UNIVERSITY OF TSUKUBA
COLLEGE OF INFORMATION SCEINCE

Referential summary

regarding

Advanced Cryptography Instances

Title:

Homomorphic Cryptography: Matrix Multiplication

Feat. by: Mamanchuk Mykola
Student ID: **202258010**

Date: **July 5, 2023**

CONTENTS

Introduction 3
Recap on Homomorphic Cryptography 3
Introduction to Matrix Multiplication using Homomorphic Encryption 3
Notations and essential background 4
Matrix notations and operators 4
General on Homomorphic Encryption 4
Plaintext packing and masking 5
Evaluation on Homomorphic Encryption operations cost 7
CKKS HE Scheme 8
Generic Homomorphic Encryption Matrix Multiplication Algorithm 10
Notations and Operations 10
Processing, Computation Algorithms, and Examples 11
Multiplication processing 11
Plaintext Encoding 12
Row and column masking 13
Matrix A computation algorithm 14
Matrix A computation example 14
Matrix B Computation algorithm 16
Matrix B computation example 17
Product Matrix $\mathbb{C}$ computation algorithm 19
Product Matrix $\mathbb{C}$ computation example 19
Complexity Analysis 20
Complexity summary 21
Conclusion 22
Sources 23

Introduction

Recap on Homomorphic Cryptography

Homomorphic Encryption (HE) is a unique class of encryption methods that enables computations to be performed directly on encrypted data, without requiring any decryption process. The results of such computations remain encrypted, and when decrypted, they match the results of the same operations performed on the original, unencrypted data.

$$\forall\, m_1, m_2 \in P\colon D(E(m_1) \oplus E(m_2)) = m_1 \oplus m_2$$

It enables the operation on the ciphered information without decrypting it, thereby preserving the privacy of the data. The resultant encrypted output, when decrypted, matches the result of operations performed on the plaintext. In this context, matrix multiplication is a common operation widely used in various computational domains. Efficiently implementing an HE-based algorithm for matrix multiplication is a complex yet crucial task.

The paper will explore notations and essential background, discuss a generic Homomorphic Encryption Matrix Multiplication Algorithm, provide an example and complexity analysis, compare it with different approaches, and talk about its implementation and evaluation. The objective is to offer a comprehensive understanding of how matrix multiplication can be conducted using Homomorphic Encryption. Essential source for the following discussion is a summary from the paper [1].

Introduction to Matrix Multiplication using Homomorphic Encryption

Conventional matrix multiplication methodologies are inadequate when working with homomorphically encrypted data, primarily due to the intricacies of encryption techniques and the need to maintain data privacy. A significant challenge is that encrypted data elements cannot be processed in the same straightforward manner as their unencrypted counterparts. This restriction necessitates the development of special algorithms, like the ones presented in this study, to perform operations such as matrix multiplication.

Moreover, computational efficiency is another critical factor to consider. Traditional algorithms, when applied directly on encrypted data, can lead to a significant increase in computational load and processing time. This is not feasible in many real-world scenarios where time and computational resources are limited. The proposed approach mitigates this issue by leveraging idle ciphertext slots and only

requiring a single ciphertext-ciphertext multiplication, thus optimizing resource usage.

Furthermore, maintaining the integrity and privacy of encrypted data during computation is paramount. Conventional methods might inadvertently expose data patterns or other sensitive information, even if the specific data values remain encrypted. This paper's methodology is designed to handle encrypted data while preserving privacy and ensuring security, making it a more suitable approach for matrix multiplication in the context of homomorphically encrypted data.

Notations and essential background

Matrix notations and operators

Matrices are denoted using uppercase letters, while vectors are represented by bold uppercase letters.

1. When A represents a $(d_1 \times d)$ matrix and B represents a $(d \times d_2)$ matrix, their linear algebra multiplication, denoted by $C = A \cdot B$, gives a $(d_1 \times d_2)$ product matrix C.
2. Another operation, denoted as '$\odot$', refers to the pair-wise multiplication of corresponding elements within two matrices. For example:

$$\begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \end{bmatrix} \odot \begin{bmatrix} b_0 & b_1 & b_2 \\ b_3 & b_4 & b_5 \end{bmatrix} = \begin{bmatrix} a_0 \cdot b_0 & a_1 \cdot b_1 & a_2 \cdot b_2 \\ a_3 \cdot b_3 & a_4 \cdot b_4 & a_5 \cdot b_5 \end{bmatrix}$$

Notation for concatenating two matrices. For instance, if A_1 and A_2 are matrices of size $(d_1 \times d)$ and $(d_2 \times d)$, respectively, their vertical concatenation is denoted by $[A_1; A_2]$ and results in a $(d_1 + d_2) \times d$ matrix.

Similarly, horizontal concatenation of two matrices B_1 and B_2 of size $(d \times d_1)$ and $(d \times d_2)$, respectively, is denoted by $[B_1 \parallel B_2]$.

Finally, the notation $A_{i,j}$ refers to the entry of a matrix A at the i-th row and j-th column, while $A_{i,:}$ denotes the entire i-th row of a matrix.

General on Homomorphic Encryption

The core of a Homomorphic Encryption (HE) scheme is made up of four integral algorithms, specifically designed for handling the different phases of encryption, decryption, and evaluation on encrypted data. Before delving into the details of these algorithms, let's define the plaintext and ciphertext spaces, represented by M and C respectively.

1. $KeyGen$ (1^{λ}): This algorithm is responsible for the generation of the keys. Given a security parameter, denoted by λ, it generates the secret key (sk), the public key (pk), and a set of evaluation keys (evk).
2. Enc_{pk} (m): This algorithm takes as input the plaintext m from the plaintext space M and encrypts it using the public key pk.
3. Dec_{sk} (c): The inverse of encryption, this algorithm takes a ciphertext c from the ciphertext space C and decrypts it using the private key sk, returning the original plaintext m.
4. $Eval_{evk}$ (f ; c_1, · · · , c_k): This algorithm enables computation directly on encrypted data. Given a circuit f : Mk → M and a tuple of ciphertexts c_1, · · · , c_k as inputs, it computes the function on the inputs, returning a ciphertext c which represents the output of the function.

The two operations form the core of the computation on encrypted data – addition and multiplication. These operations can be executed between two ciphertexts or a plaintext and a ciphertext.

The multiplication between ciphertexts and multiplication between a ciphertext and plaintext are denoted by $Mult$ (ct_1, ct_2) and $cMult$ (ct, m) respectively.

Addition is represented by Add(ct_1, ct_1) and $cAdd$(ct, m) for addition between two ciphertexts and a ciphertext and a plaintext respectively.

The security of the system is primarily based on the Learning With Errors (LWE) problem and its variant, the Ring Learning With Errors (RLWE) problem [2]. These cryptographic problems have been at the forefront of research in the past decade and have led to practical solutions in the field of HE.

Plaintext packing and masking

HE schemes often leverage a feature known as plaintext packing (also known as plaintext encoding) to significantly enhance their performance [3]. This method enables the encoding of multiple messages within a single plaintext. For instance, HE schemes predicated on the Ring Learning With Errors (RLWE) assumption, when applied with a cyclotomic polynomial of degree $2n$, facilitate the storage of up to n values within the same plaintext. These stored values are referred to as plaintext or ciphertext slots.

The plaintext, and its corresponding ciphertext, can be interpreted as a vector. With plaintext encoding, each message is stored in a slot of the vector, and homomorphic operations are executed component-wise, following a Single Instruction Multiple Data (SIMD) approach. This means that the same operation (be it addition or

multiplication) is performed between corresponding slots of the HE operation inputs, whether these are ciphertexts or a mixture of plaintext and ciphertext.

Several strategies exist for plaintext packing. For example, within the context of matrix multiplication, all elements of a matrix can be encoded in a single ciphertext, or each element can be stored in different ciphertexts. The matrix packing approach is explored in section Example and complexity analysis.

However, plaintext packing also presents several constraints. Although the plaintext can be interpreted as a vector, it does not allow for random access to a specific slot, nor the transfer of data between slots, or the application of an operation to a subset of slots due to data manipulation and encryption. To overcome these limitations, two primary operations are utilised: the cyclic rotation $Rot()$ of the slots and scalar (plaintext) multiplication $cMult()$.

HE schemes founded on the RLWE assumption take advantage of the Galois group's structure to implement left and right cyclic rotation operations.

Here, $Rot(ct, l)$ denotes the left rotation operation that transforms a ciphertext ct, which encrypts plaintext m = $(m_0, \cdots, m_{n-1}) \in$ M, into the encryption of the message $(m_l, \cdots, m_{n-1}, m_0, \cdots, m_{l-1})$. When l is a negative integer, it represents plaintext rotation to the right, i.e., by $(n - l)$.

In practical scenarios, it is often convenient to multiply a ciphertext ct with a binary plaintext.

Let Π represent a binary plaintext where all slots $i \in I_1$ are equal to 1 and all others $I_0 = Z_n \setminus I_1$ are zero. By executing $cMult\ (ct, \Pi)$, the encrypted plaintext is zero at slots I_0 and equals ct at all other slots. This implies that the new ciphertext encrypts a subset of the original slots and zero in all other slots, a technique referred to as 'masking'.

Handling arithmetic circuits with inputs from different slots of the same plaintext presents a challenge. Thus, to compute elements in varying relative positions, rotation and masking operations are combined. For example, to add values a_1 and a_2 in the first two slots of a plaintext, masking is first used to compute a ciphertext of a_2 (and other that are zeros). Then, rotation and ciphertext addition are used to compute the desired result in the first slot.

Masking and rotation are utilized to perform computations on the rows or columns of an encrypted matrix.

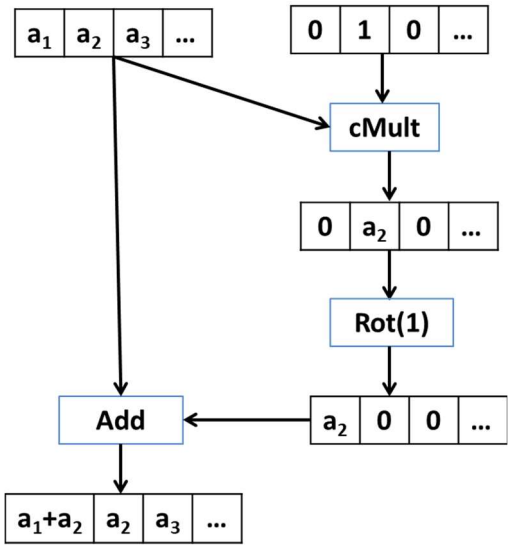

Figure 1 [1]: A simple example of adding two entries from different slots of the same ciphertext.

Evaluation on Homomorphic Encryption operations cost

Noise management poses a significant challenge for all HE schemes. In HE, the ciphertext is a noise-infused version of the corresponding plaintext. The magnitude of this noise, B, must remain within two thresholds, B_{min} and B_{max}. If $B < B_{min}$, the ciphertext fails to secure the message. Conversely, if $B > B_{max}$, the noise cannot be removed, and thus the accurate message can no longer be retrieved.

In the realm of noise management, the multiplication of ciphertexts is the most costly operation. The noise level escalates hyper exponentially to B^{2^L} following L successive multiplications. This number L, referred to as the multiplicative depth of the executed circuit, is preferably kept as low as possible.

Moreover, HE operations exhibit markedly different computational overheads. Table 1 compares the runtime of four HE operations utilizing the CKKS [4] scheme, one of the most practical HE schemes. It reveals that ciphertext-to-ciphertext multiplication and cyclic rotation consume significantly more time than scalar (plaintext-ciphertext) multiplication or ciphertext-to-ciphertext addition.

Operation	$L = 3, n = 2^{12}$	$L = 5, n = 2^{13}$	$L = 9, n = 2^{14}$
Add	16	59	239
cMult	54	212	853
Mult	1576	7681	41483
Rot 1 step	1123	5325	30355
Rot rand	4197	21887	134267

Table 1 [1]: computing the runtime (μ_{sec}) for several HE operations. The CKKS HE scheme is used for the experiments, where L the number of layers.

The computational complexity of an algorithm is characterized as a function of the resource-intensive HE operations involved (ciphertext multiplications and cyclic rotations). The multiplicative depth of the arithmetic circuit implementing the algorithm is also evaluated.

CKKS HE Scheme

The Cheon-Kim-Kim-Song (CKKS) scheme, introduced in the paper "Homomorphic Encryption for Arithmetic of Approximate Numbers", is a particular kind of HE approach. What sets it apart is its ability to perform computations on vectors of complex values, which includes real values.

The CKKS scheme builds on the concept of approximate homomorphic encryption (AHE), a type of encryption that supports the computation of approximated operations on encrypted numbers, as opposed to exact operations in other forms of HE. CKKS is specially designed to accommodate floating point arithmetic, a critical feature for many practical applications.

One of the major advantages of CKKS is its ability to handle complex and real numbers, which is not common in HE schemes. This attribute makes it highly suitable for applications that require operations on such numbers, including machine learning algorithms, scientific computations, and statistical analysis, among others. The complexity of these computations requires an encryption scheme that can handle approximation, something that the CKKS scheme does remarkably well.

However, it's important to note that the CKKS scheme deals with approximate, rather than exact, computations. As a result, there will always be a certain degree of error in the final decrypted results. This error arises from two main sources: the inherent noise in the encryption process and the rounding errors from performing computations on encrypted data. This has to be controlled after a certain amount of operations to sustain a reliable output result after the decryption process.

Despite this, the CKKS represents a significant stride in the field of HE, enabling more complex computations on encrypted data than were previously possible. This has opened up a host of potential applications, particularly in privacy-preserving data analysis and machine learning, making it an important tool in the field of data security.

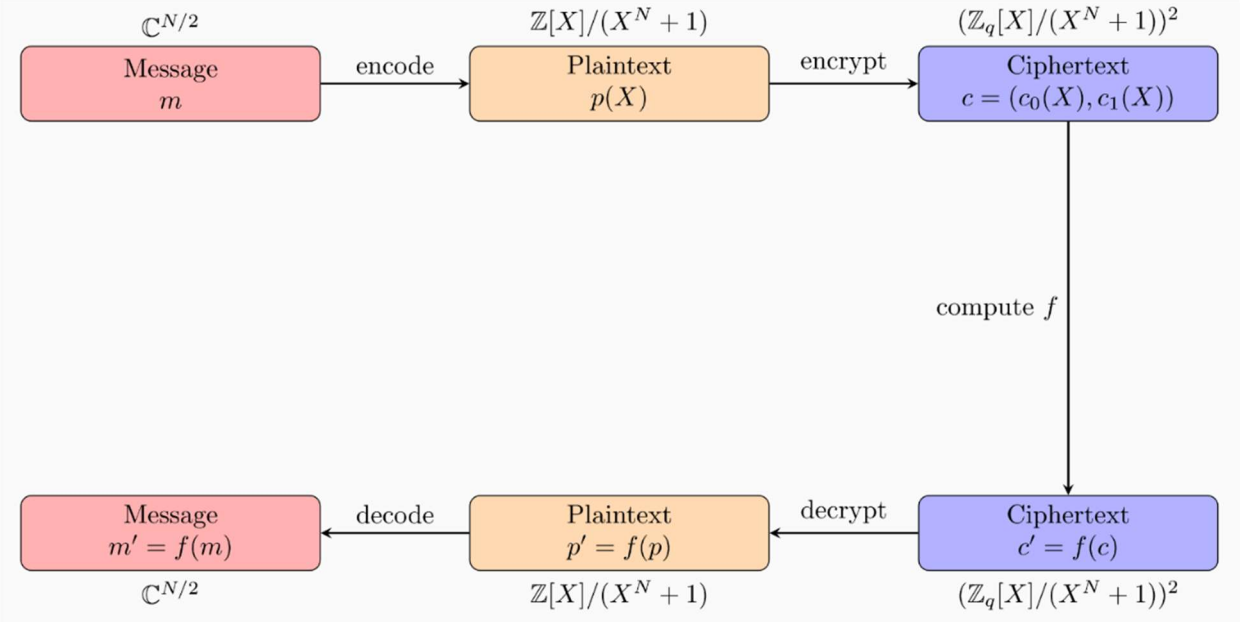

Figure 2 [5]: High level view of CKKS

The provided diagram offers an overarching representation of the CKKS. Here, a message *m*, constituting a vector of values for which certain computations are desired, is initially encoded into a plaintext polynomial *p*(*X*) and subsequently encrypted utilizing a public key.

CKKS employs polynomials given their optimal balance between security and efficiency in comparison to conventional vector computations.

Upon the encryption of the message *m* into *c* (a pair of polynomials), CKKS allows for the execution of several operations on c, including addition, multiplication, and rotation.

Assuming a function *f* embodies a combination of homomorphic operations, decrypting `c' = f(c)` using the secret key will result in `p' = f(p)`. Hence, on decoding *p'*, the output is `m' = f(m)`.

The core principle in implementing a HE scheme is to incorporate homomorphic properties into the encoder, decoder, encryptor, and decryptor. This ensures that operations executed on ciphertexts, when decrypted and decoded, yield outputs as if the operations were performed directly on the plaintexts.

Generic Homomorphic Encryption Matrix Multiplication Algorithm

Notations and Operations

Let's introduce the matrix multiplication algorithm that is implemented homomorphically.

Let A and B be two matrices of size $(d_1 \times d)$ and $(d \times d_2)$, respectively.

The product $C = A \cdot B$ has size $(d_1 \times d_2)$ and it is given by:

$$c_{i,j} = \sum_{k=0}^{d-1} a_{i,k} \cdot b_{k,j},\ 0 \le i < d_1,\ 0 \le j < d_2 \qquad (1)$$

For $0 \le k < d$, $(d_1 \times d_2)$ matrices are defined as:

$$\hat{A}_k = \{\hat{a}_{i,j}^{(k)}\}_{0 \le i < d_1, 0 \le j < d_2}$$

such that and the $(d_1 \times d_2)$ matrices $\widehat{\mathrm{B}}_k$ is defined as:

$$\hat{B}_k = \{\hat{b}_{i,j}^{(k)}\}_{0 \le i < d_1, 0 \le j < d_2}$$

such that, $\hat{b}_{i,j}^{(k)} = b_{k,j}$.

That is that, the matrix $\hat{A}_k$ is computed by horizontally concatenating d_2 times the k-th column of A, while $\widehat{\mathrm{B}}_k$ is the computed by vertically concatenating d_1 times the k-th row of B. Thus, from (1), the product C can be computed as,

$$C = A \cdot B = \sum_{k=0}^{d-1} \hat{A}_k \odot \hat{B}_k \qquad (2)$$

Example 1. Let $d_1 = d = d_2 = 4$. Then,

$$C = A \cdot B = \sum_{k=0}^{3} \hat{A}_k \odot \hat{B}_k = \begin{bmatrix} a_{0,0} & a_{0,0} & a_{0,0} & a_{0,0} \\ a_{1,0} & a_{1,0} & a_{1,0} & a_{1,0} \\ a_{2,0} & a_{2,0} & a_{2,0} & a_{2,0} \\ a_{3,0} & a_{3,0} & a_{3,0} & a_{3,0} \end{bmatrix} \odot \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \end{bmatrix}$$
$$+ \begin{bmatrix} a_{0,1} & a_{0,1} & a_{0,1} & a_{0,1} \\ a_{1,1} & a_{1,1} & a_{1,1} & a_{1,1} \\ a_{2,1} & a_{2,1} & a_{2,1} & a_{2,1} \\ a_{3,1} & a_{3,1} & a_{3,1} & a_{3,1} \end{bmatrix} \odot \begin{bmatrix} b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \end{bmatrix}$$
$$+ \begin{bmatrix} a_{0,2} & a_{0,2} & a_{0,2} & a_{0,2} \\ a_{1,2} & a_{1,2} & a_{1,2} & a_{1,2} \\ a_{2,2} & a_{2,2} & a_{2,2} & a_{2,2} \\ a_{3,2} & a_{3,2} & a_{3,2} & a_{3,2} \end{bmatrix} \odot \begin{bmatrix} b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix}$$
$$+ \begin{bmatrix} a_{0,3} & a_{0,3} & a_{0,3} & a_{0,3} \\ a_{1,3} & a_{1,3} & a_{1,3} & a_{1,3} \\ a_{2,3} & a_{2,3} & a_{2,3} & a_{2,3} \\ a_{3,3} & a_{3,3} & a_{3,3} & a_{3,3} \end{bmatrix} \odot \begin{bmatrix} b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

The result of the multiplication $C = A \cdot B$ can then be computed by summing the element-wise (pair-wise) multiplication of the matrices $\hat{A}_k$ and $\widehat{\mathrm{B}}_k$ for each value of k from 0 to $d - 1$.

The advantage of this approach is that it reduces matrix multiplication to simpler operations (sums and element-wise products), which could be more easily handled in a homomorphic context where full-fledged matrix multiplication might not be possible or more efficient.

Processing, Computation Algorithms, and Examples

Multiplication processing

Here is introduced a method for homomorphic multiplication of square matrices of size d, given that $d^3 < n$, where n is the number of plaintext slots. This condition ensures that the multiplication can be effectively processed within the available slots of the plaintext. For simplicity, d is assumed to be a power of 2.

In this method, matrix multiplication is expressed as the addition of d pointwise products of matrices $\hat{A}_k$ and $\hat{B}_k$.

These matrices are constructed from the input matrices A and B as follows:

The matrix $\hat{A}$ is obtained by vertically concatenating matrices $\hat{A}_k$, i.e.,

$\hat{A} = [\hat{A}_0; \cdots; \hat{A}_{d\text{-}1}]$

Similarly, the matrix $\hat{B}$ is obtained by vertically concatenating matrices $\hat{B}_k$, i.e.,

$\hat{B} = [\hat{B}_0; \cdots; \hat{B}_{d\text{-}1}]$

The multiplication process is divided into three algorithms:

1. Computation of the matrix $\hat{A}$;
2. Computation of the matrix $\hat{B}$;
3. Computation of the final product $C = \hat{A} \times \hat{B}$.

Plaintext Encoding

The plaintext encoding is a design requirement that ensures the input matrices A, B and the output C have the same representation. This is done to facilitate the developer's work as they only have to deal with one matrix representation. The proposed solution follows the row ordering encoding map from [6].

Thus, for a vector C of size d^2, and a ring R the encoding map $\xi : \mathrm{R}^{d^2} \rightarrow \mathrm{R}^{d \times d}$ is defined as:

$$\xi : \mathbf{C} \rightarrow \mathrm{C} = \{C_{d \cdot i + j}\}_{0 \leq i < d, 0 \leq j < d}$$

That is that, the vector **C** is the concatenation of the d row vectors of the $(d \times d)$ square matrix C.

In more details, the element $C_{i,j}$, for $0 \leq i < d$, $0 \leq j < d$,

appears in the $(i \cdot d + j)$ slot of **C**. The ciphertext that encrypts the plaintext $\mathbf{C} = \xi^{-1}(\mathrm{C})$, is referred to as the encryption of the message C. For simplicity, the matrix C (message), is denoted by **C** to refer to the corresponding plain-text/ciphertext.

Example 2. Let $d = 3$ and

let $\mathrm{C} = \begin{bmatrix} c_0 & c_1 & c_2 \\ c_3 & c_4 & c_5 \\ c_6 & c_7 & c_8 \end{bmatrix}$,

then the message C will be transformed into a vector **C** using $\xi^{-1}()$

as

$$\mathbf{C} = \boxed{c_0} \boxed{c_1} \boxed{c_2} \boxed{c_3} \boxed{c_4} \boxed{\cdots} \boxed{c_7} \boxed{c_8},$$

so the first row is mapped into the first 3 slots, the second row into the next 3 slots and so on.

Row and column masking

Next are defined two binary vectors Π and Ψ that are used to manage rows and columns of a matrix using scalar multiplication. More precisely, the binary vector $\Pi_{k1,k2}$ equals 1 in the slots with index of form $k_1 + l \cdot k_2$, for an integer l, while $\Psi_{k1,k2}$ equals 1 in the slots with index from k_1 to $k_1 + k_2 - 1$. Both vectors are zero in all the other slots. More formally,

$$\Pi_i^{(k_1,k_2)} = \begin{cases} 1, & \text{if } k_1 = i \mod k_2 \\ 0, & \text{otherwise} \end{cases}$$

and

$$\Psi_i^{(k_1,k_2)} = \begin{cases} 1, & \text{if } k_1 \le i < k_1 + k_2 \\ 0, & \text{otherwise} \end{cases}$$

Afterward, let **C** be the ciphertext of a $(d \times d)$ matrix C. Then, for $0 \le i < d$, the HE plaintext multiplication $\mathbf{L_i} = \mathbf{cMult(C, \Pi_{i,d})}$ returns a ciphertext $\mathbf{L_i}$ which has all slots equal to zero except the slots in positions $(i + l \cdot d)$ that are equal to the corresponding **C** slots. That is that, $\mathbf{L_i}$ is the encryption of the i-th column of C. Similarly, for $0 \le i < d$, $\mathbf{R_i} = \mathbf{cMult(C, \Psi_{i \cdot d,d})}$ returns an all zero ciphertext $\mathbf{R_i}$ except for the slots $[i \cdot d]$ to $[(i + 1) \cdot d - 1]$ that contain the i-th row of C.

Example 3. Following *Example* 2, the ciphertext $\mathbf{L_0} = \mathbf{cMult(C, \Pi_{0,3})}$ is given by:

$$\mathbf{L_0} = \boxed{c_0}\boxed{0}\boxed{0}\boxed{c_3}\boxed{0}\boxed{0}\boxed{c_6}\boxed{0}\boxed{0},$$

where

$$\mathbf{\Pi_{0,3}} = \boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}$$

Next, regarding the row masks, for $\mathbf{\Psi_{0,3}}$, we get $\mathbf{R_0} = \mathbf{cMult(C, \Psi_{0,3})}$

$$\mathbf{R_0} = \boxed{c_0}\boxed{c_1}\boxed{c_2}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0},$$

where

$$\mathbf{\Psi_{0,3}} = \boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}$$

Matrix $\hat{A}$ computation algorithm

Computing $\hat{A}$:

Input: matrix A of size $(d \times d)$.

Output: matrix $\hat{A}$ of size $(d^2 \times d)$.

Both matrices are encrypted homomorphically.

Algorithm 1. Computation of $\hat{A}$

```
1: procedure AlgoA(A, d)
2:     for i = 0 to d − 1 do                          ▷ First step
3:         L_i ← cMult(A, Π_{i,d})
4:     end for
5:     Â ← L_0                                        ▷ Second step
6:     for i = 1 to d − 1 do
7:         Â ← Add(Â, Rot(L_i, −i · (d^2 − 1)))
8:     end for
9:     for i = 0 to log_2(d) − 1 do                   ▷ Third step
10:        Â ← Add(Â, Rot(Â, −2^i))
11:    end for
12:    Return Â
13: end procedure
```

The detailed breakdown of this algorithm:

First Step: The d columns of A are copied to d different ciphertexts $\mathbf{L_i}$, where $0 \leq i < d$. Each $\mathbf{L_i}$ contains the i-th column of A with the same slot positions. This step is achieved by multiplying the ciphertext A by a mask $\mathbf{\Pi_{i,d}}$. The element $A_{i,j}$ appears in the slot $(i \cdot d + j)$ of A and will occupy all the slots $(j \cdot d^2 + i \cdot d + \delta)$ of $\hat{A}$, for $0 \leq \delta < d$.

Second Step: The ciphertexts $\mathbf{L_i}$ are used to compute the first column of $\hat{A}$. By rotating Li to the right by $(i \cdot d^2 - i)$ slots, the non-zero elements of $\mathbf{L_i}$ (the elements of the i-th column of A) will be moved to the slots $(i \cdot d^2 + j \cdot d)$ of $\mathbf{L_i}$, for $0 \leq j < d$. Adding all the rotated ciphertexts together results in the first column of $\hat{A}$.

Third Step: The first column of $\hat{A}$ is copied $(d - 1)$ times. This is achieved by rotating and adding the ciphertext $log_2(d)$ times.

This algorithm allows to construct the matrix $\hat{A}$, which can then be used to perform the square matrix multiplication homomorphically.

Matrix $\hat{A}$ computation example

Example 4. For a better data visualization, the message representation, i.e. the data after applying the $\xi^{-1}()$ transformation, and not the vector (plaintext encoding) representation is shown.

Let's assume that the matrix used as input in *Algorithm* 1 is:

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix}$$

After the *First Step*, the four matrices $\mathbf{L_i}$ are:

$$L_0 = \begin{bmatrix} a_0 & 0 & 0 & 0 \\ a_4 & 0 & 0 & 0 \\ a_8 & 0 & 0 & 0 \\ a_{12} & 0 & 0 & 0 \end{bmatrix}, L_1 = \begin{bmatrix} 0 & a_1 & 0 & 0 \\ 0 & a_5 & 0 & 0 \\ 0 & a_9 & 0 & 0 \\ 0 & a_{13} & 0 & 0 \end{bmatrix},$$

$$L_2 = \begin{bmatrix} 0 & 0 & a_2 & 0 \\ 0 & 0 & a_6 & 0 \\ 0 & 0 & a_{10} & 0 \\ 0 & 0 & a_{14} & 0 \end{bmatrix}, L_3 = \begin{bmatrix} 0 & 0 & 0 & a_3 \\ 0 & 0 & 0 & a_7 \\ 0 & 0 & 0 & a_{11} \\ 0 & 0 & 0 & a_{15} \end{bmatrix}$$

In the second step, the four matrices are rotated and added to produce the matrix, so the result after the *Second Step* can be represented by:

$$\hat{A} = \begin{bmatrix} a_0 & 0 & 0 & 0 \\ a_4 & 0 & 0 & 0 \\ a_8 & 0 & 0 & 0 \\ a_{12} & 0 & 0 & 0 \\ a_1 & 0 & 0 & 0 \\ a_5 & 0 & 0 & 0 \\ a_9 & 0 & 0 & 0 \\ a_{13} & 0 & 0 & 0 \\ a_2 & 0 & 0 & 0 \\ a_6 & 0 & 0 & 0 \\ a_{10} & 0 & 0 & 0 \\ a_{14} & 0 & 0 & 0 \\ a_3 & 0 & 0 & 0 \\ a_7 & 0 & 0 & 0 \\ a_{11} & 0 & 0 & 0 \\ a_{15} & 0 & 0 & 0 \end{bmatrix}$$

Finally, in the *Third Step*, $\hat{A}$ is rotated so that the first column becomes second in the rotated version and the two matrices are added. Then, it is rotated once more, and the first and second columns become third and fourth in the rotated version. By adding the two matrices, the final matrix is $\hat{A}$ obtained:

$$\hat{A} = \begin{bmatrix} a_0 & a_0 & a_0 & a_0 \\ a_4 & a_4 & a_4 & a_4 \\ a_8 & a_8 & a_8 & a_8 \\ a_{12} & a_{12} & a_{12} & a_{12} \\ a_1 & a_1 & a_1 & a_1 \\ a_5 & a_5 & a_5 & a_5 \\ a_9 & a_9 & a_9 & a_9 \\ a_{13} & a_{13} & a_{13} & a_{13} \\ a_2 & a_2 & a_2 & a_2 \\ a_6 & a_6 & a_6 & a_6 \\ a_{10} & a_{10} & a_{10} & a_{10} \\ a_{14} & a_{14} & a_{14} & a_{14} \\ a_3 & a_3 & a_3 & a_3 \\ a_7 & a_7 & a_7 & a_7 \\ a_{11} & a_{11} & a_{11} & a_{11} \\ a_{15} & a_{15} & a_{15} & a_{15} \end{bmatrix}$$

Matrix $\hat{B}$ Computation algorithm

Computing $\hat{B}$:

Input: matrix B of size $(d \times d)$.

Output: matrix $\hat{B}$ of size $(d^2 \times d)$.

Both matrices are also encrypted homomorphically.

Algorithm 2. Computation of $\hat{B}$

```
1:  procedure AlgoB(B, d)
2:      for i = 0 to d − 1 do                          ▷ First step
3:          R_i ← cMult(B, Ψ_(i·d,d))
4:      end for
5:      B̂ ← R_0                                        ▷ Second step
6:      for i = 1 to d − 1 do
7:          B̂ ← Add(B̂, Rot(R_i, −i · (d^2 − d))
8:      end for
9:      for i = 0 to log_2(d) − 1 do                   ▷ Third step
10:         B̂ ← Add(B̂, Rot(B̂, −d · 2^i)
11:     end for
12:     Return B̂
13: end procedure
```

The detailed breakdown of this algorithm:

First Step: The d rows of B are copied into d different ciphertexts $\mathbf{R_i}$, where $0 \leq i < d$. Each $\mathbf{R_i}$ contains the i-th row of B with the same slot positions. This step is accomplished by multiplying the ciphertext B by a mask $\Psi i \cdot d,d$. The element $B_{i,j}$ is in the slot $(i \cdot d + j)$ of B and will occupy the d slots of $\hat{B}$, $(i \cdot d^2 + j + \delta \cdot d)$, for $0 \leq \delta < d$.

Second Step: The ciphertexts $\mathbf{R_i}$ are then used to compute the first row of $\hat{B}$. By rotating $\mathbf{R_i}$ to the right by $[i \cdot (d^2 - d)]$ slots, the non-zero elements of the ciphertext (the elements of the i-th row of B) will appear in slots from $(i \cdot d^2)$ to $(i \cdot d^2 + d - 1)$, for $0 \leq i < d$. All the rotated ciphertexts are added together to form the first row of $\hat{B}$.

Third Step: The first row of $\hat{B}$ is then copied $d - 1$ times to fill in the rest of the rows of $\hat{B}$. This is achieved by rotating to the right and adding the ciphertext $log_2(d)$ times.

This algorithm enables the computation of the matrix $\hat{B}$, which can then be used in combination with the previously computed matrix $\hat{A}$ for homomorphic square matrix multiplication.

Matrix $\hat{B}$ computation example

Example 5. Similarly, the computation results for each step while the matrix $\hat{B}$ is obtained using the *Algorithm* 2 with the following input example can be represented as following:

$$B = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ b_8 & b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix}$$

After the *First Step*, the four matrices $\mathbf{R_i}$ are:

$$R_0 = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, R_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ b_4 & b_5 & b_6 & b_7 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$R_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ b_8 & b_9 & b_{10} & b_{11} \\ 0 & 0 & 0 & 0 \end{bmatrix}, R_3 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix}$$

During *Second Step*, after rotations, the four matrices are added, to produce the next result:

$$\hat{B} = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ b_4 & b_5 & b_6 & b_7 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ b_8 & b_9 & b_{10} & b_{11} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ b_{12} & b_{13} & b_{14} & b_{15} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Finally, on the *Third Step*, $\hat{B}$ is rotated, so that the first row becomes second, the fifth becomes sixth, the ninth becomes tenth and the fourteenth becomes sixteenth, in the rotated version. Then, the two matrices are added. At last, this matrix is rotated by two rows and added. The final matrix is now $\hat{B}$ composed:

$$\hat{B} = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_0 & b_1 & b_2 & b_3 \\ b_0 & b_1 & b_2 & b_3 \\ b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ b_4 & b_5 & b_6 & b_7 \\ b_4 & b_5 & b_6 & b_7 \\ b_4 & b_5 & b_6 & b_7 \\ b_8 & b_9 & b_{10} & b_{11} \\ b_8 & b_9 & b_{10} & b_{11} \\ b_8 & b_9 & b_{10} & b_{11} \\ b_8 & b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} \\ b_{12} & b_{13} & b_{14} & b_{15} \\ b_{12} & b_{13} & b_{14} & b_{15} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix}$$

Product Matrix $\hat{\mathbb{C}}$ *computation algorithm*

At last, the multiplication scheme is utilized to obtain the final result that represents the matrix product.

Computing $\hat{\mathbb{C}}$:

Input: matrix $\hat{A}$ of size $(d^2 \times d)$, matrix $\hat{B}$ of size $(d^2 \times d)$.

Output: matrix $\hat{\mathbb{C}} = \hat{A} \times \hat{B}$ of size $(d \times d)$.

Algorithm 3. Computation of $\hat{\mathbb{C}}$:

```
1: procedure AlgoC(Â, B̂, d)
2:     C ← Â ⊙ B̂                                  ▷ First step
3:     for i = 1 to log2(d) do                    ▷ Second step
4:         C ← Add(C, Rot(C, d³/2^i))
5:     end for
6:     Return C
7: end procedure
```

The final part of the proposed homomorphic matrix multiplication process involves computing the matrix C (or $\hat{\mathbb{C}}$) using the already computed matrices $\hat{A}$ and $\hat{B}$. From equation (2), the product C is the sum of d elementwise matrix multiplications such that $\hat{\mathbb{C}}_k = \hat{A}_k \odot \hat{B}_k$.

The intermediate matrix $\hat{\mathbb{C}}$ is formed as the product of the elementwise matrix multiplication of $\hat{A}$ and $\hat{B}$. It is computed as follows:

$$\hat{C} = \hat{A} \odot \hat{B} = \begin{bmatrix} \hat{A}_0 \odot \hat{B}_0 \\ \hat{A}_1 \odot \hat{B}_1 \\ \vdots \\ \hat{A}_{d-1} \odot \hat{B}_{d-1} \end{bmatrix}$$

The products $\hat{A}_k \odot \hat{B}_k$ are then added together to form the final matrix $\mathbb{C}$. This process involves rotating and adding the ciphertext $\hat{\mathbb{C}}$. The end result is the homomorphically encrypted matrix C that represents the product of the original input matrices A and B.

Product Matrix $\hat{\mathbb{C}}$ *computation example*

Example 6. The encrypted matrices $\hat{A}$ and $\hat{B}$ are used as input in *Algorithm* 3.

The First Step computes the following $\hat{C}$:

$$\hat{C} = \hat{A} \odot \hat{B} = \begin{bmatrix} a_0 \cdot b_0 & a_0 \cdot b_1 & a_0 \cdot b_2 & a_0 \cdot b_3 \\ a_4 \cdot b_0 & a_4 \cdot b_1 & a_4 \cdot b_2 & a_4 \cdot b_3 \\ a_8 \cdot b_0 & a_8 \cdot b_1 & a_8 \cdot b_2 & a_8 \cdot b_3 \\ a_{12} \cdot b_0 & a_{12} \cdot b_1 & a_{12} \cdot b_2 & a_{12} \cdot b_3 \\ a_1 \cdot b_4 & a_1 \cdot b_5 & a_1 \cdot b_6 & a_1 \cdot b_7 \\ a_5 \cdot b_4 & a_5 \cdot b_5 & a_5 \cdot b_6 & a_5 \cdot b_7 \\ a_9 \cdot b_4 & a_9 \cdot b_5 & a_9 \cdot b_6 & a_9 \cdot b_7 \\ a_{13} \cdot b_4 & a_{13} \cdot b_5 & a_{13} \cdot b_6 & a_{13} \cdot b_7 \\ a_2 \cdot b_8 & a_2 \cdot b_9 & a_2 \cdot b_{10} & a_2 \cdot b_{11} \\ a_6 \cdot b_8 & a_6 \cdot b_9 & a_6 \cdot b_{10} & a_6 \cdot b_{11} \\ a_{10} \cdot b_8 & a_{10} \cdot b_9 & a_{10} \cdot b_{10} & a_{10} \cdot b_{11} \\ a_{14} \cdot b_8 & a_{14} \cdot b_9 & a_{14} \cdot b_{10} & a_{14} \cdot b_{11} \\ a_3 \cdot b_{12} & a_3 \cdot b_{13} & a_3 \cdot b_{14} & a_3 \cdot b_{15} \\ a_7 \cdot b_{12} & a_7 \cdot b_{13} & a_7 \cdot b_{14} & a_7 \cdot b_{15} \\ a_{11} \cdot b_{12} & a_{11} \cdot b_{13} & a_{11} \cdot b_{14} & a_{11} \cdot b_{15} \\ a_{15} \cdot b_{12} & a_{15} \cdot b_{13} & a_{15} \cdot b_{14} & a_{15} \cdot b_{15} \end{bmatrix}$$

During the *Second Step*, two rotations and additions are performed. After the first rotation and addition, C appears in Table 2. A second rotation and addition compute the final C.

$$C = \begin{bmatrix} a_0 \cdot b_0 + a_2 \cdot b_8 & a_0 \cdot b_1 + a_2 \cdot b_9 & a_0 \cdot b_2 + a_2 \cdot b_{10} & a_0 \cdot b_3 + a_2 \cdot b_{11} \\ a_4 \cdot b_0 + a_6 \cdot b_8 & a_4 \cdot b_1 + a_6 \cdot b_9 & a_4 \cdot b_2 + a_6 \cdot b_{10} & a_4 \cdot b_3 + a_6 \cdot b_{11} \\ a_8 \cdot b_0 + a_{10} \cdot b_8 & a_8 \cdot b_1 + a_{10} \cdot b_9 & a_8 \cdot b_2 + a_{10} \cdot b_{10} & a_8 \cdot b_3 + a_{10} \cdot b_{11} \\ a_{12} \cdot b_0 + a_{14} \cdot b_8 & a_{12} \cdot b_1 + a_{14} \cdot b_9 & a_{12} \cdot b_2 + a_{14} \cdot b_{10} & a_{12} \cdot b_3 + a_{14} \cdot b_{11} \\ a_1 \cdot b_4 + a_3 \cdot b_{12} & a_1 \cdot b_5 + a_3 \cdot b_{13} & a_1 \cdot b_6 + a_3 \cdot b_{14} & a_1 \cdot b_7 + a_3 \cdot b_{15} \\ a_5 \cdot b_4 + a_7 \cdot b_{12} & a_5 \cdot b_5 + a_7 \cdot b_{13} & a_5 \cdot b_6 + a_7 \cdot b_{14} & a_5 \cdot b_7 + a_7 \cdot b_{15} \\ a_9 \cdot b_4 + a_{11} \cdot b_{12} & a_9 \cdot b_5 + a_{11} \cdot b_{13} & a_9 \cdot b_6 + a_{11} \cdot b_{14} & a_7 \cdot b_3 + a_{11} \cdot b_{15} \\ a_{13} \cdot b_4 + a_{15} \cdot b_{12} & a_{13} \cdot b_5 + a_{15} \cdot b_{13} & a_{13} \cdot b_6 + a_{15} \cdot b_{14} & a_{13} \cdot b_7 + a_{15} \cdot b_{15} \end{bmatrix}$$

Table 2 [1]: Intermediate value of matrix C in *Algorithm* 3 (following *Example* 6)

Complexity Analysis

The complexity analysis for the square matrix multiplication algorithm presented above is given in terms of HE operations needed and the multiplicative depth of the computation circuit.

The complexity of the *Algorithm* 1 (computation of $\hat{A}$) includes:

- d scalar multiplications in the *First Step*,
- $(d - 1)$ ciphertext additions and $d - 1$ rotations in *Second Step*,
- $log_2(d)$ rotations and ciphertext additions in the *Third Step*.

The complexity of the algorithm *Algorithm* 2 (computation of $\hat{B}$) includes:

- d scalar multiplications in the *First Step*,
- d – 1 ciphertext additions and d - 1 rotations in *Second Step*,
- $log_2(d)$ rotations and ciphertext additions in the *Third Step*.

It is essential to note that *Algorithms* 1 and 2 can be executed in parallel. The multiplicative depth for these steps is increased by 1 due to the scalar multiplication.

The complexity of the final *Algorithm* 3 (computation of $\hat{\mathrm{C}}$), requires:

- one ciphertext multiplication,
- $log_2(d)$ additions and rotations.

This step also increases the multiplicative depth by 1 due to the scalar multiplication.

The total number of the HE operations and the multiplicative depth are summarized in a Table 3 to give an overall view of the computational demands of the algorithm. This allows for comparison with other possible algorithms and helps in making informed decisions about the efficiency and feasibility of the approach.

Complexity summary

Algorithm	# operations	HE operation
$Algo$rithm_1$\hat{A}$ (A, d)	d $d + log_2(d) - 1$ $d + log_2(d) - 1$	cMult Rot Add
$Algo$rithm_2$\hat{B}$ (B, d)	d $d + log_2(d) - 1$ $d + log_2(d) - 1$	cMult Rot Add
$Algo$rithm_3$\hat{\mathrm{C}}$ (C, d)	$log_2(d)$ $log_2(d)$ 1	cMult Rot Add
Total HE operations	$2d$ $2d + 3\ log_2(d) - 2$ $2d + 3\ log_2(d) - 2$ 1	cMult Rot Add Mult
Multiplication Depth	1 1	cMult Mult

Table 3: Illustration of computation complexity. The total number of HE operations and the multiplicative depth per algorithm

Comparison with different approaches

***

Implementation and evaluation

***

Conclusion

The studied paper introduces a generic matrix computation algorithm for homomorphically encrypted data. The presented approach capitalizes on unoccupied ciphertext slots, a common occurrence due to varying data availability, allowing for increased computational efficiency through the use of a single ciphertext-ciphertext multiplication.

The computation model assumes an honest-but-curious service provider, a common assumption in many secure computation schemes. The techniques used for plaintext encoding are based on previous work, specifically referring to [6].

While the focus of this work has been on square matrices, an extension to rectangular matrices is under development. The proposed method can easily cover cases where the dimensions of the matrix are unequal ($d \neq d1$) with minor modifications.

It is worth noting that two scenarios are distinguished: when $d < d_2$ and when $d > d_2$. For the former, $d_2 - d$ zero columns are added to matrix A, and for the latter, $d - d_2$ zero columns are appended to matrix B. After these additions, the existing solution can be employed with slight adjustments. Initial results for this extension appear to be highly promising.

Criterial highlights of the proposed approach:

Efficiency: The proposed square matrix multiplication algorithm is efficient in terms of computational complexity. The number of critical HE operations such as ciphertext multiplications (cMult / Mult), rotations (Rot), and additions (Add) required is at most a function of d (the dimension of the input square matrix) therefore the computation complexity has an upper limited computation complexity of $O(d)$ regarding the algorithms 1 and 2 and $log_2(d)$ for the last product computation, which is reasonably modest. This makes the algorithm efficient for processing large datasets. The proposed approach is also compared with other contemporary and its result is more efficient, where the difference approaches a constant.

Paralleled Computations: The algorithms 1 and 2 can be executed in parallel, which suggests that the overall runtime can be significantly reduced if parallel computing resources are available.

Depth of Computations: The multiplicative depth of the computations is minimal. This indicates that the computations can be executed in a shallow circuit, which is beneficial in homomorphic encryption schemes where the depth of the circuit can affect the noise and, consequently, the correctness of the computations.

Flexibility: The approach appears to be flexible and can be adapted to various square matrix sizes as the computational complexity scales with the dimension d of the matrix.

In conclusion, given these characteristics, this square matrix multiplication algorithm is well-suited for encrypted computations. It can handle large amounts of data efficiently and correctly within a homomorphic encryption context. However, the practical performance will largely depend on the available computational resources and the specific HE scheme used.

Sources

[1] Panagiotis, R., Aikaterini, T. (2022). On Matrix Multiplication with Homomorphic Encryption. Proceedings of the 2022 on Cloud Computing Security Workshop, 53–61. https://doi.org/10.1145/3560810.3564267

[2] Regev, O. (2005). On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing, 84–93. https://doi.org/10.1145/1060590.1060603

[3] Nigel P. Smart, Vercauteren F. (2011). Fully Homomorphic SIMD Operations. IACR Cryptol. ePrint Arch., 133. http://eprint.iacr.org/2011/133

[4] Blatt M., Gusev A., Polyakov Y., Goldwasser S. (2020). Secure large-scale genome-wide association studies using homomorphic encryption. Proceedings of the National Academy of Sciences 117, 21, 11608–11613. https://doi.org/10.1073/pnas.1918257117

[5] OpenMined. (2020). CKKS explained: Part 1, Vanilla Encoding and Decoding. https://blog.openmined.org/ckks-explained-part-1-simple-encoding-and-decoding/

[6] Jiang X., Kim M., Kristin Lauter E., Song Y. (2018). Secure Outsourced Matrix Computation and Application to Neural Networks. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October, 15-19. https://doi.org/10.1145/3243734.3243837