



**Universidad
Europea**

LAUREATE INTERNATIONAL UNIVERSITIES

**MÁSTER UNIVERSITARIO EN SEGURIDAD DE LAS
TECNOLOGÍAS DE LA INFORMACIÓN Y LAS
COMUNICACIONES**

TRABAJO FIN DE MÁSTER

**Estudio de viabilidad del uso de librerías de
criptografía homomórfica en procesos reales**

Autor

Javier Junquera Sánchez

Directores del Trabajo Fin de Máster

Alfonso Muñoz Muñoz

CURSO 2018-2019

Resumen

Se conoce como criptografía homomórfica al conjunto de técnicas destinadas a cifrar los datos de tal forma que las operaciones que se apliquen sobre el texto cifrado se manifiesten en el texto plano al descifrar. En este trabajo analizaremos las distintas técnicas existentes para este propósito, cuáles son sus bases teóricas, y evaluaremos si es viable o no utilizar las implementaciones disponibles en sistemas destinados al uso en producción.

Las principales implementaciones de criptografía homomórfica protegen el texto utilizando un esquema de cifrado llamado *Learning With Errors*, y estarán categorizadas como *Partially Homomorphic Encryption*, *Somewhat Homomorphic Encryption* (SHE) y *Fully Homomorphic Encryption* (FHE) en función de las propiedades homomórficas que cumplan. Para nuestra evaluación construiremos un sistema que utilice una implementación de tipo SHE y otra de tipo FHE, y compararemos los resultados desde el punto de vista de la eficiencia, de la capacidad de cómputo y de la facilidad de desarrollo.

Concluiremos mostrando cómo existe cierta viabilidad desde el punto de vista tecnológico a la hora de utilizar sistemas de criptografía homomórfica, pero la viabilidad económica dependerá tanto del valor del activo como del riesgo que se busque mitigar, pues todavía no existe una solución universal.

Abstract

Homomorphic Encryption is the set of techniques designed to encrypt data so that an operation performed over encrypted data remains in the data when decryption. In this work, we will analyze the different existing techniques for this purpose, what are its theoretical bases, and we will evaluate the viability for using the different existing implementations in production systems.

The main homomorphic encryption implementations protect the text using a schema known as *Learning With Errors* and are categorized as *Partially Homomorphic Encryption*, *Somewhat Homomorphic Encryption* (SHE) and *Fully Homomorphic Encryption* (FHE) depending on the homomorphic encryption properties they fulfill. For our implementation, we will build a system using both a SHE, and an FHE implementation, and we will compare the results from the point of view of the efficiency, the computing capacity and the easiness of development.

We will conclude showing how it exists certain viability from the point of view of the technology when using homomorphic encryption systems, but the economic viability depends on both the active value and the risk to be mitigated, as there is still no a universal solution.

Agradecimientos

Aprendí que existía una cosa llamada criptografía con las lecciones de Alfonso Muñoz en Ciphertext. Julio Rilo me enseñó que había una criptografía especial, llamada homomórfica, y Jose Javier Marinez Herráiz (que además nos había presentado) me dejó pensar en ello con total libertad durante dos años.

Sería injusto decir exclusivamente que, a mis padres, les debo estos estudios. Y tú Natalia, has sido la protagonista de este relato... Y has aguantado la alarma a las 7 de la mañana durante todo el mes de agosto para que este trabajo estuviese hoy aquí.

¡Muchas gracias a todos!

Índice general

1. Introducción	17
1.1. Estandarización	17
1.1.1. Seguridad	17
1.1.2. Aplicaciones	18
1.1.3. API	18
1.2. Objetivo del trabajo	19
2. Estudio teórico	21
2.1. Tipos	21
2.2. Primitivas matemáticas	22
2.2.1. Lattice-based encryption	22
2.2.2. Learn With Errors (LWE)	24
2.3. Generaciones	25
2.3.1. Pre-HE	26
2.3.2. Primera generación	27
2.3.3. Segunda generación	28
2.3.4. Tercera generación	29
3. Implementaciones	31
3.1. Librerías	31
3.2. Microsoft SEAL	32
3.2.1. API	32
3.2.2. EncryptionParameters	33
3.2.3. SEALContext, niveles de error y realinearización	36
3.2.4. Encoders	37
3.2.5. Trabajo con vectores	38
3.2.6. Evaluator	39
3.3. TFHE	40

3.3.1.	API	41
3.3.2.	Evolución de TFHE	44
4.	Solución propuesta	47
4.1.	Funcionamiento del sistema	48
4.1.1.	Generación del modelo	48
4.1.2.	Obtención de la posición	49
4.2.	Implementación con TFHE	51
4.2.1.	Curva de regresión	52
4.2.2.	tfhe-math	53
4.3.	Implementación con SEAL	57
4.3.1.	Cálculo de la posición	57
4.4.	Implementaciones Cliente/Servidor	59
4.5.	Evaluación de límites y rendimientos	62
5.	Resultados	67
5.1.	TFHE	67
5.1.1.	Tiempos de ejecución	67
5.1.2.	Límites de cómputo	70
5.1.3.	Problemas encontrados	71
5.2.	SEAL	72
5.2.1.	Tiempos de ejecución	72
5.2.2.	Límites de cómputo	73
5.2.3.	Problemas encontrados	75
5.3.	Coste del producto	76
5.3.1.	Coste de desarrollo	76
5.3.2.	Coste de despliegue	76
6.	Conclusiones	81
7.	Trabajos futuros	83

A. Test LWE	91
A.1. Código en python	91
A.2. Ejecución	93
B. Regresión cuadrática	95
B.1. Ejemplo en python	95
B.1.1. Ejecución	97
B.2. Funciones en C++	98
C. Datos AEMET	103
C.1. Obtención de datos	103
C.2. Procesado	104
D. Tests de eficiencia de SEAL	107
D.1. BFV	107
D.2. CKKS	109
E. Servidor en Digital Ocean	113
E.1. Factura de servicios	113
F. Manual de instalación y uso	115
F.1. Prerequisitos	115
F.1.1. SEAL	115
F.1.2. TFHE	115
F.2. Instalación	117
F.2.1. tfhe-math	117
F.2.2. tfhe-cs	117
F.2.3. seal-cs	117
F.3. Uso	118
F.3.1. tfhe-math	118
F.3.2. tfhe-cs	118
F.3.3. seal-cs	118

G. Archivos de cabeceras	119
G.1. tfhe-cs	119
G.1.1. Cliente	119
G.1.2. Servidor	120
G.2. seal-cs	120
G.2.1. Cliente	120
G.2.2. Servidor	122
G.2.3. Curvas	123

Índice de cuadros

3.1. Relación <i>poly_modulus_degree/coeff_modulus_degree</i>	35
4.1. Resultados del servidor TFHE	60
5.1. Tiempo estimado de ejecución	68
5.2. Tiempo real de ejecución	69
5.3. Sub-operaciones de regresión cuadrática	69
5.4. Tiempo real de regresión	72
5.5. Tests de eficiencia de SEAL (microsegundos)	73
5.6. Profundidad computacional máxima BFV	74
5.7. Profundidad computacional máxima CKKS	75

Índice de figuras

2.1. Espacio vectorial generado por u y v	22
2.2. Short vector problem (contributors., 2019e)	23
2.3. Closest vector problem (contributors., 2019e)	24
2.4. LWE (Halevi, 2017)	25
3.1. Curva de seguridad de λ (Chillotti, Gama, Georgieva y Izabachène, 2016a)	42
3.2. Circuito lógico de suma	45
4.1. Flujo de los datos cifrados (Diagrama generado con Piktochart https://piktochart.com)	49
4.2. Curva de Regresión Cuadrática con temperaturas de Cabo de Gata	50
4.3. Temperatura vs Regresión	51
4.6. Ejecución completa del sistema de SEAL	62
4.4. Algoritmo de multiplicación	64
4.5. Algoritmo de división	65
5.1. Tiempo de ejecución por número de bits	70
5.2. Crecimiento: división y multiplicación	71
5.3. Panel web de estadísticas del servidor	78
6.1. “FHE will never be practical within the next 10 years?”	82
C.1. Panel de descarga de datos de AEMET	103
F.1. Parámetros de configuración de tfhe	116

1 Introducción

Para todo $A, B \in \chi$, una operación \oplus , y una función f ; si se cumple $f(A) \oplus f(B) = f(A \oplus B)$, f es una función homomórfica con respecto a \oplus en χ . Cuando una función criptográfica cumple esta condición con alguna operación se dice que es maleable (Dolev, Dwork y Naor, 1991). Aunque es una propiedad que podría no ser deseable en muchos ámbitos (por ejemplo, en un escenario en el que además de confidencialidad se requiera integridad, permitiría a un atacante modificar los datos), si cumple ciertas condiciones puede tener numerosas aplicaciones. Así se crea el campo de estudio de la criptografía homomórfica.

Se conoce como criptografía homomórfica al conjunto de técnicas criptográficas destinadas a permitir operar con datos cifrados y que dichas operaciones se materialicen correctamente sobre los datos al descifrarlos. En función (principalmente) de las operaciones con las que se cumple esta premisa, o el sistema utilizado para procesar los datos antes y después de operar, los esquemas con propiedades homomórficas se categorizarán de una forma u otra. Aunque pueda haber muchas variantes de cada una de estas propiedades, la comunidad científica ha establecido criterios y notaciones para su estudio.

1.1 Estandarización

El consorcio “Homomorphic Encryption Standardization” (Albrecht y col., 2018) ha ido desarrollando un estándar a lo largo de los años atendiendo a los avances en las distintas tecnologías que componen la criptografía homomórfica, y prestando un interés especial en las implementaciones necesarias para ponerla en práctica. Así, se han ido sucediendo las tres generaciones de criptografía homomórfica.

El último encuentro de trabajo del grupo se produjo el 17 de Agosto en Santa Clara, y en él se trabajará principalmente la eficiencia, la seguridad y la usabilidad de las librerías.

La documentación del estándar está dividida en tres *white papers* y una lista de implementaciones conocidas.

1.1.1 Seguridad

En el documento (Chase y col., 2017) analizan qué principios seguir para implementar esquemas de criptografía homomórfica y qué beneficios tienen estos esquemas para la seguridad de

la información. También hacen un análisis de los ataques existentes a estos esquemas y qué parámetros matemáticos son los ideales para hacer estos esquemas lo más resistentes posible tanto en el panorama actual como en un escenario *post-quantum*.

1.1.2 Aplicaciones

En el estudio (Archer y col., 2017) abordan para qué campos es útil la criptografía homomórfica. Mientras que se han estado centrando los esfuerzos en poder almacenar información en la nube de forma segura (cifrándola) se ha descuidado la seguridad de dicha información cuando sube a la nube para ser procesada. La criptografía homomórfica puede ser la solución a este problema, y en campos como:

- La computación distribuida
- La protección de datos médicos que tengan que ser procesados en equipos potentes, ajenos a la institución médica
- La consulta de información de forma anónima: por ejemplo, una consulta DNS sin revelar a qué se está accediendo

En la última reunión del estándar presentaron ejemplos de protocolos de distribución de datos y claves (Troncoso-Pastoriza y Rohloff, s.f.) utilizando nuevas implementaciones como Lattigo (*Lattigo 1.0*, 2019).

1.1.3 API

Por último el consorcio de estandarización busca establecer un modelo de almacenamiento común tanto de los datos como de las claves, y una terminología (lo llaman *lenguaje ensamblador*) que sirva de lenguaje común para transmitir las ideas y los elementos mínimos que debe tener cualquier sistema de criptografía homomórfica. Están definidos en el documento (Brenner y col., 2017), y como veremos más adelante, encajan perfectamente con los elementos de las implementaciones. Son los siguientes:

- SecKeygen, PubKeygen

Tiene que haber un método de generación de clave privada (o secreta) y pública.

- SecEncrypt, PubEncrypt

Define la posibilidad de que haya además de un sistema de cifrado público (usando la clave pública) que en determinados esquemas se pueda cifrar la información directamente con la clave privada.

- Decrypt

Tiene que haber un sistema para restaurar la información desde un texto cifrado.

- Eval

La llamada Eval será el paraguas que recoja todas las operaciones homomórficas que se puedan realizar sobre el texto cifrado para que luego se materialicen en el descifrado.

Algunos esquemas introducirán otras herramientas enfocadas a procesar los datos, pero estarían en niveles superiores de la arquitectura, y son exclusivas de cada implementación.

1.2 Objetivo del trabajo

El objetivo de este trabajo es evaluar si es viable o no utilizar las tecnologías existentes en sistemas y procesos reales. Estudiaremos qué implementaciones hay, en qué consisten, y cuales serán las más idóneas (las más avanzadas, que puedan servir de muestra para inferir la viabilidad de las demás) atendiendo a:

- La facilidad de uso: A fin de cuentas la documentación existente y la mayor o menor facilidad de uso se traduce en horas de salario de trabajadores altamente cualificados.
- Las capacidades de la tecnología: O qué problemas pueden resolverse con ella
- La eficiencia de la solución: Estudiar los tiempos de ejecución de las operaciones

2 Estudio teórico

Para la computación de criptografía homomórfica necesitaremos comprender las bases teóricas matemáticas y los distintos niveles de homomorfismo, además de las herramientas de notación y computación definidas por el estándar...

2.1 Tipos

Las generaciones publicadas por el estándar guardan una estrecha relación con las capacidades que tienen los esquemas para trabajar con los datos cifrados. Estas capacidades que van desde la posibilidad de aplicar algún homomorfismo a poder trabajar libremente con el texto cifrado están categorizadas en tres niveles. Para comenzar, veremos cuales son estos tres tipos:

- Partially Homomorphic Encryption

Existe algún homomorfismo dentro del esquema de cifrado, pero este no es explotable para realizar cálculos arbitrarios con la información. Por ejemplo, el producto en RSA (ver [2.3.1](#))

- Somewhat Homomorphic Encryption (SHE)

Se pueden realizar operaciones arbitrarias, pero el sistema hace que a medida que se procesa la información aumenta el nivel de error del resultado (como veremos, los esquemas de cifrado son semi-probabilísticos), hasta destruir la información. Hay técnicas para aumentar este umbral de error, pero sigue teniendo límites.

- Fully Homomorphic Encryption (FHE)

Los sistemas FHE son los más codiciados dentro del campo, porque permiten realizar cualquier cálculo con la información cifrada sin que aumente el nivel de error y se vuelva irrecuperable. Es cierto que actualmente estos esquemas actualmente son menos eficientes que los anteriores, pero como dicen en TFHE (Gama, [s.f.](#)): “Si Spiderman puede balancearse sobre su cuerda el tiempo suficiente para lanzar una nueva cuerda, ¡puede volar!”

2.2 Primitivas matemáticas

La seguridad de los sistemas criptográficos se basa en problemas matemáticos que, si bien pueden ser resolubles, dicha resolución no es computacionalmente viable en un tiempo razonable: factorización de enteros, logaritmo discreto, ordenación de conjuntos...

La base de los principales sistemas de criptografía homomórfica modernos son problemas relacionados con unas estructuras algebraicas conocidas como retículos.

2.2.1 Lattice-based encryption

Un retículo o red (también conocidos como lattice en inglés) es un conjunto de elementos similar a un espacio vectorial discreto generado por la combinación de una base vectorial concreta. Por ejemplo, a los vectores $u = (2, 0)$ y $v = (-1, 3)$ serían la base de la red bidimensional generada por todas sus combinaciones $n * u + m * v$ (ver 2.1).

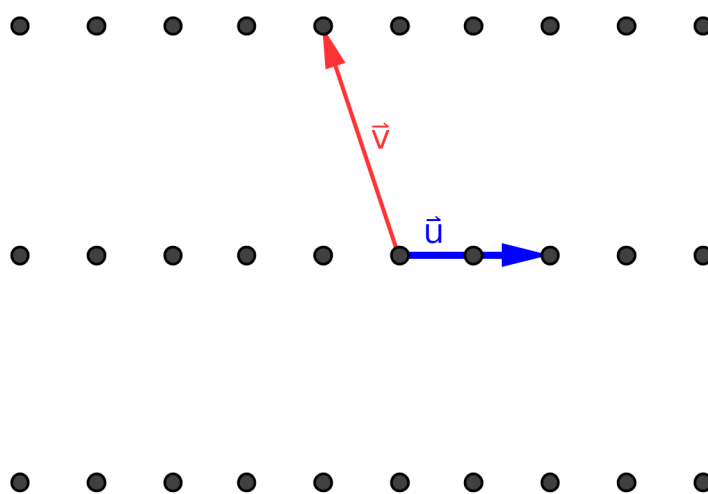


Figura 2.1: Espacio vectorial generado por u y v

Es decir, el conjunto de puntos del gráfico 2.1 sería un retículo, cuya base está formada por los vectores u y v .

En función de lo cerca o lejos que estén los vectores que forman la base del origen se dirá que

estas bases son largas o cortas (Wickr, 2018). Por ejemplo, el espacio anterior podría formarse con el mismo vector v , y con otro vector $w = (1, 0)$ más corto que u .

En la búsqueda de soluciones criptográficas resistentes a la computación cuántica se han encontrado útiles los siguientes problemas de retículos:

- Short Vector Problem (SVP)

Consiste en dada una base larga, buscar un vector corto lo más cercano al origen, sin ser el origen mismo (ver 2.2).

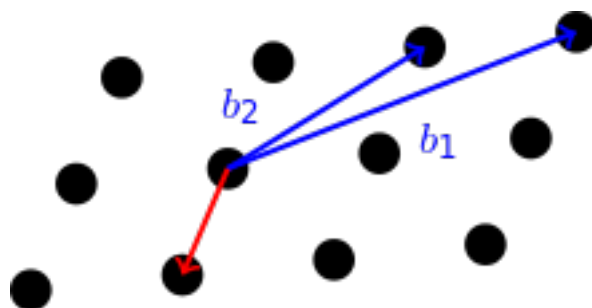


Figura 2.2: Short vector problem (contributors., 2019e)

- Short Basis Problem (SBP)

Dada una base larga buscar una base corta del mismo espacio.

- Closest Vector Problem (CVP)

Dada una base larga y un punto P de la red, buscar el vector de la red formada por la base más cercano a P (ver 2.3).

Aunque estos problemas puedan parecer triviales a simple vista, su complejidad computacional aumenta exponencialmente con el aumento del número de dimensiones hasta hacerlo computacionalmente irresoluble.

Llamaremos Lattice-based encryption (para ajustarnos a la bibliografía, que está en su práctica totalidad escrita en inglés) a la aplicación de este conjunto de problemas a la criptografía.

En cuanto a nuestro caso, el problema del aprendizaje con errores aplicado a retículos será el núcleo de la criptografía homomórfica.

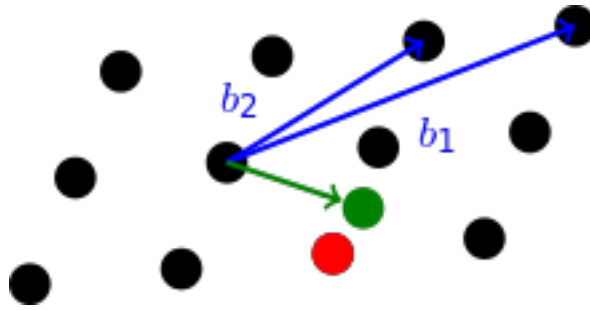


Figura 2.3: Closest vector problem (contributors., 2019e)

2.2.2 Learn With Errors (LWE)

El problema del aprendizaje con errores plantea la dificultad de determinar los componentes de una función en base a sus resultados cuando estos contienen errores (Apon, s.f.).

Dada una base modular q , n vectores $a_i \leftarrow \mathbb{Z}_q^m$, un vector $s \leftarrow \mathbb{Z}_q^n$ y n vectores $e_i \leftarrow \chi^m$ tomados de una distribución de error (distribución de Gauss (contributors., 2019c)) $\chi \subset \mathbb{Z}$:

$$b_i = (a_i \times s + e_i) \pmod{q} \quad (2.1)$$

Conociendo m pares (a_i, b_i) no se puede determinar el valor s , y no se pueden diferenciar dichos pares de una distribución aleatoria (Zijlstra, s.f.).

- Uso en criptografía

En su estudio, Regev (Regev, 2010) formula un sistema de clave pública utilizando este problema:

1. Generación de clave Pública

Se emite como clave pública una colección de valores (a_i, b_i) generada como hemos visto en 2.1. El conjunto de vectores a_i se interpretará en algunas implementaciones como una matriz $A \leftarrow \mathbb{Z}_q^{m \times n}$.

2. Cifrado

Para cada bit x a cifrar se elige una de las parejas y se realiza la siguiente operación:

$$\boxed{\vec{b}} := \boxed{\vec{s}} \times \boxed{A} + \boxed{\vec{\eta}}$$

Figura 2.4: LWE (Halevi, 2017)

$$(c1, c2) = \left(\sum_{j=1}^m a_{ij}, \sum_{j=1}^m b_{ij} + x * (q/2) \right) \quad (2.2)$$

3. Descifrado

El resultado realmente no es determinista, es decir, no devuelve realmente el valor cifrado. Para determinar el valor calcularíamos:

$$\begin{aligned} x &\approx c_2 - (c_1 * s) \\ b + x * (q/2) - a * s \\ a * s + e + x * (q/2) - a * s \\ e + x * (q/2) &\approx x * (q/2) \end{aligned} \quad (2.3)$$

Si se ha introducido s correctamente, y siendo e despreciable en comparación con $q/2$, obtendríamos como resultado $x * (q/2)$, por lo que sabremos que $x = 1$ si el resultado es cercano a $q/2$ y $x = 0$ si el resultado es cercano a 0.

En el apéndice A puede verse un ejemplo de implementación del algoritmo, y el resultado de su ejecución.

2.3 Generaciones

Las distintas generaciones de esquemas de criptografía homomórfica se han ido cerrando en base a los encuentros de estandarización. Además de la propia estandarización, estos encuentros (concretamente, de la segunda generación en adelante) han servido para crear los grupos

de trabajo que han hecho germinar los avances de la siguiente. Aunque la única categorización realmente aplicable a los esquemas es la que hemos visto al principio de este capítulo (2.1), es interesante ver las distintas fases para conocer la evolución, pues normalmente la diferencia entre unas y otras ha sido simplemente el desarrollo en profundidad de una idea; o el mecanismo de una ha surgido mediante la implementación de una idea radicalmente opuesta a la base teórica de la anterior.

Además las tecnologías de, por ejemplo, la segunda generación, no han sido sustituidas por las de la tercera, si no que este salto generacional indica la existencia de un modelo más maduro hacia la consecución del verdadero objetivo: sistemas FHE eficientes. Siguiendo con el ejemplo, los esquemas de la segunda generación se siguen utilizando para cálculos acotados que requieren cierta eficiencia. Este es el motivo de que, como veremos más adelante, hayamos elegido una tecnología de cada una de estas generaciones para desarrollar nuestra implementación.

2.3.1 Pre-HE

Dentro de la categoría de esquemas previos a la criptografía homomórfica se encuentran aquellos que, o bien tienen propiedades homomórficas de forma casual, o bien no cumplen las condiciones necesarias para que se puedan utilizar en ningún sistema. Hablaremos de RSA por lo intuitivo que es para comprender la criptografía homomórfica, y del sistema desarrollado por Boneh, Goh y Nissim por ser uno de los primeros orientados correctamente a la computación con criptografía homomórfica.

■ RSA

Las propiedades matemáticas de RSA lo convierten en un esquema con homomorfismo en el producto. Para cifrar y descifrar, en RSA se exponencia el elemento al que se le desea aplicar la operación. Siendo e la clave de cifrado y d la clave de descifrado, la aplicación de RSA (puro) sobre el mensaje m sería tal que:

$$c = m^e \pmod{n}$$

$$m = c^d \pmod{n}$$

Esto hace que, si multiplicamos dos mensajes cifrados:

$$c_1 = m_1^e \pmod n, c_2 = m_2^e \pmod n$$

$$c_1 * c_2 = m_1^e * m_2^e \pmod n = (m_1 * m_2)^e \pmod n$$

Podamos obtener el producto de los dos elementos al descifrar:

$$(m_1^e * m_2^e)^d \pmod n$$

$$((m_1 * m_2)^e)^d \pmod n$$

$$m_1 * m_2$$

- Criptosistema de Boneh–Goh–Nissim (Boneh, Goh y Nissim, 2005)

Permite evaluar circuitos lógicos en forma normal disyuntiva (reducido a puertas lógicas or, and y not) sobre texto cifrado. Se traduce en la capacidad de evaluar polinomios de segundo grado. Aunque en comparación con los esquemas actuales este parezca “de juguete”, es un gran aporte a la hora de impulsar la investigación en criptografía homomórfica.

2.3.2 Primera generación

- Bootstrapping: Fully homomorphic encryption using ideal lattices

La técnica de *bootstrapping* de Gentry (Gentry, 2009) revoluciona la criptografía homomórfica. Estipula que para crear un esquema de cifrado que permita la evaluación arbitraria de circuitos lógicos simplemente hace falta un esquema de cifrado que pueda realizar la operación de descifrado sin realmente descifrar el texto. Esto se consigue mediante algo parecido a mezclar el texto cifrado (en el que se produce ruido tras hacer una operación) con una versión cifrada de la clave secreta (conocida como clave de evaluación) para que se elimine el ruido.

Por ejemplo, cuando se evalúa usando la técnica de *bootstrapping* la operación $c_a + c_b$ con c_a, c_b textos cifrados, $e(x)$ función de cifrado y $d(x)$ función de descifrado, el resultado es $e(d(c_a + c_b))$. En la operación de descifrado intermedia se elimina el ruido, y el

propio esquema hace que el evaluador de la operación no pueda revelar la clave secreta. (“homomorphic encryption - What exactly is bootstrapping in FHE?”, [s.f.](#)).

El principal problema de la operación de *bootstrapping* es que consume mucho tiempo. En la segunda generación todos los esfuerzos se centran en crear esquemas más rápidos.

2.3.3 Segunda generación

- BGV (Brakerski, Gentry y Vaikuntanathan, [2012](#))

Esquema de cifrado (enunciado como FHE, pero finalmente categorizado como SHE) basado en LWE que permite evitar el costoso método de bootstrapping mediante la introducción del concepto de “leveled homomorphic encryption” (“homomorphic encryption - difference between leveled FHE and normal FHE scheme”, [s.f.](#)): permite hacer esquemas más eficientes (no se va eliminando el error mediante bootstrapping), pero el número de cálculos que se puede hacer sobre el texto cifrado está acotado hasta un punto en el que, el error acumulado (el error “despreciable” al que hacíamos referencia en la fórmula [2.3](#)), hace irrecuperable el mensaje.

- BFV (Fan y Vercauteren, [2012](#)) Es una implementación optimizada de BGV sobre RLWE (ring-LWE, sistema de LWE sobre elementos de un anillo algebraico, (contributors., [2019a](#))) que mejora la eficiencia del esquema y aumenta la cota de cómputo (el número de operaciones que se pueden realizar hasta que el error del cifrado lo vuelva inconsistente) de una técnica conocida como realinealización.

- CKKS (Cheon, Kim, Kim y Song, [2017](#))

No está descrito en el estándar, pero sí se plantea su implementación y se extiende su uso en varias librerías. Su funcionamiento consiste en introducir una función de reescalado del texto cifrado a medida que se va trabajando con él. El reescalado trunca el mensaje cifrado (operación equivalente a dividir para reducir su magnitud) eliminando progresivamente el error cada vez que se opera. De esta forma se pueden realizar operaciones aritméticas con números reales (pertenecientes a \mathbb{R}) e ir eliminando el error (se reduce cuando se trunca el mensaje) a medida que se va operando.

Introduce además una técnica de codificación de los datos que nos permitirá trabajar con

ellos como vectores, permitiendo aplicar técnicas SIMD (Single Instruction Multiple Data (contributors., 2017)) para paralelizar determinadas tareas, técnica que utilizaremos en nuestra implementación con Microsoft SEAL (3.2).

2.3.4 Tercera generación

- GSW (Gentry, Sahai y Waters, 2013)

El principal esquema de la tercera generación ya es considerado realmente FHE, pues permite la evaluación de todas las operaciones necesarias para poder realizar cualquier cómputo, todas las veces que hagan falta (sin cotas). En este nuevo esquema se implementa el problema LWE sobre matrices, tratando la clave como una matriz cuyo auto-vector es, aproximadamente (por el pequeño error del problema LWE), el valor secreto. En este esquema desaparece la necesidad de utilizar la clave de evaluación para operar (hace opcional la operación de bootstrapping), pudiendo trabajar directamente con el dato cifrado.

- TFHE (Chillotti, Gama, Georgieva y Izabachène, 2016a)

El éxito de GSW como esquema FHE reside en que todos los parámetros son extremadamente pequeños comparados con q (la base modular), y la implementación de mecanismos para que el error acumulado crezca muy lento. De esta forma, no es necesaria la operación de bootstrapping en la mayoría de los casos (que hasta el momento, tardaba alrededor de 6 minutos por operación (Ducas y Micciancio, 2014)), y siempre puede aplicarse cuando el cálculo vaya a crecer mucho. El esquema FHEW logra un hito reduciendo el tiempo de bootstrapping hasta 1 segundo por operación, pero no es el último paso

Basado en una aproximación a LWE conocida como TLWE (Torus Learn With Errors) en la que los parámetros de LWE se definen sobre el espacio de un toro, y siguiendo la trayectoria del esquema FHEW (Ducas y Micciancio, 2014), TFHE puede realizar la técnica de bootstrapping en menos de 0,1 segundos. Esto permite su aplicación sobre GSW sin perder eficiencia. Además, reduce el tamaño de las claves hasta 16MB (en lugar del GB que ocupaban hasta el momento) manteniendo el mismo nivel de seguridad.

Utilizaremos la librería homónima (Chillotti, Gama, Georgieva y Izabachène, 2016b) para construir parte de la implementación del trabajo.

3 Implementaciones

3.1 Librerías

Hay varias implementaciones de criptografía homomórfica, y varias soluciones por cada una de las generaciones. Aunque la mayoría de las implementaciones están escritas para C/C++/C#, recientemente han ido apareciendo algunas nuevas que buscan, principalmente, adaptar los esquemas de cifrado a otros lenguajes. Aunque esto es una buena noticia, nosotros nos centraremos en las “clásicas” evaluadas por el consorcio de estandarización:

- Segunda generación

- HELib
- Microsoft SEAL
- PALISADE
- HeaAn
- LoL
- NFLlib

- Tercera generación

- TFHE
- FHEW

Hay además dos tecnologías remarcables que parecen ser las que marquen el paso de las vías de desarrollo de criptografía homomórfica:

- cuHe (<https://github.com/vernamlab/cuHE>)

Implementación de esquemas de criptografía homomórfica con CUDA para acelerar los circuitos mediante GPU.

- Cingulata (<https://github.com/CEA-LIST/Cingulata>)

Sistema de compilación diseñado para transformar código C++ en circuitos FHE. Soporta los esquemas BFV y TFHE.

Para nuestro trabajo utilizaremos las librerías Microsoft SEAL (como representante de la segunda generación de criptografía homomórfica) y TFHE (como representante de la tercera). Para la segunda generación habría sido también una muy buena opción utilizar PALISADE, pero la documentación es mucho menor, y no incluye el esquema CKKS (necesario para trabajar con números reales).

3.2 Microsoft SEAL

Esta librería de código abierto desarrollada por Microsoft (*SEAL* de ahora en adelante) busca ofrecer una opción asequible para los desarrolladores de implementar soluciones con criptografía homomórfica.

Es muy fácil de instalar, no tiene dependencias externas, y está diseñada para ser construida en cualquier entorno con `cmake`.

Cuenta con dos esquemas de cifrado de segunda generación (BGV y CKKS) y dentro de su código fuente se incluyen varios ejemplos que muestran cómo se opera con ella. En estos ejemplos, ordenados para conocer las distintas herramientas, muestran todo lo necesario para empezar a trabajar.

A la hora de implementar una idea en SEAL, hay dos aspectos clave a tener en cuenta:

1. Elegir el esquema de cifrado que nos permita codificar todo correctamente
2. Estudiar si la operación realmente es realizable con estos esquemas (recordemos que SHE tiene una cota de cómputo)

Además, hay que ser consciente de que bajo determinados usos, puede ser insegura. Por ejemplo, no se debe permitir el descifrado desde un entorno no controlado (no es CCA seguro (Peng, 2019)).

3.2.1 API

La API para operar con SEAL está codificada en los siguientes elementos:

- **EncryptionParameters**

Parámetros descriptivos de los elementos criptográficos

- SEALContext

Contexto del texto cifrado (cambia a medida que se opera)

- Claves

- SecretKey

Clave secreta para descifrar los datos

- PublicKey

Clave pública para cifrar los datos

- RelinKeys

Claves públicas para realinearizar los datos y reducir el nivel de error acumulado tras operar

- GaloisKeys

Claves públicas para trabajar con rotaciones en los vectores cifrados (en nuestro ejemplo no las usamos)

- Encryptor, Decryptor

Sistemas para cifrar y descifrar los datos

- Evaluator

Es el componente que realiza las operaciones sobre los datos cifrados

- Encoder

Encargado de codificar los datos en función del esquema que se desee utilizar

A continuación explicaremos más detalladamente el funcionamiento de los componentes más complejos.

3.2.2 EncryptionParameters

Los parámetros criptográficos dependerán del esquema con el que se quiere trabajar.

- BFV

BFV es el esquema “básico” de SEAL. Permite trabajar con números enteros, y operar con ellos hasta que se alcanza el límite máximo de error. En BFV, este límite se podría conceptualizar como un cubo de fichas que se gastan cada vez que se opera.

Hay algunas operaciones que son casi gratuitas (la suma y la resta), y la multiplicación es muy costosa. Una vez se vacía este cubo (codificado en el parámetro `noise_budget`), nuestro cifrado quedará corrupto y no se puede recuperar el texto.

Además de `noise_budget`, hay tres parámetros configurables que guardan una estrecha relación:

- `poly_modulus_degree`

Determina el módulo del polinomio usado para realizar las operaciones criptográficas (recordemos que implementa LWE sobre un anillo de polinomios, ver [2.3.3](#)). Se expresará como una potencia de 2 que cuanto más grande sea, más operaciones permitirá hacer, pero serán más lentas.

- `coeff_modulus`

Es un vector de números primos que determina el módulo del texto cifrado. A mayor módulo, mayor será el `noise_budget`. Cuando decíamos que a mayor `poly_modulus_degree`, podíamos hacer más operaciones, era porque el número de bits de `coeff_modulus` está acotado por el de `poly_modulus_degree` de la forma indicada en la [figura 3.1](#)

- `plain_modulus`

Módulo del texto plano. El consumo de `noise_budget` se produce de forma logarítmica en base al tamaño de `plain_modulus`, por lo que cuanto más pequeño es, más operaciones podremos realizar. También debe ser menor que `poly_modulus_degree`.

- CKKS

En la implementación del esquema CKKS podremos trabajar con número decimales, pero tendremos que realizar algunas gestiones adicionales a las de BGV para evitar que se pierda la integridad de nuestros cálculos.

<code>poly_modulus_degree</code>	Número de bits de <code>coeff_modulus</code>
1024	27
2048	54
4096	109
8192	218
16384	438
32768	881

Cuadro 3.1: Relación *poly_modulus_degree*/*coeff_modulus_degree*

Comparte con BGV tanto `poly_modulus_degree` como `coeff_modulus`, pero desaparece el elemento `plain_modulus` (la integridad del texto cifrado ya no se evaluará con el `noise_budget`), y los valores elegidos para `coeff_modulus` tendrán otras implicaciones.

Aparece un elemento nuevo: la escala. De forma análoga a la que hemos implementado en la maqueta de THFE (ya lo veremos en el capítulo 4) para codificar los números decimales se aplica una escala (se multiplican por un valor muy alto, potencia de 2) y se tratan como número enteros.

Esta escala aumentará drásticamente cada vez que se multipliquen dos elementos. Siendo N el tamaño del primer factor, y M el del segundo, el resultado del producto tendrá como tamaño $M + N - 1$. Para evitar que el número desborde su tamaño máximo, tras un producto puede reducirse la escala. La operación de reescalado implementada en SEAL trunca el valor del texto cifrado tantos bits como tenga el último elemento del vector `coeff_modulus`, y elimina este elemento (cambiando el contexto de cifrado). De esta forma, siendo P el tamaño del elemento eliminado de `coeff_modulus` el texto cifrado pasaría a tener un tamaño $(M + N - 1)/P$. Una vez se terminan los elementos de `coeff_modulus` no se puede seguir operando, y se debe conservar siempre uno para poder realizar la operación de descifrado.

Para poder reescalar sin problemas se debe elegir una escala inicial menor que el número expresado por el último valor de `coeff_modulus` (cuando `coeff_modulus` está integro). Además, el primer valor de la cadena (el que se utilizará cuando se vaya a descifrar), debe ser mayor que el resto para poder tener cierta precisión: si este valor es 60, y el segundo (el que se utiliza al

realizar la última operación) es 40, tendremos 20 bits de precisión ($60 - 40$) para los decimales al retirar la escala.

A continuación veremos en qué consisten exactamente el contexto y la cadena de la que hablamos.

3.2.3 SEALContext, niveles de error y realinealización

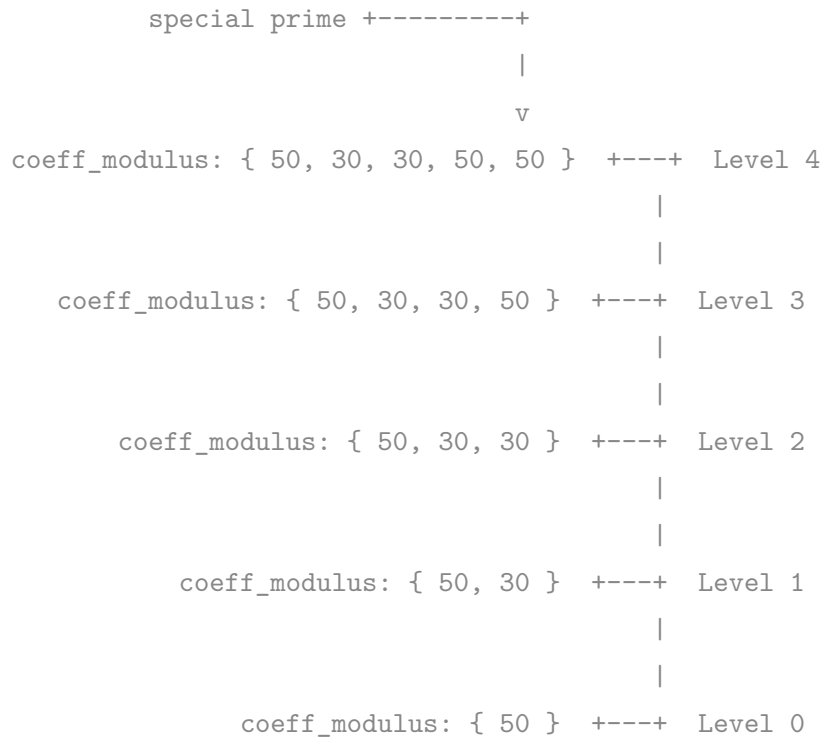
SEALContext (lo que llamamos contexto, o contexto criptográfico) es la clase en la que se codifican las propiedades del texto cifrado, desde el esquema utilizado a los distintos parámetros de cifrado. La creación del SEALContext genera una estructura similar a una cadena que guardará el estado de `coeff_modulus`. De esta forma, podremos mantener la integridad al operar entre distintos textos cifrados simplemente verificando que esta cadena es igual.

Inicialmente, la cadena se inicializa con la información de `coeff_modulus`, y va cambiando a medida que, por ejemplo, se aplican operaciones de reescalado. La cadena es una lista enlazada de los SEALContext en cada momento de operación:

Se puede descender, pero nunca ascender. Cuanto más abajo, más rápidos son los cálculos, pero menos cálculos se pueden hacer. En ocasiones, debido al tamaño del texto plano, descender no tendrá consecuencias en cuanto al nivel de error, por lo que se podrá hacer para tener mayor eficiencia. En otros, será una condición indispensable para poder seguir operando. Es por esto que, cumpliendo los requisitos sobre el tamaño de `coeff_modulus` con respecto a `poly_modulus_degree`, y siguiendo las pautas que hemos comentado con respecto al primer y último elemento de la cadena, es recomendable que elijamos el mayor número de elementos intermedios posible.

Mientras que en BGV este contexto no tiene mucha importancia (más allá de la eficiencia) en CKKS la cosa cambia. Cuando se reescala en CKKS, se está eliminando al mismo tiempo parte de esta cadena. Por lo tanto, hay que tener en cuenta que si se ha reescalado, y se ha cambiado el contexto de cifrado de un elemento, tendremos que ajustar el contexto del resto de elementos que queramos operar con él (para que estén en la misma escala y usen los mismos parámetros criptográficos), ya sea reescalando (cuando sea necesario) o haciendo al elemento “descender” en la cadena.

Otra forma de reducir la velocidad a la que aumenta el nivel de error sin perder profundidad computacional (capacidad para realizar más operaciones) es la realinealización. Como hemos



Listing 1: Cadena de SEALContext (documentación de SEAL)

visto anteriormente, a medida que se opera, el error aumenta. La suma es casi gratuita, pero con el producto de dos elementos de tamaño M y N , teniendo el resultado tamaño $M + N - 1$, el error aumenta considerablemente. Interpretaremos este tamaño como el grado polinómico del elemento. La realinearización es una operación que permite, utilizando unas claves especiales (claves de realinearización), reducir el tamaño tras una multiplicación, reduciendo así el consumo de `noise_budget` en las siguientes operaciones. Tiene un coste computacional alto en comparación con el resto de operaciones, pero permite operar de forma más eficiente, y que el elemento no crezca hasta “romperse” (o crezca de una forma más lenta).

3.2.4 Encoders

SEAL ofrece tres sistemas para codificar los datos con los que se va a trabajar:

- IntegerEncoder (para BFV)

Codifica el número descomponiéndolo en una sucesión de exponenciaciones para poder trabajar con sus subelementos de forma independiente.

- BatchEncoder (para BFV)

Crea una matriz $2 \times (N/2)$ con N el número de elementos codificables. Estos elementos son conocidos como *slots*, y su número es igual a `poly_modulus_degree`. En cada *slot* se almacena un número en base modular $\mathbb{Z}_{\text{plain_modulus}}$. Si se codifica un vector, se introducirá tal cual, y si sólo se codifica un valor, se llenarán todas las posiciones de la matriz con ese valor.

$$a = \begin{pmatrix} a & a & \dots & a \\ a & a & \dots & a \end{pmatrix} \quad \begin{pmatrix} c_1 & c_2 & c_3 \end{pmatrix} = \begin{pmatrix} c_1 & c_2 & c_3 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \end{pmatrix} \quad (3.1)$$

- CKKSEncoder

Funciona de forma similar al BatchEncoder, pero tiene la mitad de posiciones con respecto al mismo `poly_modulus_degree`.

3.2.5 Trabajo con vectores

Tanto BatchEncoder como CKKSEncoder, interpretan todos los datos como vectores. Una de las cosas más interesantes de esta forma de codificar los datos, es que se pueden rotar las columnas y las filas. Para ello será necesario el empleo de unas claves especiales llamadas *Galois Keys*. La rotación es cíclica, es decir, cuando se rota una posición a la izquierda el primer valor pasa a la última posición.

- Rotando filas

$$\begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \end{bmatrix} \rightarrow \begin{bmatrix} b_{12} & \dots & b_{1n} & b_{11} \\ b_{22} & \dots & b_{2n} & b_{21} \end{bmatrix} \quad (3.2)$$

- Rotando columnas

$$\begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \end{bmatrix} \rightarrow \begin{bmatrix} b_{21} & b_{22} & \dots & b_{2n} \\ b_{11} & b_{12} & \dots & b_{1n} \end{bmatrix} \quad (3.3)$$

Por otro lado, las operaciones aritméticas se realizan entre los valores de cada posición uno a uno, no como una operación entre matrices. Por ejemplo, el producto sería:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ a_{21} & \dots & a_{2n} \end{bmatrix} * \begin{bmatrix} b_{11} & \dots & b_{1n} \\ b_{21} & \dots & b_{2n} \end{bmatrix} = \begin{bmatrix} (a_{11} * b_{11}) & \dots & (a_{1n} * b_{1n}) \\ (a_{21} * b_{21}) & \dots & (a_{2n} * b_{2n}) \end{bmatrix} \quad (3.4)$$

3.2.6 Evaluator

Por último, tras conocer todo lo que se puede hacer con SEAL, veremos cómo hacerlo. Para trabajar con los elementos cifrados SEAL ofrece las siguientes operaciones, codificadas en la clase Evaluator:

- `negate`

Negar (equivalente a multiplicar por -1) la variable introducida.

- `add`

Función de suma

- `sub`

Función de resta

- `multiply`

Función de multiplicación

- `square`

Función para elevar al cuadrado.

- `exponentiate`

Función de exponenciación.

- `rotate_rows / rotate_columns`

Funciones para rotar los elementos codificados como matrices (con `BatchEncoder` o `CKKSEncoder`).

- `relinearize`

Función de realinearización

- `rescale`

Función para reescalar un elemento al siguiente nivel de su cadena

- `mod_switch`

Función para descender en la cadena del contexto de un elemento

Todas las operaciones aritméticas se pueden aplicar para operar tanto entre elementos cifrados como con números en texto plano (simplemente codificados). Todas devuelven el resultado en una variable adicional, a no se que se añada el sufijo `_inline`, que aplicaría la operación sobre la primera variable introducida en la función.

3.3 TFHE

En TFHE se trabaja directamente con puertas lógicas en lugar de con operaciones aritméticas. A diferencia de SEAL, y entre otras cosas por implementar un único esquema de cifrado, no hay que seleccionar más que un parámetro de seguridad (λ) y aportarle una semilla que le permita generar una clave aleatoria (o al menos, lo más aleatoria posible). En la figura 2 podemos ver lo simple que es comenzar a trabajar con TFHE.

Este parámetro “lambda” (λ) se usa para caracterizar el nivel de seguridad en función de los n bits de entropía de la clave secreta y la tasa de error α . En la figura 3.1 podemos ver cómo están relacionados estos parámetros.

Tiene una API simple pero muy potente, pero para hacer cualquier cálculo hay que implementarlo desde el nivel más bajo. Los datos cifrados se representan como un array de bits sobre el que se opera como si el dato estuviese en claro. Será este array (codificado en la clase `LweSample`) sobre el que aplicaremos los algoritmos y procedimientos que diseñaremos.

Los elementos de TFHE están codificados en sólo unos pocos grupos:

- Parámetros de cifrado (`TFheGateBootstrappingParameterSet`)


```
//generate a keyset
const int minimum_lambda = 110;
TFheGateBootstrappingParameterSet* params =
new_default_gate_bootstrapping_parameters(minimum_lambda);

//generate a random key
uint32_t seed[] = { 314, 1592, 657 };
tfhe_random_generator_setSeed(seed,3);
TFheGateBootstrappingSecretKeySet* key =
new_random_gate_bootstrapping_secret_keyset(params);
```

Listing 2: Inicialización de TFHE (documentación de TFHE)

- Claves
 - Privada (TFheGateBootstrappingSecretKeySet)
 - Pública (TFheGateBootstrappingCloudKeySet)
- Operaciones lógicas sobre texto cifrado (LweSample)
- Operaciones de cifrado y descifrado

Además, TFHE ofrece funciones para limpiar la memoria, dejando en manos del desarrollador la posibilidad de evitar fugas de información una vez no se va a utilizar un elemento.

3.3.1 API

A continuación haremos un repaso de las principales operaciones sobre el texto cifrado incluidas en la librería TFHE.

- bootsCONSTANT

Carga una constante en value en la variable cifrada result.

```
bootsCONSTANT(LweSample* result, int value,
               const TFheGateBootstrappingCloudKeySet* bk);
```

SECURITY CURVE

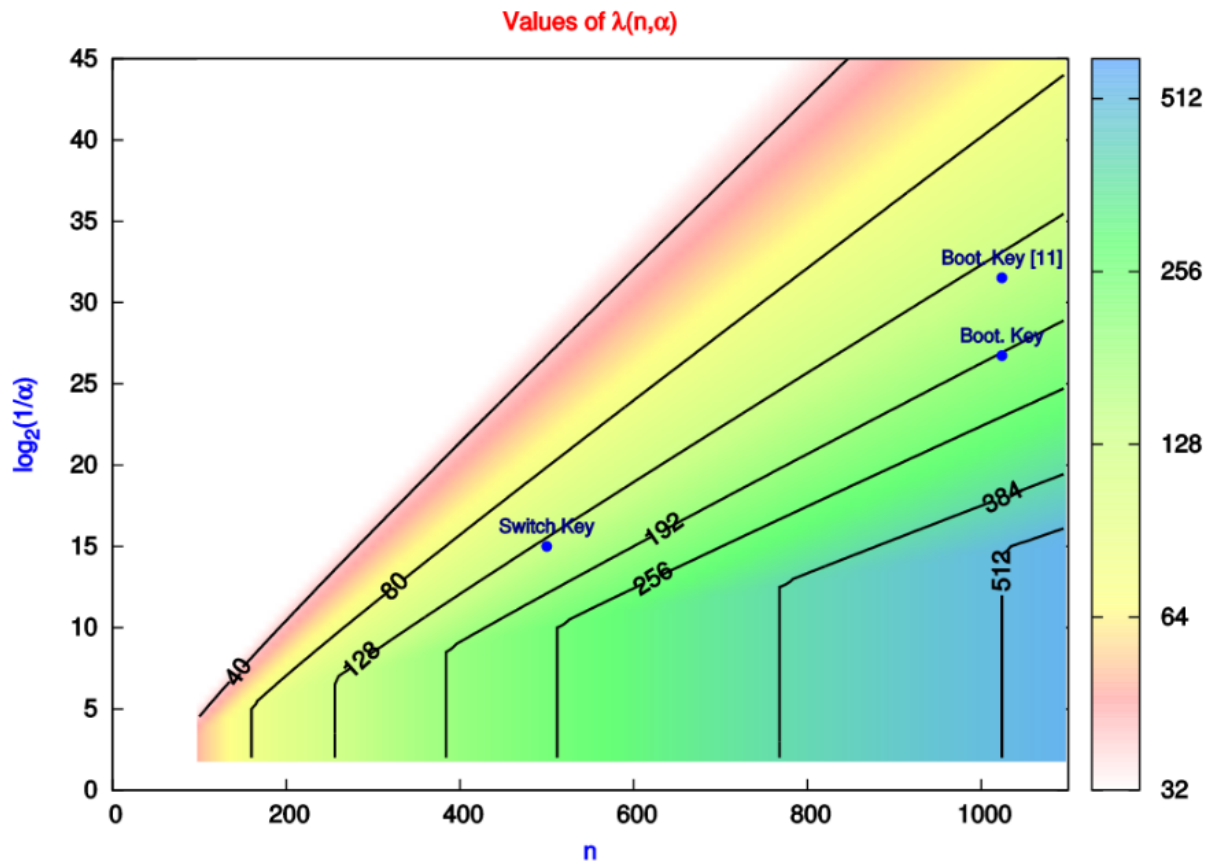


Figura 3.1: Curva de seguridad de λ (Chillotti, Gama, Georgieva y Izabachène, 2016a)

■ bootsNOT

Niega el valor de la variable `ca` y lo almacena en `result`.

```
bootsNOT(LweSample* result, const LweSample* ca,
         const TFheGateBootstrappingCloudKeySet* bk);
```

■ bootsCOPY

Copia el valor de la variable `ca` y lo almacena en `result`. Cuando se utilizan tanto esta función como las funciones `bootsCONSTANT` y `bootsNOT` se suele trabajar bit a bit iterando

sobre los dos arrays. Si no se hace así, a veces pueden tener comportamientos erráticos.

```
bootsCOPY(LweSample* result, const LweSample* ca,  
          const TFheGateBootstrappingCloudKeySet* bk);
```

■ bootsMUX

Es una implementación del operador ternario ($a ? b : c$) esencial para poder introducir cómputos con divergencias en su flujo de ejecución aunque (como hemos comentado anteriormente) no podamos modificarlo. Asigna a `result` el valor de `b` si se cumple `a`, si no, le asigna el valor de `c`.

```
bootsMUX(LweSample* result, const LweSample* a,  
         const LweSample* b, const LweSample* c,  
         const TFheGateBootstrappingCloudKeySet* bk);
```

Esta función es especialmente interesante, y es la que le da todo el valor a la librería para hacer implementaciones complejas. Por ejemplo, una función con el siguiente código:

```
while (result < 100)  
    result = result * 2;
```

No podría ser implementada sin evaluar el valor de `result`. Sin embargo, con el operador MUX podemos hacer lo siguiente (es pseudocódigo):

```
/*  
    Hasta que el menor valor que podemos  
    escribir con los bits que hemos asignado a  
    los decimales (10 bits) no sea mayor que 100  
*/  
for (int i = 0.001; i < 100; i = i*2) {  
    // es_mayor = result >= 100  
    gte(es_mayor, result, 100);  
    // factor = es_mayor ? 2 : 1
```

```
bootsMUX(factor, es_mayor, 2, 1);  
// result = result * factor  
multiplica(result, result, factor);  
}
```

■ Puertas lógicas

También implementa las siguientes puertas lógicas booleanas. La utilizaremos como base para hacer el resto de operaciones, como por ejemplo la suma (la función de suma es igual que un circuito sumador, como el de la figura 3.2, con más bits).

- NAND
- OR
- AND
- XOR
- XNOR
- NOR
- ANDNY, ANDYN
- ORNY, ORYN

3.3.2 Evolución de TFHE

En su desarrollo se plantea la implementación de un interesante modo conocido como Chimera (Boura, Gama, Georgieva y Jetchev, 2018) en el que se puedan mover los datos entre el esquema de TFHE y esquemas más eficientes como BFV y CKKS; y viceversa, sin tener que descifrarlos. Esto puede ser útil, por ejemplo, para trabajar con muchas puertas lógicas y hacer alguna operación aritmética rápida entre medias.

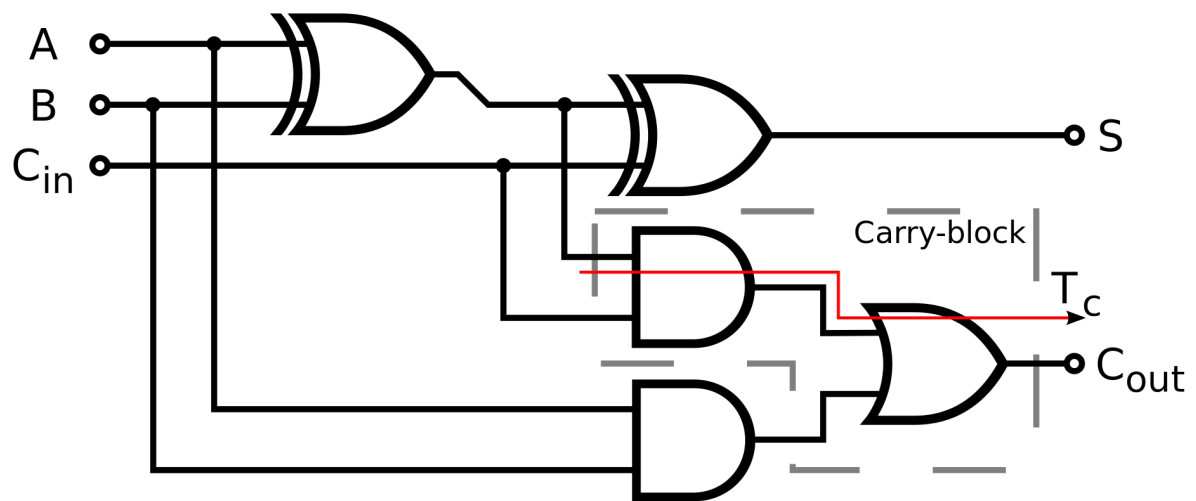


Figura 3.2: Circuito lógico de suma

4 Solución propuesta

Como hemos comentado utilizaremos una librería de la segunda generación de criptografía homomórfica (Microsoft SEAL) y otra para la tercera (TFHE). Comenzaremos con un estudio del funcionamiento de las librerías, de cuales son sus límites computacionales y finalmente implementaremos una operación “no trivial” que requiera la utilización de ambas tecnologías.

La librería SEAL cuenta con pocas operaciones aritméticas implementadas, pero ofrece la posibilidad de trabajar con matrices y números decimales. La principal complicación con SEAL es determinar cómo se está desviando el cálculo para saber cuándo parar o qué correcciones hacer. Además, el número de operaciones que puede hacer son muy limitadas (cuando se hacen, por ejemplo productos, crece mucho el error). Por último hay que tener en cuenta que el algoritmo que se implemente en SEAL no puede contener más que sumas, restas y multiplicaciones (no existe la división).

La librería TFHE ofrece una API de operaciones lógicas a nivel de bit. Aunque permitirá hacer cálculos más complejos sin añadir error al resultado, tendremos que implementar todas las operaciones a bajo nivel mediante puertas lógicas, siempre teniendo en cuenta que en ningún momento vamos a poder controlar el flujo de ejecución y nuestros algoritmos tienen que ser capaces de realizar los cálculos sin poder evaluar las variables (están cifradas).

Por lo tanto tenemos que encontrar un cálculo que sea realizable con pocas operaciones y números bajos para SEAL; y operaciones que sean implementables en un tiempo razonable con puertas lógicas para poder trabajar en TFHE.

Se han elegido tanto el sistema de ejemplo como las tecnologías específicas para cada uno de los actores en función de las capacidades y limitaciones de cada tecnología. Como resultado del proyecto se ha creado, además del análisis y el propio código del proyecto, una librería de operaciones matemáticas implementadas en TFHE (Junquera, 2019) con puertas lógicas, que es innovadora en tanto en cuanto no existe ningún código público que permita trabajar con TFHE a este nivel.

4.1 Funcionamiento del sistema

Para analizar las librerías implementaremos un sistema de posicionamiento anónimo en función de la temperatura y el mes del año. La base para determinar la posición serán dos curvas de regresión cuadrática de temperaturas de dos ciudades (generada con TFHE), que permitirá que un usuario suba a SEAL la fecha y la temperatura cifradas y averigüe su localización.

En este sistema habrá tres actores: el cliente, el servidor de posicionamiento (programado con SEAL) y un tercer servidor (programado con TFHE) que generará el modelo para calcular la posición. El cliente consultará su posición con el servidor de SEAL, que previamente habrá generado en el servidor de TFHE un modelo de posicionamiento basado en las temperaturas del último año en varias ubicaciones (ver [4.1](#)).

Generaremos dicho modelo (la curva de regresión $f(x)$) con TFHE porque para ello necesitaremos realizar operaciones aritméticas que no nos ofrece SEAL. De esta forma, podremos realizar la evaluación de los datos introducidos por el usuario con SEAL (distancia entre el valor de temperatura y especificado por el usuario, y $f(x)$ con x el mes del año) utilizando sus operaciones más básicas, y aprovechando su sistema de vectores para evaluar todas las curvas al mismo tiempo.

4.1.1 Generación del modelo

Para generar el modelo el cliente (en este caso, el servidor de posicionamiento con SEAL) y el servidor (el servidor con TFHE) seguirían el siguiente procedimiento:

1. El cliente genera un par de claves (pública y privada).
2. Cifra n pares de datos (en nuestro ejemplo el par sería $(mes, temperatura)$).
3. Sube los datos cifrados y su clave pública. El número de datos que se puede subir está limitado por el crecimiento del tamaño (en bits) de dichos datos al exponenciarlos para calcular la curva de regresión. Trabajaremos con los datos de 12 meses porque, como comentaremos más adelante (en el capítulo [5](#)), aunque el orden máximo al que llegaríamos con estos datos es de 46 bits tendremos otras limitaciones a la hora de procesar los datos.
4. El servidor procesa los datos y devuelve cifrados los parámetros de la curva

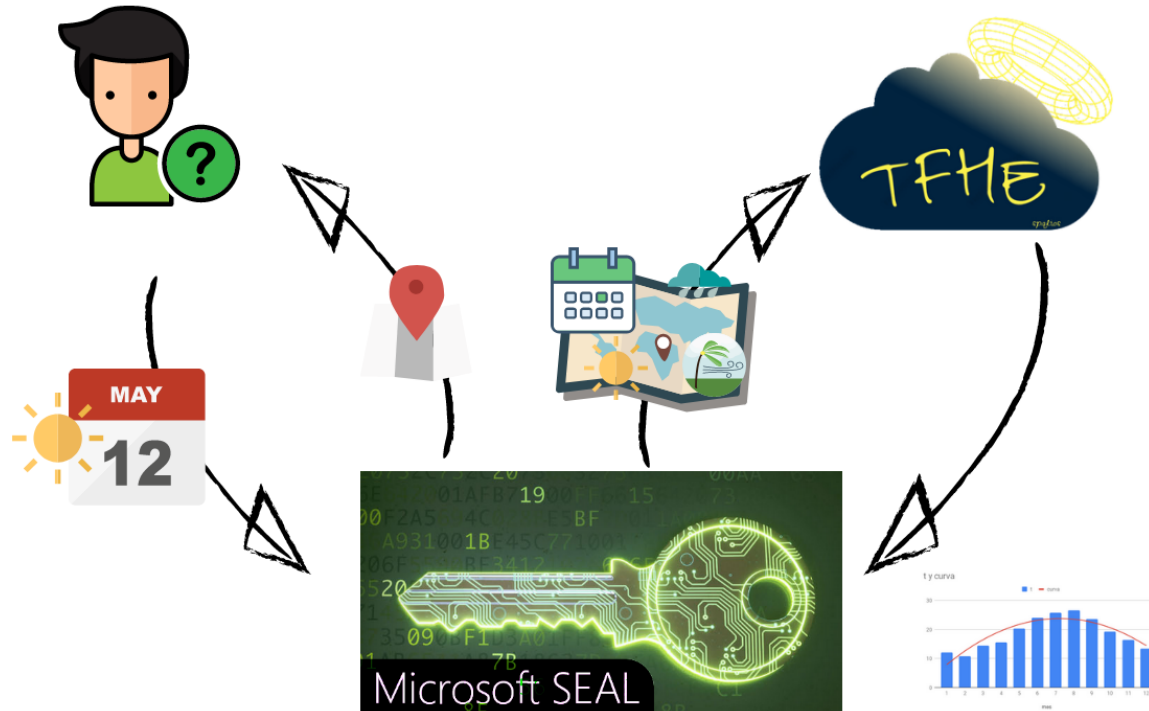


Figura 4.1: Flujo de los datos cifrados (Diagrama generado con Piktochart <https://piktochart.com>)

5. El cliente los descifra y los almacena asociados a una posición geográfica

Finalmente, con los parámetros recibidos, el cliente obtendría una curva similar a 4.2.

4.1.2 Obtención de la posición

Teniendo el servidor de SEAL ya generados los modelos de la temperatura de cada ubicación (en este caso, sólo los 2 de Cabo de Gata y Finisterre) el cliente (ahora ya sí, el usuario final) consultará su ubicación:

1. El cliente genera tres claves: pública, privada y clave de realineación.
2. El cliente cifra (con su clave pública) la temperatura y el mes del año para el que quiere hacer la consulta, generando los valores y e x respectivamente.

Regresión cuadrática (Cabo de Gata)

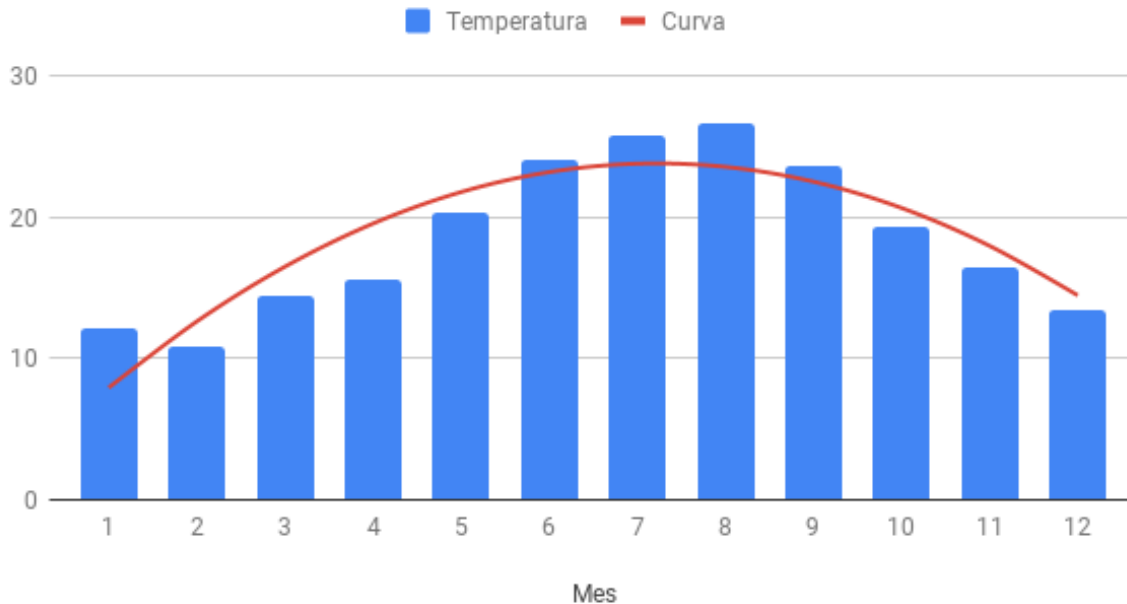


Figura 4.2: Curva de Regresión Cuadrática con temperaturas de Cabo de Gata

3. Sube ambos valores al servidor, junto con la clave de realinearización (en nuestro ejemplo, no se necesitará subir la pública).
4. El servidor calculará la diferencia (resta) entre el punto y y $f(x)$ para cada curva
5. El servidor devuelve una lista con los valores calculados y la ubicación a la que corresponde cada valor
6. El cliente descifra los valores con su clave privada, obteniendo la distancia entre la temperatura introducida y el valor de la curva de cada una de las ubicaciones ese mes.

En la figura 4.3 se puede ver un ejemplo del funcionamiento del sistema de posicionamiento. Los puntos introducidos corresponden a las temperaturas de varias ubicaciones, y el sistema de posicionamiento devolvería la distancia entre estos puntos y las dos curvas de regresión. De

esta forma se puede obtener una estimación de la posición. En el ejemplo vemos que, la temperatura en Cabo de Gata está muy próxima a su curva en el mes de junio, y daría un resultado indeterminado en abril; que la temperatura de Madrid no casaría con ninguna de las dos curvas; o que la temperatura de Finisterre en noviembre efectivamente está mucho más cerca de la curva de Finisterre que de la de Cabo de Gata.

Temperatura / Regresión Cuadrática

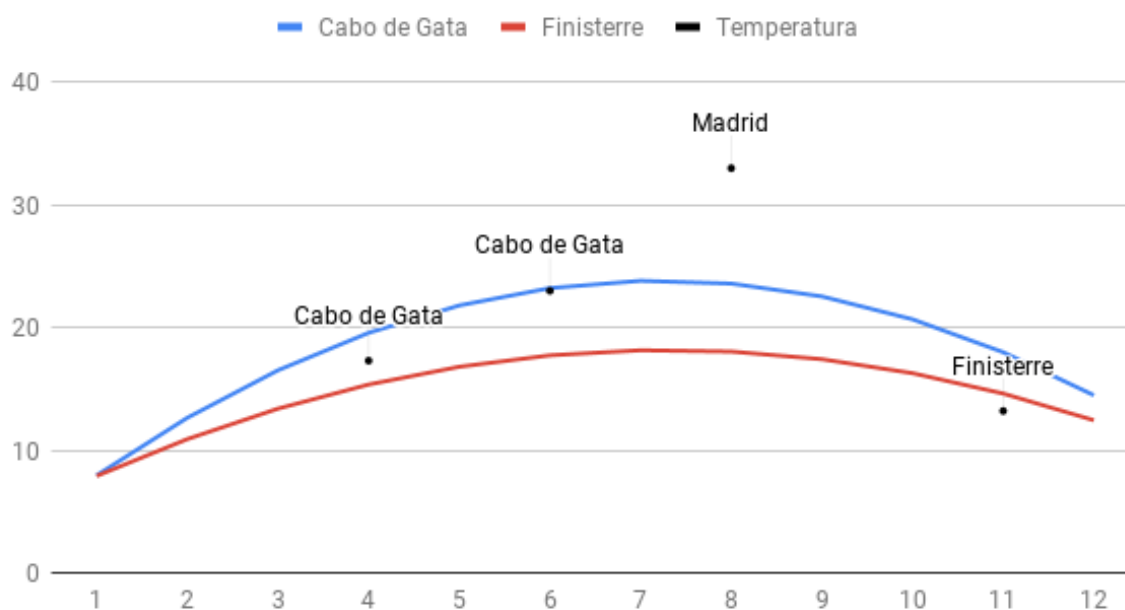


Figura 4.3: Temperatura vs Regresión

4.2 Implementación con TFHE

Para generar la curva de regresión que utilizará el servidor de SEAL para ubicar al usuario, dicho servidor cifrará los datos de temperatura del último año en dos ubicaciones distintas y se las enviará al servidor TFHE. Este procesa los datos cifrados y calcula la regresión cuadrática codificada en tres parámetros a, b, c que devolverá cifrados al servidor de SEAL.

4.2.1 Curva de regresión

Esta curva $y = f(x)$ está definida por tres parámetros (a , b y c) de forma que:

$$y = ax^2 + bx + c$$

Resultante de resolver el siguiente sistema de ecuaciones:

$$\begin{cases} \sum_{i=0}^n y_i = a * \sum_{i=0}^n x_i^2 + b * \sum_{i=0}^n x_i + n * c \\ \sum_{i=0}^n x_i * y_i = a * \sum_{i=0}^n x_i^3 + b * \sum_{i=0}^n x_i^2 + c * \sum_{i=0}^n x_i \\ \sum_{i=0}^n x_i^2 * y_i = a * \sum_{i=0}^n x_i^4 + b * \sum_{i=0}^n x_i^3 + c * \sum_{i=0}^n x_i^2 \end{cases}$$

Realizando las siguientes sustituciones:

$$\begin{aligned} i &= \sum_{i=0}^n x_i & j &= \sum_{i=0}^n x_i^2 & k &= \sum_{i=0}^n x_i^3 & l &= \sum_{i=0}^n x_i^4 \\ u &= \sum_{i=0}^n y_i & v &= \sum_{i=0}^n x_i * y_i & w &= \sum_{i=0}^n x_i^2 * y_i \end{aligned}$$

Obtendríamos un sistema lineal de ecuaciones solucionable mediante eliminación Gauss-Jordan:

$$\begin{cases} u = a * j + b * i + n * c \\ v = a * k + b * j + c * i \\ w = a * l + b * k + c * j \end{cases}$$

Además de la propia implementación con TFHE (codificada en el archivo `reg2.cpp`), se ha desarrollado un código de ejemplo en python mucho más legible (anexo B).

TFHE sólo ofrece operadores lógicos, así que tenemos que escribir la operaciones aritméticas necesarias: suma, resta, multiplicación y división. Además tendremos que dar la posibilidad de trabajar con números reales, por lo que tendremos que determinar la codificación más apropiada. Para todo ello se ha desarrollado la librería `tfhe-math`

4.2.2 tfhe-math

El desarrollo de esta librería ha sido quizás la parte más costosa del trabajo, pero también la que más resultados arroja sobre la implementación de criptografía homomórfica en un entorno real. Como hemos comentado, hemos desarrollado las operaciones aritméticas necesarias para poder realizar la regresión cuadrática teniendo en cuenta que necesitábamos codificar los valores como número reales.

TFHE sólo trabaja con arrays de bits cifrados, y los datos no se pueden evaluar durante la ejecución del programa, así que hemos tenido que meter algunas funciones redundantes para cubrir todos los supuestos: Por ejemplo, en la multiplicación trabajamos asumiendo que los dos factores tienen signo positivo y corregimos el resultado utilizando la puerta lógica MUX. Por otro lado, como hay algunos casos en los que vamos a tener la certeza del signo de los operandos (porque lo hemos gestionado ya antes), y la lógica de tratamiento del signo es muy pesada, hemos creado algunas funciones para trabajar con números sin signo, cuyo nombre hemos precedido de `u_` (de *unsigned*).

A la hora de trabajar con los números con coma flotante, para proteger los decimales al codificar el número como entero, les asignamos un número determinado de bits. Por defecto, cuando trabajamos con números de 64 bits, les asignamos 10 (y así podemos guardar 3 decimales). Así por ejemplo, si queremos trabajar con el número 3,14, lo multiplicamos por 1024 (2^{10} , el equivalente a moverlo 10 bits hacia la izquierda) y trabajamos con 3215.

Lo hacemos de esta forma (en lugar de, por ejemplo, multiplicar por 10) porque la operación de desplazamiento de bits es casi gratuita (en cuanto a tiempo de cómputo), mientras que la multiplicación y la división son muy costosas. Cuantos más bits introduzcamos, obtendremos mayor precisión, pero bajará la eficiencia.

Tenemos que tener precaución principalmente en dos aspectos:

1. En el producto y en la división también se multiplican y dividen los desplazamientos. Por ejemplo si trabajamos con dos números (a y b) a los que les hemos aplicado el desplazamiento de 10 bits ($a' = a * 2^{10}$, $b' = b * 2^{10}$). Hay que restaurar la escala antes de la división:

$$a'/b' = (a * 2^{10}) / (b * 2^{10}) = (a/b) \quad (4.1)$$

$$(a/b) \neq (a/b) * 2^{10} \quad (4.2)$$

Para que al restaurar el número no haya errores quitando las posiciones de los decimales:

$$a'/b' \rightarrow (a' * 2^{10}) / b' = (a/b) * 2^{10} \quad (4.3)$$

Y hay que restaurar la escala tras la multiplicación:

$$a' * b' = (a * 2^{10}) * (b * 2^{10}) = a * b * 2^{20} \quad (4.4)$$

$$a * b * 2^{20} \neq a * b * 2^{10} \quad (4.5)$$

Para, además de evitar errores al restaurar el número, no se produzcan desbordamientos (si a es de 4 bits, y b es de 3, $a * b$ ocupará 7 bits):

$$a' * b' \rightarrow (a' * b') / 2^{10} = a * b * 2^{10} \quad (4.6)$$

2. La gestión del signo

Para trabajar con el signo de los números se codifican en complemento a 2 (contributors., 2019b). En la suma y la resta podemos operar libremente, pero a la hora de hacer la división y el producto (nuevamente) hemos tenido que implementar algoritmos que “no entienden de signo”. Siempre que necesitamos trabajar con algoritmos sin signo, y como no podemos cambiar el flujo de ejecución en función del mismo (porque no podemos verlo), realizamos el siguiente proceso con cada operando:

- a) Negamos el operando: Para ello hemos creado la función `negativo` que devuelve el negativo del número si es positivo, y el positivo si es negativo.

- b) Guardamos (usando la puerta MUX) el mayor de los dos para asegurarnos de trabajar con el número en positivo
- c) Guardamos en un bit (cifrado, un bit que no vemos pero que MUX será capaz de interpretar) si el número es positivo (guardamos 0) o negativo (guardamos 1)
- d) Tras operar comparamos con la puerta XOR los bits que indican si los operandos eran positivos o negativos. Nos devolverá 0 sólo si ambos eran positivos o negativos, indicando que no hay que hacer ninguna corrección. Llamaremos a este bit *corrector*.
- e) Aplicamos la función `negativo` sobre el resultado, y esta vez determinamos (usando MUX) qué devolver con el parámetro `corrector`: Si es 1 devolvemos el resultado negado, si es 0 el mismo que ha devuelto el cálculo.

El código fuente se encuentra disponible en <https://gitlab.com/junquera/tfhe-math>.

- Operaciones

A continuación veremos cómo trabajan nuestras principales funciones:

- `gte`

Compara los `nb_bits` bits de `a` y `b` y marcar el bit `result` con el valor de la expresión lógica `a >= b`.

```
void gte(LweSample* result, const LweSample* a,  
         const LweSample* b, const int nb_bits,  
         const TFheGateBootstrappingCloudKeySet* bk);
```

- `is_negative`

Devuelve el valor del bit más significativo de `a` (que es 1 si es negativo y 0 si no por la codificación en complemento a 2).

```
void is_negative(LweSample* result, const LweSample* a,  
                const int nb_bits,  
                const TFheGateBootstrappingCloudKeySet* bk);
```

- **negativo**

Realiza el cambio de valor de `a` en complemento a 2: Niega todos los bits y suma 1 al resultado.

```
void negativo(LweSample* result, const LweSample* a,  
              const int nb_bits,  
              const TFheGateBootstrappingCloudKeySet* bk);
```

- **minimum / maximum**

Guarda en `result` el valor mínimo o máximo entre `a` y `b`.

```
void minimum(LweSample* result, const LweSample* a,  
             const LweSample* b, const int nb_bits,  
             const TFheGateBootstrappingCloudKeySet* bk);  
void maximum(LweSample* result, const LweSample* a,  
            const LweSample* b, const int nb_bits,  
            const TFheGateBootstrappingCloudKeySet* bk);
```

- **shiftrl / shiftr**

Almacena en `result` el valor de `a` desplazado (hacia la izquierda o la derecha) el número de veces especificado en `posiciones`. Es equivalente a multiplicar (mover a la izquierda) o dividir (mover a la derecha) entre dos. La operación sin signo (con `u_`) es casi instantánea, y el coste de la implementación normal es casi exclusivamente el de la gestión del signo.

```
void shiftrl(LweSample* result, const LweSample* a,  
            const int posiciones, const int nb_bits,  
            const TFheGateBootstrappingCloudKeySet* bk);  
void shiftr(LweSample* result, const LweSample* a,  
           const int posiciones, const int nb_bits,  
           const TFheGateBootstrappingCloudKeySet* bk);
```

- **add / sub**

`add` suma `a` y `b`. `sub` aplica la función `negativo` sobre `b` y luego suma `a` y `-b`.


```
void add(LweSample* result , const LweSample* a ,  
         const LweSample* b, const int nb_bits ,  
         const TFheGateBootstrappingCloudKeySet* bk);  
void sub(LweSample* result , const LweSample* a ,  
         const LweSample* b, const int nb_bits ,  
         const TFheGateBootstrappingCloudKeySet* bk);
```

■ mult

Operación de multiplicación (ver figura 4.4). A diferencia de las anteriores, esta operación y `div_float` tienen en cuenta los bits asignados a los decimales mediante el protocolo comentado anteriormente (ver 4.1).

```
void mult_float(LweSample* result ,  
               const LweSample* a, const LweSample* b,  
               const int float_bits , const int nb_bits ,  
               const TFheGateBootstrappingCloudKeySet* bk);
```

■ div

Operación de división (ver figura 4.5). Es la operación más costosa porque requiere duplicar el tamaño (en bits) de los elementos, pero es esencial para realizar nuestro cálculo, y es el principal elemento que marca la diferencia con respecto al resto de implementaciones.

```
void div_float(LweSample* result ,  
              const LweSample* a, const LweSample* b,  
              const int float_bits , const int nb_bits ,  
              const TFheGateBootstrappingCloudKeySet* bk);
```

4.3 Implementación con SEAL

4.3.1 Cálculo de la posición

El sistema CKKS ofrece la posibilidad de trabajar con varios datos a la vez codificándolos como vectores. Hemos aprovechado esta característica para poder resolver la distancia entre el punto

introducido por el usuario y las distintas curvas en una sola llamada, aplicando las operaciones necesarias en todas las curvas al mismo tiempo.

Procedemos de la siguiente manera:

1. Generamos un vector A con los valores a de todas las curvas, otro B con los valores b y otro C con los valores c de las curvas.
2. Codificamos todos estos vectores con `CKKSEncoder`
3. Determinamos el punto y correspondiente al valor x introducido por el usuario. Como sólo podemos hacer una operación en cada paso, tenemos que descomponer la sustitución en varias operaciones. En cada producto, al estar utilizando el esquema CKKS, tendremos que reescalar (y realinearizar, cuando los cálculos pendientes así lo requieran) nuestro resultado. La operación de reescalado trunca el número (le quita bits, ver [2.3.3](#)) y modifica los parámetros de cifrado reduciendo el número de operaciones restantes que podemos realizar con él. Por lo tanto, hay que intentar realizar el menor número de productos posible. Por ejemplo, si hubiese que hacer la operación $x_4 = x^4$, sería preferible hacer $x_2 = x^2, x_4 = x_2^2$ (con sólo dos productos) que $x_4 = x * x * x * x$ (con cuatro).

Además, al realizar el reescalado estamos obteniendo un número con una escala menor que el número que hemos cifrado inicialmente, por lo que cuando queramos sumar dos variables tenemos que asegurarnos de ajustarlas entre sí o saltará una excepción. Estas operaciones realmente se corresponden con eliminar valores de una cadena que gestiona `SEALContext` en la que se especifican los parámetros de cifrado de cada fase.

Tras realizar este estudio, procedemos a calcular:

a) $p_1 = x^2$

b) Realinearización y reescalado de p_1 . Aquí realinearizamos porque este elemento es el que más productos va a tener ($A * x^2$ conlleva tres productos, mientras que $B * x$ sólo uno).

c) $p_2 = p_1 * A$

d) Reescalado de p_2

e) $p_3 = B * x$

f) Reescalado de p_3

g) $p = p_2 + p_3$

h) Doble reescalado de C para poder sumarlo con p

i) $y = p + C$

$$\begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix} = \begin{pmatrix} y \\ y \\ \vdots \\ y \end{pmatrix} - (x^2 * \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} + x * \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} + \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}) \quad (4.7)$$

4. Finalmente, como nuestro y es el resultado de dos reescalados, tendremos que reescalar dos veces el y introducido por el usuario (al que llamaremos y_0) para poder operar. Tras hacerlo, devolvemos el resultado `encrypted_result` igual a $y - y_0$.

Cuando el usuario descifre `encrypted_result`, obtendrá un vector con las distancias entre el punto que ha introducido y todas las curvas, similar al de la figura 4.7.

4.4 Implementaciones Cliente/Servidor

Finalmente, para cada una de las tecnologías se han desarrollado dos programas que hacen las funciones de cliente y servidor. De esta forma separamos la lógica de cifrado y descifrado (perteneciente a los clientes) y la lógica de procesamiento de los datos cifrados (propia de los servidores).

Para ambas, se ha programado una clase cliente, y una clase servidor, como hemos comentado cada una con las funciones que le son propias por su rol en la arquitectura, y para el desarrollo del trabajo hemos creado cuatro ejecutables que, apoyándose en las clases cliente/servidor implementarán la lógica para generar la curva de regresión y evaluar en el resultado los valores. De ahora en adelante cuando hablemos de cliente y servidor nos referiremos a estos ejecutables.

- TFHE

El cliente TFHE ofrece la posibilidad de:

Ubicación	a	b	c
Cabo de Gata	−0,413	5,928	2,423
Finisterre	−0,261	3,801	5,041

Cuadro 4.1: Resultados del servidor TFHE

1. Generar datos cifrados de la temperatura en Cabo de Gata guardando los n valores de las x con el formato `ciudad_AXn.data` y los de las y como `ciudad_AYn.data`).
2. Generar datos cifrados de la temperatura en Finisterre siguiendo el mismo formato que anteriormente, pero con el prefijo `ciudad_B`.
3. Analizar los resultados generados por el servidor (indicándole la ruta en la que están, busca los archivos `{a,b,c}.data`).
4. Descifrar un archivo cifrado con TFHE

Cuando genera los datos cifrados, además exporta al sistema de archivos las claves para poder descifrar estos archivos cuando sea necesario, y poder subir la pública al servidor para procesar los datos.

El módulo que hace las veces de servidor con TFHE generará los parámetros a , b y c de los que llevamos hablando toda la sección en base a los archivos `ciudad_...` que haya en la carpeta que se le indique. Como el proceso es muy lento, almacenará los resultados de las operaciones intermedias para recurrir a ellos si se corta la ejecución (ver figura 3), hasta generar los archivos `a.data`, `b.data` y `c.data`.

Para hacer una prueba lo más realista posible de todo el sistema, he cifrado los datos que he obtenido de AEMET (ver anexo C) de las temperaturas en las dos ciudades con el cliente de TFHE, y los he subido a la nube de Digital Ocean (<https://www.digitalocean.com/>) para procesarlos con el código del servidor TFHE.

Tras finalizar el cálculo, los he descargado y descifrado con el cliente TFHE, y he insertado manualmente los resultados (ver tabla 4.1) en el código del servidor SEAL.

El código fuente se encuentra disponible en <https://gitlab.com/junquera/tfhe-cs>.

```

junquera@opa:~/UEM/TFM/results/tfhe/ciudad_A/resultados\$ ls -la
total 4240
drwxrwxr-x 2 junquera junquera 4096 ago 27 22:29 .
drwxrwxr-x 3 junquera junquera 4096 ago 16 10:06 ..
-rw-rw-r-- 1 junquera junquera 129024 ago 27 20:33 a.data
-rw-rw-r-- 1 junquera junquera 129024 ago 27 08:22 b.data
-rw-rw-r-- 1 junquera junquera 129024 ago 25 23:22 c.data
-rw-r--r-- 1 junquera junquera 129024 ago 16 10:06 i2.data
-rw-rw-r-- 1 junquera junquera 129024 ago 24 07:08 i2l2.data
...
-rw-rw-r-- 1 junquera junquera 129024 ago 23 20:06 ul.data
-rw-rw-r-- 1 junquera junquera 129024 ago 16 10:06 v.data
-rw-rw-r-- 1 junquera junquera 129024 ago 23 20:06 vl.data
-rw-rw-r-- 1 junquera junquera 129024 ago 16 10:06 w.data
-rw-rw-r-- 1 junquera junquera 129024 ago 23 20:06 wj.data

```

Listing 3: Archivos generados por el servidor TFHE

• SEAL

El servidor (`server.main`) lee de la carpeta en la que se está ejecutando los archivos cifrados `x.data` y `y.data` generados por el cliente, y los elementos criptográficos necesarios (clave pública, clave de realinealización y parámetros de cifrado). Tras procesar estos datos con los valores de las curvas que ha calculado previamente en TFHE, genera el archivo `result.data` y un archivo de texto con los nombres de las curvas (para que el cliente pueda interpretar los resultados).

El `main` del cliente (codificado en el archivo `client-main.cpp`) tiene dos funcionalidades:

1. En la primera pide que se introduzcan dos valores (temperatura y fecha), los cifra y genera dos archivos: `x.data` y `y.data`.
2. La otra funcionalidad, que se ejecuta tras la ejecución del código del servidor lee el archivo `result.data` de la carpeta en la que se ejecuta, lo descifra e imprime por pantalla el resultado combinado del vector descifrado con el archivo `curve_names.data`. En [4.6](#) se

puede ver la ejecución del sistema completo.

```

junquera@opa:~/UEM/TFM/seal-cs/data$ ../client
Encrypt(0) or decrypt(1)? > 0
Inserte temperatura > 23.4
Inserte mes > 6
junquera@opa:~/UEM/TFM/seal-cs/data$ ls -la
total 3104
drwxrwxr-x 2 junquera junquera 4096 ago 28 11:27 .
drwxrwxr-x 6 junquera junquera 4096 ago 28 11:25 ..
-rw-rw-r-- 1 junquera junquera 49 ago 28 11:27 params.data
-rw-rw-r-- 1 junquera junquera 524361 ago 28 11:27 public.key
-rw-rw-r-- 1 junquera junquera 1573131 ago 28 11:27 relin.key
-rw-rw-r-- 1 junquera junquera 262192 ago 28 11:27 secret.key
-rw-rw-r-- 1 junquera junquera 393289 ago 28 11:27 x.data
-rw-rw-r-- 1 junquera junquera 393289 ago 28 11:27 y.data
junquera@opa:~/UEM/TFM/seal-cs/data$ ../server
junquera@opa:~/UEM/TFM/seal-cs/data$ ls -la
total 3304
drwxrwxr-x 2 junquera junquera 4096 ago 28 11:28 .
drwxrwxr-x 6 junquera junquera 4096 ago 28 11:25 ..
-rw-rw-r-- 1 junquera junquera 24 ago 28 11:28 curve_names.data
-rw-rw-r-- 1 junquera junquera 49 ago 28 11:27 params.data
-rw-rw-r-- 1 junquera junquera 524361 ago 28 11:27 public.key
-rw-rw-r-- 1 junquera junquera 1573131 ago 28 11:27 relin.key
-rw-rw-r-- 1 junquera junquera 196681 ago 28 11:28 result.data
-rw-rw-r-- 1 junquera junquera 262192 ago 28 11:27 secret.key
-rw-rw-r-- 1 junquera junquera 393289 ago 28 11:27 x.data
-rw-rw-r-- 1 junquera junquera 393289 ago 28 11:27 y.data
junquera@opa:~/UEM/TFM/seal-cs/data$ cat curve_names.data
cabo_de_gata
finisterre
junquera@opa:~/UEM/TFM/seal-cs/data$ ../client
Encrypt(0) or decrypt(1)? > 1
cabo_de_gata: 0.210004
finisterre: 5.34

```

Figura 4.6: Ejecución completa del sistema de SEAL

El código fuente se encuentra disponible en <https://gitlab.com/junquera/seal-cs>. Para más información consultar el apéndice F.

4.5 Evaluación de límites y rendimientos

Por último se han desarrollado varios archivos de prueba para evaluar los límites y el rendimiento de las librerías. Estas pruebas son la culminación de este estudio, y buscan aportar un enfoque cuantitativo a su funcionamiento, más allá de los cálculos que se puedan hacer con el conocimiento de las tecnologías o la visión empírica desde el punto de vista del desarrollador.

- TFHE

Para evaluar la eficiencia de TFHE ejecutamos todas las funciones aritméticas que hemos escrito en `tfhe-math` con distintos tamaños de texto cifrado (desde 4 hasta 64 bits) calculando el tiempo que tardan en realizarse. En el capítulo de resultados (capítulo 5) veremos la comparativa entre los resultados obtenidos y el tiempo estimado en base a la teoría.

- SEAL

Con SEAL hemos desarrollado dos tipos de pruebas:

- Al igual que con TFHE, hemos evaluado el tiempo que tarda en realizar las operaciones aritméticas (en este caso, sólo suma y multiplicación) con distintos tamaños de texto cifrado.
- También hemos evaluado cuáles son los límites de cómputo con distintos tamaños de texto cifrado, con y sin aplicar realinealizaciones.

Además entre los códigos de ejemplo de SEAL se encuentra un programa para evaluar la eficiencia de sus esquemas. Lo incluiremos entre nuestras pruebas.

Ya sólo queda analizar los resultados de todos nuestros experimentos.

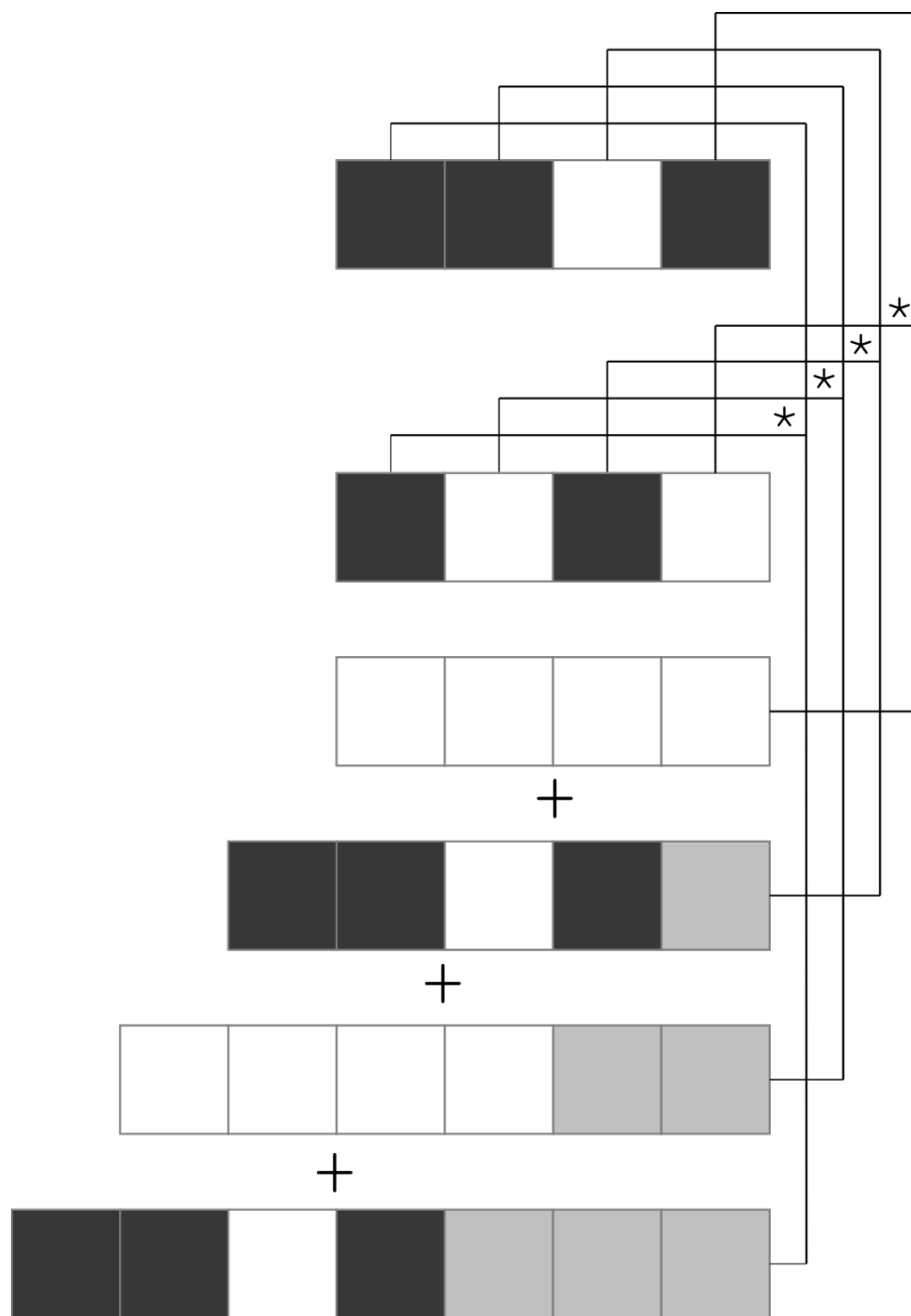


Figura 4.4: Algoritmo de multiplicación

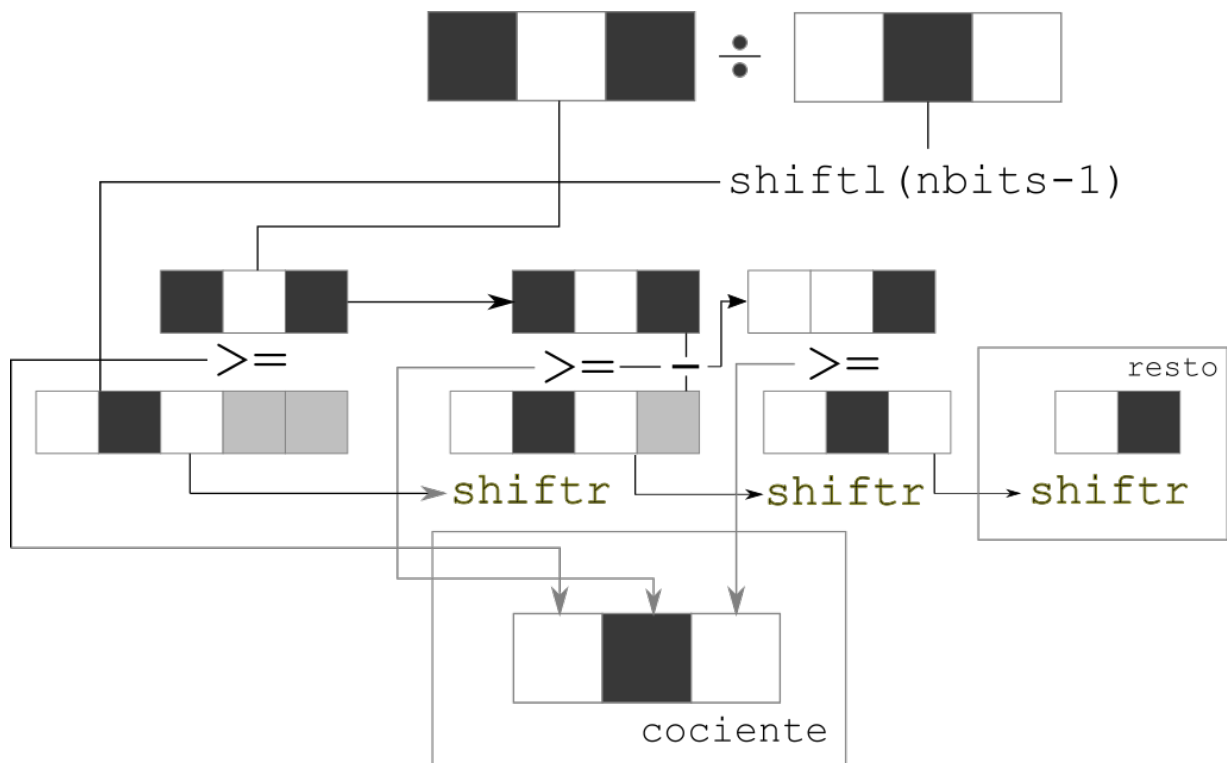


Figura 4.5: Algoritmo de división

5 Resultados

5.1 TFHE

5.1.1 Tiempos de ejecución

Al ser una ejecución “especulativa”, la optimización de las funciones está muy limitada (se tiene que operar con todos los bits, gestionando el flujo de los datos eligiendo los valores con los que se opera mediante la puerta MUX). Sin embargo, siempre que el circuito lógico sea cerrado para un número concreto de bits (de forma similar a un circuito físico), pueden utilizarse métodos formales y algebraicos para aumentar su eficiencia.

Por otro lado, crear algoritmos que sean versátiles con respecto al número de bits hace que estos circuitos lógicos tengan que hacer operaciones redundantes, o introducir demasiada complejidad ciclomática para estar preparado para cualquier tamaño de palabra.

- Operaciones aritméticas

Según la documentación de TFHE, cada puerta lógica tarda 13ms en ejecutarse, y las puertas MUX tardan 26ms. Sabiendo que sólo el 13 % de las puertas lógicas en nuestro código son puertas MUX podemos establecer un tiempo por puerta de unos 15ms para calcular el tiempo teórico que deberían tardar nuestras operaciones en función del número de puertas lógicas que emplean. En la tabla 5.1 podemos ver los resultados del cálculo.

Hemos medido los tiempos de ejecución de las distintas operaciones aritméticas con varios tamaños de palabra (de 4, 8, 16, 32 y 64 bits). En la figura 5.1 podemos observar como prácticamente todas tienen un crecimiento lineal con mayor o menor pendiente: hay algunas que son casi gratuitas, como las operaciones `shift`; y hay otras como la resta, que crecen un poco más rápido (porque están compuestas por varias operaciones crecientes) pero sin perder su comportamiento lineal.

Sin embargo hay dos operaciones que crecen de una forma más preocupante: la multiplicación y la división. En ambas funciones se juntan tres factores:

1. Requiere el uso de un gran número de funciones adicionales: sumas, restas, comparaciones, etc; haciendo que crezca el número de puertas lógicas.

operación	puertas logicas	tiempo estimado
compare_bit	2	0,04
equal	128	2,56
is_negative	1	0,02
minimum/maximum	388	7,76
add_bit	5	0,1
sum	320	6,4
negativo	192	3,84
resta	512	10,24
multiply	46826	936,52
mayor_igual	128	2,56
shiftr/shiftr	771	15,42
u_shiftr/u_shiftr	129	2,58
divide	85776	1715,52

Cuadro 5.1: Tiempo estimado de ejecución

- Debido a su complejidad, tiene varios bucles anidados cuyas iteraciones están relacionadas con el número de bits.
- Abordamos las operaciones trabajando con el doble de bits de la entrada. Es decir, un número de 64 bits se opera como si tuviese 128 para evitar desbordamientos.

En la tabla 5.2 vemos cuáles son estos tiempos con números de 64 bits.

En la figura 5.2 se puede ver cómo el crecimiento de cada valor con respecto al anterior ($f(2^x)/f(2^{x-1})$ con x número de bits) de ambas funciones corresponde con un crecimiento potencial con respecto al número de bits.

Por último, calcularemos el tiempo teórico que tardarían en realizarse las operaciones de la curva de regresión (sumando los tiempos de sus sumas, multiplicaciones, divisiones...) y lo compararemos con el tiempo que han tardado realmente.

En la tabla 5.3 analizamos las operaciones contenidas en cada uno de los cálculos (ver anexo B) y el tiempo estimado que tardarían en función de los tiempos mostrados en la tabla 5.2 .

Operación	Tiempo (s)
compare_bit	0
is_negative	0
u_shift	0
add_bit	0
equal	3
mayor_igual	4
min / max	14
sum	8
negativo	8
resta	16
shift	19
multiply	1257
divide	11662

Cuadro 5.2: Tiempo real de ejecución

Cálculo	Sumas	Multiplicaciones	Divisiones	Tiempo estimado (s)
initVectores	7	5	0	76092
calcCuadrados	0	4	0	5028
calcDuplas	0	9	0	11313
calcComplejos	0	10	0	12570
CalcC	6	0	4	46776
CalcB	4	2	1	14208
CalcA	2	2	1	14192
Total	9	32	6	180179

Cuadro 5.3: Sub-operaciones de regresión cuadrática

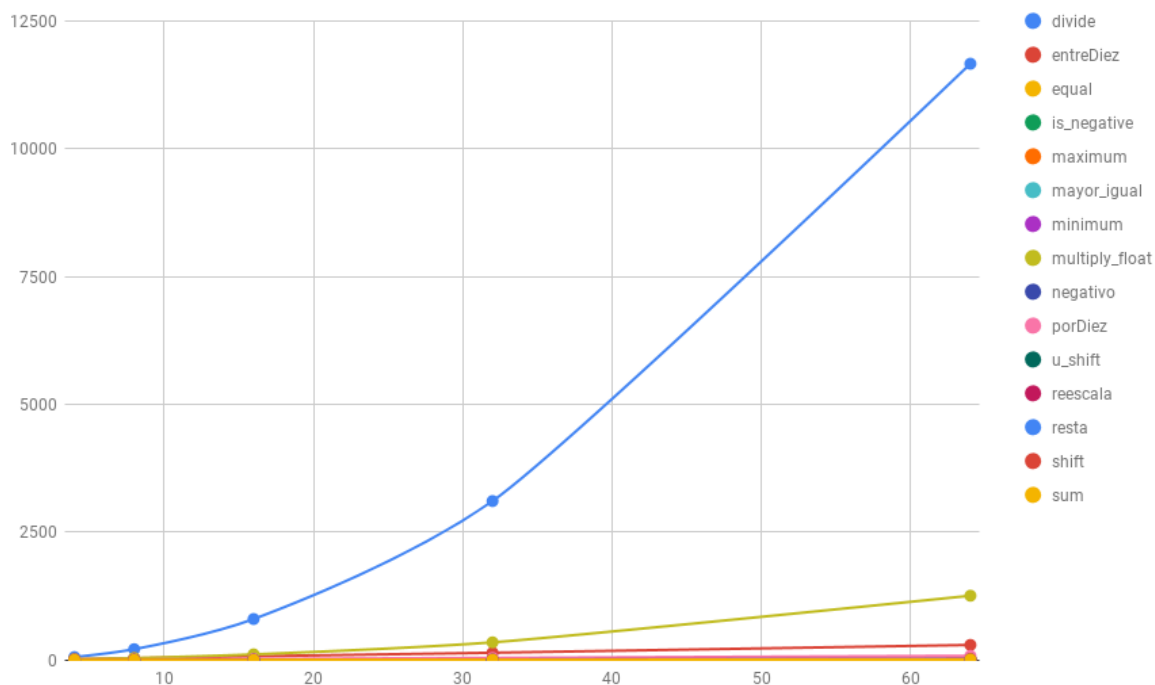


Figura 5.1: Tiempo de ejecución por número de bits

En la tabla 5.4 vemos el tiempo total que han tardado en realizarse estas operaciones frente al calculado.

Se puede apreciar que apenas hay ligeras diferencias con respecto a los datos teóricos.

5.1.2 Límites de cómputo

No los hay, más allá de los relacionados con la codificación al devolver el resultado plano, pero hay que estar pendiente (muy pendiente) del crecimiento cuando se opera, y a medida que aumenta el tamaño de los datos, aumenta notablemente el tiempo de ejecución.

Por ejemplo, para determinar cuántos bits necesitábamos para nuestra regresión cuadrática calculamos con X el mayor valor de las x (en nuestro caso 12), n el mayor grado alcanzable (el mayor número de multiplicaciones que se podían acumular sobre ese valor), d el número de bits para decimales y añadiéndole 1 (el bit del signo):

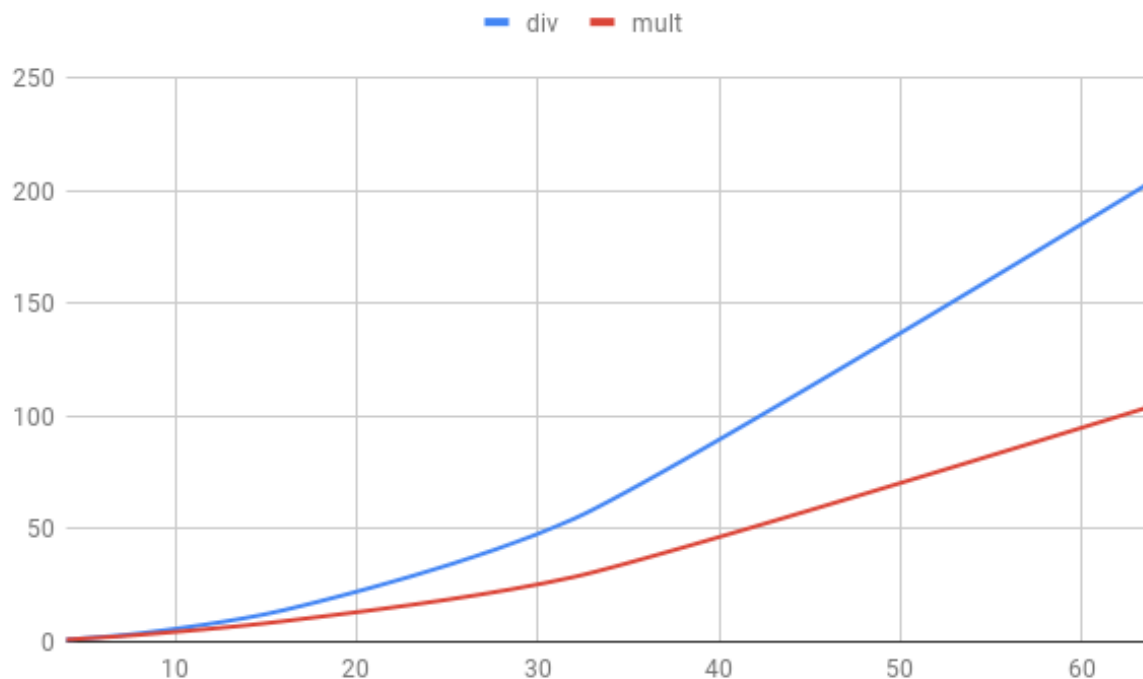


Figura 5.2: Crecimiento: división y multiplicación

$$bits_necesarios = n * (\log_2 X) + d + 1 = 51 \quad (5.1)$$

Podríamos haber implementado la solución con números de 51 bits sin problemas, pero decidimos hacerlo con 64 para poder contrastar los resultados con los obtenidos evaluando las operaciones de forma individual.

5.1.3 Problemas encontrados

A medida que hemos ido desarrollando la solución hemos encontrado principalmente problemas derivados de tener que trabajar con número en un sistema de puertas lógicas: implementar los algoritmos, gestionar el signo, los decimales, etc; sin tener casi documentación ni librerías de apoyo. Es decir, uno de los principales problemas de TFHE es que cualquier solución se tiene que construir desde la base, como si efectivamente se estuviese programando en ensamblador

Cálculo	Tiempo estimado (s)	Tiempo real (s)	Proporción
initVectores	76092	68940	0.90
calcCuadrados	5028	4555	0.90
calcDuplas	11313	10431	0.92
calcComplejos	12570	11522	0.91
CalcC	46776	39818	0.85
CalcB	14208	12222	0.86
CalcA	14192	12200	0.86
Total	180179	159688	0.89

Cuadro 5.4: Tiempo real de regresión

y sin poder acudir a librerías de código abierto o documentación de ningún tipo (mas allá de la propia del sistema).

Aunque este es un problema importante a la hora de realizar una implementación destinada al uso comercial, desde el punto de vista del tiempo de implantación (hay que invertir muchas horas), es principalmente un problema para la estabilidad de las soluciones, que se basan en sistemas que no han sido probados previamente por nadie (se ha invertido casi el mismo tiempo en desarrollar nuestra solución que en depurarla y arreglar errores).

Pero sin lugar a dudas el mayor problema es el de la eficiencia: si bien nuestra solución es optimizable, el cálculo de una curva de regresión con la maqueta en *python* (ver [B](#)) tarda menos de 50 ms, frente a las 44,35 horas (1,84 días) del cálculo cifrado.

5.2 SEAL

5.2.1 Tiempos de ejecución

Hemos probado los tiempos de ejecución con las siguiente operaciones.

- Codificación de variable
- Codificación de variable como matriz
- Cifrado
- Cifrado de matriz

Operación	BFV	CKKS
encode	410	12753
decode	359	34802
encrypt	15324	20628
decrypt	7145	1055
add	344	300
multiply	76506	3567
multiply plain	12462	1140
square	54013	2662
relinearize	31034	33513
rotate vector	31823	36967
rotate columns	31432	-
rescale	-	11607

Cuadro 5.5: Tests de eficiencia de SEAL (microsegundos)

- Multiplicación
- Suma

En todas ellas, y con números de 8192, 16384 y 32768 bits, la ejecución se realiza de forma prácticamente instantánea (menos de un segundo). En el apéndice [D](#) se pueden ver los resultados en bruto de la ejecución de los tests incluidos en la documentación de SEAL. En la tabla [5.5](#) podemos ver un resumen.

5.2.2 Límites de cómputo

Para cada uno de los esquemas veremos un límite de cómputo distinto, dependiendo de qué parámetro determina la capacidad de operación:

- El parámetro `noise_budget` en el esquema BFV
- Los elementos dentro de la cadena de CKKS

Además cuando hablamos de límites no hablamos sólo del número de operaciones, sino que

<i>poly_modulus_degree</i>	Multiplicaciones realizables
2^{13}	1
2^{14}	4
2^{15}	9

Cuadro 5.6: Profundidad computacional máxima BFV

debemos incluir que en SEAL sólo podremos trabajar con números, y en resumen con sólo tres tipos de operación: suma, resta y multiplicación.

- BFV

Con BFV la limitación de trabajar con números se amplía: sólo podemos trabajar con números enteros. El parámetro que determina cuántos cálculos se pueden hacer es `noise_budget`. Como comentamos en el capítulo 2, este parámetro se va reduciendo a medida que se opera, y si llega a 0 el número se vuelve irrecuperable.

Hemos evaluado estos límites mediante un programa que multiplica mientras el nivel sea mayor que cero. Probando con distintos tamaños (`poly_modulus_degree`) bajo los parámetros recomendados por SEAL, obtenemos los resultados de la tabla 5.6.

Vemos que el número de productos realizable es similar al que podemos realizar con 64 bits en TFHE, pero es mucho más bajo realmente. Mientras que en nuestro ejemplo de TFHE podíamos realizar 10 productos sin ningún problema, en este sólo podemos realizar 8 (si nuestro test nos ha devuelto 9 significa que con 9 el resultado está corrupto), sólo con número enteros de hasta 60 bits.

- CKKS

Con CKKS, por tamaño de la cadena. El primer y el último número de la cadena tienen que ser mayores que el número a cifrar/descifrar, y la suma de estos dos con los intermedios tiene que ser menos que `max_coeff_modulus_bit-length`. Para calcular el tamaño de la cadena seguiremos el siguiente procedimiento:

1. Reservaremos 120 bits: 60 para el primer número de la cadena (el número que determinará la precisión del descifrado) y otros 60 para el último (que tiene que tener al menos

poly_modulus_degree Multiplicaciones realizables

2^{13}	2
2^{14}	7
2^{15}	19

Cuadro 5.7: Profundidad computacional máxima CKKS

el mismo número de bits que el más grande del resto de la cadena)

2. Rellenamos los espacios intermedios con números de 40 bits mientras la suma de toda la cadena sea menor que el número máximo de bits asignable a `coeff_modulus` (ver 3.1).

Tras realizar este cálculo, obtendremos el tamaño de la cadena equivalente al número de multiplicaciones realizables de la tabla 5.7.

Sin embargo, la profundidad de cómputo está también limitada por el tamaño del número en texto plano y por su escala, rompiéndose al alcanzar cifras que realmente son pequeñas (y más en comparación con las cifras que promete poder abordar). En la figura 4 podemos ver el resultado de ejecutar varias iteraciones $a = a * \pi$ con CKKS. Vemos que a partir del tercer cálculo se corrompe el valor de la constante π (lo que parece ser un error en el reescalado, ¡o incluso puede tener origen en algún tipo de overflow en la memoria!), y al llegar al décimo producto (que equivaldría a un número relativamente pequeño: 93174, 37) salta una excepción.

5.2.3 Problemas encontrados

SEAL es una librería extremadamente limitada:

- Por un lado, no se pueden hacer apenas operaciones, y con un conjunto de datos muy limitados
- Por otro, los límites de cómputo son muy reducidos, y no da la posibilidad de ampliarlos más (aún a riesgo de hacer un programa menos eficiente).

Es justo decir, de todas formas, que la posibilidad de implementar los datos como matrices, y de poder operar con todos ellos a la vez es una solución muy interesante y aplicable a muchos pro-

blemas (nuestra prueba de concepto es un ejemplo de ello). Pero los límites en la profundidad de cálculo hacen inviable cualquier solución real.

Los principales problemas a la hora de trabajar con SEAL han sido los relacionados con la selección de parámetros. Aunque las guías de SEAL son impecables (de hecho son un muy buen apoyo para entender la parte teórica) hay que comprender muy bien cómo se codifican los datos y cómo funciona el esquema para que el programa funcione bien, e incluso en ese caso, la ejecución sigue teniendo determinados comportamientos erráticos que llevan a tener que desarrollar mediante ensayo y error.

5.3 Coste del producto

A continuación desarrollaremos los costes que ha tenido el desarrollo de este trabajo.

5.3.1 Coste de desarrollo

Para el desarrollo de este trabajo se han invertido las siguientes horas de trabajo:

- Estudio teórico (mediados de abril, mediados de junio de 2019): 2 meses, 2 horas a la semana. Total: 18 horas
- Estudio de las librerías y herramientas (mediados de junio, finales de julio de 2019): 1 mes y medio, 2 horas a la semana. Total: 12 horas
- Implementación de la solución (mes de agosto de 2019): Un mes, promedio de 6 horas al día, de lunes a viernes. Total: 90 horas

Contabilizadas 120 horas, y con un sueldo mínimo de 30000€ al año (fuente: Universidad Europea de Madrid ¹), podríamos estimar el coste del desarrollo en 1800€.

5.3.2 Coste de despliegue

Para la ejecución del servidor de TFHE hemos contratado una máquina en Digital Ocean (<https://www.digitalocean.com/>) durante una semana. Así utilizamos sus servicios:

- Durante un día tuvimos contratada una instancia con un núcleo de CPU, con un precio de 5 dólares al mes. En esta instancia instalamos las librerías necesarias, nuestro código, y

¹Cuánto gana un ingeniero informático, <https://universidadeuropea.es/blog/cuanto-gana-un-ingeniero-informatico>

lanzamos las pruebas de eficiencia de TFHE (con las que hemos generado los resultados de la figura 5.1).

- Tras esta prueba reescalamos las características de la máquina para que tuviese 2 CPU y poder realizar los cálculos de las dos curvas de regresión de forma paralela. Esta máquina tiene un coste de 15 dólares al mes, y estuvo corriendo durante otros 7 días y medio (el tiempo que tardó en calcular las curvas, más el tiempo de repetir algunos cálculos a medida que íbamos depurando errores en nuestro código), consumiendo el 100 % de sus recursos casi todo el tiempo.

El coste final ha sido de 5,07 dólares (4,6€). En un entorno real este coste dependería del número de máquinas que fuesen necesarias para calcular los valores de cada curva, durante unos 2 días por cada una de ellas. La parte positiva es que, con el modelo que hemos construido para el sistema, sólo sería necesario realizar estos cálculos una vez para cada curva.

En el anexo E pueden verse todos los detalles de facturación, y en la figura 5.3 el panel de control web con las estadísticas de la máquina.

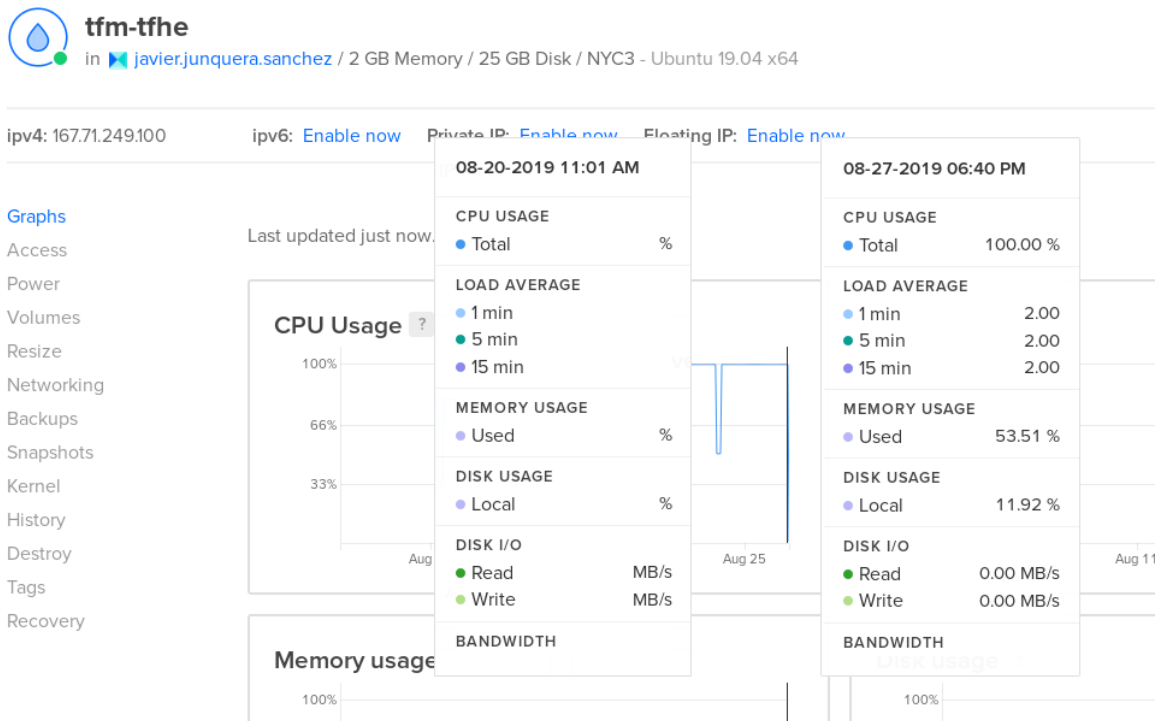


Figura 5.3: Panel web de estadísticas del servidor

```
# Mult: 0
3.14 * 3.14 = 9.8596
# Mult: 1
9.8596 * 3.14 = 30.9591
# Mult: 2
30.9591 * 3.14 = 97.2117
# Mult: 3
97.2117 * -1.27512e+06 = -1.23957e+08
# Mult: 4
-1.23957e+08 * -1.4063e+18 = -2.23609e+25
# Mult: 5
-2.23609e+25 * -1.56344e+30 = 6.43309e+56
# Mult: 6
6.43309e+56 * -1.7182e+42 = -2.09111e+99
# Mult: 7
-2.09111e+99 * -1.87436e+54 = 3.45003e+153
# Mult: 8
3.45003e+153 * -2.07686e+66 = 2.8223e+219
# Mult: 9
2.8223e+219 * -2.27126e+78 = -4.74131e+298
# Mult: 10
terminate called after throwing an instance of 'std::invalid_argument'
  what():  scale out of bounds
Abortado (`core' generado)
```

Listing 4: Potencias de π con CKKS

6 Conclusiones

Aunque el sistema que hemos desarrollado, en concreto, no sirva para resolver ningún problema real (la ubicación en base a la temperatura no es un sistema fiable de posicionamiento), nuestro modelo sí que cumple con las características de un modelo real (en concreto, de machine learning) implementado con criptografía homomórfica:

- Un sistema de generación de modelo lento, que sólo se tiene que ejecutar una vez
- Otro sistema rápido de evaluación de datos, que se ejecuta múltiples veces

Esto nos permite extraer conclusiones acerca de la viabilidad del uso de librerías de criptografía homomórfica en procesos reales:

1. El problema de la eficiencia es un problema realmente grave para aplicarlo a sistemas reales: por mucho que optimizásemos nuestra solución, los 50 ms que tarda en calcularse la curva de regresión en *python* (un lenguaje que ya de por sí es lento comparado con los lenguajes compilados) equivalen a ejecutar dos puertas lógicas de TFHE.
2. Independientemente de la eficiencia, es difícil programar soluciones con estas librerías, que funcionen, y que realmente sean seguras. Hay que saber mucho para hacerlo correctamente, y se pueden cometer errores que comprometan seriamente la seguridad (Peng, 2019).

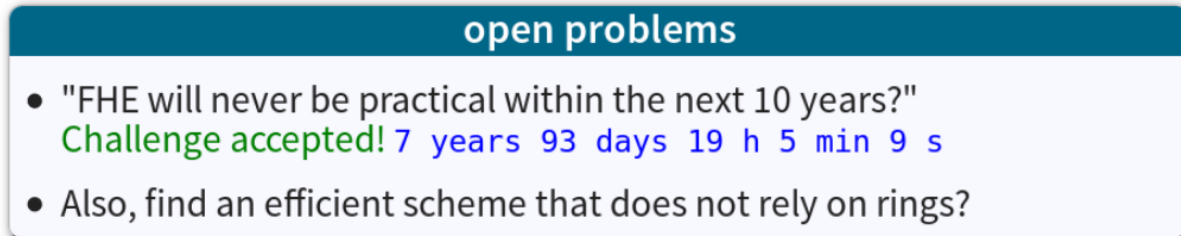
Sin embargo, existen ciertos aspectos esperanzadores en todo el asunto. Es muy buena idea en TFHE que limiten los parámetros a elegir. El desarrollador debe poder trabajar con las librerías de forma segura sin tener que elegir los parámetros o comprender qué hacen (eso debe ser trabajo de los criptógrafos).

Como hemos visto a lo largo del máster, en un ámbito profesional las medidas de seguridad tienen que implementarse en función del riesgo del sistema. Estas librerías pueden ser útiles, por ejemplo, para operaciones extremadamente confidenciales que tengan que ser ejecutadas en nubes públicas, siempre que el valor del activo así lo requiera (en comparación con el coste de desarrollar la solución, y el coste computacional extra).

En el estadio actual de las implementaciones de criptografía homomórfica no hay una solución clara que sirva para cualquier problema. Lo ideal es buscar dicha solución para cada caso concreto. Si bien esto puede incrementar el coste (es difícil reutilizar código), hace que el uso de criptografía homomórfica sea viable. Por ejemplo, mientras que nuestra solución pretende ser muy versátil para experimentar con las tecnologías, si se va a trabajar con 64 bits es mucho más eficiente implementar las operaciones en circuitos cerrados de 64 bits (como se haría en un circuito físico).

En cualquier caso, estas tecnologías prometen ser la solución a varios problemas: además de la criptografía homomórfica, la criptografía basada en *lattices* es una de las aproximaciones más prometedoras en un panorama *post-quantum*.

En su presentación en el año 2016, los desarrolladores de TFHE aceptaron un reto: crear sistemas FHE prácticos en menos de 10 años (figura 6.1, (Chillotti y col., 2016b)). Los experimentos que hemos realizado con sus tecnologías, y los trabajos futuros con las tecnologías más recientes, muestran visos de optimismo.



open problems

- "FHE will never be practical within the next 10 years?"
Challenge accepted! 7 years 93 days 19 h 5 min 9 s
- Also, find an efficient scheme that does not rely on rings?

Figura 6.1: "FHE will never be practical within the next 10 years?"

7 Trabajos futuros

Las vías más interesantes a explorar en busca de sistemas válidos y eficientes para implementaciones con criptografía homomórfica son:

- Paralelización de tareas mediante GPU con librerías como cuHE. Aunque es una solución menos versátil (depende del uso de GPUs) puede ser una buena aproximación “intermedia”, al menos hasta que haya sistemas más eficientes.
- Implementación de sistemas utilizando el compilador Cingulata (ver referencia en el capítulo 3). Sería especialmente interesante la comparación entre las funciones aritméticas que hemos implementado y el uso de circuitos lógicos. En algunas ponencias de TFHE apuntan al uso de los circuitos documentados en el framework ABY (<https://github.com/encryptogroup/ABY/tree/public/bin/circ>) como una alternativa al desarrollo de librerías. Aunque son menos usables desde el punto de vista de desarrollo (el circuito divisor tiene más de 53000 líneas de código), pueden ser una aproximación más eficiente.

Otro buen aporte podría ser el estudio de formas más eficientes de codificar los datos en TFHE. Se podrían implementar clases que puedan operar con menos bits, que gestionen de otra forma la lógica del signo o los decimales. Quizás la mejor manera de estudiar estas formas de codificación sea tratando de implementar soluciones con criptografía homomórfica, para detectar y evaluar nuevas vías de optimización.

Por último, siendo un campo todavía en una fase tan temprana de desarrollo, conviene estar al tanto de los avances en el estándar (la publicación de una posible cuarta generación), en especial del desarrollo de la versión 2 de TFHE con el modo Chimera.

Bibliografía

- Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., ... Vaikuntanathan, V. (2018). *Homomorphic Encryption Security Standard*. HomomorphicEncryption.org. Toronto, Canada.
- Apon, D. (s.f.). An intro to lattices and learning with errors, 89.
- Archer, D. (2017). Revolution and Evolution: Fully Homomorphic Encryption. Recuperado el 2 de agosto de 2019, desde <https://galois.com/blog/2017/12/revolution-evolution-fully-homomorphic-encryption/>
- Archer, D., Chen, L., Cheon, J. H., Gilad-Bachrach, R., Hallman, R. A., Huang, Z., ... Wang, S. (2017). *Applications of Homomorphic Encryption*. HomomorphicEncryption.org. Redmond WA, USA.
- Barak, B. (s.f.). Lecture 15: Fully homomorphic encryption : Introduction and bootstrapping. Recuperado el 10 de mayo de 2019, desde https://www.boazbarak.org/cs127spring16/chap15_fhe
- Boneh, D., Goh, E.-J. & Nissim, K. (2005). Evaluating 2-DNF Formulas on Ciphertexts. En D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, ...J. Kilian (Eds.), *Theory of Cryptography* (Vol. 3378, pp. 325-341). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:[10.1007/978-3-540-30576-7_18](https://doi.org/10.1007/978-3-540-30576-7_18)
- Boura, C., Gama, N., Georgieva, M. & Jetchev, D. (2018). *CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes* (inf. téc. N.º 758). Recuperado el 27 de agosto de 2019, desde <https://eprint.iacr.org/2018/758>
- Brakerski, Z., Gentry, C. & Vaikuntanathan, V. (2012). (Leveled) Fully Homomorphic Encryption Without Bootstrapping. En *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (pp. 309-325). ITCS '12. event-place: Cambridge, Massachusetts. New York, NY, USA: ACM. doi:[10.1145/2090236.2090262](https://doi.org/10.1145/2090236.2090262)
- Brenner, M., Dai, W., Halevi, S., Han, K., Jalali, A., Kim, M., ... Sunar, B. (2017). *A Standard API for RLWE-based Homomorphic Encryption*. HomomorphicEncryption.org. Redmond WA, USA.
- Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Hoffstein, J., ... Vaikuntanathan, V. (2017). *Security of Homomorphic Encryption*. HomomorphicEncryption.org. Redmond WA, USA.
- Cheon, J. H., Kim, A., Kim, M. & Song, Y. (2017). Homomorphic Encryption for Arithmetic of Approximate Numbers. En T. Takagi & T. Peyrin (Eds.), *Advances in Cryptology – ASIACRYPT 2017* (pp. 409-437). Lecture Notes in Computer Science. Springer International Publishing.

- Chillotti, I., Gama, N., Georgieva, M. & Izabachène, M. (2016a). Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. En J. H. Cheon & T. Takagi (Eds.), *Advances in Cryptology – ASIACRYPT 2016* (Vol. 10031, pp. 3-33). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:[10.1007/978-3-662-53887-6_1](https://doi.org/10.1007/978-3-662-53887-6_1)
- Chillotti, I., Gama, N., Georgieva, M. & Izabachène, M. (2016b). *TFHE: Fast Fully Homomorphic Encryption Library*.
- contributors., W. (2017). SIMD. Page Version ID: 103930179. Recuperado el 27 de agosto de 2019, desde <https://es.wikipedia.org/w/index.php?title=SIMD&oldid=103930179>
- contributors., W. (2018a). Base (álgebra). Page Version ID: 112556453. Recuperado el 19 de agosto de 2019, desde [https://es.wikipedia.org/w/index.php?title=Base_\(%C3%A1lgebra\)&oldid=112556453](https://es.wikipedia.org/w/index.php?title=Base_(%C3%A1lgebra)&oldid=112556453)
- contributors., W. (2018b). Cifrado maleable. Page Version ID: 107189153. Recuperado el 20 de agosto de 2019, desde https://es.wikipedia.org/w/index.php?title=Cifrado_maleable&oldid=107189153
- contributors., W. (2018c). Red (grupo). Page Version ID: 112557175. Recuperado el 19 de agosto de 2019, desde [https://es.wikipedia.org/w/index.php?title=Red_\(grupo\)&oldid=112557175](https://es.wikipedia.org/w/index.php?title=Red_(grupo)&oldid=112557175)
- contributors., W. (2019a). Anillo (matemática). Page Version ID: 116724200. Recuperado el 27 de agosto de 2019, desde [https://es.wikipedia.org/w/index.php?title=Anillo_\(matem%C3%A1tica\)&oldid=116724200](https://es.wikipedia.org/w/index.php?title=Anillo_(matem%C3%A1tica)&oldid=116724200)
- contributors., W. (2019b). Complemento a dos. Page Version ID: 117847084. Recuperado el 23 de agosto de 2019, desde https://es.wikipedia.org/w/index.php?title=Complemento_a_dos&oldid=117847084
- contributors., W. (2019c). Generalized normal distribution. Page Version ID: 898098236. Recuperado el 20 de agosto de 2019, desde https://en.wikipedia.org/w/index.php?title=Generalized_normal_distribution&oldid=898098236
- contributors., W. (2019d). Homomorphic encryption. Page Version ID: 894868556. Recuperado el 10 de mayo de 2019, desde https://en.wikipedia.org/w/index.php?title=Homomorphic_encryption&oldid=894868556
- contributors., W. (2019e). Lattice problem. Page Version ID: 907089742. Recuperado el 19 de agosto de 2019, desde https://en.wikipedia.org/w/index.php?title=Lattice_problem&oldid=907089742

- contributors., W. (2019f). Lattice-based cryptography. Page Version ID: 908847210. Recuperado el 2 de agosto de 2019, desde https://en.wikipedia.org/w/index.php?title=Lattice-based_cryptography&oldid=908847210
- Costache, A. & Smart, N. P. (2015). *Which Ring Based Somewhat Homomorphic Encryption Scheme is Best?* Published: Cryptology ePrint Archive, Report 2015/889.
- CryptoWiki. (s.f.). Recuperado el 10 de mayo de 2019, desde http://cryptowiki.net/index.php?title=Main_Page
- Dai, W. & Sunar, B. (2015). *cuHE: A Homomorphic Encryption Accelerator Library*. Published: Cryptology ePrint Archive, Report 2015/818.
- Dolev, D., Dwork, C. & Naor, M. (1991). Non-malleable Cryptography. En *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing* (pp. 542-552). STOC '91. event-place: New Orleans, Louisiana, USA. New York, NY, USA: ACM. doi:[10.1145/103418.103474](https://doi.org/10.1145/103418.103474)
- Ducas, L. & Micciancio, D. (2014). *FHEW: Bootstrapping Homomorphic Encryption in less than a second* (inf. téc. N.º 816). Recuperado el 27 de agosto de 2019, desde <https://eprint.iacr.org/2014/816>
- Fan, J. & Vercauteren, F. (2012). *Somewhat Practical Fully Homomorphic Encryption*. Published: Cryptology ePrint Archive, Report 2012/144.
- Fully Homomorphic Encryption and Signatures | Simons Institute for the Theory of Computing. (s.f.). Recuperado el 2 de agosto de 2019, desde <https://simons.berkeley.edu/talks/wichs-brakerski-2015-07-06>
- Gama, N. (s.f.). TFHE: past, present, future. Recuperado el 25 de agosto de 2019, desde <http://lab.algonics.net/slides/index-ccs.html#/>
- Gentry, C. (2009). Fully Homomorphic Encryption Using Ideal Lattices. En *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing* (pp. 169-178). STOC '09. event-place: Bethesda, MD, USA. New York, NY, USA: ACM. doi:[10.1145/1536414.1536440](https://doi.org/10.1145/1536414.1536440)
- Gentry, C., Sahai, A. & Waters, B. (2013). *Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based*.
- Green, M. (2012). A very casual introduction to Fully Homomorphic Encryption. Recuperado el 10 de mayo de 2019, desde <https://blog.cryptographyengineering.com/2012/01/02/very-casual-introduction-to-fully/>

- Halevi, S. (2017). Homomorphic Encryption. En Y. Lindell (Ed.), *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich* (pp. 219-276). Information Security and Cryptography. Cham: Springer International Publishing. doi:[10.1007/978-3-319-57048-8_5](https://doi.org/10.1007/978-3-319-57048-8_5)
- Homomorphic cryptosystems in RSA. (s.f.). Recuperado el 2 de agosto de 2019, desde <https://crypto.stackexchange.com/questions/3555/homomorphic-cryptosystems-in-rsa>
- Homomorphic Encryption Breakthrough - Schneier on Security. (s.f.). Recuperado el 12 de mayo de 2019, desde https://www.schneier.com/blog/archives/2009/07/homomorphic_enc.html
- homomorphic encryption - difference between leveled FHE and normal FHE scheme. (s.f.). Recuperado el 27 de agosto de 2019, desde <https://crypto.stackexchange.com/questions/15794/difference-between-leveled-fhe-and-normal-fhe-scheme>
- homomorphic encryption - What exactly is bootstrapping in FHE? (s.f.). Recuperado el 25 de agosto de 2019, desde <https://crypto.stackexchange.com/questions/42666/what-exactly-is-bootstrapping-in-fhe>
- Junquera, J. (2019). *TFHE Math Library*. Recuperado desde <https://gitlab.com/junquera/tfhe-math>
- Lattigo 1.0*. (2019). Published: Online: <http://github.com/lca1/lattigo>.
- Micciancio, D. & Regev, O. [Oded]. (2009). Lattice-based Cryptography. En D. J. Bernstein, J. Buchmann & E. Dahmen (Eds.), *Post-Quantum Cryptography* (pp. 147-191). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:[10.1007/978-3-540-88702-7_5](https://doi.org/10.1007/978-3-540-88702-7_5)
- Microsoft SEAL (release 3.3)*. (2019). Recuperado desde <https://github.com/Microsoft/SEAL>
- OBE, P. B. B. (2018). Learning With Errors and Ring Learning With Errors. Recuperado el 12 de mayo de 2019, desde <https://medium.com/asecuritysite-when-bob-met-alice/learning-with-errors-and-ring-learning-with-errors-23516a502406>
- Peng, Z. (2019). Danger of using fully homomorphic encryption: A look at Microsoft SEAL. *arXiv:1906.07127 [cs]*. arXiv: 1906.07127. Recuperado el 21 de agosto de 2019, desde <http://arxiv.org/abs/1906.07127>
- Poyiatzis, A. (2018). Homomorphic Encryption — The holy grail of online confidentiality. Recuperado el 2 de agosto de 2019, desde <https://medium.com/@apogiatzis/homomorphic-encryption-the-holy-grail-of-online-confidentiality-f6e505365039>
- Regev, O. [O.]. (2010). The Learning with Errors Problem (Invited Survey). En *2010 IEEE 25th Annual Conference on Computational Complexity* (pp. 191-204). doi:[10.1109/CCC.2010.26](https://doi.org/10.1109/CCC.2010.26)

- Somewhat Homomorphic Encryption versus Fully Homomorphic Encryption? (s.f.). Recuperado el 10 de mayo de 2019, desde <https://crypto.stackexchange.com/questions/39591/somewhat-homomorphic-encryption-versus-fully-homomorphic-encryption>
- Thaine, P. (2019). Homomorphic Encryption for Beginners: A Practical Guide (Part 1). Recuperado el 2 de agosto de 2019, desde <https://medium.com/privacy-preserving-natural-language-processing/homomorphic-encryption-for-beginners-a-practical-guide-part-1-b8f26d03a98a>
- Troncoso-Pastoriza, J. & Rohloff, K. (s.f.). Homomorphic Encryption Standard Applications, 14.
- Wickr. (2018). What is Lattice-based cryptography & why should you care. Recuperado el 12 de mayo de 2019, desde <https://medium.com/cryptoblog/what-is-lattice-based-cryptography-why-should-you-care-dbf9957ab717>
- Zijlstra, T. (s.f.). Learning with Errors- based Cryptography · Semidoc. Recuperado el 20 de agosto de 2019, desde <https://semidoc.github.io/zijlstra-lwe>

A Test LWE

A.1 Código en python

```
import random

Z_MAX = 1e50
MOD = 681029
q=MOD
DIMENSIONS = 10

s = []
aes = []
es = []

for i in range(DIMENSIONS):
    s.append(random.randint(0, Z_MAX))

for i in range(100):
    a = []
    e = []
    for j in range(DIMENSIONS):
        a.append(random.randint(0, Z_MAX) % MOD)
        e.append(random.randint(0, 10))
    aes.append(a)
    es.append(e)

bs = []
for i in range(len(aes)):
    b = []
    for j in range(DIMENSIONS):
```

```
        b.append((aes[i][j]*s[j]+e[j]) % MOD)
    bs.append(b)

x = int(input("x > "))

# Cifrar
v = []
for i in range(DIMENSIONS):
    '''
    Si v es aleatorio, queda delatada la b porque si x=0
    y v=0 a=0 y b=x*q/2. v tiene que ser 1
    '''
    v.append(1)

reses_a = []
reses_b = []
for i in range(len(aes)):
    res_a = []
    res_b = []
    for j in range(DIMENSIONS):
        res_a.append((aes[i][j]*v[j]))
        res_b.append((bs[i][j]*v[j] + x*int(q/2)))

    reses_a.append(res_a)
    reses_b.append(res_b)

# Descifrar
dec = []
for i in range(len(reses_a)):
    for j in range(DIMENSIONS):
```

```
c1 = reses_a[i][j]
c2 = reses_b[i][j]
c1_aux = c1 * s[j]
dec.append((c2 - c1_aux) % MOD)

print("c1: ", reses_a[0])
print("c2: ", reses_b[0])
print("Result: ", dec[:3])
```

A.2 Ejecución

```
junquera@opa:~/tfm/lwe_tests$ python3 test_lwe.py
x > 1
c1: [583949, 280901, 385393, 215313, 3921, 664402,
     669495, 470378, 548011, 15699]
c2: [597668, 851152, 692041, 618895, 590806, 410594,
     694835, 830543, 381247, 582842]
Result: [340514, 340518, 340516]
junquera@opa:~/tfm/lwe_tests$ python3 test_lwe.py
x > 0
c1: [482569, 656029, 171078, 505801, 228565, 444569, 45811,
     394863, 509682, 357903]
c2: [106231, 198539, 660839, 461502, 350107, 178714, 450856,
     646125, 522843, 106305]
Result: [8, 6, 7]
```


B Regresión cuadrática

B.1 Ejemplo en python

```
from math import sqrt
```

```
xys = [  
    (1.2, 4.5),  
    (1.8, 5.9),  
    (3.1, 7.0),  
    (4.9, 7.8),  
    (5.7, 7.2),  
    (7.1, 6.8),  
    (8.6, 4.5),  
    (9.8, 2.7)  
]
```

```
# finisterre
```

```
xys = [  
    (1.0, 10.6), (2.0, 9.5), (3.0, 13.4), (4.0, 11.7),  
    (5.0, 16.4), (6.0, 17.9), (7.0, 18.8), (8.0, 20.4),  
    (9.0, 19.4), (10.0, 16.5), (11.0, 12.4), (12.0, 12.4)  
]
```

```
# cabogata
```

```
xys = [  
    (1.0, 12.2), (2.0, 10.8), (3.0, 14.5), (4.0, 15.6),  
    (5.0, 20.4), (6.0, 24.1), (7.0, 25.8), (8.0, 26.7),  
    (9.0, 23.7), (10.0, 19.3), (11.0, 16.4), (12.0, 13.5)  
]
```

```
i = 0
j = 0
k = 0
l = 0

u = 0
v = 0
w = 0

n = len(xys)

for x, y in xys:
    i += x
    j += x**2
    k += x**3
    l += x**4

    u += y
    v += (x * y)
    w += ((x**2) * y)

print("i", i)
print("j", j)
print("k", k)
print("l", l)
print("u", u)
print("v", v)
print("w", w)
```



```

c = (u*l - w*j)/(l*n - j**2)
c1 = (j*k*l*v - j*(k**2)*w - i*(l**2)*v + i*k*l*w)
c2 = (j*(l**2)*n - (k**2)*l*n - (j**3)*l + (j**2)*(k**2))
c += c1 / c2
r1 = ((j**2)*(k**2)-2*(i*j*k*l)+(i**2)*(l**2))
r2 = (j*(l**2)*n - (k**2)*l*n - (j**3)*l + (j**2)*(k**2))
r = r1 / r2
c = c / (1 - r)

b = (v*l - k*w + c*j*k - c*i*l) / (j*l - k**2)
a = (w - b*k - c*j) / l

print("a", a)
print("b", b)
print("c", c)

```

```

print("%sx2 + %sx + %s" % (a, b, c))

```

B.1.1 Ejecución

Ejemplo de ejecución con los datos de Finisterre:

```

i 78.0
j 650.0
k 6084.0
l 60710.0
u 186.60000000000002
v 1268.9
w 10486.1
a -0.2617882117882111
b 3.7948551448551346
c 5.063636363636395
-0.2617882117882111x2 + 3.7948551448551346x + 5.063636363636395

```

B.2 Funciones en C++

```
#ifndef _REG2_H_INCLUDED_
#define _REG2_H_INCLUDED_

#include <iostream>
#include <vector>

#include <tfhe/tfhe.h>
#include <tfhe/tfhe_io.h>

#include "tfhe-math/arithmetic.h"
using namespace std;

class RegresionCuadratica {
public:
    RegresionCuadratica(int _nb_bits, int _float_bits,
                        TFheGateBootstrappingCloudKeySet* _bk);
    void calcula(LweSample* a, LweSample* b, LweSample* c,
                const vector<LweSample*> xs, const vector<LweSample*> ys,
                string results_path);
private:
    int nb_bits;
    int float_bits;

    TFheGateBootstrappingCloudKeySet* bk;

    /*
        Inicialización de valores: i, j, k, l, u, v y w
    */
    void initVectores(const vector<LweSample*> xs,
```

```
        const vector<LweSample*> ys,
        string results_path);

/*
    Cálculo de potencias con vectores iniciales:  $i^2, j^2...$ 
*/
void calcCuadrados(string results_path);

/*
    Cálculo de parejas de valores:  $il, kv, ln...$ 
*/
void calcDuplas(string results_path);

/*
    Cálculo de combinaciones de duplas:  $ijkl, i2l2, j3l...$ 
*/
void calcComplejos(string results_path);

void calcC(LweSample* c,
           string results_path);
void calcB(LweSample* b, LweSample* c,
           string results_path);
void calcA(LweSample* a, LweSample* b, LweSample* c,
           string results_path);

LweSample* aux1;
LweSample* aux2;
LweSample* aux3;
LweSample* aux4;

LweSample* n;
LweSample* uno;
```

```
// Vectores iniciales
```

```
LweSample* i;  
LweSample* j;  
LweSample* k;  
LweSample* l;
```

```
LweSample* u;  
LweSample* v;  
LweSample* w;
```

```
// Cuadrados de las variables
```

```
LweSample* i2;  
LweSample* j2;  
LweSample* k2;  
LweSample* l2;
```

```
// Duplas
```

```
LweSample* il;  
LweSample* jk;  
LweSample* jl;  
LweSample* kw;  
LweSample* kv;  
LweSample* ln;  
LweSample* ul;  
LweSample* vl;  
LweSample* wj;
```

```
// Complejos
```

```
LweSample* ijkl;
```

```
LweSample* i2l2;  
LweSample* iklw;  
LweSample* il2v;  
LweSample* j2k2;  
LweSample* jk2w;  
LweSample* jklv;  
LweSample* j3l;  
LweSample* jl2n;  
LweSample* k2ln;  
};  
  
#endif
```


C Datos AEMET

C.1 Obtención de datos

Para descargar los datos de las ciudades ha sido necesario obtener una API key para acceder al sistema de Open Data (<https://opendata.aemet.es/centrodedescargas/inicio>, ver figura C.1).

Figura C.1: Panel de descarga de datos de AEMET



Valores Climatológicos				
Climatologías diarias	Seleccione una provincia	Seleccione una estación	Fecha inicio: <input type="text"/> Fecha fin: <input type="text"/>	Obtener
Climatologías mensuales/anuales	A Coruña	Cargando...	Año (AAAA): <input type="text"/>	Obtener
Valores normales	Seleccione una provincia	Seleccione una estación		Obtener
Extremos registrados	Seleccione una provincia	Seleccione una estación	Seleccione una variable	Obtener
Inventario de estaciones de Valores Climatológicos				Obtener

Una vez descargados, hemos obtenido para cada ciudad un JSON como el siguiente:

```
[ {
  "fecha" : "2012-1",
  "indicativo" : "63250",
  "p_max" : "15.0(16)",
  "...",
  "tm_mes" : "12.2",
  "tm_max" : "16.6",
  "nv_0100" : "0.0",
  "q_min" : "1005.2(16)",
  "np_010" : "2.0"
}, {
  "fecha" : "2012-2",
  "indicativo" : "63250",
```

```

    ...
    "tm_mes" : "14.5",
    "tm_max" : "19.1",
    "nv_0100" : "0.0",
    "q_min" : "1007.5(31)",
    "np_010" : "2.0"
  },
  ...
  {
    "fecha" : "2012-13",
    ...
    "q_mar" : "1016.9",
    "q_med" : "1014.5",
    "tm_min" : "14.7",
    "ta_max" : "39.0(17/ago)",
    "ts_min" : "25.3",
    ...
    "p_sol" : "71.0",
    "nw_91" : "1.0",
    "np_001" : "36.0",
    "ta_min" : "1.8(13/feb)",
    "w_rec" : "320.0",
    "e" : "151.0",
    ...
    "q_min" : "993.3(31/oct)",
    "np_010" : "23.0"
  } ]

```

C.2 Procesado

Para cada mes, para cada archivo, se extrae el parámetro `tm_mes` (que indica la temperatura media del mes) y se genera un documento CSV para poder tratarlo mejor:

ft ,2012 ,1 ,10.6

...

cg ,2018 ,12 ,14.2

D Tests de eficiencia de SEAL

D.1 BFV

```
+-----+
|          BFV Performance Test with Degrees: 4096, 8192, and 16384          |
+-----+
/
| Encryption parameters :
|   scheme: BFV
|   poly_modulus_degree: 4096
|   coeff_modulus size: 109 (36 + 36 + 37) bits
|   plain_modulus: 786433
\
```

Generating secret/public keys: Done

Generating relinearization keys: Done [1503 microseconds]

Generating Galois keys: Done [32975 microseconds]

Running tests Done

Average batch: 113 microseconds

Average unbatch: 99 microseconds

Average encrypt: 1376 microseconds

Average decrypt: 502 microseconds

Average add: 17 microseconds

Average multiply: 5359 microseconds

Average multiply plain: 767 microseconds

Average square: 3793 microseconds

Average relinearize: 1090 microseconds

Average rotate rows one step: 1098 microseconds

Average rotate rows random: 4374 microseconds

Average rotate columns: 1107 microseconds

/

```
| Encryption parameters :  
|   scheme: BFV  
|   poly_modulus_degree: 8192  
|   coeff_modulus size: 218 (43 + 43 + 44 + 44 + 44) bits  
|   plain_modulus: 786433  
\  

```

Generating secret/public keys: Done

Generating relinearization keys: Done [9157 microseconds]

Generating Galois keys: Done [207176 microseconds]

Running tests Done

Average batch: 221 microseconds

Average unbatch: 163 microseconds

Average encrypt: 4150 microseconds

Average decrypt: 1812 microseconds

Average add: 66 microseconds

Average multiply: 19627 microseconds

Average multiply plain: 3140 microseconds

Average square: 13518 microseconds

Average relinearize: 5348 microseconds

Average rotate rows one step: 5277 microseconds

Average rotate rows random: 25931 microseconds

Average rotate columns: 5203 microseconds

/

```
| Encryption parameters :  

```

```
|  scheme: BFV
|  poly_modulus_degree: 16384
|  coeff_modulus size: 438 (48 + 48 + 48 + 49 + 49 + 49 + 49 + 49 + 49) bits
|  plain_modulus: 786433
\
```

```
Generating secret/public keys: Done
Generating relinearization keys: Done [55666 microseconds]
Generating Galois keys: Done [1376466 microseconds]
Running tests ..... Done
```

```
Average batch: 410 microseconds
Average unbatch: 359 microseconds
Average encrypt: 15324 microseconds
Average decrypt: 7145 microseconds
Average add: 344 microseconds
Average multiply: 76506 microseconds
Average multiply plain: 12462 microseconds
Average square: 54013 microseconds
Average relinearize: 31034 microseconds
Average rotate rows one step: 31823 microseconds
Average rotate rows random: 166624 microseconds
Average rotate columns: 31432 microseconds
```

D.2 CKKS

```
+-----+
|          CKKS Performance Test with Degrees: 4096, 8192, and 16384          |
+-----+
/
| Encryption parameters :
|  scheme: CKKS
```

```
| poly_modulus_degree: 4096
| coeff_modulus size: 109 (36 + 36 + 37) bits
\
```

```
Generating secret/public keys: Done
Generating relinearization keys: Done [3381 microseconds]
Generating Galois keys: Done [36900 microseconds]
Running tests ..... Done
```

```
Average encode: 1853 microseconds
Average decode: 2540 microseconds
Average encrypt: 1832 microseconds
Average decrypt: 75 microseconds
Average add: 19 microseconds
Average multiply: 204 microseconds
Average multiply plain: 74 microseconds
Average square: 125 microseconds
Average relinearize: 1181 microseconds
Average rescale: 513 microseconds
Average rotate vector one step: 1424 microseconds
Average rotate vector random: 6035 microseconds
Average complex conjugate: 1390 microseconds
```

```
/
| Encryption parameters :
|   scheme: CKKS
|   poly_modulus_degree: 8192
|   coeff_modulus size: 218 (43 + 43 + 44 + 44 + 44) bits
\
```

```
Generating secret/public keys: Done
Generating relinearization keys: Done [8152 microseconds]
Generating Galois keys: Done [186742 microseconds]
Running tests ..... Done
```

```
Average encode: 4297 microseconds
Average decode: 8697 microseconds
Average encrypt: 5611 microseconds
Average decrypt: 255 microseconds
Average add: 74 microseconds
Average multiply: 770 microseconds
Average multiply plain: 278 microseconds
Average square: 519 microseconds
Average relinearize: 5396 microseconds
Average rescale: 2609 microseconds
Average rotate vector one step: 6441 microseconds
Average rotate vector random: 29696 microseconds
Average complex conjugate: 6378 microseconds
```

```
/
| Encryption parameters :
|   scheme: CKKS
|   poly_modulus_degree: 16384
|   coeff_modulus size: 438 (48 + 48 + 48 + 49 + 49 + 49 + 49 + 49 + 49) bits
\
```

```
Generating secret/public keys: Done
Generating relinearization keys: Done [49346 microseconds]
Generating Galois keys: Done [1325534 microseconds]
Running tests ..... Done
```

Average encode: 12753 microseconds
Average decode: 34802 microseconds
Average encrypt: 20628 microseconds
Average decrypt: 1055 microseconds
Average add: 300 microseconds
Average multiply: 3567 microseconds
Average multiply plain: 1140 microseconds
Average square: 2662 microseconds
Average relinearize: 33513 microseconds
Average rescale: 11607 microseconds
Average rotate vector one step: 36967 microseconds
Average rotate vector random: 199463 microseconds
Average complex conjugate: 36792 microseconds

E Servidor en Digital Ocean

E.1 Factura de servicios



Final invoice for the August 2019 billing period

From

DigitalOcean
101 Avenue of the Americas, 10th Floor
New York, NY 10013
VAT ID: EU528002224

Details

Invoice number:

Date of issue:

Payment due on:

September 1, 2019

September 1, 2019

For

junquera
<javier@junquera.xyz>

Summary

Total usage charges	\$4.19
Subtotal	\$4.19
VAT Spain (21.00%)	\$0.88
Total due	\$5.07

If you have a credit card on file, it will be automatically charged within 24 hours

Product usage charges

Detailed usage information is available in a CSV usage report accessible from the billing overview in your account

Droplets	Hours	Start	End	\$4.19
tfm-tfhe (s-1vcpu-1gb)	26	08-19 07:18	08-20 09:24	\$0.19
tfm-tfhe (s-2vcpu-2gb)	179	08-20 09:24	08-27 20:34	\$4.00

F Manual de instalación y uso

Manual de instalación y uso de la solución del trabajo (4). Estos pasos han sido probados en un sistema Ubuntu 19.04 de 64 bits.

F.1 Prerequisitos

F.1.1 SEAL

Como hemos comentado, Microsoft SEAL no necesita ninguna dependencia. Para instalarlo (generar las librerías y los archivos de cabeceras) seguimos los siguientes pasos:

```
$ git clone https://github.com/Microsoft/SEAL
$ cd SEAL/native/src
$ cmake .
$ make
```

Generará la librería compilada en la ruta SEAL/native/lib, y tendremos sus archivos .h en el directorio SEAL/native/src.

Para más detalles, ver la guía de instalación oficial en <https://github.com/Microsoft/SEAL>.

F.1.2 TFHE

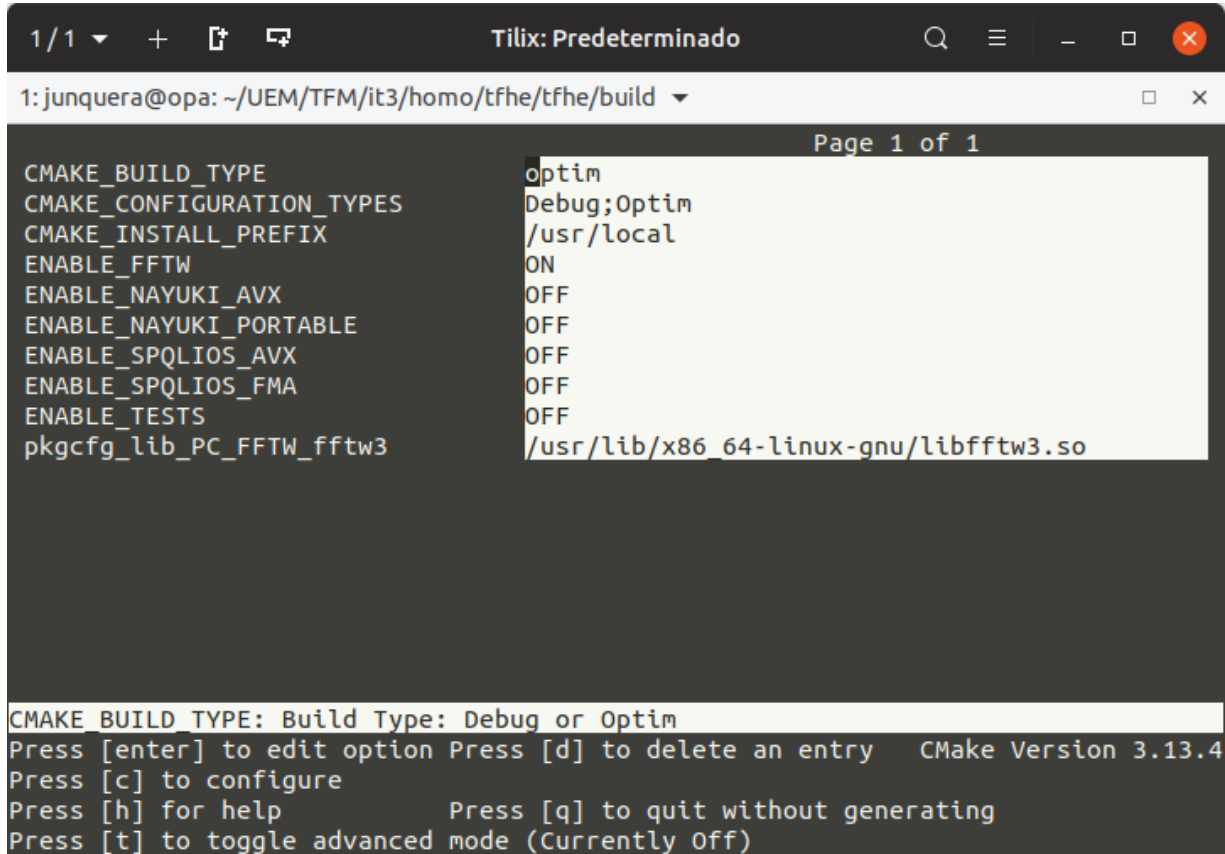
La librería TFHE sí que tiene algunos prerequisites, que satisfaremos ejecutando:

```
$ sudo apt-get install build-essential cmake cmake-curses-gui
```

Una vez instalados, descargamos el código y lo configuramos para compilarlo:

```
$ #clone the tfhe repository
$ git clone --recurse-submodules \
  --branch=master https://github.com/tfhe/tfhe.git
$ cd tfhe
$ #configure the build options
$ mkdir build
$ cd build
$ cmake ../src
```

Abrirá un menú con varias opciones dependiendo de las características que queramos utilizar. En la figura F.1 se muestran las que hemos utilizado en nuestra instalación.



```
1/1  +  [?]  [x]  Tilix: Predeterminado  [?]  [≡]  [–]  [□]  [X]
1: junquera@opa: ~/UEM/TFM/it3/homo/tfhe/tfhe/build  [□]  [X]
Page 1 of 1
CMAKE_BUILD_TYPE                optim
CMAKE_CONFIGURATION_TYPES        Debug;Optim
CMAKE_INSTALL_PREFIX             /usr/local
ENABLE_FFTW                      ON
ENABLE_NAYUKI_AVX                OFF
ENABLE_NAYUKI_PORTABLE           OFF
ENABLE_SPQLIOS_AVX               OFF
ENABLE_SPQLIOS_FMA               OFF
ENABLE_TESTS                     OFF
pkgcfg_lib_PC_FFTW_fftw3         /usr/lib/x86_64-linux-gnu/libfftw3.so

CMAKE BUILD TYPE: Build Type: Debug or Optim
Press [enter] to edit option Press [d] to delete an entry  CMake Version 3.13.4
Press [c] to configure
Press [h] for help          Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figura F.1: Parámetros de configuración de tfhe

Tras guardar esta configuración, simplemente ejecutamos:

```
$ #build the library
$ make
```

Como resultado obtendremos la librería compilada dentro de `tfhe/build/libtfhe`. En función de los parámetros que hayamos elegido tendrá un nombre u otro. Con nuestros parámetros se llamará `libtfhe-fftw.so`

Para más detalles, ver la guía de instalación oficial en <https://tfhe.github.io/tfhe/installation.html>.

F.2 Instalación

F.2.1 tfhe-math

Para instalar `tfhe-math` necesitaremos haber compilado `tfhe`. Primero descargamos el código y entramos en la carpeta para configurarlo:

```
$ git clone git@gitlab.com:junquera/tfhe-math.git
$ cd tfhe-math
```

Editamos el archivo `src/Makefile` y especificamos en la variable `TFHE_PREFIX` la ruta en la que están la librería de `tfhe` y sus archivos `.h`. Tras guardar, entramos en la carpeta `src` y ejecutamos el comando `make`. Creará una carpeta llamada `prefix` en la raíz del proyecto con todos los archivos necesarios para que utilicemos la librería en otros proyectos.

F.2.2 tfhe-cs

Descargaremos el código con las funcionalidades de cliente y servidor desde su repositorio en GitLab:

```
$ git clone --recurse-submodules git@gitlab.com:junquera/tfhe-cs.git
$ cd tfhe-cs
```

De esta forma, además se descargará la librería `tfhe-math` para que la usemos (si no la hemos instalado aún).

En el archivo `Makefile` tenemos que especificar la ruta a las carpetas de `prefix` (las carpetas que contengan las librerías y los archivos de cabeceras `.h`) de `tfhe` y de `tfhe-math`.

Ejecutando el comando `make` generaremos el programa cliente, el programa servidor y el ejecutable con los test.

F.2.3 seal-cs

Haremos exactamente lo mismo que con `tfhe-cs`, pero eligiendo su repositorio correspondiente:

```
$ git clone git@gitlab.com:junquera/seal-cs.git  
$ cd tfhe-cs
```

Esta vez, en lugar de especificar la ruta a `tfhe`, indicamos en el archivo `Makefile` la ruta a SEAL y ejecutamos el comando `make`.

F.3 Uso

A continuación veremos los distintos usos que podemos darle a nuestra implementación, ya sea para ejecutar el código del trabajo o realizar nuestra propia implementación. Las cabeceras de referencia de cada programa se pueden ver en el anexo [G](#).

F.3.1 `tfhe-math`

Para utilizar las funciones de `tfhe-math` únicamente tenemos que enlazar nuestro código con la librería, e incluir sus archivos de cabecera en él. Por ejemplo, para utilizar las funciones aritméticas escribiríamos:

```
#include "tfhe-math/arithmetic.h"
```

Después trabajaríamos con sus métodos, que funcionan tal y como hemos indicado en la sección [4.2.2](#), teniendo en cuenta únicamente que:

- Los métodos precedidos con `u_` son más rápidos, pero no tienen en cuenta el signo.
- Si se trabaja con números reales, para multiplicar y dividir usaremos `multiply_float` y `divide_float` respectivamente (en lugar de `multiply` y `divide`)

F.3.2 `tfhe-cs`

Si quisiésemos usar la funcionalidad del cliente o del servidor (no del programa, si no de la clase) bastaría con utilizar los códigos `client.cpp` o `server.cpp` (respectivamente) incluyendo sus archivos de cabeceras.

F.3.3 `seal-cs`

Al igual que con `tfhe-cs` podemos utilizar nuestros códigos de cliente y servidor para hacer una implementación propia con sus funcionalidades, o ejecutar los programas generados (`client`, `server` y `test`).

G Archivos de cabeceras

A continuación mostraremos las cabeceras (archivos .h) de los programas cliente y servidor, tanto de TFHE como de SEAL.

G.1 tfhe-cs

G.1.1 Cliente

```
#ifndef _CLIENT_H_INCLUDED_
#define _CLIENT_H_INCLUDED_

#include "common/hefile.h"

class HClient {
public:
    HClient(int _nb_bits, int _float_bits);
    HClient(int _nb_bits, int _float_bits, string sec_key_path);
    void genKeys();
    void setKeysFromFile(string keyFileName);
    void cifra(LweSample* answer, int64_t input);
    int64_t descifra(LweSample* answer);
    void exportSecretKeyToFile(string name);
    void exportCloudKeyToFile(string name);
    void getCloudKey(TFheGateBootstrappingCloudKeySet* &cloudKey);
private:
    int nb_bits;
    int float_bits;
    TFheGateBootstrappingSecretKeySet* key;
};

#endif
```

G.1.2 Servidor

```
#ifndef _SERVER_H_INCLUDED_
#define _SERVER_H_INCLUDED_

#include "common/hefile.h"
#include "reg2.h"

class HServer {
public:
    HServer(int _nb_bits, int _float_bits);
    // TODO cloud_key_path > bk
    void regresionCuadratica(LweSample* a, LweSample* b, LweSample* c,
        const vector<LweSample*> xs, const vector<LweSample*> ys,
        string cloud_key_path, string results_path);
private:
    int nb_bits;
    int float_bits;
};

#endif
```

G.2 seal-cs

G.2.1 Cliente

```
#ifndef _CLIENT_H_INCLUDED_
#define _CLIENT_H_INCLUDED_

#include <iostream>
#include <fstream>          // std::ofstream

#include "seal/seal.h"
```



```
#include "constants.h"
#include "common/sealfile.h"

#include <vector>

#include "server.h"
#include "constants.h"

using namespace std;
using namespace seal;

class Distance {
public:
    string name;
    double distance;

    Distance(string name, double distance)
        : name{name}, distance{distance} {
    }
};

class SClient {
public:
    SClient();
    SClient(string config_mask);
    SClient(string config_mask,
            size_t poly_modulus_degree, vector<int> coeff_modulus);
    void genKeys();
```

```
void setKeysFromFile(string keyFileName);
Ciphertext encrypt(double a);
Ciphertext encrypt(vector<double> a);
vector<double> decrypt(Ciphertext a);
void saveConfig();
void saveConfig(string config_mask);
PublicKey public_key;
RelinKeys relin_keys;
EncryptionParameters* parms;
private:
    SecretKey secret_key;
    Encryptor* encryptor;
    Evaluator* evaluator;
    Decryptor* decryptor;
    CKKSEncoder* encoder;
    string config_mask;
};
```

#endif

G.2.2 Servidor

```
#ifndef _SERVER_H_INCLUDED_
#define _SERVER_H_INCLUDED_

#include <iostream>
#include <fstream>          // std::ofstream

#include "seal/seal.h"
#include <vector>
#include <string>
```

```
#include "constants.h"
#include "common/sealfile.h"
#include "curva.h"

using namespace std;
using namespace seal;

class SServer {
public:
    SServer();
    SServer(string config_mask);
    Ciphertext distance(Ciphertext x_encrypted,
                       Ciphertext y_encrypted,
                       RelinKeys relin_keys);
    void addCurva(Curva c);
    vector<string> getCurveNames();
private:
    Evaluator* evaluator;
    CKKSEncoder* encoder;
    vector<Curva> curvas;
};

#endif
```

G.2.3 Curvas

```
#ifndef _CURVA_H_INCLUDED_
#define _CURVA_H_INCLUDED_

#include <string>
#include <vector>
```

```
#include <iostream>
#include <fstream>          // std::ofstream

using namespace std;

class Curva {
public:
    double a, b, c;
    std::string name = "";
    Curva(double a, double b, double c) : a{a}, b{b}, c{c} {
    }

    Curva(std::string name, double a, double b, double c)
        : name{name}, a{a}, b{b}, c{c} {
    }
};

inline void saveCurveNames(vector<Curva> curvas, string filename){
    ofstream res;
    res.open(filename);

    for(Curva c: curvas)
        res << c.name << endl;

    res.close();
};

inline void saveCurveNames(vector<string> curvas, string filename){
```

```
ofstream res;
res.open(filename);

for(string name: curvas)
    res << name << endl;

res.close();
};

inline vector<string> loadCurveNames(string filename){
    vector<string> names;
    string line;
    ifstream res(filename);
    if (res.is_open()) {
        while ( getline (res, line) ) {
            names.push_back(line);
            cout << line << '\n';
        }
        res.close();
    }
    return names;
};

#endif
```