The Journal of Supercomputing (2025) 81:1534
https://doi.org/10.1007/s11227-025-08032-w

New algorithms for fully homomorphic matrix addition and multiplication

Shang Ci[1] · Yihan Wang[1] · Sen Hu[1] · Donghai Guan[1] · Çetin Kaya Koç[1,2,3]

Received: 3 April 2025 / Accepted: 1 November 2025

Abstract
New algorithms are introduced for embedding matrices with integer or fixed-point real number entries into plaintexts and for performing matrix addition and multiplication on encrypted matrices. The encryption algorithms used for this purpose are fully homomorphic, such as BGV, BFV, and CKKS. These algorithms have the property of performing SIMD style parallel operations on plaintext values encrypted into a single ciphertext vector by a technique called "ciphertext packing" using the Chinese Remainder Theorem. This concept was introduced by Halevi and Shoup, and algorithms for homomorphic matrix operations were further improved by Jiang et al. (JKLS). Our algorithms improve both Halevi and Shoup and JKLS algorithms in terms of arithmetic and data rotation complexity. The proposed algorithms are designed on top of the BFV fully homomorphic encryption scheme. We describe our algorithms in detail step by step in this article, providing numerical examples in Appendix A. Experimental results demonstrate that our matrix multiplication algorithm achieves superior efficiency in terms of running time compared to JKLS algorithm. Real-world applications of FHE-based matrix computations often require matrix dimensions in the hundreds of thousands. Given typical FHE parameter settings (polynomial degree $N = 2^{13}$, plaintext modulus $t = 65537$), the total number of arithmetic and data-rotation operations can easily reach the order of 10^{12}. This scale necessitates state-of-the-art high-performance computing practices.

Keywords Matrix addition and multiplication · Homomorphic encryption

1 Introduction

This paper addresses secure outsourcing of matrix computations, which involves executing matrix operations (such as addition, multiplication, and inversion) on cloud platforms while preserving the confidentiality and integrity of the matrix

Yihan Wang, Sen Hu and Donghai Guan have contributed equally to this work.

Extended author information available on the last page of the article

Published online: 07 November 2025

data. This approach is becoming increasingly relevant due to the rising demand for secure data processing across a variety of applications, including cloud computing in finance, healthcare, machine learning, data analytics, and bioinformatics.

The concept of secure outsourcing of computation was first explored by Atallah, Pantazopoulos, Rice, and Spafford in their seminal paper [1, 2]. The security is provided by *disguising* or *obscuring* the data using various ad hoc methods, rather than encryption. They proposed several methods, for example, disguising matrices by multiplying them with 0–1 permutation matrices whose inverses are readily available. The data owner then sends the disguised matrices to the cloud server, which performs matrix additions or multiplications using $O(k^3)$ operations for matrices of dimension k. Disguising and un-disguising the matrices are performed by the data owner in a local server, with only $O(k^2)$ operations. This computational efficiency is the motivation behind such techniques. However, the security of this approach is admittedly limited, as the disguised operands can often be recovered either completely or approximately [1, 2]. To enhance security, various other disguising methods incorporating randomization with secret parameters have been proposed by subsequent researchers, including Seitkulov [3], as well as several others referenced therein.

The most secure outsourcing algorithms are based on Homomorphic Encryption (HE), which enables arithmetic operations to be performed directly on encrypted data. Partially Homomorphic Encryption (PHE) schemes [4] support either addition or multiplication, but not both, and are therefore unsuitable for matrix multiplication, which requires both operations. In contrast, Fully Homomorphic Encryption (FHE) supports both addition and multiplication, as well as general algebraic computations through techniques such as Taylor expansions. Consequently, FHE is a natural candidate for privacy-preserving outsourced computation. Since Gentry's breakthrough in 2009 [5], FHE has attracted worldwide attention, leading to significant advances in both the efficiency of FHE schemes and their applications across various practical domains.

Despite its theoretical potential, FHE encounters significant obstacles, including performance limitations and increased complexity. The complexity of homomorphic operations is significant mostly because of the size of operands in FHE plaintexts, ciphertexts, and keys. Consider the BFV algorithm, which is based on the ring of polynomials with integer coefficients mod $x^n + 1$, that is, $\mathcal{R} = \mathcal{Z}[x]/(x^n + 1)$. Plaintexts and ciphertexts are polynomials whose coefficients are reduced mod q for a selected modulus q, that is, they belong to $\mathcal{R}_q = \mathcal{Z}_q[x]/(x^n + 1)$. Many cryptographic frameworks suggest aiming for security equivalent to at least 128-bit security against classical attacks and 64-bit security against quantum attacks, as discussed in [6]. To reach this level of security or beyond, the modulus q is selected to be at least 512 bits, up to 2048 bits. Moreover, the size of polynomials denoted as n is selected between 16, 384 and 65, 536. A single ciphertext requires two polynomials from $\mathcal{Z}_q[x]/(x^n + 1)$; thus, the FHE operands are indeed very large, just copying them between memory locations requires a significant amount of time and energy.

The complexity of homomorphic matrix operations increases significantly with the matrix dimension and the number of elements involved. For example, a

100×100 matrix has 10, 000 entries, and thus **an element-wise encrypted matrix** would require 10, 000 times more memory than a plaintext matrix. It is evident that more innovative approaches are required to achieve progress in homomorphic matrix computations. While there are good algorithms for performing a single homomorphic addition and multiplication, **parallel homomorphic operations** with multiple ciphertexts would be more effective as pointed out by several researchers [7–9].

In 2010, Smart and Vercauteren introduced a variant of Gentry's fully homomorphic encryption algorithm that supports SIMD (single-instruction multiple-data) style parallelism [7]. Their algorithm is based on the algebra $\mathcal{F}_2[x]/F(x)$ such that $F(x)$ splits as $\prod_{i=1}^{t} f_i \pmod 2$ with coprime f_i and $\deg(f_i) = d_i$. By using the CRT (Chinese Remainder Theorem), one obtains

$$\mathcal{F}_2[x]/F(x) \equiv \mathcal{F}_{2^{d_1}} \times \mathcal{F}_{2^{d_2}} \times \cdots \times \mathcal{F}_{2^{d_t}} \ . \tag{1}$$

By carefully selecting $F(x)$, one obtains SIMD style homomorphic encryption in multiple finite fields of characteristic 2 at the same time. Gentry and Halevi [10] optimized the Smart and Vercauteren algorithm; however, their scheme does not support SIMD style computation. A subsequent work by Smart and Vercauteren [8] fixes their algorithm so that it supports SIMD style operations on non-trivial finite fields of characteristic 2.

A significant advancement in the direction of parallel homomorphic computations with implications in linear algebra and matrix computation came from Halevi and Shoup [9]. They introduced algorithms for the HElib [11, 12] software library, which implements the fully homomorphic encryption algorithm BGV (Brakerski–Gentry–Vaikuntanathan) [13, 14]. A list of plaintext values can be packed (encrypted) into a single ciphertext vector by a technique called **ciphertext packing** using the CRT [15]. Homomorphic operations are performed on these vectors, component-wise in an SIMD fashion. Thus, SIMD homomorphic operations act element-wise between the rows or columns of encrypted matrices, bringing speed-ups of several orders of magnitude.

Ciphertext packing and SIMD computation have opened up new avenues in homomorphic matrix computations, enabling a broad spectrum of applications across various fields. For example, packing techniques were used to accelerate encrypted statistical computations in [16–18]. Moreover, matrices are integral to a wide array of applications, including finance, data analytics, and machine learning. As a result, homomorphic matrix computations are set to be extensively applied in contexts where privacy is critically important.

2 Homomorphic matrix computations

The relevant notations used in this paper are shown in Table 1. Consider the Naive method of representing a $k \times k$ matrix using k^2 distinct ciphertexts, that is, every entry of the matrix is independently encrypted (without ciphertext packing). In order to multiply two matrices, homomorphic additions and multiplications of the

Table 1 Notations

Symbols	Definition
k, n	The dimension of the square matrix, and $n = k^2$
$A_{k\times l}$, $B_{l\times m}$	Matrices A and B of dimension $k \times l$ and $l \times m$
$\boldsymbol{v}$	Vector $\boldsymbol{v}$, bold indicate a vector
Mult	Homomorphic multiplication of two ciphertexts
CMult	Homomorphic multiplication of a ciphertext by a constant
$[x]y$	x mod y
$\mathcal{R}$	Ring of polynomials
t	Plaintext polynomials coefficients module
q	Ciphertext polynomials coefficients module
p	Random integer
$\boldsymbol{sk}$	Secret key
$\boldsymbol{pk}$	Public key
$\boldsymbol{rk}$	Relinearization key
$\boldsymbol{w}$	Public key component
$\boldsymbol{e}$	Error vector
$\boldsymbol{r}$	Random vector
Δ	Scaling factor
$\boldsymbol{m}$	Message vector
$\boldsymbol{c}$	Ciphertext vector
$\oplus$	Element-wise addition
$\odot$	Element-wise multiplication
$\mathrm{Enc}_{\boldsymbol{pk}}(\boldsymbol{m})$	**Encryption function**
$\mathrm{Dec}_{\boldsymbol{sk}}(\boldsymbol{c})$	**Decryption function**
$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{d}$	Ciphertext vectors in column order

individual ciphertexts are performed according to the standard matrix multiplication algorithm. The total number of ciphertext multiplications is $O(k^3)$, without employing any parallelism. However, the Naive method is already computationally expensive due to the large number of ciphertexts that must be handled.

On the other hand, the Halevi and Shoup algorithm [9] takes the matrix elements as row, column and diagonal vectors and uses **a single ciphertext to represent each vector**. Therefore, the matrix A is represented using k ciphertexts. In order to perform the matrix–vector multiplication $\boldsymbol{\theta} = A \cdot \boldsymbol{v}$, the algorithm performs $O(k)$ ciphertext multiplications and data rotations (Figs. 1, 2).

Using the Halevi and Shoup algorithm, the matrix–vector multiplication requires $k + 1$ ciphertexts and achieves a computational complexity of $O(k)$ with a multiplicative depth of $O(1)$. We can extend the Halevi and Shoup algorithm to the matrix–matrix multiplication by applying the matrix–vector multiplication with the columns of the second matrix k times, obtaining the number of homomorphic multiplications and additions as $O(k^2)$.

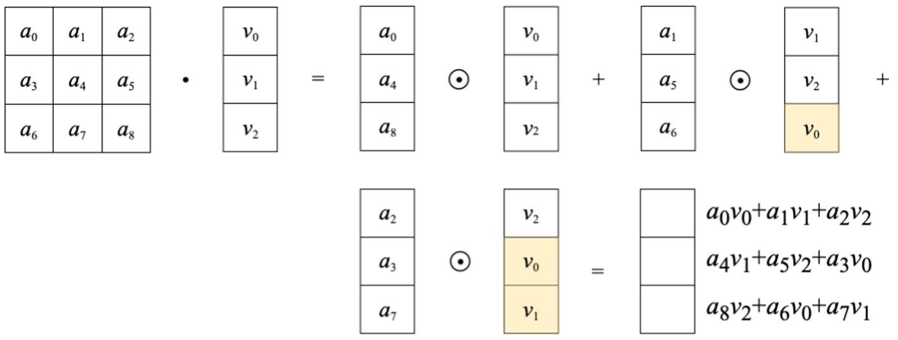

Fig. 1 Halevi and Shoup's Algorithm [9] illustrated for $k = 3$

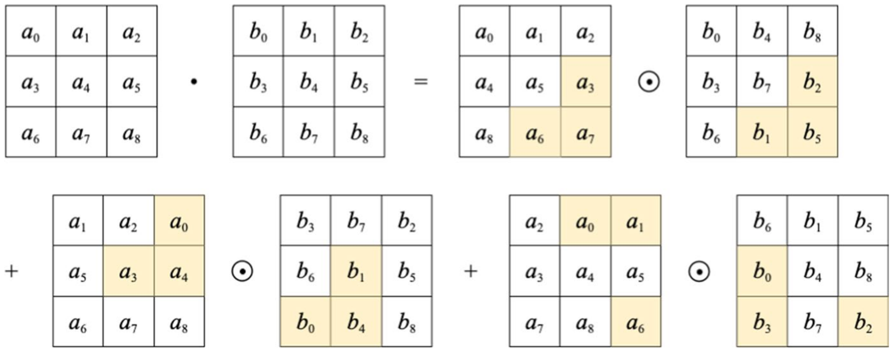
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

Fig. 2 JKLS Algorithm [19] illustrated for $k = 3$

A more efficient algorithm was introduced by Jiang et al. in [19]. Their algorithm encodes an arbitrary square matrix of size $k \times k$ as a vector of dimension $n = k^2$ having the same entries. Let $\odot$ represent component-wise multiplication of two matrices. The multiplication of two matrices A, B can be written as $A \cdot B = \sum_{i=0}^{k} A_i \odot B_i$ where the matrices A_i, B_i are obtained by taking specific transformations of the matrices A and B. The first two matrices A_0, B_0 require $O(k)$ rotations from the initial matrices A, B; however, subsequent matrices A_i, B_i for $i \geq 1$ are computed using only $O(1)$ permutations.

A brief complexity analysis of the three methods are summarized in Table 2. The arithmetic complexity and the depth of the circuit (total parallel time) are closely related to the representation of the encrypted matrices, that is, how the ciphertext packing is performed. In Table 2, Mult denotes homomorphic multiplication of two ciphertexts, while CMult refers homomorphic multiplication of a ciphertext by a constant.

The review of the recent algorithms [19–26] for homomorphic matrix multiplication highlights several important properties.

Table 2 Comparing the Naive, Halevi and Shoup [9], and JKLS [19] algorithms for homomorphic multiplication of $k \times k$ matrices

	Naive	Halevi & Shoup	JKLS
Condition	–	$k^2 < n$	$k^2 < n$
Input ordering	–	Diagonal	Row
Output ordering	–	Diagonal	Row
Ciphertexts	$2k^2$	$2k$	1
Additions	$k^2(k-1)$	$k(k-1)$	$6k$
Multiplications	k^3	k^2	k
Data rotations	0	k	$3k + 5\sqrt{k}$
Arithmetic depth	1 Mult	1 Mult	1 Mult + 2 CMult

k denotes the dimension of square matrices and $n = k^2$. Mult is homomorphic multiplication of two ciphertexts, while CMult is homomorphic multiplication of a ciphertext by a constant

1. Since the ciphertext packing often requires changes in the original order of the input matrix elements, there are costs associated with changing the locations of the matrix elements between row, column and generalized diagonal formats. Thus, these rotation costs must be taken into account. Researchers have considered storing the matrix elements or submatrices in hypercubes [9, 18, 23] or similar architectures in order to minimize communication costs.
2. If the input matrices are taken in a particular order for a given algorithm, the resulting encrypted matrix after the multiplication must also be brought to that particular ordering if subsequent multiplications are needed. This is particularly important when the product of a series of matrices is to be computed, as addressed in [20, 21]. Therefore, algorithms that produce the output matrices in the same format as the input matrices would be desirable. This property is called "composability" by Huang and Zong in their paper [23].
3. A common formulation underlying many recent algorithms that leverage the SIMD property and ciphertext packing for homomorphic matrix multiplication is surprisingly simple and is based on the JKLS algorithm [19]. In this formulation, the multiplication of two matrices A, B is expressed as

$$A \cdot B = \sum_{i=0}^{k} A_i \odot B_i, \tag{2}$$

where the matrices A_i, B_i are obtained by taking **specific transformations** of the matrices A and B. For example, the algorithm of Gao et al. [24] considers the multiplication of the rectangular matrices A and B of dimension $k \times l$ and $l \times m$ to obtain the matrix C of dimension $k \times m$ as $C_{k\times m} = A_{k\times l} \times B_{l\times m}$. This is accomplished by computing $C_{k\times m} = \sum_{i=0}^{k} A_i \odot B_i$ using 4 different **transformations** $(\sigma, \tau, \epsilon^k, \omega^k)$ of A and B as follows

$$\begin{aligned}\sigma(A)_{i,j} &= A_{i,[i+j]l} \text{ for } 0 \le i < k \text{ and } 0 \le j < l,\\ \tau(B)_{i,j} &= B_{[i+j]l,j} \text{ for } 0 \le i < l \text{ and } 0 \le j < m,\\ \epsilon^k(A)_{i,j} &= A_{i,[j+k]l} \text{ for } 0 \le i < k \text{ and } 0 \le j < m,\\ \omega^k(B)_{i,j} &= B_{[i+k]l,j} \text{ for } 0 \le i < k \text{ and } 0 \le j < m,\end{aligned} \tag{3}$$

where $[x]y$ denotes $x \bmod y$. The final formula for computing the product matrix is given as

$$A_{k\times l} \times B_{l\times m} = \sum_{n=0}^{l-1} \epsilon^k(\sigma(A)) \odot \omega^k(\tau(B)). \tag{4}$$

These transformations and the final computation of the product matrix are illustrated in Figures 1 and 2 of Gao et al. [24].

4. One final but important issue is the need for algorithms that can efficiently multiply rectangular matrices, rather than being limited to square matrices, as is often the case in scientific computing. Addressing this challenge, our paper extends existing algorithms for square matrices and introduces new methods that support rectangular matrix multiplication without incurring excessive computational costs.

3 Our contributions

In this paper, we introduce new homomorphic matrix addition and multiplication algorithms that utilize BFV homomorphic encryption method [27]. Our formulation differs from the JKLS algorithm [19], which has subsequently been adopted and extended by other researchers, including Gao et al. [24], as discussed in Sect. 2.

BFV algorithm is based on a ring of polynomials with integer coefficients mod $x^n + 1$. We describe the algorithmic details of secret and public key generation, homomorphic addition and multiplication operations. Most importantly, it is shown that the CRT algorithm allows multiple homomorphic addition and multiplication operations to be performed by ciphertext packing, thereby enabling SIMD parallelism. This method establishes the algorithmic foundations of homomorphic matrix addition and multiplication operations. We describe the homomorphic matrix algorithms in detail, giving numerical examples and performance analyses.

We conducted an experimental evaluation to provide a quantitative comparison between our algorithm, JKLS algorithm [19], and Zheng et al.'s (ZLW) algorithm [26]. Compared with the JKLS algorithm, although our method incurs slightly higher encryption time and memory usage, it achieves a significant reduction in matrix multiplication time, which is the most computationally intensive step. The decryption time remains comparable. In comparison with the ZLW algorithm, our approach demonstrates better runtime performance on small-dimensional matrix multiplications. These results indicate that our algorithm delivers superior performance compared to previous works.

Our contributions can be formally summarized as follows:

- Introduce a homomorphic matrix addition algorithm `BFVMatrixAdd` for matrices $A \in \mathbb{Z}^{k \times k}, B \in \mathbb{Z}^{k \times k}$, computing $D = A + B$ directly in encrypted form.
- Develop a homomorphic matrix multiplication algorithm `BFVMatrixMultiply` for $A \in \mathbb{Z}^{k \times k}, B \in \mathbb{Z}^{k \times k}$ computing $D = A \cdot B$ in encrypted form by extending JKLS algorithm.
- Leverage CRT-based ciphertext packing to enable SIMD parallelism, reducing the multiplication runtime from T_{JKLS} to T_{ours}, where $T_{\text{ours}} < T_{\text{JKLS}}$.
- Provide quantitative evaluation: encryption time $E_{\text{ZLW}} > E_{\text{ours}} > E_{\text{JKLS}}$, decryption time $D_{\text{ZLW}} > D_{\text{ours}} \approx D_{\text{JKLS}}$ and memory usage $M_{\text{ours}} = M_{\text{ZLW}} > M_{\text{JKLS}}$.
- We present that matrix multiplication based on the BFV framework has a computational complexity of $O(k)$ and provide precise formulas for homomorphic operations.

4 BFV algorithm

BFV algorithm is based on a ring of polynomials with integer coefficients mod $x^n + 1$, that is $\mathcal{R} = \mathcal{Z}[x]/(x^n + 1)$. Plaintexts are polynomials, whose coefficients are reduced mod t, that is, they belong to $\mathcal{R}_t = \mathcal{Z}_t[x]/(x^n + 1)$. On the other hand, ciphertexts are pairs of polynomials whose coefficients are reduced mod q, that is, they belong to $\mathcal{R}_q^2 = (\mathcal{Z}_q[x]/(x^n + 1))^2$. For example, if the ring is $\mathcal{R} = \mathcal{Z}[x]/(x^8 + 1)$, both the plaintext and ciphertext polynomials are of length $n = 8$. If the ring is $\mathcal{Z}_5[x]/(x^8 + 1)$, the polynomial coefficients are in $\mathcal{Z}_5$, expressed in the least magnitude as $\{-2, -1, 0, 1, 2\}$.

The addition and multiplication operations in the ring $\mathcal{Z}_q[x]/(x^n + 1)$ are performed by adding and multiplying the input polynomials mod $x^n + 1$ and their coefficients mod q. The addition of two polynomials does not increase the power of the sum polynomial and therefore reduction mod $x^n + 1$ is not needed, only the coefficients are reduced mod q. However, the multiplication of two polynomials produces a product polynomial whose largest power exceeds n; therefore, polynomial reduction mod $x^n + 1$ and coefficient reduction mod q are required. The addition operation $c = a + b$ in the ring is performed as

$$\begin{aligned} a =& a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}, \\ b =& b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}, \\ c =& c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1}, \\ c_i =& a_i + b_i \pmod q \text{ for } i = 0, 1, \ldots, n-1. \end{aligned} \tag{5}$$

Therefore, the ring elements can also be represented as row vectors of integers mod q, and added element-wise mod q.

$$\begin{aligned}
\boldsymbol{a} =&[a_0, a_1, a_2, \dots, a_{n-1}],\\
\boldsymbol{b} =&[b_0, b_1, b_2, \dots, b_{n-1}],\\
\boldsymbol{c} =&[c_0, c_1, c_2, \dots, c_{n-1}],\\
c_i =&a_i + b_i \pmod{q} \text{ for } i = 0, 1, \dots, n-1.
\end{aligned} \tag{6}$$

On the other hand, the multiplication operation $c = a \cdot b$ produces a product polynomial whose largest power may be larger than n,

$$\begin{aligned}
c =&(a_0 \cdot b_0) + (a_0 \cdot b_1 + a_1 \cdot b_0)x + (a_0 \cdot b_1 + a_1 \cdot b_1 + a_2 \cdot b_0)x^2\\
&+ \cdots + (a_0 \cdot b_{n-1} + a_1 \cdot b_{n-2} + \cdots + a_{n-1} \cdot b_0)x^{n-1}\\
&+ \cdots + (a_{n-2} \cdot b_{n-1} + a_{n-1} \cdot b_{n-2})x^{2n-3} + (a_{n-1} \cdot b_{n-1})x^{2n-2}.
\end{aligned} \tag{7}$$

The complexity of the multiplication is $O(n^2)$, and furthermore, the product polynomial is reduced mod $x^n + 1$ and the coefficients are reduced mod q. However, a special property of the modulus $x^n + 1$ allows us to employ the Chinese Remainder Theorem (CRT) to represent the ring elements as vectors and reduce the multiplication complexity. Let ζ be $2n$-th primitive root of unity mod q, that is, $\zeta^{2n} = 1 \pmod{q}$ and $\zeta^i \neq 1 \pmod{q}$ for $1 \leq i \leq 2n-1$. The existence of such a root requires that $q \equiv 1 \pmod{2n}$, i.e., $2n$ must divide $q-1$. The polynomial $x^n + 1$ can be factored to polynomials of degree 1, which are of the form $x - \zeta^i$ for $i = 1, 3, 5, \dots, 2n-1$ as follows:

$$x^n + 1 = (x-\zeta)(x-\zeta^3)(x-\zeta^5)\cdots(x-\zeta^{2n-1}) \pmod{q}. \tag{8}$$

Thus, the reduction of a polynomial mod $x^n + 1$ is equivalent to the reduction with respect to mod $(x - \zeta^i)$ for $i = 1, 3, 5, \dots, 2n-1$. The reduction $a(x)$ mod $(x - \zeta^i)$ is easily obtained by evaluating the polynomial $a(x)$ at $x = \zeta^i$, that is, $a(x)$ mod $(x - \zeta^i) = a(\zeta^i)$.

The ring element $a(x)$ can be represented as a vector consisting of its remainders mod $(x - \zeta^i)$, which correspond to the values of $a(x)$ at $x = \zeta^i \pmod{q}$.

$$\begin{aligned}
a(x) =&a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1}x^{n-1} \pmod{x^n + 1},\\
\boldsymbol{a} =&[a(\zeta^1), a(\zeta^3), \dots, a(\zeta^{2n-1})] \pmod{q}.
\end{aligned} \tag{9}$$

The CRT representation allows us to element-wise add and multiply vectors representing the ring elements,

$$\begin{aligned}
\boldsymbol{a} =&[a(\zeta^1), a(\zeta^3), \dots, a(\zeta^{2n-1})] \pmod{q},\\
\boldsymbol{b} =&[b(\zeta^1), b(\zeta^3), \dots, b(\zeta^{2n-1})] \pmod{q},\\
\boldsymbol{a} + \boldsymbol{b} =&[a(\zeta^1) + b(\zeta^1), a(\zeta^3) + b(\zeta^3), \dots, a(\zeta^{2n-1}) + b(\zeta^{2n-1})] \pmod{q},\\
\boldsymbol{a} \cdot \boldsymbol{b} =&[a(\zeta^1) \cdot b(\zeta^1), a(\zeta^3) \cdot b(\zeta^3), \dots, a(\zeta^{2n-1}) \cdot b(\zeta^{2n-1})] \pmod{q}.
\end{aligned} \tag{10}$$

There is a one-to-one relationship between the coefficients of the polynomial representation and the vector CRT representation of a ring element,

$$
\begin{aligned}
a(x) =& a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}, \\
a(\zeta^1) =& a_0 + a_1 \zeta^{1\cdot 1} + a_2 \zeta^{1\cdot 2} + \cdots + a_{n-1} \zeta^{1\cdot(n-1)} \pmod{q}, \\
a(\zeta^3) =& a_0 + a_1 \zeta^{3\cdot 1} + a_2 \zeta^{3\cdot 2} + \cdots + a_{n-1} \zeta^{3\cdot(n-1)} \pmod{q}, \\
a(\zeta^5) =& a_0 + a_1 \zeta^{5\cdot 1} + a_2 \zeta^{5\cdot 2} + \cdots + a_{n-1} \zeta^{5\cdot(n-1)} \pmod{q}, \\
&\cdots \\
a(\zeta^{2n-1}) =& a_0 + a_1 \zeta^{(2n-1)\cdot 1} + a_2 \zeta^{(2n-1)\cdot 2} + \cdot + a_{n-1} \zeta^{(2n-1)\cdot(n-1)} \pmod{q}.
\end{aligned} \tag{11}
$$

The relationship between the coefficient of the matrix representation and evaluation points of the polynomial at the roots of unity can be expressed as a matrix vector product.

$$
\begin{bmatrix} a(\zeta^1) \\ a(\zeta^3) \\ a(\zeta^5) \\ \vdots \\ a(\zeta^{2n-1}) \end{bmatrix} = \begin{bmatrix} \zeta^{1\cdot 0} & \zeta^{1\cdot 1} & \zeta^{1\cdot 2} & \cdots & \zeta^{1\cdot(n-1)} \\ \zeta^{3\cdot 0} & \zeta^{3\cdot 1} & \zeta^{3\cdot 2} & \cdots & \zeta^{3\cdot(n-1)} \\ \zeta^{5\cdot 0} & \zeta^{5\cdot 1} & \zeta^{5\cdot 2} & \cdots & \zeta^{5\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \zeta^{(2n-1)\cdot 0} & \zeta^{(2n-1)\cdot 1} & \zeta^{(2n-1)\cdot 2} & \cdots & \zeta^{(2n-1)\cdot(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}. \tag{12}
$$

Whenever necessary, the above matrix equation can be used to switch between the polynomial and CRT vector representations. For the rest of the paper, the CRT vector representation is adopted for all elements of the ring $\mathcal{R}_q = \mathcal{Z}_q[x]/(x^n + 1)$. All variables, including plaintexts, ciphertexts, and keys, are treated as integer vectors of dimension n mod q, that is, $\mathcal{R}_q \equiv \mathcal{Z}_q^n$.

4.1 BFV key generation

BFV algorithm uses random elements, which are sampled from particular random distributions. The secret key $\boldsymbol{sk} \leftarrow \mathcal{HW}(h)$ is a vector of dimension n and Hamming weight h, with entries from $\{0, +1, -1\}$. Here, $\mathcal{HW}(\cdot)$ is Hamming weight function. For a set $h \in \{0, 1, -1\}^n$, $\mathcal{HW}(h) = \sum_{i=1}^{n} h_i$, i.e., the number of nonzero entries in h. The public key component $\boldsymbol{w} \leftarrow \mathcal{U}(\mathcal{R}_q)$ is a vector of dimension n with entries mod q in the range $(-q/2, q/2)$. Here, $\mathcal{U}(\cdot)$ is uniform distribution. $\mathcal{U}(\chi)$ denotes a random variable sampled uniformly from the set χ. For example, $\mathcal{U}(\{0, 1\}^n)$ represents a uniformly random binary vector of length n. The error vector $\boldsymbol{e} \leftarrow \mathcal{DG}(\sigma^2)$ is of dimension n with entries in $\mathcal{Z}_q$. Here, $\mathcal{DG}(\cdot)$ is discrete Gaussian distribution. $\mathcal{DG}(\sigma)$ denotes a discrete Gaussian distribution over $\mathbb{Z}$ with standard deviation σ, i.e., $\Pr[x] \propto \exp\left(-\frac{x^2}{2\sigma^2}\right), x \in \mathcal{Z}$.

The **secret key** is a vector of dimension n denoted as $\boldsymbol{sk} \in \mathcal{R}_q$. The polynomial $\boldsymbol{u}$ is computed using the random polynomials

$$
\boldsymbol{u} = -(\boldsymbol{w} \cdot \boldsymbol{sk} + \boldsymbol{e}) \pmod{q}. \tag{13}
$$

The **public key** is defined as a pair: $\boldsymbol{pk} = (\boldsymbol{u}, \boldsymbol{w}) \in \mathcal{R}_q^2$. Also sample $\boldsymbol{w}' \leftarrow \mathcal{U}(\mathcal{R}_{p\cdot q})$ and $\boldsymbol{e}' \leftarrow \mathcal{DG}(\sigma^2)$, and compute

$$\boldsymbol{u}' = -(\boldsymbol{w}' \cdot \boldsymbol{sk} + \boldsymbol{e}') + p \cdot \boldsymbol{sk}^2 \pmod{p \cdot q}. \tag{14}$$

Here, p is a random integer used for computing the **relinearization** key, which is equal to $\boldsymbol{rk} = (\boldsymbol{u}', \boldsymbol{w}') \in \mathcal{R}_q^2$

4.2 BFV encryption and decryption

We select $\boldsymbol{r} \leftarrow \mathcal{ZO}(0.5)$ and $\boldsymbol{e}_0, \boldsymbol{e}_1 \leftarrow \mathcal{DG}(\sigma^2)$. The scaling factor is defined as $\Delta = \lfloor q/t \rfloor$. Given the plaintext $\boldsymbol{m} \in \mathcal{R}_t$, the **encryption function** $\mathrm{Enc}_{\boldsymbol{pk}}(\boldsymbol{m})$ computes the ciphertext $\boldsymbol{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_q^2$ from the message $\boldsymbol{m}$ using the public key $\boldsymbol{pk}$ as

$$\begin{aligned}
\boldsymbol{c} =& \boldsymbol{r} \cdot \boldsymbol{pk} + (\boldsymbol{m} \cdot \Delta + \boldsymbol{e}_0, \boldsymbol{e}_1) \pmod{q} \\
=& \boldsymbol{r} \cdot (\boldsymbol{u}, \boldsymbol{w}) + (\boldsymbol{m} \cdot \Delta + \boldsymbol{e}_0, \boldsymbol{e}_1) \pmod{q} \\
=& (\boldsymbol{r} \cdot \boldsymbol{u} + \boldsymbol{m} \cdot \Delta + \boldsymbol{e}_0, \boldsymbol{r} \cdot \boldsymbol{w} + \boldsymbol{e}_1) \pmod{q} \\
=& (\boldsymbol{c}_0, \boldsymbol{c}_1), \\
\boldsymbol{c}_0 =& \boldsymbol{r} \cdot \boldsymbol{u} + \boldsymbol{m} \cdot \Delta + \boldsymbol{e}_0, \\
\boldsymbol{c}_1 =& \boldsymbol{r} \cdot \boldsymbol{w} + \boldsymbol{e}_1.
\end{aligned} \tag{15}$$

The **decryption function** $\mathrm{Dec}_{\boldsymbol{sk}}(\boldsymbol{c})$ computes the plaintext polynomial m' from the ciphertext $\boldsymbol{c}$ using the secret key $\boldsymbol{sk}$ as

$$\boldsymbol{m}' = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{sk} \pmod{q}. \tag{16}$$

The approximate equality can be verified as

$$\begin{aligned}
\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{sk} =& (\boldsymbol{r} \cdot \boldsymbol{u} + \boldsymbol{m} \cdot \Delta + \boldsymbol{e}_0) + (\boldsymbol{r} \cdot \boldsymbol{w} + \boldsymbol{e}_1) \cdot \boldsymbol{sk} \pmod{q} \\
=& \boldsymbol{r} \cdot (-\boldsymbol{w} \cdot \boldsymbol{sk} - \boldsymbol{e}) + \boldsymbol{m} \cdot \Delta + \boldsymbol{e}_0 + \boldsymbol{r} \cdot \boldsymbol{w} \cdot \boldsymbol{sk} + \boldsymbol{e}_1 \cdot \boldsymbol{sk} \pmod{q} \\
=& \boldsymbol{m} \cdot \Delta - \boldsymbol{r} \cdot \boldsymbol{e} + \boldsymbol{e}_0 + \boldsymbol{e}_1 \cdot \boldsymbol{sk} \pmod{q} \\
\approx& \boldsymbol{m} \cdot \Delta \ = \ \boldsymbol{m}'.
\end{aligned} \tag{17}$$

The final step of the decryption function computes the plaintext

$$\boldsymbol{m} = \left\lfloor \frac{1}{\Delta} \cdot \boldsymbol{m}' \right\rceil \pmod{t}. \tag{18}$$

4.3 Homomorphic addition algorithm

The homomorphic addition of two ciphertexts $\boldsymbol{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ and $\boldsymbol{c}' = (\boldsymbol{c}_0', \boldsymbol{c}_1')$ is performed using

$$\begin{aligned}
\boldsymbol{c}_{add} =& (\boldsymbol{c}_0, \boldsymbol{c}_1) \oplus (\boldsymbol{c}_0', \boldsymbol{c}_1'), \\
(\boldsymbol{d}_0, \boldsymbol{d}_1) =& (\boldsymbol{c}_0 + \boldsymbol{c}_0', \boldsymbol{c}_1 + \boldsymbol{c}_1').
\end{aligned} \tag{19}$$

The input values $\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}'_0, \boldsymbol{c}'_1$ and the output values $\boldsymbol{d}_0, \boldsymbol{d}_1$ are the elements of the ring $\mathcal{R}_q$ represented as CRT vectors. The arithmetic is performed in the CRT ring $\mathcal{Z}_q^n$. The ring elements are expressed as row vectors and added element-wise.

4.4 Homomorphic multiplication algorithm

The homomorphic multiplication of two ciphertexts $\boldsymbol{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ and $\boldsymbol{c}' = (\boldsymbol{c}'_0, \boldsymbol{c}'_1)$ is computed as

$$\begin{aligned}
\boldsymbol{c}_{mul} &= \boldsymbol{c} \odot \boldsymbol{c}', \\
\boldsymbol{d}'_0 &= \left\lfloor \frac{1}{\Delta} \cdot \boldsymbol{c}_0 \cdot \boldsymbol{c}'_0 \right\rceil, \\
\boldsymbol{d}'_1 &= \left\lfloor \frac{1}{\Delta} \cdot (\boldsymbol{c}_0 \cdot \boldsymbol{c}'_1 + \boldsymbol{c}'_0 \cdot \boldsymbol{c}_1) \right\rceil, \\
\boldsymbol{d}'_2 &= \left\lfloor \frac{1}{\Delta} \cdot \boldsymbol{c}_1 \cdot \boldsymbol{c}'_1 \right\rceil, \\
\boldsymbol{c}_{mul} &= (\boldsymbol{d}'_0, \boldsymbol{d}'_1) + \left\lfloor p^{-1} \cdot \boldsymbol{d}'_2 \cdot \boldsymbol{rk} \right\rceil \\
&= (\boldsymbol{d}_0, \boldsymbol{d}_1).
\end{aligned} \tag{20}$$

Multiplications by $\frac{1}{\Delta}$ and p^{-1} are performed in approximate arithmetic. The real numbers are then rounded to integers, denoted as $\lfloor \cdot \rceil$.

5 Homomorphic matrix operations with BFV

We introduce algorithms which embed an **entire** $k \times k$ matrix into a single plaintext. Embedding of a square matrix into a single plaintext requires representing it in row, column, or diagonal ordering. Consider the $k \times k$ matrix $A = [a_{ij}]$ for $1 \leq i, j \leq k$ such that a_{ij} are integers or fixed-point numbers. The plaintexts of BFV algorithm are CRT row vectors of dimension n as $\boldsymbol{a} = [a_1, a_2, \ldots, a_n]$. The entire matrix A is represented as a single element of the ring. For simplicity, k^2 is taken to be n; however, it is required that $k^2 \leq n$ in order to fit the $k \times k$ matrix into a plaintext over the CRT ring $\mathcal{Z}_q^n$. Elements of matrix A are written in row, column, or diagonal ordering, as shown in Fig. 3.

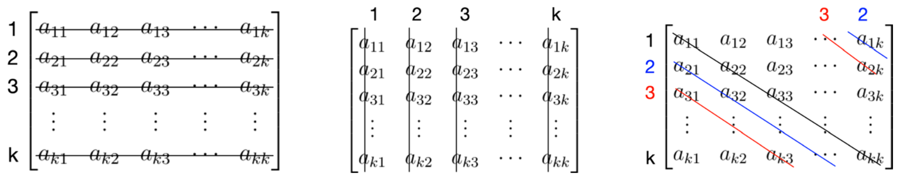

Fig. 3 Row, column or diagonal ordering of matrix elements

The rows or columns or diagonals of the matrix A are expressed in a row vector:

$$\begin{aligned} \boldsymbol{a}_{row} =&[a_{11}, a_{12}, \dots, a_{1k}, a_{21}, a_{22}, \dots, a_{2k}, \dots, a_{k1}, a_{k2}, \dots, a_{kk}], \\ \boldsymbol{a}_{col} =&[a_{11}, a_{21}, \dots, a_{k1}, a_{12}, a_{22}, \dots, a_{k2}, \dots, a_{1k}, a_{2k}, \dots, a_{kk}], \\ \boldsymbol{a}_{diag} =&[a_{11}, a_{22}, \dots, a_{kk}, a_{21}, a_{32}, \dots, a_{1k}, \dots, a_{k1}, a_{12}, \dots, a_{k-1,k}]. \end{aligned} \tag{21}$$

To clarify the concept of diagonal ordering, consider a $k \times k$ matrix $A = [a_{ij}]_{1 \le i,j \le k}$. The idea is to linearize the matrix elements along its diagonals instead of the conventional row or column ordering. More specifically, we first extract the main diagonal $(a_{11}, a_{22}, \dots, a_{kk})$, followed by the next diagonal starting at the second row, i.e., $(a_{21}, a_{32}, \dots, a_{1k})$, then the diagonal starting at the third row $(a_{31}, a_{42}, \dots, a_{2k})$. This process continues until all the diagonals parallel to the main diagonal have been traversed, with the final diagonal represented as $(a_{k1}, a_{12}, \dots, a_{k-1,k})$. In this way, the entire matrix is reorganized into a one-dimensional vector by collecting entries along each diagonal in sequence. Formally, the diagonal ordering can be expressed as Equation (21).

If a matrix is given in row, column or diagonal ordering, it can be converted to another ordering using $O(k^2)$ **read-write operations**. Since there are three orderings, there will be six conversion functions: `Col2Row`, `Col2Diag`, `Row2Col`, `Row2Diag`, `Diag2Col`, `Diag2Row`. We will only make use of the `Col2Diag` function in this paper. Given a ciphertext vector $\boldsymbol{x}$ expressed in column order, the function $\boldsymbol{d} = \texttt{Col2Diag}(\boldsymbol{x})$ converts it to diagonal order

$$\begin{aligned} \boldsymbol{x} =&[x_{1,1}, x_{2,1}, \dots, x_{k,1}, x_{1,2}, x_{2,2}, \dots, x_{k,2}, \dots, x_{1,k}, x_{2,k}, \dots, x_{k,k}], \\ \boldsymbol{d} =&[x_{1,1}, x_{2,2}, \dots, x_{k,k}, x_{2,1}, x_{3,2}, \dots, x_{1,k}, \dots, x_{k,1}, x_{1,2}, \dots, x_{k-1,k}]. \end{aligned} \tag{22}$$

5.1 Homomorphic matrix addition

Assume two input matrices $A, B \in \mathcal{Z}_q^{k \times k}$ are represented in the **same** ordering (row, column, or diagonal) giving the plaintext vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathcal{Z}_q^n$ with $k^2 \le n$. Each matrix occupies a single BFV plaintext. The plaintext vectors $\boldsymbol{a}, \boldsymbol{b}$ are encrypted to obtain the ciphertext vectors $\boldsymbol{x}, \boldsymbol{y}$ using the same key $\boldsymbol{pk}$.

$$\begin{aligned} \boldsymbol{x} =&(\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1}) = \text{Enc}_{\boldsymbol{pk}}(\boldsymbol{a}), \\ \boldsymbol{y} =&(\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1}) = \text{Enc}_{\boldsymbol{pk}}(\boldsymbol{b}). \end{aligned} \tag{23}$$

`BFVMatrixAdd` algorithm homomorphically adds two matrices A and B, by adding the ciphertext pairs $\boldsymbol{x}$ and $\boldsymbol{y}$. The result $z = \texttt{BFVMatrixAdd}(\boldsymbol{x}, \boldsymbol{y})$ is the encryption of $\boldsymbol{a} + \boldsymbol{b}$. In other words $z = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1}) = \text{Enc}_{\boldsymbol{pk}}(\boldsymbol{a} + \boldsymbol{b})$.

`BFVMatrixAdd` algorithm requires the same ordering (row, column or diagonal) for the input matrices. The result is the encryption of the sum of the input matrices represented in the same ordering. If the input matrices are of different ordering, one of the matrices needs to be converted, requiring $O(k^2)$ read-write operations. `BFVMatrixAdd` algorithm performs **a single homomorphic addition** of the two ciphertexts $\boldsymbol{x} = (\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1})$ and $\boldsymbol{y} = (\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1})$. An homomorphic addition is performed by adding the respective components of two pairs of ciphertexts. `BFVMatrixAdd` algorithm is defined in Algorithm 1.

Algorithm 1 BFVMatrixAdd

```
Input: $\boldsymbol{x}, \boldsymbol{y}$: ciphertext of two matrices in column ordering
Output: $\boldsymbol{z}$: the encryption of $\boldsymbol{a} + \boldsymbol{b}$
1: $\boldsymbol{x} = (\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1})$ and $\boldsymbol{y} = (\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1})$
   $\boldsymbol{z} = \boldsymbol{x} \oplus \boldsymbol{y}$
2: $\boldsymbol{c}_{z0} = \boldsymbol{c}_{x0} + \boldsymbol{c}_{y0}$
3: $\boldsymbol{c}_{z1} = \boldsymbol{c}_{x1} + \boldsymbol{c}_{y1}$
4: $\boldsymbol{z} = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1})$
5: return $\boldsymbol{z}$
```

The result is equal to $z = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1}) = \mathrm{Enc}_{\boldsymbol{pk}}(\boldsymbol{a} + \boldsymbol{b})$. The decryption of z with the secret key $\boldsymbol{sk}$ would produce $\boldsymbol{d} = \mathrm{Dec}_{\boldsymbol{sk}}(z)$ such that $\boldsymbol{d} = \mathrm{ColumnOrder}(D)$. Here, $D = A + B$ and $D \in \mathcal{Z}_q^{k\times k}$ and $k^2 \leq n$.

5.2 Homomorphic matrix multiplication

In this section, a homomorphic matrix multiplication algorithm is introduced to multiply two encrypted square matrices given in column order. We assume $A, B \in \mathcal{Z}_q^{k\times k}$ are given in their column ordering as $\boldsymbol{a}, \boldsymbol{b}$, and also their encryptions $\boldsymbol{x} = \mathrm{Enc}_{\boldsymbol{pk}}(\boldsymbol{a}) = (\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1})$ and $\boldsymbol{y} = \mathrm{Enc}_{\boldsymbol{pk}}(\boldsymbol{b}) = (\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1})$ are given. `BFVMatrixMultiply` algorithm given below takes $\boldsymbol{x}$ and $\boldsymbol{y}$ in column order and produces their product z in column order. The decryption of z produces the product matrix D, in other words $\mathrm{Dec}_{\boldsymbol{sk}}(z) = \boldsymbol{d} = \mathtt{ColumnOrder}(D)$ and $D = A \cdot B$.

The input ciphertexts $\boldsymbol{x}$ and $\boldsymbol{y}$ are given in column order:

$$\begin{aligned} \boldsymbol{c}_{x0} &= [c_{x0,11}, c_{x0,21}, \cdots, c_{x0,k1}, \cdots, c_{x0,1k}, c_{x0,2k}, \cdots, c_{x0,kk}], \\ \boldsymbol{c}_{x1} &= [c_{x1,11}, c_{x1,21}, \cdots, c_{x1,k1}, \cdots, c_{x1,1k}, c_{x1,2k}, \cdots, c_{x1,kk}], \\ \boldsymbol{c}_{y0} &= [c_{y0,11}, c_{y0,21}, \cdots, c_{y0,k1}, \cdots, c_{y0,1k}, c_{y0,2k}, \cdots, c_{y0,kk}], \\ \boldsymbol{c}_{y1} &= [c_{y1,11}, c_{y1,21}, \cdots, c_{y1,k1}, \cdots, c_{y1,1k}, c_{y1,2k}, \cdots, c_{y1,kk}]. \end{aligned} \tag{24}$$

$\boldsymbol{c}_{x0}$ and $\boldsymbol{c}_{y0}$, $\boldsymbol{c}_{x1}$ and $\boldsymbol{c}_{y1}$ are arranged into k vectors of length k as $\bar{c}_{x0,i}$ and $\bar{c}_{x1,i}$,

$$\begin{aligned} \boldsymbol{c}_{x0} &= [\bar{c}_{x0,1}, \bar{c}_{x0,2}, \cdots, \bar{c}_{x0,k}] \text{ and } & \boldsymbol{c}_{y0} &= [\bar{c}_{y0,1}, \bar{c}_{y0,2}, \cdots, \bar{c}_{y0,k}], \\ \boldsymbol{c}_{x1} &= [\bar{c}_{x1,1}, \bar{c}_{x1,2}, \cdots, \bar{c}_{x1,k}] \text{ and } & \boldsymbol{c}_{y1} &= [\bar{c}_{y1,1}, \bar{c}_{y1,2}, \cdots, \bar{c}_{y1,k}]. \end{aligned} \tag{25}$$

so that, for $i = 1, 2, \ldots, k$, we have

$$\begin{aligned} \bar{c}_{x0,i} &= [c_{x0,1i}, c_{x0,2i}, \cdots, c_{x0,ki}] \text{ and } & \bar{c}_{y0,i} &= [c_{y0,1i}, c_{y0,2i}, \cdots, c_{y0,ki}], \\ \bar{c}_{x1,i} &= [c_{x1,1i}, c_{x1,2i}, \cdots, c_{x1,ki}] \text{ and } & \bar{c}_{y1,i} &= [c_{y1,1i}, c_{y1,2i}, \cdots, c_{y1,ki}]. \end{aligned} \tag{26}$$

The **first step** of `BFVMatrixMultiply` is to convert $\boldsymbol{c} = (\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1})$ from column to diagonal order $\boldsymbol{d} = (\boldsymbol{d}_{x0}, \boldsymbol{d}_{x1}) = \mathtt{Col2Diag}(\boldsymbol{c})$,

$$\begin{aligned}\boldsymbol{d}_{x0} =&[d_{x0,11}, d_{x0,22}, \cdots, d_{x0,kk}, \cdots, d_{x0,k1}, d_{x0,12}, \cdots, d_{x0,k-1,k}],\\ \boldsymbol{d}_{x1} =&[d_{x1,11}, d_{x1,22}, \cdots, d_{x1,kk}, \cdots, d_{x1,k1}, d_{x1,12}, \cdots, d_{x1,k-1,k}].\end{aligned} \tag{27}$$

$\boldsymbol{d}_{x0}$ and $\boldsymbol{d}_{x1}$ are arranged into k vectors of length k as $\bar{d}_{x0,i}$ and $\bar{d}_{x1,i}$,

$$\begin{aligned}\boldsymbol{d}_{x0} =&[\bar{d}_{x0,1}, \bar{d}_{x0,2}, \dots, \bar{d}_{x0,k}],\\ \boldsymbol{d}_{x1} =&[\bar{d}_{x1,1}, \bar{d}_{x1,2}, \dots, \bar{d}_{x1,k}].\end{aligned} \tag{28}$$

These $\bar{d}_{x0,i+1}$ and $\bar{d}_{x1,i+1}$ vectors can be written for $i = 0, 1, \dots, k-1$ as

$$\begin{aligned}\bar{d}_{x0,i+1} =&[d_{x0,i,1}, d_{x0,i+1 \bmod k,2}, d_{x0,i+2 \bmod k,3}, \cdots, d_{x0,i+k-1 \bmod k,k}],\\ \bar{d}_{x1,i+1} =&[d_{x1,i,1}, d_{x1,i+1 \bmod k,2}, d_{x1,i+2 \bmod k,3}, \cdots, d_{x1,i+k-1 \bmod k,k}].\end{aligned} \tag{29}$$

The **second step** of BFVMatrixMultiply algorithm generates the vectors $\vec{\boldsymbol{d}}_{x0i}$ and $\vec{\boldsymbol{d}}_{x1i}$ for $i = 1, 2, \dots, k$, using $\bar{d}_{x0,i}$ and $\bar{d}_{x1,i}$. Let $R^j(x)$ denote the right rotation of the vector x by j times. For example, if $x = [x_1, x_2, \dots, x_k]$, then $R^1(x) = [x_k, x_1, x_2, \dots, x_{k-1}]$. The vectors $\bar{d}_{x0,i}$ and $\bar{d}_{x1,i}$ are right rotated and replicated to generate the vectors $\vec{\boldsymbol{d}}_{x0i}$ and $\vec{\boldsymbol{d}}_{x1i}$ for $i = 1, 2, \dots, k$ as follows:

$$\begin{aligned}\vec{\boldsymbol{d}}_{x0i} =&[R^{i-1}(\bar{d}_{x0,i}), R^{i-1}(\bar{d}_{x0,i}), \dots, R^{i-1}(\bar{d}_{x0,i})],\\ \vec{\boldsymbol{d}}_{x1i} =&[R^{i-1}(\bar{d}_{x1,i}), R^{i-1}(\bar{d}_{x1,i}), \dots, R^{i-1}(\bar{d}_{x1,i})].\end{aligned} \tag{30}$$

Here, $\bar{d}_{x0,i}$ and $\bar{d}_{x1,i}$ are of length k, while $\boldsymbol{d}_{x0i}$ and $\boldsymbol{d}_{y1i}$ are of length k^2. Note that the two consecutive terms $R^{i-1}(\bar{d}_{x0,i})$ and $R^{i-1}(\bar{d}_{x1,i})$ are intentionally written in this form, as they correspond to different components in the rotation process rather than a typographical error. The computation of $(\vec{\boldsymbol{d}}_{x0i}, \vec{\boldsymbol{d}}_{x1i})$ for $i = 1, 2, \dots, k$ is denoted as

$$(\vec{\boldsymbol{d}}_{x0i}, \vec{\boldsymbol{d}}_{x1i}) = \texttt{Generate}(\boldsymbol{d}_{x0}, \boldsymbol{d}_{x1}). \tag{31}$$

The **third step** of BFVMatrixMultiply algorithm uses the column ordering of $(\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1})$,

$$\begin{aligned}\boldsymbol{c}_{y0} &= [\bar{c}_{y0,1}, \bar{c}_{y0,2}, \cdots, \bar{c}_{y0,k}] \text{ for } \quad \bar{c}_{y0,i} = [c_{y0,1i}, c_{y0,2i}, \cdots, c_{y0,ki}],\\ \boldsymbol{c}_{y1} &= [\bar{c}_{y1,1}, \bar{c}_{y1,2}, \cdots, \bar{c}_{y1,k}] \text{ for } \quad \bar{c}_{y1,i} = [c_{y1,1i}, c_{y1,2i}, \cdots, c_{y1,ki}].\end{aligned} \tag{32}$$

Here, $\bar{c}_{y0,i}$ and $\bar{c}_{y1,i}$ are of length k, while $\boldsymbol{c}_{y0}$ and $\boldsymbol{c}_{y1}$ are of length k^2. The vectors $\bar{c}_{y0,i}$ and $\bar{c}_{y1,i}$ are right rotated to generate $\vec{\boldsymbol{c}}_{y0i}$ and $\vec{\boldsymbol{c}}_{y1i}$ vectors for $i = 1, 2, \dots, k$,

$$\begin{aligned}\vec{\boldsymbol{c}}_{y0i} =&[R^{i-1}(\bar{c}_{y0,1}), R^{i-1}(\bar{c}_{y0,2}), \dots, R^{i-1}(\bar{c}_{y0,k})],\\ \vec{\boldsymbol{c}}_{y1i} =&[R^{i-1}(\bar{c}_{y1,1}), R^{i-1}(\bar{c}_{y1,2}), \dots, R^{i-1}(\bar{c}_{y1,k})].\end{aligned} \tag{33}$$

The computation of $(\vec{\boldsymbol{c}}_{y0i}, \vec{\boldsymbol{c}}_{y1i})$ for $i = 1, 2, \dots, k$ is denoted as

$$(\vec{\boldsymbol{c}}_{y0i}, \vec{\boldsymbol{c}}_{y1i}) = \texttt{Generate}(\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1}). \tag{34}$$

The **fourth step** of `BFVMatrixMultiply` algorithm uses $\vec{\boldsymbol{d}}_{x0i}$ and $\vec{\boldsymbol{d}}_{x1i}$, and $\vec{\boldsymbol{c}}_{y0i}$ and $\vec{\boldsymbol{c}}_{y1i}$ components for $i = 1, 2, 3, \dots, k$, and performs homomorphic multiplications and additions to obtain $\boldsymbol{z}$:

$$\begin{aligned} z_1 &= (\vec{\boldsymbol{d}}_{x01}, \vec{\boldsymbol{d}}_{x11}) \odot (\vec{\boldsymbol{c}}_{y01}, \vec{\boldsymbol{c}}_{y11}), \\ z_2 &= (\vec{\boldsymbol{d}}_{x02}, \vec{\boldsymbol{d}}_{x12}) \odot (\vec{\boldsymbol{c}}_{y02}, \vec{\boldsymbol{c}}_{y12}), \\ &\cdots \\ z_k &= (\vec{\boldsymbol{d}}_{x0k}, \vec{\boldsymbol{d}}_{x1k}) \odot (\vec{\boldsymbol{c}}_{y0k}, \vec{\boldsymbol{c}}_{y1k}), \\ z &= z_1 \oplus z_2 \oplus \cdots \oplus z_k. \end{aligned} \tag{35}$$

The resulting ciphertext $\boldsymbol{z} = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1})$ is the encryption of the column representation of the product matrix $D = A \cdot B$. The BFVMatrixMultiply algorithm is described in Algorithm 2.

Algorithm 2 BFVMatrixMultiply

```
Input: $\boldsymbol{x}, \boldsymbol{y}$: ciphertext of two matrices in column ordering
Output: $\boldsymbol{z}$: the encryption of $\boldsymbol{a} \times \boldsymbol{b}$
 1: $\boldsymbol{x} = (\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1})$ and $\boldsymbol{y} = (\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1})$
    $\boldsymbol{z} = \boldsymbol{x} \odot \boldsymbol{y}$
 2: $\boldsymbol{d} = \texttt{Col2Diag}(\boldsymbol{x})$
 3: $(\vec{\boldsymbol{d}}_{x0i}, \vec{\boldsymbol{d}}_{x1i}) = \texttt{Generate}(\boldsymbol{d}_{x0}, \boldsymbol{d}_{x1})$
 4: $(\vec{\boldsymbol{c}}_{y0i}, \vec{\boldsymbol{c}}_{y1i}) = \texttt{Generate}(\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1})$
 5: for $i = 1, 2, \dots, k$ do
 6:     $\boldsymbol{z}_i = (\vec{\boldsymbol{d}}_{x0i}, \vec{\boldsymbol{d}}_{x1i}) \odot (\vec{\boldsymbol{c}}_{y0i}, \vec{\boldsymbol{c}}_{y1i})$
 7: end for
 8: $\boldsymbol{z} = \boldsymbol{z}_1$
 9: for $i = 2, 3, \dots, k$ do
10:     $\boldsymbol{z} = \boldsymbol{z} \oplus \boldsymbol{z}_i$
11: end for
12: return $\boldsymbol{z}$
```

The result is equal to $\boldsymbol{z} = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1}) = \text{Enc}_{\boldsymbol{pk}}(\boldsymbol{a} \cdot \boldsymbol{b})$. The decryption of $\boldsymbol{z}$ with the secret key $\boldsymbol{sk}$ would produce $\boldsymbol{d} = \text{Dec}_{\boldsymbol{sk}}(\boldsymbol{z})$ such that $\boldsymbol{d} = \text{ColumnOrder}(D)$. Here, $D = A \cdot B$ and $D \in \mathcal{Z}_q^{k \times k}$ and $k^2 \leq n$.

6 Complexity analysis

The `BFVMatrixAdd` algorithm assumes the input ciphertexts $\boldsymbol{x}$ and $\boldsymbol{y}$ are in the **column order**, and computes the resulting matrix in the column order. If the input ciphertexts are not in the column order, they need to be converted to the column order, which requires $O(k^2)$ read-write operations. Given the two input ciphertexts in the column order, the `BFVMatrixAdd` algorithm performs a single BFV homomorphic addition of two ciphertexts. This is accomplished using two additions in

$\mathcal{Z}_q^n$. The output ciphertext is the encryption of the sum matrix $A + B$ in the column order.

On the other hand, given the two input ciphertexts in the column order, the `BFV-MatrixMultiply` algorithm performs k BFV homomorphic multiplications and $k-1$ BFV homomorphic additions, as shown in Steps 5-11 of Algorithm 2. The complexity for matrix size $k \times k$ of our homomorphic matrix multiplication algorithm is given in the last column of Table 3. Furthermore, the multiplicative complexity values for the Naive, Halevi and Shoup, and JKLS algorithms are included in the same table. Table 3 shows that the arithmetic complexity of our algorithm is slightly better than that of the JKLS algorithm; however, they are asymptotically the same.

While it has nearly the same arithmetic complexity as our algorithm, the JKLS algorithm incurs significantly higher costs for changing the order of matrices. The main reason for this cost is that the JKLS algorithm encrypts two matrices into a single ciphertext. This reduces the number of ciphertexts, but changing the order of ciphertext elements increases its complexity. Here, we demonstrate their algorithm for changing the matrix from the row order to diagonal order. For example, two matrices of size 3×3, as shown in Fig. 2, are encoded into a single vector $\boldsymbol{v}$:

$$\boldsymbol{v} = [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8].$$

First, we perform three masking operations to obtain $\boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2$:

$$\begin{aligned}
\boldsymbol{v}_0 &= [a_0, 0, 0, 0, a_4, 0, 0, 0, a_8, b_0, 0, 0, 0, b_4, 0, 0, 0, b_8],\\
\boldsymbol{v}_1 &= [0, a_1, 0, 0, 0, a_5, a_6, 0, 0, 0, b_1, 0, 0, 0, b_5, b_6, 0, 0],\\
\boldsymbol{v}_2 &= [0, 0, a_2, a_3, 0, 0, 0, a_7, 0, 0, 0, b_2, b_3, 0, 0, 0, b_7, 0].
\end{aligned}$$

The complexity of the **first step** is three.

Second, these three vectors are rotated. For vector $\hat{\boldsymbol{v}}_i$, $i = 0, 1, 2$, it has $2 \cdot 3 = 6$ non-zero elements. Thus the number of rotation for vector $\hat{\boldsymbol{v}}_i$ is six. The total complexity of the **second step** is $6 \cdot 3 = 18$.

Table 3 The complexity of the proposed algorithm for matrix multiplication along with the Naive, Halevi and Shoup, and JKLS algorithms for dimension k square matrices

	Naive	Halevi & Shoup	JKLS	Our algorithm
Condition	–	$k^2 < n$	$k^2 < n$	$k^2 < n$
Input ordering	–	Diagonal	Row	Column
Output ordering	–	Vector	Row	Column
Ciphertexts	$2k^2$	$2k$	1	2
Additions	$k^2(k-1)$	$k(k-1)$	$6k$	$k-1$
Multiplications	k^3	k^2	k	k
Data rotations	0	k	$3k + 5\sqrt{k}$	$2k$
Arithmetic depth	1 Mult	1 Mult	1 Mult + 2 CMult	1 Mult

k denotes the dimension of square matrices and $n = k^2$. Mult is homomorphic multiplication of two ciphertexts, while CMult is homomorphic multiplication of a ciphertext by a constant

$$\begin{aligned}\hat{\boldsymbol{v}}_0 =&[a_0, a_4, a_8, 0, 0, 0, 0, 0, 0, b_0, b_4, b_8, 0, 0, 0, 0, 0, 0],\\ \hat{\boldsymbol{v}}_1 =&[0, 0, 0, a_1, a_5, a_6, 0, 0, 0, 0, 0, 0, b_1, b_5, b_6, 0, 0, 0],\\ \hat{\boldsymbol{v}}_2 =&[0, 0, 0, 0, 0, 0, a_2, a_3, a_7, 0, 0, 0, 0, 0, 0, b_2, b_3, b_7].\end{aligned}$$

Third, these vectors are added to obtain the vector $\hat{\boldsymbol{v}}$:

$$\begin{aligned}\hat{\boldsymbol{v}} =&\hat{\boldsymbol{v}}_0 + \hat{\boldsymbol{v}}_1 + \hat{\boldsymbol{v}}_2\\ =&[a_0, a_4, a_8, a_1, a_5, a_2, a_3, a_7, b_0, b_4, b_8, b_1, b_5, b_6, b_2, b_3, b_7].\end{aligned} \tag{36}$$

The complexity of the **third step** is two.

Therefore, the complexity of the first masking step is k, the second rotation step is $2k^2$, and the third addition step is $k-1$. The total complexity of changing the order of JKLS algorithm becomes $2k^2 + 2k - 1$. The results are summarized in Table 4.

As compared to the JKLS algorithm, our algorithm only needs to rotate k^2 nonzero elements. The complexity of rotation step is k^2. The total complexity of changing the order of our algorithm is $k^2 + 2k - 1$.

While our proposed BFV-based matrix multiplication algorithm demonstrates correctness and SIMD-enabled parallelism, several limitations should be acknowledged. First, homomorphic rotations and multiplications introduce considerable computational and storage overhead, which may limit practicality for very large matrices. Second, although our zero-padding method allows for handling rectangular matrices, extremely large dimensions still incur substantial computational cost. Third, BFV encoding inherently introduces small approximation errors, which should be taken into account in precision-sensitive applications.

7 Rectangular matrices

The importance of extending the square matrix multiplication algorithms to rectangular matrices, with matching dimensions such as $A_{j\times k} \times B_{k\times l}$, has been emphasized. We now describe an algorithm for performing rectangular matrix multiplication. Specifically, the matrices $A_{j\times k}$ and $B_{k\times l}$ are expanded into square matrices

Table 4 Complexities of JKLS algorithm and our algorithm changing the matrix order

	JKLS	Our algorithm
Condition	$k^2 < n$	$k^2 < n$
Input ordering	Row	Column
Output ordering	Row	Column
Ciphertexts	1	2
Additions	$k-1$	$k-1$
Multiplications	k	k
Data rotations	$2k^2$	k^2

k denotes the dimension of square matrices and $n = k^2$

$A_{K\times K}$ and $B_{K\times K}$ by padding with zero rows and columns to both matrices, such that $K = \max(j, k, l)$. Two cases are distinguished under this scenario: $K = j$ and $K = l$.

- For $K = j$, the number of rows of the matrix $A_{j\times k}$ is greater than that of columns. For example, consider $j = 4, k = 2, l = 3$, then $K = max(4, 2, 3) = 4$. We embed matrices $A_{4\times 2}$ and $B_{2\times 3}$ into $A_{4\times 4}$ and $B_{4\times 4}$ square matrices by padding zero rows or columns. Matrix $A_{4\times 2}$ is padded to $A_{4\times 4}$ by appending two columns of zeros on the right. Matrix $B_{2\times 3}$ is padded to $B_{4\times 4}$ by appending one column of zeros on the right and two rows of zeros on the bottom.

$$A_{4\times 2} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \longrightarrow A_{4\times 4} = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & 0 & 0 \\ a_{41} & a_{42} & 0 & 0 \end{bmatrix}$$

$$B_{2\times 3} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \longrightarrow B_{4\times 4} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & 0 \\ b_{21} & b_{22} & b_{23} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- For $K = l$, the number of columns of the matrix $B_{k\times l}$ is greater than that of rows. For example, consider $j = 2, k = 3, l = 4$, then $K = max(2, 3, 4) = 4$. We embed matrices $A_{2\times 3}$ and $B_{3\times 4}$ into $A_{4\times 4}$ and $B_{4\times 4}$ square matrices by padding zero rows or columns. Matrix $A_{2\times 3}$ is padded to $A_{4\times 4}$ by appending one column of zeros on the right and two rows of zeros on the bottom. Matrix $B_{3\times 4}$ is padded to $B_{4\times 4}$ by appending one row of zeros on the bottom.

$$A_{2\times 3} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \longrightarrow A_{4\times 4} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B_{3\times 4} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix} \longrightarrow B_{4\times 4} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The complexity values of our algorithm and the JKLS algorithm are summarized in Table 5. JKLS algorithm considers two special cases of rectangular matrices multiplication $A_{j\times k} \times B_{k\times k}$ and $A_{k\times k} \times B_{k\times j}$, respectively, where $j < k$. They fill $k - j$ rows in a vertical direction of the left matrix $A_{j\times k}$ so that it becomes $A_{k\times k}$. Also, they fill $k - j$ columns in a vertical direction of the right matrix $B_{k\times j}$ so that it becomes $B_{k\times k}$. Their algorithm requires $O(j)$ homomorphic multiplications in the square matrix multiplication. However, it is unclear that their algorithm could be extended to the other rectangular matrices multiplication, such as $A_{k\times j} \times B_{j\times k}$.

Table 5 Comparing JKLS algorithm to the proposed homomorphic multiplication algorithm for rectangular matrices

	JKLS	Our Algorithm
Condition	$k^2 < n$	$K^2 < n$
Input ordering	Row	Column
Output ordering	Row	Column
Ciphertexts	1	2
Additions	$3k + 2j + \log(k/j)$	$K - 1$
Multiplications	j	K
Data rotations	$3j + 5\sqrt{k} + \log(k/j)$	$2K$

j, k, l denote the dimensions of the rectangular matrices in JKLS algorithm. K represents the dimension of the square matrix in our algorithm. n denotes K^2

8 Experimental evaluation

In this section, we evaluate the performance of our secure matrix multiplication algorithm through experiments and compare it with related works.

8.1 Setup

Our algorithm is based on BFV fully homomorphic encryption scheme and is implemented using the *Pyfhel* library [28], version 3.4.1. The library is based on Microsoft's open source *SEAL* library. The machine equipped for the experiment is an Apple Silicon M2 running with 8 cores (3.5GHz) and 16GB of RAM, with macOS Sonoma operating system installed. Our implementation is openly accessible at https://github.com/KocLab2023/MatMult.

For the parameter selection of the BFV algorithm, we follow the homomorphic encryption standard [29]. The security parameter is set to $\lambda = 128$ bits; a larger value increases security but reduces computational efficiency. We choose a polynomial modulus of degree $N = 2^{13}$, which allows each plaintext to be encrypted into a single ciphertext with 2^{13} slots. The plaintext modulus t is selected to ensure both security and efficient encoding. It is required to be prime, and $t - 1$ must be divisible by 2^N. In our experiments, the bit-length of t is set to 20, yielding a plaintext modulus of $t = 65537$.

In our algorithm, the dimension of a square matrix is k, requiring k^2 plaintext slots, and k^2 slots are required for matrix multiplication. Therefore, a single plaintext can perform up to $\delta \leq \frac{2^{13}}{k^2}$ homomorphic multiplication operations. The maximum utilization rate of the slots in our proposed matrix multiplication algorithm is $\frac{1}{\delta}$%.

In our experiments, we selected an optimal number of plaintext slots. Since the matrix dimension is not too large (≤ 20), it is easy to find such a parameter. For the computation of matrices with larger dimensions, the upper limit on the number of plaintext slots can be increased accordingly.

Table 6 The basic characteristics of datasets

Dataset	Record	Attribute
Adult	32,561	6
Winequality	4898	12
Communities	1994	20

8.2 Datasets and comparison

Experiments are conducted on three datasets from the UCI Machine Learning Repository.[1] By selecting matrices with different numbers of rows and columns from these three datasets, we analyze the impact of different matrix dimensions and matrix elements on the performance of our matrix multiplication algorithm. These three datasets are Adult, Winequality, and Communities, respectively. Specifically, the Adult dataset contains 32,561 records, and for each record, we select six attributes of continuous data; the Winequality dataset contains 4,898 records, each with 12 attributes; the Communities dataset contains 1,994 records, and 20 attributes are selected for each record. The basic characteristics of the dataset are shown in Table 6. For floating-point data, it cannot be directly used for BFV homomorphic encryption. Therefore, we multiply floating-point data by an amplification factor of 100 to convert it into integer data before performing encryption operations.

Comparison. Existing secure matrix multiplication algorithms are discussed in Sect. 2, which require multiple expensive ciphertext multiplication operations. However, our scheme reduces the number of ciphertext multiplication operations and rotations, improving computational efficiency. Therefore, we evaluate the performance of JKLS algorithm [19], ZLW algorithm [26], and our algorithm. Each dataset and algorithm is tested 20 times, and the average of these 20 repeated experiments is reported.

8.3 Performance evaluation

In this section, the time performance and memory usage of our algorithm and the comparison algorithm are tested. For the three selected datasets, we randomly select matrices of different dimensions to satisfy the experimental requirements.

In Table 7, we compare the performance of our secure matrix multiplication algorithm with JKLS algorithm and ZLW algorithm. The results of our algorithm are shown in **bold**. It can be observed that our matrix multiplication algorithm consistently achieves better running time across matrices of different dimensions and datasets compared to JKLS algorithm. Moreover, our algorithm achieves better running time than the ZLW algorithm in small-dimensional matrix multiplication, and its encryption and decryption times are also superior.

[1] http://archive.ics.uci.edu/ml.

Table 7 Comparison of time performance and memory usage between JKLS algorithm [19], ZLW algorithm [26], and our algorithm

Dataset	Dimension	Algorithm	Encrypt (ms)	Mat. Mult (s)	Decrypt (ms)	Memory (KB)
Adult	$A_{2\times4}\cdot B_{4\times5}$	[19]	15.395	5.683	1.628	512
		[26]	51.648	0.433	5.492	1024
		Ours	**17.444**	**0.116**	**1.575**	**1024**
	$A_{8\times6}\cdot B_{6\times2}$	[19]	16.482	8.312	1.546	512
		[26]	53.218	0.722	5.790	1024
		Ours	**17.676**	**2.159**	**1.662**	**1024**
	$A_{10\times6}\cdot B_{6\times4}$	[19]	17.355	10.029	1.726	512
		[26]	50.304	1.479	5.585	1024
		Ours	**18.819**	**3.998**	**1.677**	**1024**
Winequality	$A_{4\times6}\cdot B_{6\times3}$	[19]	15.827	7.251	1.657	512
		[26]	55.619	0.889	5.801	1024
		Ours	**17.278**	**0.487**	**1.693**	**1024**
	$A_{9\times8}\cdot B_{8\times6}$	[19]	17.748	10.894	1.881	512
		[26]	52.235	1.241	5.755	1024
		Ours	**19.801**	**3.241**	**1.711**	**1024**
	$A_{8\times12}\cdot B_{12\times12}$	[19]	18.201	16.128	1.583	512
		[26]	55.013	1.083	5.634	1024
		Ours	**19.255**	**2.151**	**1.696**	**1024**
Communities	$A_{11\times10}\cdot B_{10\times8}$	[19]	14.854	13.528	1.989	512
		[26]	53.281	1.537	5.816	1024
		Ours	**17.256**	**5.439**	**1.742**	**1024**
	$A_{10\times15}\cdot B_{15\times14}$	[19]	16.664	19.452	1.635	512
		[26]	55.936	1.827	5.727	1024
		Ours	**17.149**	**4.073**	**1.685**	**1024**
	$A_{20\times20}\cdot B_{20\times16}$	[19]	16.792	47.163	1.908	512
		[26]	56.113	2.016	5.824	1024
		Ours	**17.662**	**18.538**	**1.881**	**1024**

- Encryption Time (milliseconds): The encryption time of JKLS algorithm is faster than that of our algorithm and ZLW algorithm. This is because their method requires only a single ciphertext to store the encrypted matrix, whereas our algorithm and ZLW algorithm require two ciphertexts. Moreover, our algorithm achieves significantly faster encryption time compared to ZLW algorithm, since it requires tensor ring encoding and zero-padding, which incur substantial initial encryption overhead.
- Matrix Multiplication Time (seconds): The matrix multiplication time of our algorithm is largely consistent with the homomorphic operation complexity described in Table 3. As expected, the computation time of our algorithm is significantly lower than that of JKLS algorithm. For instance, on the Communities dataset with $A_{10\times15}\cdot B_{15\times14}$, the matrix multiplication time decreases from

19.452 s to 4.073 s. This improvement is due to our algorithm requiring fewer rotations than JKLS algorithm. Moreover, our algorithm outperforms ZLW algorithm in small-dimensional matrix multiplication, for example, on the Winequality dataset with $A_{4\times 6} \cdot B_{6\times 3}$, the matrix multiplication time decreases from 0.889 s to 0.487 s.
- Decryption Time (milliseconds): Our algorithm achieves a decryption time comparable to the JKLS algorithm and superior to the ZLW algorithm. Moreover, decryption is faster than encryption, since only the single ciphertext corresponding to the product matrix needs to be decrypted, whereas the encryption process involves encrypting both original matrices.
- Memory (KB): We measured the storage cost of encrypting two matrices into ciphertexts. For the three datasets, when varying the dimensions of the matrices, the storage cost of JKLS algorithm consistently remained 512 KB. This is because each plaintext matrix is encoded into a single BFV ciphertext; therefore, as long as the number of elements does not exceed the number of plaintext slots ($N = 2^{13} = 8192$), the ciphertext size remains unchanged. In contrast, our algorithm and ZLW algorithm require encrypting two matrices, whereas JKLS algorithm requires only one, resulting in the storage cost of JKLS algorithm being approximately half that of our algorithm and ZLW algorithm.
- Need for Supercomputing: The results of our experiments shown in Table 7 are far from real-world scenarios where the dimensions of matrices can range from 1000 to several 100,000. Furthermore, achieving a 128-bit security level for BFV requires correspondingly large setup parameters: the modulus q is selected between 512 and 2,048, while the size of the polynomials n is selected between 16,384 and 65,536. As we stated in Section 1, ciphertexts are polynomials from the ring $\mathcal{Z}_q[x]/(x^n + 1)$. From Table 3, we observe that although we reduce the number of homomorphic additions, multiplications, and data rotations to $O(k)$, the required number of arithmetic and memory operations are still insurmountable: $k \times q \times n = 100,000 \times 2,048 \times 65,536 \approx 13 \cdot 10^{12}$. In order to bring down such levels of complexity, there have been several projects, for example, DPRIVE is one prime example [30]. We hope our algorithms among many other similar ones will bring the attention of the supercomputing community to the FHE matrix problems for real-world scientific applications.

Table 8 summarizes the performance analysis of JKLS, ZLW, and our algorithms, highlighting their relative strengths and weaknesses in terms of encryption time, matrix multiplication time, decryption time, and memory usage.

9 Conclusions and future work

We introduced two new algorithms `BFVMatrixAdd` and `BFVMatrixMultiply` for homomorphic matrix addition and multiplication of matrices encrypted using the fully homomorphic encryption method BFV [27]. We describe our proposed

Table 8 Performance summary of JKLS algorithm, ZLW algorithm, and Our algorithm

Metric	JKLS algorithm	ZLW algorithm	Our algorithm
Encryption time (ms)	Fastest (1 ciphertext per matrix)	Slowest (tensor ring encoding + zero-padding)	Slower than JKLS, but significantly faster than ZLW
Matrix multiplication time (s)	19.452 s on Communities dataset	0.889 s on Winequality (slower on small matrices)	4.073 s on Communities dataset; 0.487 s on Winequality
Decryption time (ms)	Comparable to ours	Slowest	Comparable to JKLS, faster than ZLW
Memory (KB)	512 KB (1 ciphertext/matrix)	2× JKLS (2 ciphertexts/matrix)	2× JKLS (2 ciphertexts/matrix)

algorithms in detail from the encryption steps, to the homomorphic arithmetic operations, and finally to the decryption, in order to facilitate programming efforts. The multiplicative arithmetic complexity of our algorithm is superior to that of the Naive, Halevi and Shoup, and JKLS algorithms. We also improve the complexity of changing the order of matrices between row, column and diagonal order. The experimental results indicate that our algorithm achieves better performance.

Furthermore, our algorithms accept input matrices in the column order, producing the output sum or product matrix in the column order, allowing a series of matrix operations found in more complex matrix algorithms such as Singular Value Decomposition (SVD) and Eigenvalue (EV) and Eigenmatrix (EM) computations. We are currently working on homomorphic SVD, EV, and EM computations on encrypted matrices, which will be reported in the near future. Our planned method is to build upon the proposed matrix multiplication algorithm by combining iterative algorithms such as the power iteration and Lanczos method, which are amenable to homomorphic implementation, and polynomial approximations (e.g., Chebyshev or Taylor expansions) to approximate nonlinear functions like normalization or reciprocal square roots. This strategy is expected to enable homomorphic implementations of matrix decomposition methods.

Appendix A.

We give numerical examples for every step of the homomorphic encryption, decryption, matrix addition, and matrix multiplication.

A.1 Encryption and decryption

Consider the parameters $\mathcal{R}$ with $n = 4$, $t = 7$, $q = 897$, and compute $\Delta = \lfloor q/t \rfloor = 128$. Select $\boldsymbol{sk} = [-1, 1, 1, 0]$, $\boldsymbol{w} = [84, -60, -23, 117]$, and $\boldsymbol{e} = [-1, -4, 1, -3]$. We compute $\boldsymbol{u} = -(\boldsymbol{w} \cdot \boldsymbol{sk} + \boldsymbol{e})$ as

$$\begin{aligned} \boldsymbol{u} =& - ([84, -60, -23, 117] \cdot [-1, 1, 1, 0] + [-1, -4, 1, -3]) \\ =& - ([-84, -60, -23, 0] + [-1, -4, 1, -3]) \\ =& [85, 64, 22, 3]. \end{aligned} \tag{37}$$

The **public key** is the pair $(\boldsymbol{u}, \boldsymbol{w}) \in \mathcal{R}_q^2$. Now the message is considered as $\boldsymbol{m}_0 = [1, 2, 3, 4]$, and the random numbers are selected as $\boldsymbol{r} = [0, -1, 0, 1]$, $\boldsymbol{e}_0 = [3, -1, 4, 2]$, and $\boldsymbol{e}_1 = [-4, 1, 5, -3]$. The first ciphertext polynomial is computed as $\boldsymbol{c}_0 = \boldsymbol{r} \cdot \boldsymbol{u} + \boldsymbol{m}_0 \cdot \Delta + \boldsymbol{e}_0$,

$$\begin{aligned} \boldsymbol{c}_0 =& [0, -1, 0, 1] \cdot [85, 64, 22, 3] + [1, 2, 3, 4] \cdot 128 + [3, -1, 4, 2] \\ =& [131, 191, 388, 517]. \end{aligned} \tag{38}$$

The second ciphertext polynomial is computed as $\boldsymbol{c}_1 = \boldsymbol{r} \cdot \boldsymbol{w} + \boldsymbol{e}_1$,

$$\begin{aligned} \boldsymbol{c}_1 =&[0,-1,0,1]\cdot[84,-60,-23,117]+[-4,1,5,-3] \\ =&[-4,61,5,114]. \end{aligned} \tag{39}$$

The ciphertext is $(\boldsymbol{c}_0,\boldsymbol{c}_1)=([131,191,388,517],[-4,61,5,114])$.

The BFV decryption computes the plaintext vector $\boldsymbol{m}'$ from the ciphertext $\boldsymbol{c}$ using the secret key $\boldsymbol{sk}$ as $\boldsymbol{m}'=\boldsymbol{c}_0+\boldsymbol{c}_1\cdot\boldsymbol{sk}$,

$$\begin{aligned} \boldsymbol{m}' =&[131,191,388,517]+[-4,61,5,114]\cdot[-1,1,1,0] \\ =&[135,252,393,517]. \end{aligned} \tag{40}$$

The final step of the decryption computes $\boldsymbol{m}_0$ as

$$\begin{aligned} \boldsymbol{m}_0 =&\frac{1}{128}\cdot[135,252,393,517] \\ \approx&[1,2,3,4]. \end{aligned} \tag{41}$$

A.2 Homomorphic addition

Another message is considered as $\boldsymbol{m}_1=[3,1,2,0]$, and the random numbers are selected as $\boldsymbol{r}'=[1,0,1,-1]$, $\boldsymbol{e}_0'=[-1,-4,1,3]$, and $\boldsymbol{e}_1'=[3,-1,4,2]$. The first component of the ciphertext is computed as $\boldsymbol{c}_0'=\boldsymbol{r}'\cdot\boldsymbol{u}+\boldsymbol{m}_1\cdot\Delta+\boldsymbol{e}_0'$,

$$\begin{aligned} \boldsymbol{c}_0' =&[1,0,1,-1]\cdot[85,64,22,3]+[3,1,2,0]\cdot128+[-1,-4,1,3] \\ =&[468,124,279,0]. \end{aligned} \tag{42}$$

The second component of the ciphertext is computed as $\boldsymbol{c}_1'=\boldsymbol{r}'\cdot\boldsymbol{w}+\boldsymbol{e}_1'$,

$$\begin{aligned} \boldsymbol{c}_1' =&[1,0,1,-1]\cdot[84,-60,-23,117]+[3,-1,4,2] \\ =&[87,-1,-19,-115]. \end{aligned} \tag{43}$$

The ciphertext is obtained as $(\boldsymbol{c}_0',\boldsymbol{c}_1')=([468,124,279,0],[87,-1,-19,-115])$. The homomorphic addition of two ciphertexts $\boldsymbol{c}=(\boldsymbol{c}_0,\boldsymbol{c}_1)$ and $\boldsymbol{c}'=(\boldsymbol{c}_0',\boldsymbol{c}_1')$ is performed using $(\boldsymbol{d}_0,\boldsymbol{d}_1)=(\boldsymbol{c}_0+\boldsymbol{c}_0',\boldsymbol{c}_1+\boldsymbol{c}_1')$ as

$$\begin{aligned} \boldsymbol{d}_0 =&[131,191,388,517]+[468,124,279,0] \\ =&[599,315,667,517], \\ \boldsymbol{d}_1 =&[-4,61,5,114]+[87,-1,-19,-115] \\ =&[83,60,-14,-1]. \end{aligned} \tag{44}$$

The homomorphic addition result is $\boldsymbol{c}_{add}=(\boldsymbol{d}_0,\boldsymbol{d}_1)$. The plaintext is obtained by decrypting $\boldsymbol{c}_{add}=(\boldsymbol{d}_0,\boldsymbol{d}_1)$ with $\boldsymbol{sk}$ as

$$\begin{aligned} \boldsymbol{m}_{add}' =&\boldsymbol{d}_0+\boldsymbol{d}_1\cdot\boldsymbol{sk} \\ =&[599,315,667,517]+[83,60,-14,-1]\cdot[-1,1,1,0] \\ =&[516,375,653,517]. \end{aligned} \tag{45}$$

The scaling factor Δ is then applied to obtain

$$\begin{aligned} \boldsymbol{m}_{add} =& \frac{1}{128} \cdot [516, 375, 653, 517] \\ \approx& [4, 3, 5, 4]. \end{aligned} \tag{46}$$

The result is correct, since the addition of the plaintexts is equal to

$$\boldsymbol{m}_0 + \boldsymbol{m}_1 = [1, 2, 3, 4] + [3, 1, 2, 0] = [4, 3, 5, 4]. \tag{47}$$

A.3 Homomorphic multiplication

Let $p = 15001$, select $\boldsymbol{w}' = [43, -5, -102, 61]$, and $\boldsymbol{e}' = [1, -3, 3, -2]$. We compute $\boldsymbol{u}' = -(\boldsymbol{w}' \cdot \boldsymbol{sk} + \boldsymbol{e}') + p \cdot \boldsymbol{sk}^2$,

$$\begin{aligned} \boldsymbol{u}' =& -([43, -5, -102, 61] \cdot [-1, 1, 1, 0] + [1, -3, 3, -2]) + 15001 \cdot [1, 1, 1, 0] \\ =& [15043, 15009, 15100, 2]. \end{aligned} \tag{48}$$

The **relinearization key** is the pair $\boldsymbol{rk} = (\boldsymbol{u}', \boldsymbol{w}')$, which is obtained as

$$(\boldsymbol{u}', \boldsymbol{w}') = ([15043, 15009, 15100, 2], [43, -5, -102, 61]). \tag{49}$$

The homomorphic multiplication of two ciphertexts $\boldsymbol{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ and $\boldsymbol{c}' = (\boldsymbol{c}'_0, \boldsymbol{c}'_1)$ produces 3 intermediate terms: $(\boldsymbol{d}'_0, \boldsymbol{d}'_1, \boldsymbol{d}'_2)$. The computation of $\boldsymbol{d}'_0 = \frac{1}{\Delta} \cdot \boldsymbol{c}_0 \cdot \boldsymbol{c}'_0$ is as follows:

$$\begin{aligned} \boldsymbol{d}'_0 =& \frac{1}{128} \cdot [131, 191, 388, 517] \cdot [468, 124, 279, 0] \\ =& [479, 185, 846, 0]. \end{aligned} \tag{50}$$

Similarly, the computation of $\boldsymbol{d}'_1 = \frac{1}{\Delta} \cdot (\boldsymbol{c}_0 \cdot \boldsymbol{c}'_1 + \boldsymbol{c}'_0 \cdot \boldsymbol{c}_1)$ is performed as

$$\begin{aligned} \boldsymbol{d}'_1 =& \frac{1}{128} \cdot ([131, 191, 388, 517] \cdot [87, -1, -19, -115] + [-4, 61, 5, 114] \cdot [468, 124, 279, 0]) \\ =& [74, 58, -47, -464]. \end{aligned} \tag{51}$$

Finally, the computation of $\boldsymbol{d}'_2 = \frac{1}{\Delta} \cdot \boldsymbol{c}_1 \cdot \boldsymbol{c}'_1$ is performed as

$$\begin{aligned} \boldsymbol{d}'_2 =& \frac{1}{128} \cdot [-4, 61, 5, 114] \cdot [87, -1, -19, -115] \\ =& [-3, 0, -1, -102]. \end{aligned} \tag{52}$$

The temporary result of the homomorphic multiplication is $(\boldsymbol{d}'_0, \boldsymbol{d}'_1, \boldsymbol{d}'_2)$. Relinearization is then performed as

$$\boldsymbol{c}_{mul} = (\boldsymbol{d}_0, \boldsymbol{d}_1) = (\boldsymbol{d}'_0, \boldsymbol{d}'_1) + \left\lfloor p^{-1} \cdot \boldsymbol{d}'_2 \cdot \boldsymbol{rk} \right\rceil, \tag{53}$$

such that $\boldsymbol{rk} = (\boldsymbol{u}', \boldsymbol{w}')$. The first term is $\boldsymbol{d}_0 = \boldsymbol{d}_0' + \left\lfloor p^{-1} \cdot \boldsymbol{d}_2' \cdot \boldsymbol{u}' \right\rceil \bmod q$,

$$\begin{aligned}\boldsymbol{d}_0 =&[479, 185, 846, 0] + \frac{1}{15001} \cdot [-3, 0, -1, -102] \cdot [15043, 15009, 15100, 2]\\ \approx&[476, 185, 845, 0].\end{aligned} \tag{54}$$

The second term is $\boldsymbol{d}_1 = \boldsymbol{d}_1' + \left\lfloor p^{-1} \cdot \boldsymbol{d}_2' \cdot \boldsymbol{w}' \right\rceil \bmod q$,

$$\begin{aligned}\boldsymbol{d}_1 =&[74, 58, -47, -464] + \frac{1}{15001} \cdot [-3, 0, -1, -102] \cdot [43, -5, -102, 61]\\ \approx&[74, 58, -47, -464].\end{aligned} \tag{55}$$

The decryption of $\boldsymbol{c}_{mul} = (\boldsymbol{d}_0, \boldsymbol{d}_1)$ using $\boldsymbol{sk}$ gives back $\boldsymbol{m}_{mul}$:

$$\begin{aligned}\boldsymbol{m}_{mul} =&\frac{1}{\Delta} \cdot (\boldsymbol{d}_0 + \boldsymbol{d}_1 \cdot \boldsymbol{sk})\\ =&\frac{1}{128} \cdot ([476, 185, 845, 0] + [74, 58, -47, -464] \cdot [-1, 1, 1, 0])\\ \approx&[3, 2, 6, 0].\end{aligned} \tag{56}$$

The result is correct, since the product of the plaintexts is equal to

$$\boldsymbol{m}_0 \cdot \boldsymbol{m}_1 = [1, 2, 3, 4] \cdot [3, 1, 2, 0] \approx [3, 2, 6, 0]. \tag{57}$$

A.4 Homomorphic matrix addition

Consider these two 3×3 matrices with integer entries

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 2 & 0 & 4 \\ 1 & 0 & 3 \end{bmatrix}, B = \begin{bmatrix} 3 & 1 & 0 \\ 2 & 4 & 1 \\ 0 & 2 & 3 \end{bmatrix}. \tag{58}$$

The column order representations of A, B are obtained as

$$\begin{aligned}\boldsymbol{a} =&[2, 2, 1, 3, 0, 0, 1, 4, 3],\\ \boldsymbol{b} =&[3, 2, 0, 1, 4, 2, 0, 1, 3].\end{aligned} \tag{59}$$

The ring $\mathcal{Z}_q^n$ with $q = 1472$ and $n = 9$ is used, and the parameters for the BFV algorithm are selected as

$$\begin{aligned}t =&23,\\ \Delta' =&\lfloor q/t \rfloor = 64,\\ \boldsymbol{sk} =&[-1, 0, -1, 1, 1, 0, 1, 0, 1],\\ \boldsymbol{w} =&[24, 17, -5, -10, 25, 41, -3, 12, 8],\\ \boldsymbol{e} =&[-1, -3, -3, -4, 4, 1, 1, -2, 2].\end{aligned} \tag{60}$$

We now compute $\boldsymbol{u} = -(\boldsymbol{w} \cdot \boldsymbol{sk} + \boldsymbol{e})$,

$$\begin{aligned} \boldsymbol{u} =& -([24,17,-5,-10,25,41,-3,12,8] \cdot [-1,0,-1,1,1,0,1,0,1] + [-1,-3,-3,-4,4,1,1,-2,2]),\\ =&[25,3,-2,14,-29,-1,2,2,-10]. \end{aligned} \tag{61}$$

The public key is the pair $\boldsymbol{pk} = (\boldsymbol{u}, \boldsymbol{w})$. To compute the relinearization key, we select $p = 150001$, $\boldsymbol{w}' = [43,-5,-102,61,34,90,-18,77,-52]$, and $\boldsymbol{e}' = [1,-3,3,-2,-4,-1,4,-2,-3]$. We compute $\boldsymbol{u}' = -(\boldsymbol{w}' \cdot \boldsymbol{sk} + \boldsymbol{e}') + p \cdot \boldsymbol{sk}^2$,

$$\begin{aligned} \boldsymbol{u}' =& -([43,-5,-102,61,34,90,-18,77,-52] \cdot [-1,0,-1,1,1,0,1,0,1]\\ &+[1,-3,3,-2,-4,-1,4,-2,-3]) + 150001 \cdot [1,0,1,1,1,0,1,0,1]\\ =&[150043,3,149896,149942,149971,1,150015,2,150056]. \end{aligned} \tag{62}$$

The relinearization key is the pair $\boldsymbol{rk} = (\boldsymbol{u}', \boldsymbol{w}')$. Encryption of $\boldsymbol{a} = \texttt{ColumnOrder}(A)$ is performed by first selecting the random numbers

$$\begin{aligned} \boldsymbol{r} =&[0,1,0,-1,1,1,0,1,1],\\ \boldsymbol{e}_0 =&[3,2,1,-1,3,-4,4,1,2],\\ \boldsymbol{e}_1 =&[-4,17,4,11,18,-40,5,2,-2]. \end{aligned} \tag{63}$$

The first component of the ciphertext is computed as $\boldsymbol{c}_{x0} = \boldsymbol{r} \cdot \boldsymbol{u} + \boldsymbol{a} \cdot \Delta' + \boldsymbol{e}_0$,

$$\begin{aligned} \boldsymbol{c}_{x0} =&[0,1,0,-1,1,1,0,1,1] \cdot [25,3,-2,14,-29,-1,2,2,-10]\\ &+ [2,2,1,3,0,0,1,4,3] \cdot 64 + [3,2,1,-1,3,-4,4,1,2]\\ =&[131,133,65,177,-26,-5,68,259,184]. \end{aligned} \tag{64}$$

The second component of the ciphertext is computed as $\boldsymbol{c}_{x1} = \boldsymbol{r} \cdot \boldsymbol{w} + \boldsymbol{e}_1$,

$$\begin{aligned} \boldsymbol{c}_{x1} =&[0,1,0,-1,1,1,0,1,1] \cdot [24,17,-5,-10,25,41,-3,12,8] + [-4,17,4,11,18,-40,5,2,-2]\\ =&[-4,34,4,21,43,1,5,14,6]. \end{aligned} \tag{65}$$

The ciphertext is $\boldsymbol{x} = (\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1}) = \text{Enc}_{\boldsymbol{pk}}(\boldsymbol{a})$.

On the other hand, the encryption of $\boldsymbol{b} = \texttt{ColumnOrder}(B)$ is obtained by first selecting the random numbers

$$\begin{aligned} \boldsymbol{r}' =&[1,-1,-1,0,1,-1,1,0,0],\\ \boldsymbol{e}'_0 =&[-2,-1,2,1,3,4,-3,-2,-1],\\ \boldsymbol{e}'_1 =&[2,-1,-1,-3,-4,-2,1,3,2]. \end{aligned} \tag{66}$$

The first component of the ciphertext is computed as $\boldsymbol{c}_{y0} = \boldsymbol{r}' \cdot \boldsymbol{u} + \boldsymbol{b} \cdot \Delta' + \boldsymbol{e}'_0$,

$$\begin{aligned} \boldsymbol{c}_{y0} =&[1,-1,-1,0,1,-1,1,0,0] \cdot [25,3,-2,14,-29,-1,2,2,-10]\\ &+ [3,2,0,1,4,2,0,1,3] \cdot 64 + [-2,-1,2,1,3,4,-3,-2,-1]\\ =&[215,124,4,65,230,133,-1,62,191]. \end{aligned} \tag{67}$$

The second component of the ciphertext is computed as $\boldsymbol{c}_{y1} = \boldsymbol{r}' \cdot \boldsymbol{w} + \boldsymbol{e}'_1$,

$$\begin{aligned}\boldsymbol{c}_{y1} =&[1,-1,-1,0,1,-1,1,0,0]\cdot[24,17,-5,-10,25,41,-3,12,8]\\ &+[2,-1,-1,-3,-4,-2,1,3,2]\\ =&[26,-18,4,-3,21,-43,-2,3,2].\end{aligned} \tag{68}$$

The ciphertext is $\boldsymbol{y} = (\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1}) = \text{Enc}_{\boldsymbol{pk}}(\boldsymbol{b})$.

The ciphertexts $\boldsymbol{x} = (\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1})$ and $\boldsymbol{y} = (\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1})$ are now homomorphically added

$$\boldsymbol{z} = \boldsymbol{x} \oplus \boldsymbol{y} \quad \rightarrow \quad (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1}) = (\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1}) \oplus (\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1}). \tag{69}$$

The components of $\boldsymbol{z} = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1})$ are obtained as

$$\begin{aligned}\boldsymbol{c}_{x0} + \boldsymbol{c}_{y0} =&[131,133,65,177,-26,-5,68,259,184]+[215,124,4,65,230,133,-1,62,191],\\ \boldsymbol{c}_{z0} =&[346,257,69,242,204,128,67,321,375],\\ \boldsymbol{c}_{x1} + \boldsymbol{c}_{y1} =&[-4,34,4,21,43,1,5,14,6]+[26,-18,4,-3,21,-43,-2,3,2],\\ \boldsymbol{c}_{z1} =&[22,16,8,18,64,-42,3,17,8].\end{aligned} \tag{70}$$

The homomorphic addition result is $\boldsymbol{z} = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1})$ To verify the correctness, $\boldsymbol{z}$ is decrypted using the secret key $\boldsymbol{sk} = [-1,0,-1,1,1,0,1,0,1]$ as follows:

$$\begin{aligned}\boldsymbol{d} &= \text{Dec}_{\boldsymbol{sk}}(\boldsymbol{z})\\ &=\frac{1}{\Delta'}\cdot(\boldsymbol{c}_{z0} + \boldsymbol{c}_{z1}\cdot\boldsymbol{sk})\\ &=\frac{1}{64}\cdot([346,257,69,242,204,128,67,321,375]\\ &\quad+[22,16,8,18,64,-42,3,17,8]\cdot[-1,0,-1,1,1,0,1,0,1])\\ &=\frac{1}{64}\cdot[324,257,61,260,268,128,70,321,383]\\ &\approx[5,4,1,4,4,2,1,5,6].\end{aligned} \tag{71}$$

The result is correct since the sum $A + B = D$ is computed as

$$D = \begin{bmatrix}2&3&1\\2&0&4\\1&0&3\end{bmatrix} + \begin{bmatrix}3&1&0\\2&4&1\\0&2&3\end{bmatrix} = \begin{bmatrix}5&4&1\\4&4&5\\1&2&6\end{bmatrix}. \tag{72}$$

The column ordering of the matrix D is obtained as

$$\boldsymbol{d} = [5,4,1,4,4,2,1,5,6]. \tag{73}$$

A.5 Homomorphic matrix multiplication

We consider these two 3×3 matrices with integer entries in Equation (58). Their column order representations are

$$\begin{aligned}\boldsymbol{a} =&[2,2,1,3,0,0,1,4,3],\\ \boldsymbol{b} =&[3,2,0,1,4,2,0,1,3].\end{aligned} \tag{74}$$

Here, we have $k = 3$ and $k^2 = 9$. Their ciphertexts are given as

$$\begin{aligned}\boldsymbol{x} = \ \mathrm{Enc}_{\boldsymbol{pk}}(\boldsymbol{a}) =&(\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1})\\ \boldsymbol{c}_{x0} =&[131,133,65,177,-26,-5,68,259,184]\\ \boldsymbol{c}_{x1} =&[-4,34,4,21,43,1,5,14,6],\\ \boldsymbol{y} = \ \mathrm{Enc}_{\boldsymbol{pk}}(\boldsymbol{b}) =&(\boldsymbol{c}_{y0}, \boldsymbol{c}_{y1})\\ \boldsymbol{c}_{y0} =&[215,124,4,65,230,133,-1,62,191]\\ \boldsymbol{c}_{y1} =&[26,-18,4,-3,21,-43,-2,3,2].\end{aligned} \tag{75}$$

The **first step** of `BFVMatrixMultiply` is to convert $\boldsymbol{x} = (\boldsymbol{c}_{x0}, \boldsymbol{c}_{x1})$ from column to diagonal order $\boldsymbol{d} = (\boldsymbol{d}_{x0}, \boldsymbol{d}_{x1}) = \texttt{Col2Diag}(\boldsymbol{c})$,

$$\begin{aligned}\boldsymbol{d}_{x0} =&[131,-26,184,133,-5,68,65,177,259],\\ \boldsymbol{d}_{x1} =&[-4,43,6,34,1,5,4,21,14].\end{aligned} \tag{76}$$

$\boldsymbol{d}_{x0}$ and $\boldsymbol{d}_{x1}$ are arranged into 3 vectors of length 3 as $\bar{d}_{x,0}$ and $\bar{d}_{x,1}$,

$$\boldsymbol{d}_{x0} = [\bar{d}_{x0,1}, \bar{d}_{x0,2}, \bar{d}_{x0,3}], \boldsymbol{d}_{x1} = [\bar{d}_{x1,1}, \bar{d}_{x1,2}, \bar{d}_{x1,3}]. \tag{77}$$

The vectors $\bar{d}_{x0,i}$ and $\bar{d}_{x1,i}$ are

$$\begin{aligned}\bar{d}_{x0,1} = [131,-26,184], \bar{d}_{x1,1} = [-4,43,6],\\ \bar{d}_{x0,2} = [133,-5,68], \bar{d}_{x1,2} = [34,1,5],\\ \bar{d}_{x0,3} = [65,177,259], \bar{d}_{x1,3} = [4,21,14].\end{aligned} \tag{78}$$

In the **second step**, the vectors $\vec{\boldsymbol{d}}_{x0i}$ and $\vec{\boldsymbol{d}}_{x1i}$ are generated as follows

$$\begin{aligned}\vec{\boldsymbol{d}}_{x0i} =&[R^{i-1}(\bar{d}_{x0,i}), R^{i-1}(\bar{d}_{x0,i}), R^{i-1}(\bar{d}_{x0,i})],\\ \vec{\boldsymbol{d}}_{x1i} =&[R^{i-1}(\bar{d}_{x1,i}), R^{i-1}(\bar{d}_{x1,i}), R^{i-1}(\bar{d}_{x1,i})],\\ \vec{\boldsymbol{d}}_{x01} =&[131,-26,184,131,-26,184,131,-26,184],\\ \vec{\boldsymbol{d}}_{x11} =&[-4,43,6,-4,43,6,-4,43,6],\\ \vec{\boldsymbol{d}}_{x02} =&[68,133,-5,68,133,-5,68,133,-5],\\ \vec{\boldsymbol{d}}_{x12} =&[5,34,1,5,34,1,5,34,1],\\ \vec{\boldsymbol{d}}_{x03} =&[177,259,65,177,259,65,177,259,65],\\ \vec{\boldsymbol{d}}_{x13} =&[21,14,4,21,14,4,21,14,4].\end{aligned} \tag{79}$$

During the **third step**, we start with the column ordering of the $\boldsymbol{c}_{y0}$ and $\boldsymbol{c}_{y1}$, and generate $\vec{\boldsymbol{c}}_{y0i}$ and $\vec{\boldsymbol{c}}_{y1i}$ for $i = 1, 2, 3$ as follows:

$$\begin{aligned} \boldsymbol{c}_{y0} &= [215, 124, 4, 65, 230, 133, -1, 62, 191], \\ \boldsymbol{c}_{y1} &= [26, -18, 4, -3, 21, -43, -2, 3, 2]. \end{aligned} \tag{80}$$

$\boldsymbol{c}_{y0}$ and $\boldsymbol{c}_{y1}$ are arranged into 3 vectors of length 3 as $\bar{c}_{y,0}$ and $\bar{c}_{y,1}$,

$$\boldsymbol{c}_{y0} = [\bar{c}_{y0,1}, \bar{c}_{y0,2}, \bar{c}_{y0,3}], \boldsymbol{c}_{y1} = [\bar{c}_{y1,1}, \bar{c}_{y1,2}, \bar{c}_{y1,3}]. \tag{81}$$

The vectors $\bar{c}_{y0,i}$ and $\bar{c}_{y1,i}$ are

$$\begin{aligned} \bar{c}_{y0,1} = [215, 124, 4], \bar{c}_{y1,1} &= [26, -18, 4], \\ \bar{c}_{y0,2} = [65, 230, 133], \bar{c}_{y1,2} &= [-3, 21, -43], \\ \bar{c}_{y0,3} = [-1, 62, 191], \bar{c}_{y1,3} &= [-2, 3, 2]. \end{aligned} \tag{82}$$

We then generate $\vec{\boldsymbol{c}}_{y0i}$ and $\vec{\boldsymbol{c}}_{y1i}$ for $i = 1, 2, 3$ as follows:

$$\begin{aligned} \vec{\boldsymbol{c}}_{y0i} &= [R^{i-1}(\bar{c}_{y0,1}), R^{i-1}(\bar{c}_{y0,2}), R^{i-1}(\bar{c}_{y0,3})], \\ \vec{\boldsymbol{c}}_{y1i} &= [R^{i-1}(\bar{c}_{y1,1}), R^{i-1}(\bar{c}_{y1,2}), R^{i-1}(\bar{c}_{y1,3})], \\ \vec{\boldsymbol{c}}_{y01} &= [215, 124, 4, 65, 230, 133, -1, 62, 191], \\ \vec{\boldsymbol{c}}_{y11} &= [26, -18, 4, -3, 21, -43, -2, 3, 2], \\ \vec{\boldsymbol{c}}_{y02} &= [4, 215, 124, 133, 65, 230, 191, -1, 62], \\ \vec{\boldsymbol{c}}_{y12} &= [4, 26, -18, -43, -3, 21, 2, -2, 3], \\ \vec{\boldsymbol{c}}_{y03} &= [124, 4, 215, 230, 133, 65, 62, 191, -1], \\ \vec{\boldsymbol{c}}_{y13} &= [-18, 4, 26, 21, -43, -3, 3, 2, -2]. \end{aligned} \tag{83}$$

Finally, the **fourth step** of `BFVMatrixMultiply` uses $\vec{\boldsymbol{d}}_{x0i}$ and $\vec{\boldsymbol{d}}_{x1i}$, and $\vec{\boldsymbol{c}}_{y0i}$ and $\vec{\boldsymbol{c}}_{y1i}$ for $i = 1, 2, 3$ and first computes z_1, z_2, z_3.

The computation of z_1,

$$
\begin{aligned}
z_1 =& (\vec{\boldsymbol{d}}_{x01}, \vec{\boldsymbol{d}}_{x11}) \odot (\vec{\boldsymbol{c}}_{y01}, \vec{\boldsymbol{c}}_{y11}),\\
\vec{\boldsymbol{d}}_{x01} =& [131, -26, 184, 131, -26, 184, 131, -26, 184],\\
\vec{\boldsymbol{d}}_{x11} =& [-4, 43, 6, -4, 43, 6, -4, 43, 6],\\
\vec{\boldsymbol{c}}_{y01} =& [215, 124, 4, 65, 230, 133, -1, 62, 191],\\
\vec{\boldsymbol{c}}_{y11} =& [26, -18, 4, -3, 21, -43, -2, 3, 2],\\
z'_{01} =& \frac{1}{\Delta'} \cdot \vec{\boldsymbol{d}}_{x01} \cdot \vec{\boldsymbol{c}}_{y01}\\
=& [440, -50, 12, 133, -93, 382, -2, -25, 549],\\
z'_{11} =& \frac{1}{\Delta'} \cdot (\vec{\boldsymbol{d}}_{x01} \cdot \vec{\boldsymbol{c}}_{y11} + \vec{\boldsymbol{c}}_{y01} \cdot \vec{\boldsymbol{d}}_{x11})\\
=& [40, 91, 12, -10, 146, -111, -4, 40, 24],\\
z'_{21} =& \frac{1}{\Delta'} \cdot \vec{\boldsymbol{d}}_{x11} \cdot \vec{\boldsymbol{c}}_{y11}\\
=& [-2, -12, 0, 0, 14, -4, 0, 2, 0],\\
z_{01} =& z'_{01} + \left\lfloor p^{-1} \cdot z'_{21} \cdot \boldsymbol{rk} \right\rceil\\
=& [438, -50, 12, 133, -79, 382, -2, -25, 549],\\
z_{11} =& z'_{11} + \left\lfloor p^{-1} \cdot z'_{21} \cdot \boldsymbol{rk} \right\rceil\\
=& [40, 91, 12, -10, 146, -111, -4, 40, 24],\\
z_1 =& (z_{01}, z_{11}).
\end{aligned}
\tag{84}
$$

The computation of z_2,

$$\begin{aligned}
z_2 =&(\vec{\boldsymbol{d}}_{x02}, \vec{\boldsymbol{d}}_{x12}) \odot (\vec{\boldsymbol{c}}_{y02}, \vec{\boldsymbol{c}}_{y12}),\\
\vec{\boldsymbol{d}}_{x02} =&[68, 133, -5, 68, 133, -5, 68, 133, -5],\\
\vec{\boldsymbol{d}}_{x12} =&[5, 34, 1, 5, 34, 1, 5, 34, 1],\\
\vec{\boldsymbol{c}}_{y02} =&[4, 215, 124, 133, 65, 230, 191, -1, 62],\\
\vec{\boldsymbol{c}}_{y12} =&[4, 26, -18, -43, -3, 21, 2, -2, 3],\\
z'_{02} =&\frac{1}{\Delta'} \cdot \vec{\boldsymbol{d}}_{x02} \cdot \vec{\boldsymbol{c}}_{y02}\\
=&[4, 447, -10, 141, 135, -18, 203, -2, -5],\\
z'_{12} =&\frac{1}{\Delta'} \cdot (\vec{\boldsymbol{d}}_{x02} \cdot \vec{\boldsymbol{c}}_{y12} + \vec{\boldsymbol{c}}_{y02} \cdot \vec{\boldsymbol{d}}_{x12})\\
=&[5, 168, 3, -35, 28, 2, 17, -5, 1],\\
z'_{22} =&\frac{1}{\Delta'} \cdot \vec{\boldsymbol{d}}_{x12} \cdot \vec{\boldsymbol{c}}_{y12}\\
=&[0, 14, 0, -3, -2, 0, 0, -1, 0],\\
z_{02} =&z'_{02} + \left\lfloor p^{-1} \cdot z'_{22} \cdot \boldsymbol{rk} \right\rceil\\
=&[4, 447, -10, 138, 133, -18, 203, -2, -5],\\
z_{12} =&z'_{12} + \left\lfloor p^{-1} \cdot z'_{22} \cdot \boldsymbol{rk} \right\rceil\\
=&[5, 168, 3, -35, 28, 2, 17, -5, 1],\\
z_2 =&(z_{02}, z_{12}).
\end{aligned} \tag{85}$$

The computation of z_3,

$$\begin{aligned}
z_3 =&(\vec{\boldsymbol{d}}_{x03}, \vec{\boldsymbol{d}}_{x13}) \odot (\vec{\boldsymbol{c}}_{y03}, \vec{\boldsymbol{c}}_{y13}),\\
\vec{\boldsymbol{d}}_{x03} =&[177, 259, 65, 177, 259, 65, 177, 259, 65],\\
\vec{\boldsymbol{d}}_{x13} =&[21, 14, 4, 21, 14, 4, 21, 14, 4],\\
\vec{\boldsymbol{c}}_{y03} =&[124, 4, 215, 230, 133, 65, 62, 191, -1],\\
\vec{\boldsymbol{c}}_{y13} =&[-18, 4, 26, 21, -43, -3, 3, 2, -2],\\
z'_{03} =&\frac{1}{\Delta'} \cdot \vec{\boldsymbol{d}}_{x03} \cdot \vec{\boldsymbol{c}}_{y03}\\
=&[343, 16, 218, 636, 538, 66, 171, 773, -1],\\
z'_{13} =&\frac{1}{\Delta'} \cdot (\vec{\boldsymbol{d}}_{x03} \cdot \vec{\boldsymbol{c}}_{y13} + \vec{\boldsymbol{c}}_{y03} \cdot \vec{\boldsymbol{d}}_{x13})\\
=&[-9, 17, 40, 134, -145, 1, 29, 50, -2],\\
z'_{23} =&\frac{1}{\Delta'} \cdot \vec{\boldsymbol{d}}_{x13} \cdot \vec{\boldsymbol{c}}_{y13}\\
=&[-6, 1, 2, 7, -9, 0, 1, 0, 0],\\
z_{03} =&z'_{03} + \left\lfloor p^{-1} \cdot z'_{23} \cdot \boldsymbol{rk} \right\rceil\\
=&[337, 16, 220, 643, 529, 66, 172, 773, -1],\\
z_{13} =&z'_{13} + \left\lfloor p^{-1} \cdot z'_{23} \cdot \boldsymbol{rk} \right\rceil\\
=&[-9, 17, 40, 134, -145, 1, 29, 50, -2],\\
z_3 =&(z_{03}, z_{13}).
\end{aligned} \tag{86}$$

These ciphertexts are added to obtain the ciphertext $\boldsymbol{z} = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1})$,

$$\begin{aligned}
(\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1}) =&z_1 \oplus z_2 \oplus z_3\\
=&(z_{01}, z_{11}) \oplus (z_{02}, z_{12}) \oplus (z_{03}, z_{13})\\
=&(z_{01} + z_{02} + z_{03}, z_{11} + z_{12} + z_{13}),\\
\boldsymbol{c}_{z0} =&z_{01} + z_{02} + z_{03},\\
\boldsymbol{c}_{z1} =&z_{11} + z_{12} + z_{13}.
\end{aligned} \tag{87}$$

Here, $\boldsymbol{z}$ is the encryption of the matrix product D in the column order. $\boldsymbol{z} = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1})$ is computed as

$$\begin{aligned}
\boldsymbol{c}_{z0} =&[438, -50, 12, 133, -79, 382, -2, -25, 549] + [4, 447, -10, 138, 133, -18, 203, -2, -5]+\\
&+ [337, 16, 220, 643, 529, 66, 172, 773, -1]\\
=&[779, 413, 222, 914, 583, 430, 373, 746, 543],\\
\boldsymbol{c}_{z1} =&[40, 91, 12, -10, 146, -111, -4, 40, 24] + [5, 168, 3, -35, 28, 2, 17, -5, 1]+\\
&+ [-9, 17, 40, 134, -145, 1, 29, 50, -2]\\
=&[36, 276, 55, 89, 29, -108, 42, 85, 23].
\end{aligned} \tag{88}$$

The decryption of the ciphertext $z = (\boldsymbol{c}_{z0}, \boldsymbol{c}_{z1})$ using the secret key is obtained as

$$\begin{aligned}\mathrm{Dec}_{\boldsymbol{sk}}(z) =& \frac{1}{\Delta'} \cdot (\boldsymbol{c}_{z0} + \boldsymbol{c}_{z1} \cdot \boldsymbol{sk}) \\ =& \frac{1}{64} \cdot ([779, 413, 222, 914, 583, 430, 373, 746, 543] + \\ & + [36, 276, 55, 89, 29, -108, 42, 85, 23] \cdot [-1, 0, -1, 1, 1, 0, 1, 0, 1]) \\ \approx& [12, 6, 3, 16, 10, 7, 6, 12, 9].\end{aligned} \tag{89}$$

To verify the result, we show that it is equal to the column ordering of the product $D = A \cdot B$. To show that, the matrix product $D = A \cdot B$ is computed as

$$D = \begin{bmatrix} 2 & 3 & 1 \\ 2 & 0 & 4 \\ 1 & 0 & 3 \end{bmatrix} \cdot \begin{bmatrix} 3 & 1 & 0 \\ 2 & 4 & 1 \\ 0 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 12 & 16 & 6 \\ 6 & 10 & 12 \\ 3 & 7 & 9 \end{bmatrix}. \tag{90}$$

The column ordering of the matrix D is indeed equal to

$$\boldsymbol{d} = [12, 6, 3, 16, 10, 7, 6, 12, 9]. \tag{91}$$

Acknowledgements This research is supported in part by the Jiangsu Province 100 Foreign Experts Introduction Plan BX2022012. This research is also supported in part by TÜBİ TAK Projects 2232-118C332 and 1001-121F348.

Author contributions This manuscript is part of the PhD thesis of Shang Ci. The author Cetin Kaya Koc is the PhD supervisor of Shang Ci. The authors Shang Ci and Cetin Kaya Koc designed the algorithms and wrote most of the paper. The other 3 authors Yihan Wang, Sen Hu, and Donghai Guan made contributions by analyzing and implementing the algorithms, and contributing to the writing of the manuscript. We used no AI in designing the algorithms or writing the manuscript.

Data availability No datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare no Conflict of interest.

References

1. Atallah MJ, Pantazopoulos KN, Rice JR, Spafford EH (1999) Secure outsourcing of scientific computations. Preprint
2. Atallah MJ, Pantazopoulos KN, Rice JR, Spafford EH (2002) Secure outsourcing of scientific computations. Adv Comput 54:215–272
3. Seitkulov YN (2013) New methods of secure outsourcing of scientific computations. J Supercomput 65:469–482
4. Koç ÇK, Özdemir F, Özger ZÖ (2021) Partially homomorphic encryption. Springer, Switzerland
5. Gentry C (2009) A fully homomorphic encryption scheme. PhD thesis, Stanford University
6. Baseri Y, Chouhan V, Ghorbani AA, Chow A (2025) Evaluation framework for quantum security risk assessment: a comprehensive strategy for quantum-safe transition. Comput Secur 150:104272. https://doi.org/10.1016/J.COSE.2024.104272

7. Smart NP, Vercauteren F (2010) Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Public-Key Cryptography—PKC 2010, pp 420–443
8. Smart NP, Vercauteren F (2014) Fully homomorphic SIMD operations. Des Codes Crypt 71:57–81
9. Halevi S, Shoup V (2014) Algorithms in HElib. In: Garay JA, Gennaro R (eds) Advances in Cryptology—CRYPTO 2014, pp 554–571. Springer, LNCS Nr. 8616
10. Gentry C, Halevi S (2011) Implementing Gentry's fully-homomorphic encryption scheme. In: Advances in Cryptology, Eurocrypt, pp 129–148. Springer, LNCS Nr. 6632
11. Halevi S, Shoup V (2013) Homomorphic Encryption library (HElib). https://en.wikipedia.org/wiki/HElib
12. Sathya SS, Vepakomma P, Raskar R, Ramachandra R, Bhattacharya S (2018) A Review of Homomorphic Encryption Libraries for Secure Computation. arXiv:1812.02428
13. Brakerski Z, Gentry C, Vaikuntanathan V (2011) Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Paper 2011/277. https://eprint.iacr.org/2011/277
14. Brakerski Z, Gentry C, Vaikuntanathan V (2012) (Leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp 309–325
15. Brakerski Z, Gentry C, Halevi S (2013) Packed ciphertexts in LWE-based homomorphic encryption. In: Public-Key Cryptography—PKC 2013, pp 1–13
16. Yasuda M, Shimoyama T, Kogure J, Yokoyama K, Koshiba T (2015) Secure statistical analysis using RLWE-based homomorphic encryption. In: ACISP 2015, pp 471–487. Springer, LNCS Nr. 9144
17. Lu W-J, Kawasaki S, Sakuma J (2016) Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data. Cryptology ePrint Archive, Paper 2016/1163. https://eprint.iacr.org/2016/1163
18. Rathee D, Mishra PK, Yasuda M (2018) Faster PCA and linear regression through hypercubes in HElib. In: Proceedings of the 2018 Workshop on Privacy in the Electronic Society, pp 42–53
19. Jiang X, Kim M, Lauter K, Song Y (2018) Secure outsourced matrix computation and application to neural networks. In: ACM SIGSAC Conference on Computer and Communications Security, pp 1209–1222
20. Mishra PK, Duong DH, Yasuda M (2017) Enhancement for secure multiple matrix multiplications over Ring-LWE homomorphic encryption. In: Liu JK, Samarati P (eds) Information Security Practice and Experience. Springer, pp 320–330, LNCS Nr. 8616
21. Wang S, Huang H (2019) Secure outsourced computation of multiple matrix multiplication based on fully homomorphic encryption. KSII Trans Internet Inf Syst 13(1):5616–5630
22. Rizomiliotis P, Triakosia A (2022) On matrix multiplication with homomorphic encryption. In: Proceedings of the 2022 on Cloud Computing Security Workshop, pp 53–61
23. Huang H, Zong H (2023) Secure matrix multiplication based on fully homomorphic encryption. J Supercomput 79(5):5064–5085
24. Gao Y, Quan G, Homsi S, Wen W, Wang L (2024) Secure and efficient general matrix multiplication on cloud using homomorphic encryption. arXiv:2405.02238
25. Aikata A, Roy SS (2024) Secure and efficient outsourced matrix multiplication with homomorphic encryption. Cryptology ePrint Archive, Paper 2024/1730. https://eprint.iacr.org/2024/1730
26. Zheng X, Li H, Wang D (2025) A new framework for fast homomorphic matrix multiplication. Des Codes Cryptogr 93(7):2671–2693. https://doi.org/10.1007/S10623-025-01614-Y
27. Fan J, Vercauteren F (2012) Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144
28. Ibarrondo A, Viand A (2021) Pyfhel: Python for homomorphic encryption libraries. In: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, pp 11–16
29. Albrecht M, Chase M, Chen H, Ding J, Goldwasser S, Gorbunov S, Halevi S, Hoffstein J, Laine K, Lauter K, Lokam S, Micciancio D, Moody D, Morrison T, Sahai A, Vaikuntanathan V (2018) Homomorphic encryption security standard. HomomorphicEncryption.org, Toronto, Canada, Technical report
30. Corporation I (2021) Intel to Collaborate with Microsoft on DARPA Program. https://www.intc.com/news-events/press-releases/detail/1445/intel-to-collaborate-with-microsoft-on-darpa-program

Authors and Affiliations

Shang Ci[1] · Yihan Wang[1] · Sen Hu[1] · Donghai Guan[1] · Çetin Kaya Koç[1,2,3]

✉ Çetin Kaya Koç
cetinkoc@ucsb.edu

Shang Ci
cishang@nuaa.edu.cn

Yihan Wang
yihanwang@nuaa.edu.cn

Sen Hu
hu_sen@nuaa.edu.cn

Donghai Guan
dhguan@nuaa.edu.cn

1 Nanjing University of Aeronautics and Astronautics, Nanjing, China

2 Düzce University, Düzce, Turkey

3 University of California Santa Barbara, Santa Barbara, USA