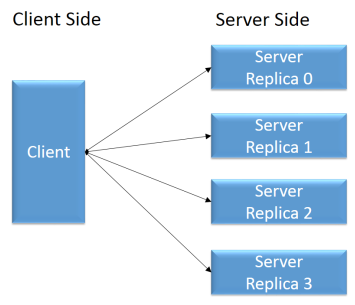


Fig. 1. DepSpace architecture with 4 server replicas

A. DepSpace

DepSpace (Dependable Tuple Space) is a fault and intrusion-tolerant *tuple space* [2]. Architecturally it is client-server system implemented in Java (see Figure 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 [7]. A stable prototype is available online.[1]

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, hi). 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, hi) and (1, 7, a, 14), but neither (1, 2, b, 4) nor (1, 2, a, hi, 5).

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.

B. 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.

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

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.

C. 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:

- $\%\ word_1 \ldots word_n$ – 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 III-A).

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 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 multiplication instead of sum.

D. Architecture and Functioning

Architecturally the HomomorphicSpace is similar to DepSpace, with a client-side and a server-side. Figure 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.

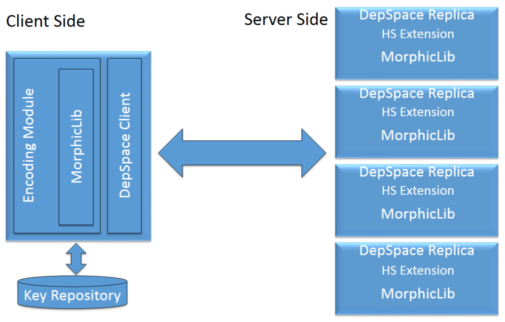


Fig. 2. HomomorphicSpace architecture

a) Client side: When the `crypt` command is issued, the library generates keys for every field of the tuple for which homomorphic properties are desired. These keys are stored in the key repository, associated with the *id* that identifies the tuple encryption type.

All the other commands (`out`, `inp`, etc.) include the tuple type *id*, that the library uses to retrieve the corresponding 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 client 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 client receives a reply from the servers, it does the opposite, i.e., it decrypts the encrypted fields using the corresponding schemes and keys.

b) 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.

IV. HomoFuse 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. The files are stored encrypted in a server, accessed using a web interface.

A. 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, he 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.

B. Design

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 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.

The basic architecture for the implementation of the system, shown in Figure 3, 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,

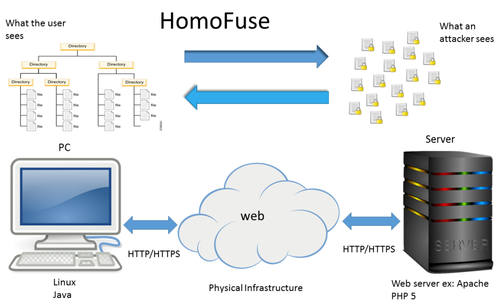


Fig. 3. Basic Architecture of HomoFuse

C. Implementation

The implementation of the client side is based on the FUSE library, specifically on its implementation in Java, *jnr-fuse* [27].

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 [15].

jnr-fuse is an implementation of Java, using Java Native Runtime (JNR), written by Sergey Tselovalnikov, and available on GitHub [27]. 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.

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 is to implement the relevant methods invoked by the kernel. In our case we have implemented (overriding) the following methods: getattr(); mkdir(); readdir(); create(); write(); read(); rmdir(); unlink().

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.

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 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, [21] 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.

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:

a) 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.

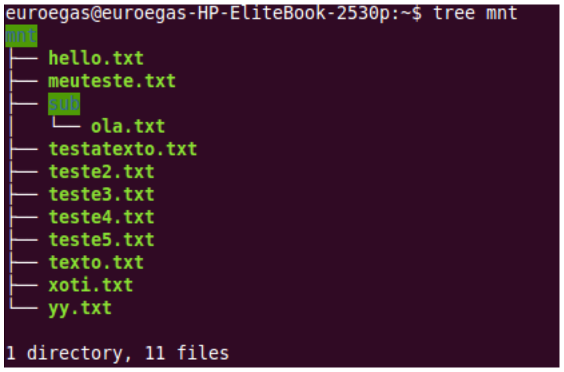
As the date of file creation is contained in the line of metadata, encrypted with a OPE scheme, the server can select the desired files without any decoding.

b) 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 4 shows a folder tree of a virtual file system, obtained with the *tree* command.

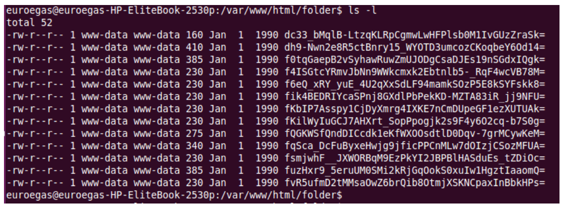


```
euroegas@euroegas-HP-EliteBook-2530p:~$ tree mnt
mnt
├── hello.txt
├── meuteste.txt
├── sub
│   └── ola.txt
├── testatexto.txt
├── teste2.txt
├── teste3.txt
├── teste4.txt
├── teste5.txt
├── texto.txt
├── xoti.txt
└── yy.txt

1 directory, 11 files
```

Fig. 4. Virtual File System

Figure 5 shows the corresponding folder list on the server.

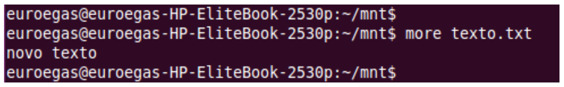


```
euroegas@euroegas-HP-EliteBook-2530p:/var/www/html/folder$ ls -l
total 52
-rw-r--r-- 1 www-data www-data 160 Jan  1  1990 dc33_bMqlB-LtzqKLRpCgmwLwHFPlsb0M1IvGUzZraSk=
-rw-r--r-- 1 www-data www-data 410 Jan  1  1990 dh9-Nwn2e8R5ctBnry15_WYOTD3umcozCKoqbeY6Od14=
-rw-r--r-- 1 www-data www-data 385 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$
```

Fig. 5. Real Files on the Server

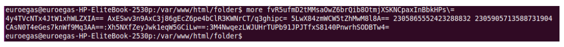
Note that the names of the actual files on the server are unintelligible and that the creation date is always January 1, 1990.

A file whose content is clear on the virtual file system as shown in the Figure 6 corresponds, on the server, to the content shown on Figure 7.



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

Fig. 6. 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$
```

Fig. 7. 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.

V. Experimental Evaluation

We did a set of experiments to evaluate the performance of both MorphicLib and HomomorphicSpace. 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).

A. *MorphicLib Library Evaluation*

For each method of the library 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.

Let us compare the values obtained for a single encryption/decryption with all schemes in the Linux machine (see Table II). 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 an implementation like the HomomorphicSpace, the insertion and retrieval of summable tuples will be much slower than other operations.

TABLE II
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 III shows the time for individual homomorphic operations. Text searches and comparisons are the faster. Sums and multiplications are orders of magnitude slower. Anyway those times can be acceptable by much applications.

TABLE III
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

B. *HomomorphicSpace Evaluation*

To deploy the architecture depicted in Figure 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.

a) 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, which are encrypted in the cases of Determinist and Order Preserving encryptions;
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).

The tests made were all exact match (equality), independently of the encryption scheme or no encryption used (see Step 2 above). However, the performance for inequalities (different, greater than, greater or equal to, ...) would be very similar as all of them are simple byte comparisons. Each test was executed 30 times and the times for the three steps were measured. The results are in Table IV.

A first conclusion from the table is that encryption has no impact in the match operations, as the comparisons without encryption and with encryption take very similar times (column for command `rdp`).

A second conclusion is that the use of encryption did not cause observable delay in the experiments. This result is consistent with the values obtained for the library, 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 IV
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

b) 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 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 simillar, except that the encryption scheme used was RSA and the command executed `rdProd *`. The result was also different (36288×10^{32}). The average time was 97 milliseconds, with a standard deviation of 19 milliseconds.

c) 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 8 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. Note also that the search time does not vary with the number of identical words, as only the distinct words are inserted in the encrypted search string.

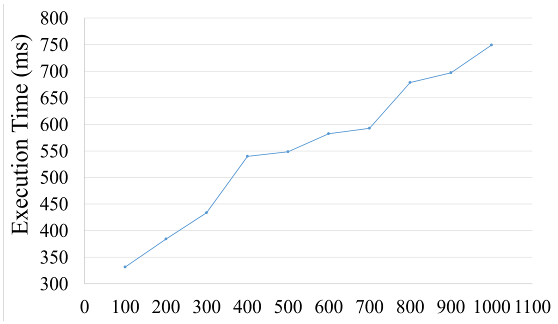


Fig. 8. Search execution time versus number of distinct words

d) Performance of the ordered operations: In order to evaluate the performance of the ordered operations we 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 V and decrypt it;
3) execute an `inAll` (read and delete all matched tuples), with the parameters indicated in the first column of Table V 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`.

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.

TABLE V
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

For the `inAll` operation we can see that the slower operation in the one that reads all the tuples ($\star$), 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.

C. HomoFuse

This section aims to assess the usefulness, in terms of execution times, of the HomoFuse filesystem.

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.

a) command tree *- Reading folders and attributes -:* To assess how the system behaves when reading folders and file metadata, the command *tree* (Figure 4) was executed both on the virtual system and on the copy created on the local filesystem. The command was executed one hundred times in each filesystem, in order to determine the average and the standard deviation. The times were:

- 181.46 ± 100.86 for the virtual filesystem
- 2.88 ± 1.31 for the local filesystem

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 for some applications.

b) 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 local 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 Figure 9.

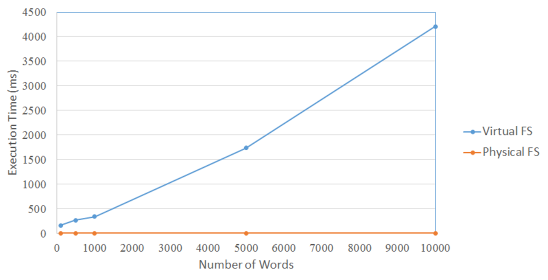


Fig. 9. Execution times of the copy into the folder (ms)

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 operations. Remarkably, the file content is encrypted with the *searchable encryption scheme* which number of encryptions is proportional to the number of distinct words.

c) Reading from the Virtual File System -: Proceeding in the same way with copies from the virtual file system, to a local folder on the machine, in order to test the reading, we got the values shown in the graph of the figure 10.

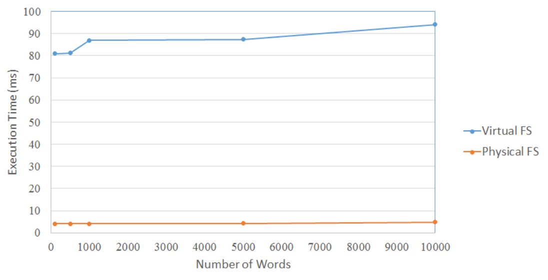


Fig. 10. Execution times of the copy from the folder (ms)

It is evident that the copy runtimes out of the folder, which correspond to read the repository are much smaller than those of the writing to the repository. At the same time 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 distinct words a file contains, the higher the number of operations required, for the encryption made at writing time. By other hand, for reading, there is no need to do word by word decryption. Only the file net content, encrypted with *Random*, has to be decrypted.

d) 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 VI.

TABLE VI
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 VII.

TABLE VII
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.

VI. CONCLUSION

We 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, metadata storage. In addition, we have shown that is possible to create a cloud file system where the data and metadata is encrypted with homomorphic encryption schemes, allowing heavy file search operations running at the server side, without conducting any decryption.

REFERENCES

[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] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

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

[9] 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.

[10] B. Ferreira, J. Rodrigues, J. Leitão, and H. Domingos. Privacy-preserving content-based image retrieval in the cloud. *CoRR*, abs/1411.4862, 2014.

[11] Y. Gahi, M. Guennoun, and K. El-Khatib. A secure database system using homomorphic encryption schemes. *CoRR*, abs/1512.03498, 2015.

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

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

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

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

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

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

[18] 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.

[19] 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.

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

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

[22] R. A. Popa et al. CryptDB webpage. https://css.csail.mit.edu/cryptdb/.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

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