Homomorphic Matrix Operations under Bicyclic Encoding

Jingwei Chen, Linhan Yang, Wenyuan Wu, Yang Liu, Yong Feng

Abstract

Homomorphically encrypted matrix operations are extensively used in various privacy-preserving applications. Consequently, reducing the cost of encrypted matrix operations is a crucial topic on which numerous studies have been conducted. In this paper, we introduce a novel matrix encoding method, named *bicyclic encoding*, under which we propose two new algorithms BMM-I and BMM-II for encrypted matrix multiplication. BMM-II outperforms the stat-of-the-art algorithms in theory, while BMM-I, combined with the *segmented* strategy, performs well in practice, particularly for matrices with high dimensions. Another noteworthy advantage of bicyclic encoding is that it allows for transposing an encrypted matrix entirely free. A comprehensive experimental study based on our proof-of-concept implementation shows that each algorithm introduced in this paper has specific scenarios outperforming existing algorithms, achieving speedups ranging from 2x to 38x.

Index Terms

Secure matrix multiplication, fully homomorphic encryption, SIMD, bicyclic encoding.

I. Introduction

Privacy-preserving computing or privacy-enhancing technologies, capable of protecting data privacy and fully exploiting data value, is an exceptionally popular area of research. Fully Homomorphic Encryption (FHE) enables computations to be performed on encrypted data without the need for decryption [1], [2], and hence offering a powerful tool for privacy-preserving computation. Among the numerous privacy-preserving applications enabled by homomorphic encryption, matrix operations emerge as a core fundamental. Therefore, the importance of matrix multiplication over encrypted data is self-evident. In this paper, we investigate matrix multiplication over data encrypted using an FHE scheme that supports SIMD (Single Instruction Multiple Data), such as the BGV [3] and B/FV [4], [5] schemes for integer arithmetic, and the CKKS scheme [6] for approximate arithmetic.

Since Gentry's pioneering work [2], FHE has rapidly developed, giving rise to various schemes such as BGV [3], B/FV [4], [5], CKKS [6], FHEW [7], TFHE [8], and numerous optimizations like data packing for SIMD [9], bootstrapping [10]–[13], etc. However, homomorphic matrix multiplication remains a bottleneck in practice. For example, Huang et al. reported in [14] that it takes nearly 6 minutes for an encrypted matrix multiplication with dimensions 2048×8 and 8×2048.

Almost all existing work about homomorphic matrix multiplication employed schemes that support SIMD, i.e., BGV, B/FV, or CKKS. For these schemes, computational efficiency is primarily influenced by two key factors. The first is multiplicative depth (including *ciphertext-ciphertext multiplication* (Mul) and *plaintext-ciphertext multiplication* (CMul)), a metric that directly impacts the computational efficiency of all known FHE schemes. This is because more multiplicative depths imply a larger ciphertext modulus or an increased number of bootstrappings. The second is the number of required *ciphertext rotation*s (Rot) on the packed ciphertext. According to our test, for the CKKS scheme implemented in Microsoft SEAL [15], the efficiency ratio between a ciphertext rotation and a ciphertext multiplication can be as large as $8:1$. Similar observations can be found in [16] as well. Consequently, the primary objective of this paper is to optimize both the multiplicative depth and the required number of ciphertext rotations for encrypted matrix multiplication.

A. Related Work

Secure matrix multiplication has plenty of applications, making it a highly active and impactful research area in privacy-preserving computation. The subject offers diverse perspectives for investigation, including secure multi-party computation [17]–[21], information-theoretically privacy [22], etc. Here, we mainly focus on encrypted matrix multiplication methods based on FHE schemes supporting SIMD, although there exist many other algorithms, e.g., [23], [24], based on a matrix version of GSW [25].

Jingwei Chen, Wenyuan Wu and Yong Feng are with Chongqing Key Laboratory of Secure Computing for Biology, Chongqing Institute of Green and Intelligent Technology, CAS and Chongqing College, University of Chinese Academy of Sciences. Email: {chenjingwei, wuwenyuan, yongfeng}@cigit.ac.cn

Linhan Yang (corresponding author) and Yang Liu are with School of Information Science and Engineering, Chongqing Jiaotong University. This work was done when Linhan Yang was visiting Chongqing Key Laboratory of Secure Computing for Biology, Chongqing Institute of Green and Intelligent Technology, CAS. Email: linhanyang@mails.cqjtu.edu.cn, liuyang13@cqjtu.edu.cn

This work was partially supported by National Key Research and Development Program of China (2020YFA0712303), Natural Science Foundation of Chongqing (2022yszx-jcx0011cstb, cstb2023yszx-jcx0008, cstb2023nscq-msx0441), and Western Young Scholars Program of CAS.

A plaintext of an FHE scheme that supports SIMD is usually an element in a certain polynomial ring, say $R = \mathbb{Z}[X]/\langle X^N + 1\rangle$. Polynomials in R can be represented in two equivalent ways. One is referred to *coefficients-encoding*, corresponding to the coefficient vector of the polynomial. The other is *slots-encoding*, i.e., the polynomial evaluations at certain ℓ points, where ℓ is the number of slots the FHE scheme supports. Roughly speaking, all existing algorithms for encrypted matrix multiplication can be categorized into two types, both of which support SIMD operations.

1) Coefficients-encoding algorithms: Duong et al. [26] generalize Yasuda et al.'s method for secure inner product [27], [28], and present two algorithms for encrypted matrix multiplication. Suppose that a ciphertext encodes a vector of dimension at most ℓ. Then the two algorithms of Duong et al. support maximal dimensions of $O(\ell^{1/2})$ and $O(\ell^{1/3})$, and require $O(\ell^{1/2})$ and only one Mul, respectively. The drawback of this method is its lack of efficiency in handling consecutive matrix multiplications. Later on, Mishra et al. [29] extended it to support k matrix multiplications successively. However, the maximal dimension is limited only to $O(\ell^{1/(k+1)})$, which makes it impractical. Zheng et al. [30] recently proposed a new framework for homomorphic matrix multiplication under BGV, which supports consecutive matrix multiplication and requires only constant Muls and CMuls, and $O(\log d)$ Rots for square matrix multiplication of dimension d when $d = O(\sqrt[3]{N})$, achieving the best theoretical complexity bound, where N is the ring dimension used in BGV. Zheng et al. also presented a Strassen variant [31] for matrices with large dimensions. However, since their algorithm relies on a hypercube structure of the plaintext space, it does not support B/FV or CKKS. In addition, for all the coefficient encoded methods, to reuse partial entries of the encrypted resulting matrix, we are typically compelled to resort to extra operations for switching between the coefficients and slots encoding, which may slow down the computation.

TABLE I
THE COST OF SLOTS-ENCODING ALGORITHMS FOR A (n, m, p) MATRIX MULTIPLICATION OVER ENCRYPTED DATA, WHERE THE TWO MATRICES TO BE MULTIPLIED ARE OF DIMENSIONS $n \times m$ AND $m \times p$, RESPECTIVELY, AND ℓ IS THE NUMBER OF PLAINTEXT SLOTS.

Method	Max. dim.	#Ctxts	#Mul	#CMul	#Rot	#Mult. depth
[10], [32]*	ℓ	(d, p)	pd	0	$2pd^{\frac{1}{2}}$	1 Mul
[33], [34]	ℓ	(n, m)	mn	mn	$mn \log p$	1 Mul + 1 CMul
[35]†	$\sqrt{\ell}$	$(1, 1)$	d	d	$d \log d + d$	1 Mul + 1 CMul
[36]†	$\sqrt{\ell}$	$(1, 1)$	d	$5d$	$3d + 5d^{\frac{1}{2}}$	1 Mul + 2 CMul
[37]†	$\sqrt{\ell}$	$(1, 1)$	d	$2d$	$2d + 4d^{\frac{1}{2}}$	1 Mul + 1 CMul
[38]†	$\sqrt[3]{\ell}$	$(1, 1)$	1	$2d$	$2d + 3 \log d - 2$	1 Mul + 1 CMul
[39], [40]‡	$\sqrt{\ell}$	$(1, 1)$	m	m	$m \log d + m$	1 Mul + 1 CMul
[41]*	$O(\sqrt{\ell})$	$(1, 1)$	d	$2d$	$4d$	1 Mul + 1 CMul
[42]†	$O(\sqrt{\ell})$	$(1, 1)$	1	d	$4d$	1 Mul + 1 CMul
BMM-I§	$\sqrt{\ell/2}$	$(1, 1)$	m	0	$2m + 2$	1 Mul
BMM-II†, **	$\sqrt[3]{\ell}$	$(1, 1)$	1	0	$3 \log d$	1 Mul

* $d = \max(n, m)$. † $d = \max(n, m, p)$. ‡ $d = \max(m, p)$.
§ (n, m, p) are pairwise coprime and $\max(n, p) < m$.
** (n, m, p) are pairwise coprime and m is a power-of-two integer.

2) Slots-encoding algorithms: There are also algorithms that encode matrix data in evaluations of plaintext polynomials, which naturally support consecutive matrix multiplication. We summarize these algorithms in Table I. Halevi and Shoup investigate linear transformation on encrypted vector [10], [32], i.e., matrix-vector multiplication. Their methods can be directly extended to matrix multiplication. Lu et al. [33] and Wang and Huang [34] extended Halevi and Shoup's method for matrix-matrix multiplication based on the row-order and column-order encoding methods, respectively. Rathee et al. [35] considered an encrypted version of a matrix multiplication algorithm presented in [43]. Jiang et al. presented an algorithm for matrix multiplication over encrypted data in [36]. It uses SIMD operations and the technique for linear transformation [10], [32]. A recent survey [44] identifies Jiang et al.'s algorithm [36] as the state-of-the-art for FHE-based matrix multiplication. Based on Jiang et al.'s algorithm [36], Huang et al. [14] improved the block matrix multiplication for rectangular matrices with special shapes. Chiang [39] and Huang and Zong [40] presented a scheme for non-square matrices, which can be considered as a generalization and optimization of [35]. Zhu et al. [41] have made further improvements based on the method in [40]. Jang et al. [37] presented an adapted CKKS scheme to support data with tensor structure better and improved Jiang et al.'s algorithm [36] in the number of required Rots and CMuls. However, the security of Jang et al.'s variant of CKKS is based on a non-standard hardness assumption called multivariate polynomial learning with errors (m-RLWE). Rizomiliotis and Triakosia [38] introduced a new method for matrix multiplication over encrypted data. This method fully leverages packing techniques, reducing the required number of Muls to just one. However, it only supports matrix dimensions $\ell^{1/3}$.

3) Other optimizations: In addition to optimizing the number of rotations, another optimization direction is to accelerate Rot itself. For example, the hoisting technique proposed in [10] optimizes scenarios involving multiple rotations on the same ciphertext, and the double hoisting introduced by Bossuat et al. [45] makes further progress. These techniques were recently used to accelerate related matrix operations in PCA (Principal Component Analysis) [46]. Gao and Gao [42] construct a fully homomorphic encryption scheme, named GMS, for matrices based on the n-secret LWE assumption [47], which is suitable for matrix multiplication. For encrypted matrices with large dimensions, the Strassen algorithm [31] has been applied recently to this area in [30], [48], [49].

B. Contribution

Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be two matrices with dimensions $n \times m$ and $m \times p$, respectively. Denote by (n, m, p) *matrix multiplication* the multiplication between $\boldsymbol{A}$ and $\boldsymbol{B}$.

1) For small-dimensional matrices: We prove the following

Theorem 1. *Let (n, m, p) be pairwise coprime integers and ℓ the number of slots that the FHE scheme supports.*

- *If $\ell > 2 \cdot \max\{mn, mp, np\}$, there exists an algorithm (*BMM-I*) that computes a homomorphically encrypted (n, m, p) matrix multiplication within m ciphertext-ciphertext multiplications (*Mul*s), and $2(m+\log\lceil p/m\rceil+\log\lceil n/m\rceil+1)$ rotations on ciphertexts (*Rot*s), and costs only one* Mul *multiplicative depth.*
- *If $\ell > mnp$, there exists an algorithm (*BMM-II*) that computes a homomorphically encrypted (n, m, p) matrix multiplication with only one* Mul *and* CMul*, and at most $\log\lceil m\rceil + \log\lceil n\rceil + \log\lceil p\rceil$* Rot*s, and costs one* Mul *and one* CMul *multiplicative depth. In particular, if m is a power-of-two integer, the algorithm can finish the computation with only one* Mul *and at most $\log\lceil m\rceil + \log\lceil n\rceil + \log\lceil p\rceil$* Rot*s, without* CMul*, and hence costs only one* Mul *multiplicative depth.*

Both BMM-I and BMM-II support all SIMD-supported FHE schemes (e.g., BGV, B/FV, CKKS) and features the following:

a) The required number of ciphertext operations: As indicated in Table I, BMM-I and BMM-II has their own advantages on the number of ciphertext operations. In particular, BMM-II requires only one Mul and $O(\log d)$ Rots without CMul, better than all other existing algorithms in Table I. Although a similar result can be achieved by Zheng et al.'s algorithm [30], theirs only apply to BGV, not BFV or CKKS.

b) Multiplicative depth: Both BMM-I and BMM-II can achieve optimal multiplicative depth, i.e., only one Mul depth. Compared to Halevi-Shoup's method [10], [32], our approach utilizes fewer ciphertexts. For instance, our computation results in a single ciphertext, while theirs produces p ciphertexts.

c) Transpose for free: Our algorithms rely on a novel matrix encoding method given in Section III-A. As a benefit of this encoding, the transpose of an encrypted matrix can be computed for completely free; see Corollary 7. In previous algorithms, e.g., [36], the transpose of an encrypted matrix is reduced to a higher-dimensional linear transformation. This feature is expected to accelerate those applications involving matrix transpose, e.g., computing the covariance matrix in PCA, backpropagation in deep learning, etc.

2) Matrix with high dimension: BMM-I (resp. BMM-II) has a limitation: It requires $\ell > 2 \cdot \max\{mn, mp, np\}$ (resp. $\ell > mnp$). Assuming $n \approx m \approx p$ gives $n \approx \sqrt{\ell/2}$ (resp. $n \approx \sqrt[3]{\ell}$), while Jiang et al.'s algorithm [36] supports encrypted matrix multiplication of dimension $\sqrt{\ell}$. Thus, for handling high-dimensional matrices in a block-wise manner, the number of blocks would be a bit more than that of some existing algorithms, e.g., Jiang et al.'s [36]. This may lead to a lower efficiency of the block-wise algorithms based on BMM-I or BMM-II. To address this problem, we fully exploit the properties of our novel encoding method (on which BMM-I and BMM-II depend) and introduce a segmented version of BMM-I for multiplying high-dimensional matrices (BMM-III), where the utilization rate of slots achieves nearly 100%, similar to, e.g., Jiang et al.'s algorithm. This leads to our second result:

Theorem 2. *Assume that (n, m, p) are pairwise coprime integers and ℓ is the number of slots the FHE scheme supports. Then there exists an algorithm (*BMM-III*) that computes the homomorphically encrypted (n, m, p) matrix multiplication within $m \cdot \left\lceil \frac{np}{\ell} \right\rceil$ ciphertext-ciphertext multiplications (*Mul*), and $2m \cdot \left\lceil \frac{np}{\ell} \right\rceil$ rotations on ciphertexts (*Rot*), $(4 \cdot \left\lceil \frac{np}{\ell} \right\rceil + 2)m + n + p$ plaintext-ciphertext multiplications (*CMul*), and costs only one* Mul *and one* CMul *multiplicative depth.*

As a consequence, BMM-III reduces the number of Rots by a factor $\frac{1}{3}$ and saves one depth of CMul compared with the state-of-the-art algorithm for high-dimensional encrypted rectangular matrices [14]. Furthermore, it depends on a so-called *segmented* matrix multiplication, which is different from the traditional block matrix multiplication.

3) Experimental study: We implement all BMM-I, BMM-II and BMM-III with CKKS in SEAL [15], including the naïve (textbook) block version and the Strassen [31] version of BMM-I as well. The code is available at https://github.com/hangenba/bicyclic_mat_mul. A comprehensive experimental study demonstrates the performances of these algorithms and identifies how to select different algorithms for different cases to achieve optimal efficiency. In particular, our implementation of BMM-III can compute a $(2048, 8, 2048)$ encrypted matrix multiplication in about 81s, achieving a 2.6x speedup compared with the state-of-the-art algorithm for rectangular matrices [14], and for a task with dimension $(1024, 1024, 1024)$, it costs about 1200s, 5x faster than the block version of Jiang et al.'s algorithm [36]; BMM-II can compute a $(15, 16, 17)$ encrypted matrix multiplication in about 13ms, about 16x faster than the algorithm presented in [38]. SegLKS (obtained from applying the segment strategy to an algorithm presented by Lu et al. [33]) with some optimizations can compute a $(32, 33, 13847)$ encrypted matrix multiplication within 8.24s, about 38x faster the block algorithm based on [36].

C. Technique Overview

Our results mainly rely on two techniques: bicyclic encoding for matrices and segmented matrix multiplication.

1) Bicyclic encoding: We first define a novel encoding map that identifies an $n \times m$ matrix as a vector of dimension mn, provided n and m coprime. We call it the *bicyclic encoding*, which can be roughly viewed as an extension of the diagonal vector of a matrix employed in Halevi-Shoup's algorithm [32]. It follows from the Chinese Remainder Theorem (CRT) that the coprime restriction on n and m guarantees that $\mathbb{Z}/(mn\mathbb{Z}) \cong \mathbb{Z}/(n\mathbb{Z}) \otimes \mathbb{Z}/(m\mathbb{Z})$, which implies that a single vector is enough to traverse all elements of an $n \times m$ matrix. For instance, the bicyclic encoding for

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix} \text{ and } \boldsymbol{B} = \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \\ J & K & L \\ M & N & O \end{pmatrix} \tag{1}$$

are $\boldsymbol{a} = (0, 6, 2, 8, 4, 5, 1, 7, 3, 9)$ and $\boldsymbol{b} = (A, E, I, J, N, C, D, H, L, M, B, F, G, K, O)$, respectively. In particular, the k-th entry of $\boldsymbol{a}$ for $k = 0, 1, \ldots, 9$ is the (i, j)-entry of $\boldsymbol{A}$ with $i = k \mod 2$ and $j = k \mod 5$.

TABLE II
An illustrative example for Algorithm 1, where s_a (resp. s_b) is the step size for rotating $\boldsymbol{a}$ (resp. $\boldsymbol{b}$) to the left.

i	(s_a, s_b)	$\boldsymbol{a}_i$	$\boldsymbol{b}_i$	$\boldsymbol{a}_i \odot \boldsymbol{b}_i$
0	(0, 0)	$(0,6,2,8,4,5)$	(A,E,I,J,N,C)	$(0A,6E,2I,8J,4N,5C)$
1	(8, 3)	$(3,9,0,6,2,8)$	(J,N,C,D,H,L)	$(3J,9N,0C,6D,2H,8L)$
2	(6, 6)	$(1,7,3,9,0,6)$	(D,H,L,M,B,F)	$(1D,7H,3L,9M,0B,6F)$
3	(4, 9)	$(4,5,1,7,3,9)$	(M,B,F,G,K,O)	$(4M,5B,1F,7G,3K,9O)$
4	(2,12)	$(2,8,4,5,1,7)$	(G,K,O,A,E,I)	$(2G,8K,4O,5A,1E,7I)$

Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be two matrices to be multiplied, with dimensions (n, m) and (m, p), respectively, satisfying (n, m, p) pairwise coprime and $m > \max(n, p)$, where the latter condition is just for simplicity and will be removed in our main algorithm. Assume that vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ are obtained via the bicyclic encoding of $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. Then the bicyclic encoding of the resulting matrix $\boldsymbol{X} = \boldsymbol{AB}$ is exactly $\sum_{0 \le i < m} \boldsymbol{a}_i \odot \boldsymbol{b}_i$, where $\odot$ denotes component-wise multiplication, and $\boldsymbol{a}_i$ and $\boldsymbol{b}_i$ are of dimension np obtained by rotating the original $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively. Taking $\boldsymbol{A}$ and $\boldsymbol{B}$ as in Eq. (1), i.e., $(n, m, p) = (2, 5, 3)$, we give in Table II an illustrative example for a matrix multiplication algorithm (Algorithm 1) under bicyclic encoding. We should note that the number of rotation positions for $\boldsymbol{a}_i$ and $\boldsymbol{b}_i$ is nontrivial; see Section III-B for details. Clearly, this algorithm requires 5 component-wise vector multiplications and 8 vector rotations.

2) Segmented matrix multiplication: For matrices with high dimension, the usual approach involves block recursive algorithms such as the Strassen algorithm [31]. However, when applied to encrypted matrix multiplication, using a recursive algorithm leads to a significant memory overhead, while rewriting the recursive algorithm into a loop increases the required multiplication depth. To address this problem, we introduce a technique called *segmented matrix multiplication*.

The basic idea of segmented matrix multiplication is to implement encrypted matrix multiplication strictly following the plaintext algorithm for matrix multiplication under bicyclic encoding. Continuing with the matrices from equation (1) as an example, suppose the number of plaintext slots $\ell = 2$. Then, the bicyclic encoding of $\boldsymbol{A}$ will be encrypted into five ciphertexts $(\mathsf{ct}.\boldsymbol{a}_i)_{0 \le i < 5}$ corresponding to encryptions of $(0, 6)$, $(2, 8)$, $(4, 5)$, $(1, 7)$ and $(3, 9)$. The primary task then becomes how to perform rotations on these ciphertexts. To fulfill this function, we design a subroutine called *Long Rotation* (LongRot). For instance, running a LongRot on $(\mathsf{ct}.\boldsymbol{a}_i)_{0 \le i < 5}$ with step size one (i.e., rotating one position towards left) returns ciphertexts of $(6, 2)$, $(8, 4)$, $(5, 1)$, $(7, 3)$ and $(9, 0)$. Based on LongRot, we present BMM-III, which supports high-dimensional encrypted matrix multiplication. For encrypted rectangular matrix multiplication, BMM-III reduces the number of Rots by a factor $\frac{1}{3}$ and saves one CMul depth, compared with the state-of-the-art algorithm [14]. The experiment in Section VII shows that BMM-III is efficient for high-dimensional matrix multiplication. Applying the segmented technique with several optimizations to an algorithm in [33] leads to an algorithm SegLKS requiring the least number of Rots asymptotically among all existing algorithms.

Outline: In Section II, we provide the necessary preliminaries. In Section III, we introduce the bicyclic encoding method and discuss matrix operations under this encoding, including matrix transpose, matrix multiplication, and switching between bicyclic encoding and the commonly used row/column encoding. In Section IV, we present the encrypted version of the aforementioned matrix multiplication algorithms and analyze their cost. In Section V, we present the encrypted segmented matrix multiplication and apply the segmented strategy to Lu et al.'s algorithm [33] in Section VI. We present a comprehensive experimental study in Section VII and conclude with Section VIII.

II. Homomorphic Operations

A typical FHE scheme consists of the following algorithms:

- $\mathsf{Setup}(1^\lambda)$. Given a security parameter λ, output parms.

- $\mathsf{KeyGen}(\mathsf{parms})$. Output a secret key $\mathsf{sk} = \boldsymbol{s}$ and the corresponding public key pk. (For convenience, we also let pk include one or more evaluation keys.)
- $\mathsf{Enc}_{\mathsf{pk}}(b)$. Given a message $b \in \mathcal{M}$, output a ciphertext $c \in \mathcal{C}$, where $\mathcal{M}$ and $\mathcal{C}$ are the plaintext space and ciphertext space, respectively.
- $\mathsf{Dec}_{\mathsf{sk}}(\boldsymbol{c})$. Given $c \in \mathcal{C}$ as input, output $b \in \mathcal{M}$.
- $\mathsf{Eval}_{\mathsf{pk}}(f, (c_1, \cdots, c_k))$. Given a function f in k variables, and $(c_i)_{i \le k}$ with $c_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(b_i)$, output $c \in \mathcal{C}$ such that $\mathsf{Dec}_{\mathsf{sk}}(c) \neq f(b_1, \cdots, b_k)$ holds with negligible probability.

We omit pk or sk for simplicity without ambiguity. An FHE scheme is said to be *secure* if it is IND-CPA secure. The security of almost all existing FHE schemes is based on the assumptions of LWE [50], RLWE [51], or their variants.

Now we recall some homomorphic operations of an SIMD-supported FHE scheme. For convenience, we take CKKS [6] as an example. In CKKS, the plaintext space is $\mathcal{M} = \mathbb{Z}[X]/\langle X^N + 1\rangle =: R$ while messages are complex vectors in $\mathbb{C}^\ell$ with $\ell = N/2$, where N is a power-of-two integer. The ciphertext space of CKKS is $\mathcal{C} = R/qR$, where q is the *ciphertext modulus*, a large integer. The restriction of the canonical embedding $\mathbb{R}[X]/\langle X^N + 1\rangle \to \mathbb{C}^\ell$ on R maps $m(X) \in R$ into $\boldsymbol{m} \in \mathbb{C}^\ell$ by evaluating $m(X)$ at the primitive $2N$-roots of unity $\xi_j = \xi^{5^j}$ for $0 \le j < \ell$. The inverse of the canonical embedding encodes a message $\boldsymbol{m}$ as a plaintext $m(X)$. Thus, CKKS naturally supports SIMD operations, i.e., performing an operation on a ciphertext corresponds to performing the same operation on $\ell = N/2$ entries of $\boldsymbol{m}$ in parallel. Each entry of the message $\boldsymbol{m} \in \mathbb{C}^\ell$ is called a *plaintext slot*.

For $\boldsymbol{x} = (x_i)_{0 \le i < \ell}$ and $\boldsymbol{y} = (y_i)_{0 \le i < \ell}$, let $\mathsf{ct}.\boldsymbol{x}$ and $\mathsf{ct}.\boldsymbol{y}$ be the ciphertext encrypted by CKKS under the same public key. CKKS supports the following basic operations:

- $\mathsf{Add}(\mathsf{ct}.\boldsymbol{x}, \mathsf{ct}.\boldsymbol{y})$: $\mathsf{Dec}(\mathsf{Add}(\mathsf{ct}.\boldsymbol{x}, \mathsf{ct}.\boldsymbol{y})) = \boldsymbol{x} + \boldsymbol{y}$.
- $\mathsf{Mul}(\mathsf{ct}.\boldsymbol{x}, \mathsf{ct}.\boldsymbol{y})$: $\mathsf{Dec}(\mathsf{Mul}(\mathsf{ct}.\boldsymbol{x}, \mathsf{ct}.\boldsymbol{y})) = \boldsymbol{x} \odot \boldsymbol{y}$, where $\odot$ is for component-wise multiplication.
- $\mathsf{CMul}(\boldsymbol{m}, \mathsf{ct}.\boldsymbol{x})$: $\mathsf{Dec}(\mathsf{CMul}(\boldsymbol{m}, \mathsf{ct}.\boldsymbol{x})) = \boldsymbol{m} \odot \boldsymbol{x}$, where $\boldsymbol{m}$ is a message in $\mathbb{C}^\ell$; for $m \in \mathbb{C}$, $\mathsf{CMul}(m, \mathsf{ct}.\boldsymbol{x})$ is a special case of $\mathsf{CMul}(\boldsymbol{m}, \mathsf{ct}.\boldsymbol{x})$ with $\boldsymbol{m} = (m, \ldots, m)$.
- $\mathsf{Sl}_{[i,j]}(\mathsf{ct}.\boldsymbol{x})$ convert a ciphertext $\mathsf{ct}.\boldsymbol{x} = \mathsf{Enc}(x_0, \ldots, x_{\ell-1})$ into a ciphertext that encrypts $(\boldsymbol{0}, x_i, x_{i+1}, \ldots, x_j, \boldsymbol{0})$, equivalent to $\mathsf{CMul}(\boldsymbol{m}, \mathsf{ct}.\boldsymbol{x})$ for $\boldsymbol{m} = (\boldsymbol{0} \in \mathbb{Z}^{i-1}, \boldsymbol{1} \in \mathbb{Z}^{j-i+1}, \boldsymbol{0})$.
- $\mathsf{Rot}_k(\mathsf{ct}.\boldsymbol{x})$ convert $\mathsf{ct}.\boldsymbol{x} = \mathsf{Enc}(x_0, \ldots, x_{\ell-1})$ into a new ciphertext $\mathsf{Enc}(x_k, \ldots, x_{\ell-1}, x_0, \ldots, x_{k-1})$.

III. Bicyclic Encoding for Matrices

In this section, we introduce a novel encoding method for matrices, disclose an intriguing property of this new encoding for matrix transpose, and present two algorithms for matrix multiplication under this new encoding.

To this end, we fix some notations. Let $\boldsymbol{A} \in \mathcal{R}^{n \times m}$ and $\boldsymbol{B} \in \mathcal{R}^{m \times p}$ be two matrices over some ring $\mathcal{R} \subseteq \mathbb{C}$. Denote by $\boldsymbol{X} \in \mathcal{R}^{n \times p}$ the resulting matrix of their multiplication, i.e., $\boldsymbol{X} = \boldsymbol{AB}$. The transpose of a matrix $\boldsymbol{A}$ is denoted by $\boldsymbol{A}^{\mathrm{T}}$. Let $[i]_k$ be the non-negative representation of the residue class of i in $\mathbb{Z}/(k\mathbb{Z})$. All indices of vectors and matrices start from 0 unless otherwise specified. For an integer k, we define a *rotation* of a vector $\boldsymbol{v} = (v_i)_{0 \le i < n} \in \mathcal{R}^n$ as $\rho_k(\boldsymbol{v}) = (v_{[k]_n}, v_{[k+1]_n}, \ldots, v_{[k+n-1]_n}) \in \mathcal{R}^n$, i.e., $\rho_k(\boldsymbol{v})$ rotates $\boldsymbol{v}$ to the left by $[k]_n$ positions.

A. Bicyclic Encoding

Let $\boldsymbol{A} = (a_{i,j}) \in \mathcal{R}^{n \times m}$ be a matrix. Define a map $\varphi_r : \mathcal{R}^{n \times m} \to \mathcal{R}^r$ with a positive integer $r \le m \cdot n$ as follows: $\varphi_{n,m,r}(\boldsymbol{A})$ = $(a_{[0]_n,[0]_m}, a_{[1]_n,[1]_m}, \ldots, a_{[r-1]_n,[r-1]_m})$. In particular, if $\gcd(n, m) = 1$ then the elements of $\varphi_{n,m,n \cdot m}(\boldsymbol{A}) \in \mathcal{R}^{n \cdot m}$ exactly traverse each element of $\boldsymbol{A}$ once. The reason is that the indices of the resulting vector are decided by the map $k \mapsto (k \mod n, k \mod m)$, which is an isomorphism between $\mathbb{Z}/(mn\mathbb{Z})$ and $\mathbb{Z}/(n\mathbb{Z}) \otimes \mathbb{Z}/(m\mathbb{Z})$ by CRT, provided $\gcd(n, m) = 1$.

We call $\varphi_{n,m,n \cdot m}(\boldsymbol{A})$ the *bicyclic encoding* of $\boldsymbol{A}$, denoted by $\boldsymbol{d}(\boldsymbol{A})$. For $\ell \ge m \cdot n$, the *bicyclic decoding* map $\psi_{\ell,n,m} : \mathcal{R}^\ell \to \mathcal{R}^{n \times m}$ that maps a vector $\boldsymbol{x} \in \mathcal{R}^\ell$ to a matrix $\boldsymbol{X} = (X_{i,j})$ with $X_{i,j} = x_k$ and $k = [i \cdot t \cdot m + j \cdot s \cdot n]_{n \cdot m}$, where (s, t) is a pair of *Bézout coefficients* for (n, m), i.e., two integers such that $s \cdot n + t \cdot m = 1$. In particular, for the bicyclic encoding $\boldsymbol{d}(\boldsymbol{A}) = (d_k)_{0 \le k < n \cdot m}$ of a matrix $\boldsymbol{A} \in \mathcal{R}^{n \times m}$ we have $\psi_{n \cdot m, n, m}(\boldsymbol{d}(\boldsymbol{A})) = \boldsymbol{A}$.

The following are some remarks on bicyclic encoding:

- In bicyclic encoding, the traversal of row and column indices of $\boldsymbol{A} \in \mathcal{R}^{n \times m}$ forms two different cycles: one modulo n and the other modulo m. This is where the name comes from. In fact, we can easily generalize the bicyclic encoding to *d-cyclic encoding* for d multidimensional arrays (tensor) $\boldsymbol{A} \in \mathcal{R}^{n_1 \times \cdots \times n_d}$ if $n_1, \ldots, n_d$ are pairwise coprime, since CRT still holds in this case.
- The bicyclic encoding for matrices can be viewed as an extension of the diagonal vectors [52, Fig. 1-35] employed in Halevi-Shoup's algorithm [32]. The primary difference lies in the fact that diagonal vectors are defined for square matrices, and a d-dimensional square matrix has d diagonal vectors. In contrast, the bicyclic encoding presented in this paper is effective for matrices with coprime dimensions, and the bicyclic encoding of a matrix is exactly a single vector.
- For a matrix $\boldsymbol{A} \in \mathcal{R}^{n \times m}$ with (n, m) coprime, the first component of $\boldsymbol{d}(\boldsymbol{A})$ is the $(0, 0)$-entry of $\boldsymbol{A}$, which can actually be adjusted to start from any entry of $\boldsymbol{A}$.

B. Matrix Multiplication under Bicyclic Encoding

Now we present two algorithms for matrix multiplication under bicyclic encoding.

Algorithm 1

```
Input: $\boldsymbol{A} \in \mathcal{R}^{n\times m}$, $\boldsymbol{B} \in \mathcal{R}^{m\times p}$, $(n, m, p)$ pairwise coprime.
Output: Matrix $\boldsymbol{X} = \boldsymbol{AB}$.
1) Initialize $\boldsymbol{a} := \boldsymbol{d}(\boldsymbol{A})$, $\boldsymbol{b} := \boldsymbol{d}(\boldsymbol{B})$, and $\boldsymbol{x} := \boldsymbol{0} \in \mathcal{R}^{n\cdot p}$.
2) Update $\boldsymbol{a} := \overbrace{(\boldsymbol{a}, \dots, \boldsymbol{a})}^{\lceil p/m\rceil \text{ times}}$ and $\boldsymbol{b} := \overbrace{(\boldsymbol{b}, \dots, \boldsymbol{b})}^{\lceil n/m\rceil \text{ times}}$.
3) Compute the smallest positive integer $r$ satisfying $r \cdot m - n > 0$ and $p \mid (r \cdot m - n)$.
4) For $0 \le i < m$ do
   a) Set $\underline{\boldsymbol{a}}_i := \rho_{-i\cdot n}(\boldsymbol{a})$, update $\underline{\boldsymbol{a}}_i$ as its first $np$ entries.
   b) Set $\underline{\boldsymbol{b}}_i := \rho_{i\cdot(r\cdot m-n)}(\boldsymbol{b})$, update $\underline{\boldsymbol{b}}_i$ as its first $np$ entries.
   c) Update $\boldsymbol{x} := \boldsymbol{x} + \underline{\boldsymbol{a}}_i \odot \underline{\boldsymbol{b}}_i$.
5) Decode $\boldsymbol{X} := \psi_{np,n,p}(\boldsymbol{x})$.
```

In Step 4a and 4b, the step size of rotations should be modulo mn and mp, respectively. In Step 4b, the step size of rotation is $i(rm - n)$ with $p \mid (rm - n)$ instead of ip. This nontrivial condition plays a key role in the proof of the correctness of Algorithm 1.

Proposition 3. *Algorithm 1 is correct. It requires at most* $2(m+\log\lceil p/m\rceil+\log\lceil n/m\rceil+1)$ *vector rotations and* m *component-wise vector products.*

Proof. Since (m, p) are coprime, n can be represented as an integral linear combination of m and p, which guarantees the existence of r in Step 3. Now we assume that (s, t) is a pair of Bézout coefficients for (n, p). Then according to the definition of bicyclic decoding, the (i, j)-element of $\boldsymbol{X}$ is $x_k = \sum_{0\le l<m} \underline{a}_{l,k} \cdot \underline{b}_{l,k}$, where x_k, $\underline{a}_{l,k}$ and $\underline{b}_{l,k}$ are the k-th element of $\boldsymbol{x}$, $\underline{\boldsymbol{a}}_l$, and $\underline{\boldsymbol{b}}_l$, respectively, and $k = [i \cdot t \cdot p + j \cdot s \cdot n]_{np}$. Furthermore, we have x_k is equal to

$$\begin{aligned}
&\sum_{0\le l<m} \underline{a}_{l,k} \cdot \underline{b}_{l,k} \\
=&\sum_{0\le l<m} a_{[k-l\cdot n]_n,[k-l\cdot n]_m} \cdot b_{[k+l(r\cdot m-n)]_m,[k+l(r\cdot m-n)]_p} &(2)\\
=&\sum_{0\le l<m} a_{[k]_n,[k-l\cdot n]_m} \cdot b_{[k-l\cdot n]_m,[k+l(r\cdot m-n)]_p} &(3)\\
=&\sum_{0\le l<m} a_{i,[k-l\cdot n]_m} \cdot b_{[k-l\cdot n]_m,j} &(4)\\
=&\sum_{0\le l<m} a_{i,l} \cdot b_{l,j}. &(5)
\end{aligned}$$

Eq. (2) follows from the definitions of bicyclic encoding and the rotation operator, and the construction of $\boldsymbol{a}$ and $\boldsymbol{b}$ in Step 2 of Algorithm 1. Eq. (3) easily follows from the modulo arithmetic. Eq. (4) follows from the fact that $[k]_n = i$ and $[k+i(r\cdot m-n)]_p = j$. In fact, according to the definition of k, there exists an integer q such that $k = i\cdot t\cdot p+j\cdot s\cdot n+q\cdot n\cdot p$. So we have $[k]_n = [i \cdot t \cdot p]_n = i$ because of $s \cdot n + t \cdot p = 1$. Similarly, $[k + i(r \cdot m - n)]_p = [j \cdot s \cdot n + i(r \cdot m - n)]_p = j$, where the last equality follows from $s \cdot n + t \cdot p = 1$ and $p \mid (r \cdot m - n)$. To prove (5), we only need to prove that the set $\{[k - l \cdot n]_m : 0 \le l < m\}$ forms a complete residue system modulo m. Assume that it is not the case, i.e., there exist l and l' such that $0 \le l' < l < m$ and $[k - l \cdot n]_m \neq [k - l' \cdot n]_m$. This assumption implies that there exists a nonzero integer u such that $k - l \cdot n + m \cdot u = k - l' \cdot n$, so we have $(l - l')n = m \cdot u$. Recalling $\gcd(n, m) = 1$, it gives $n \mid u$, i.e., there exists a nonzero integer v such that $u = n\cdot v$. Hence, $(l-l') = m\cdot v$, which implies $\left|l - l'\right| > m$. This contradicts with $0 \le l' < l < m$, which completes the proof.

The for loop of Algorithm 1 requires $2m-2$ vector rotations and m vector Hadamard products, and Step 2 requires at most $2(\log\lceil p/m\rceil + \log\lceil n/m\rceil + 2)$ extra vector rotations (see Proposition 8), which completes the proof. □

Definition 4. For $\boldsymbol{c} = (c_i)_i \in \mathcal{R}^n$ and an integer k with k dividing n, the *segment-sum* of $\boldsymbol{c}$ *with length* k is defined to be $\boldsymbol{s} = (s_i)_i \in \mathcal{R}^k$ with $s_i = \sum_{0\le j<n/k} c_{j\cdot k+i}$.

Now we propose the second matrix multiplication algorithm (Algorithm 2) under bicyclic encoding.

Proposition 5. *Algorithm 2 is correct. It requires at most* $\log\lceil n\rceil+\log\lceil m\rceil+\log\lceil p\rceil$ *vector rotations and only one component-wise vector product of dimension* mnp.

Algorithm 2

```
Input: $\boldsymbol{A} \in \mathcal{R}^{n\times m}$, $\boldsymbol{B} \in \mathcal{R}^{m\times p}$, $(n,m,p)$ pairwise coprime.
Output: Matrix $\boldsymbol{X} = \boldsymbol{A}\boldsymbol{B}$.
1) Initialize $\boldsymbol{a} := \boldsymbol{d}(\boldsymbol{A})$ and $\boldsymbol{b} := \boldsymbol{d}(\boldsymbol{B})$.
2) Set $\boldsymbol{a} := (\boldsymbol{a},\dots,\boldsymbol{a}) \in \mathcal{R}^{mnp}$, $\boldsymbol{b} := (\boldsymbol{b},\dots,\boldsymbol{b}) \in \mathcal{R}^{mnp}$.
3) Compute $\boldsymbol{x} := \boldsymbol{a} \odot \boldsymbol{b}$.
4) Compute the segment-sum of the vector $\boldsymbol{x}$ with length $np$.
5) Decode $\boldsymbol{X} := \psi_{mnp,n,p}(\boldsymbol{x})$.
```

From the perspective of matrix multiplication in plaintext, both Algorithm 1 and 2 require $O(mnp)$ arithmetic operations. However, their vectorized calculation style may be employed to accelerate matrix multiplication on certain heterogeneous platforms, such as FPGA and GPU.

C. Encrypted Matrix Operations under Bicyclic Encoding

We now start to discuss some matrix operations on encrypted data under bicyclic encoding, but we defer the encrypted matrix multiplication to the next section.

1) Switching between encrypted bicyclic encoding and row encoding: The first operation is how to convert between the bicyclic encoding and the commonly used row encoding or column encoding. Here, we only discuss the row encoding, since the discussion for the column encoding can be obtained similarly. For a matrix $\boldsymbol{A} = (a_{i,j}) \in \mathcal{R}^{n\times m}$ with (n,m) coprime, the *row encoding* of $\boldsymbol{A}$ is defined as a vector $\boldsymbol{r}(\boldsymbol{A}) = (a_{\lfloor k'/m\rfloor,[k']_m})_{0\le k'<mn}$. Suppose that $\boldsymbol{d}(\boldsymbol{A})$ is the bicyclic encoding of $\boldsymbol{A}$. Then there exists a linear transformation $\boldsymbol{T}$ between $\boldsymbol{r}(\boldsymbol{A})$ and $\boldsymbol{d}(\boldsymbol{A})$. In particular, $\boldsymbol{d}(\boldsymbol{A}) = \boldsymbol{T}\cdot\boldsymbol{r}(\boldsymbol{A})$, where $t_{k,k'} = 1$ with k and k' determined as follows. For $0 \le i < n$ and $0 \le j < m$, we first decide $k' = i\cdot m + j$, and then compute $k = [i\cdot t\cdot m + j\cdot s\cdot n]_{mn}$, where (s,t) is a pair of Bézout coefficients for (n,m), as in the bicyclic decoding process. One can decide a matrix $\boldsymbol{T}'$ satisfying $\boldsymbol{r}(\boldsymbol{A}) = \boldsymbol{T}'\cdot\boldsymbol{d}(\boldsymbol{A})$ similarly.

Given an encryption of $\boldsymbol{r}(\boldsymbol{A})$, we can use the diagonal encoding introduced by Halevi-Shoup to perform linear transformations on ciphertexts [32], thereby accomplishing the conversion between the two encodings. Assuming the matrix $\boldsymbol{T}$ has d non-zero diagonal vectors, this transformation can be computed within d CMuls and $2\sqrt{d}$ Rots, and requires only one level of CMul multiplication depth [36].

2) Encrypted matrix transpose under bicyclic encoding: Under bicyclic encoding, we show that one can transpose a matrix for free, either in plaintext or encrypted form.

Proposition 6. *For a matrix $\boldsymbol{A} \in \mathcal{R}^{n\times m}$ with $\gcd(n,m) = 1$, we have $\boldsymbol{d}(\boldsymbol{A}) = \boldsymbol{d}(\boldsymbol{A}^{\mathrm{T}})$.*

Corollary 7. *Let $(\mathsf{ct}.\boldsymbol{a}_i)_{i<\lceil \frac{mn}{\ell}\rceil}$ be ciphertexts (under an FHE scheme that supports ℓ slots) of the bicyclic encoding of a matrix $\boldsymbol{A} \in \mathcal{R}^{n\times m}$ with $\gcd(n,m) = 1$. Then $(\mathsf{ct}.\boldsymbol{a}_i)_{i<\lceil \frac{mn}{\ell}\rceil}$ are also ciphertexts of the bicyclic encoding of $\boldsymbol{A}^{\mathrm{T}}$.*

IV. Encrypted Matrix Multiplication under Bicyclic Encoding

In this section, we always assume that (n,m,p) are coprime, which means bicyclic encoding applies to all matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{X} = \boldsymbol{A}\boldsymbol{B}$, denoted by $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{x}$ the encoded vectors, respectively. We also assume that all the encoded vectors can be encrypted into a single ciphertext, denoted by $\mathsf{ct}.\boldsymbol{a}$, $\mathsf{ct}.\boldsymbol{b}$, and $\mathsf{ct}.\boldsymbol{x}$, respectively.

A. Building Blocks

For convenience, we first present two building blocks.

1) The Repeat *Operation:* According to Step 2 of Algorithm 1, we need first to convert a ciphertext $\mathsf{ct}.\boldsymbol{a}$ of $\boldsymbol{a}$ to a ciphertext of $(\boldsymbol{a},\dots,\boldsymbol{a})$, i.e., repeated with certain times.

Algorithm 3 Repeat

```
Input: A ciphertext $\mathsf{ct}.\boldsymbol{a}$ of $(\boldsymbol{a},\boldsymbol{0}) \in \mathcal{R}^{\ell}$ (where $\boldsymbol{a} \in \mathcal{R}^{d}$) and an integer $t = \sum_{i=0}^{\lfloor\log t\rfloor} t_i\cdot 2^i \ge 1$ satisfying $td < \ell$.
Output: A updated ciphertext $\mathsf{ct}.\boldsymbol{c}$ that encrypts $(\boldsymbol{a},\dots,\boldsymbol{a},\boldsymbol{0}) \in \mathcal{R}^{\ell}$ with $\boldsymbol{a}$ repeated $t$ times.
1) Initialize $\mathsf{ct}.\boldsymbol{a}_0 \leftarrow \mathsf{ct}.\boldsymbol{a}$.
2) For $1 \le i \le \lfloor\log t\rfloor$ do
    a) Compute $\mathsf{ct}.\boldsymbol{a}_i \leftarrow \mathsf{Add}(\mathsf{ct}.\boldsymbol{a}_{i-1}, \mathsf{Rot}_{-2^{i-1}d}(\mathsf{ct}.\boldsymbol{a}_{i-1}))$.
3) Set $k = 2^{\lfloor\log t\rfloor}\cdot d$ and $\mathsf{ct}.\boldsymbol{c} := \mathsf{ct}.\boldsymbol{a}_{\lfloor\log t\rfloor}$.
4) For $i = \lfloor\log t\rfloor - 1, \lfloor\log t\rfloor - 2, \dots, 1, 0$ do
    a) If $t_i \ne 0$ then compute $\mathsf{ct}.\boldsymbol{c} \leftarrow \mathsf{Add}(\mathsf{ct}.\boldsymbol{c}, \mathsf{Rot}_{-k}(\mathsf{ct}.\boldsymbol{a}_i))$ and update $k := k + t_i\cdot 2^i\cdot d$.
```

Proposition 8. *The* Repeat *algorithm is correct and requires at most* $2\log t$ Rot*s and* Add*s, respectively. In particular, if t is a power-of-two integer, it only requires* $\log t$ *such operations.*

2) The SegSum *Operation:* In Step 4 of Algorithm 2, we need to compute the segment-sum (Definition 4) of a vector $\boldsymbol{a} = (a_i)_i \in \mathcal{R}^n$ with length k satisfying $n = k \cdot m$ for an integer m. The following algorithm is an encrypted form of this process.

Algorithm 4 SegSum

```
Input: A ciphertext ct.$\boldsymbol{a}$ of $(\boldsymbol{a}, \boldsymbol{0}) \in \mathcal{R}^\ell$ (where $\boldsymbol{a} \in \mathcal{R}^n$) and an integer $k$ satisfying $n = k \cdot m$ for an integer $m$.
Output: A ciphertext ct.$\boldsymbol{c}$ that encrypts $(\boldsymbol{c}, \boldsymbol{0}) \in \mathcal{R}^\ell$, where $\boldsymbol{c} \in \mathcal{R}^k$ is the segment-sum of $\boldsymbol{a}$ with length $k$.
1) Initialize ct.$\boldsymbol{c}$ ← ct.$\boldsymbol{a}$ and $m := n/k$.
2) For $i = \lceil \log m \rceil - 1, \lceil \log m \rceil - 2, \ldots, 1, 0$ do
   a) Set ct.$\boldsymbol{t}$ := $\mathsf{Rot}_{2^i \cdot k}$(ct.$\boldsymbol{c}$)
   b) If $i = \lceil \log m \rceil - 1$ then ct.$\boldsymbol{t}$ ← $\mathsf{Sl}_{[0, n-2^i-1]}$(ct.$\boldsymbol{t}$).
      // This can be omitted if $m$ is a power-of-two integer.
   c) Update ct.$\boldsymbol{c}$ ← Add(ct.$\boldsymbol{c}$, ct.$\boldsymbol{t}$).
```

Proposition 9. *The* SegSum *algorithm is correct and requires at most* $\lceil \log m \rceil$ Rot*s and* Add*s, and one* CMul. *In particular, if* m *is a power-of-two integer, it only requires* $\log m$ Rot*s and* Add*s, without* CMul.

B. BMM-I (Encrypted Version of Algorithm 1)

We now propose an encrypted version of Algorithm 1. In comparison to the plaintext algorithm (Algorithm 1), the most significant difference lies in Step 2, where the number of repetitions for vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is doubled. This directly leads to the requirement of $\ell > 2 \cdot \max\{mn, mp, np\}$. The reason is that Rot operations are indeed performed on vectors of dimension ℓ. In contrast, in the plaintext algorithm, the rotation operations for $\boldsymbol{a}$ (resp. $\boldsymbol{b}$) are performed on a vector of dimension mn (resp. mp). Thus, we have to double the number of repetitions of the original vectors to guarantee the correctness of the results.

Algorithm 5 BMM-I (Bicyclic Matrix Multiplication)

```
Input: ct.$\boldsymbol{a}$ and ct.$\boldsymbol{b}$, which are ciphertexts of the bicyclic encoding of matrices $\boldsymbol{A} \in \mathcal{R}^{n \times m}$ and $\boldsymbol{B} \in \mathcal{R}^{m \times p}$, respectively,
    where $(n, m, p)$ are coprime and the number of slots $\ell > 2 \cdot \max\{mn, mp, np\}$.
Output: ct.$\boldsymbol{x}$, whose first $n \cdot p$ slots correspond to the bicyclic encoding of the resulting matrix $\boldsymbol{X} \in \mathcal{R}^{n \times p}$.
1) Initialize ct.$\boldsymbol{x}$ ← Enc($\boldsymbol{0}$). Compute the smallest positive integer $r$ satisfying $p \mid (r \cdot m - n)$ and $r \cdot m - n > 0$.
2) Update ct.$\boldsymbol{a}$ ← Repeat(ct.$\boldsymbol{a}$, $2\lceil p/m \rceil$) and ct.$\boldsymbol{b}$ ← Repeat(ct.$\boldsymbol{b}$, $2\lceil n/m \rceil$).
3) For $0 \le i < m$ do
   a) Compute ct.$\underline{\boldsymbol{a}}_i$ ← $\mathsf{Rot}_{[-i \cdot n]_{mn}}$(ct.$\boldsymbol{a}$).
   b) Compute ct.$\underline{\boldsymbol{b}}_i$ ← $\mathsf{Rot}_{[i \cdot (r \cdot m - n)]_{mp}}$(ct.$\boldsymbol{b}$).
   c) Update ct.$\boldsymbol{x}$ ← Add(ct.$\boldsymbol{x}$, Mul(ct.$\underline{\boldsymbol{a}}_i$, ct.$\underline{\boldsymbol{b}}_i$)).
```

Another difference lies in the for loop: neither ct.$\underline{\boldsymbol{a}}_i$ nor ct.$\underline{\boldsymbol{b}}_i$ is truncated to keep only the first np elements. Even after the for loop, ct.$\boldsymbol{x}$ is still not truncated to contain only the first np elements. In fact, if we denote the ℓ-dimensional vector $\boldsymbol{x}$ by the decryption of ct.$\boldsymbol{x}$ returned by BMM-I, then the first np components of $\boldsymbol{x}$ exactly correspond to the bicyclic encoding of the result matrix $\boldsymbol{X} = \boldsymbol{AB}$. Hence, the truncation can be delayed until decryption. This property implies that BMM-I does not need any plaintext-ciphertext multiplication CMul.

Proof of Theorem 1 item 1. Under the assumptions on n, m, p and ℓ, bicyclic encoding is available for each matrix, and each bicyclic encoding vector can be encrypted in one ciphertext. Further, the assumption $\ell > 2 \cdot \max\{mn, mp, np\}$ guarantees the correctness of Rots in Step 3a and 3b. Thus, BMM-I exactly follows the plaintext Algorithm 1, which implies the correctness. The cost of BMM-I follows from the cost of Repeat and counting. □

Remark 1. Although BMM-I does not use plaintext-ciphertext multiplication CMul, users should know that the computation results are contained only in the first np slots. Suppose that decrypting the ct.$\boldsymbol{x}$ obtains $\boldsymbol{x} \in \mathcal{R}^\ell$. Then running bicyclic decoding $\psi_{\ell,n,p}(\boldsymbol{x})$ gives the resulting matrix $\boldsymbol{X} = \boldsymbol{AB}$. If necessary, these results can be extracted through a single $\mathsf{Sl}_{[0,np-1]}$(ct.$\boldsymbol{x}$) operation. In fact, performing this selection remains unnecessary unless there is a need to utilize the latter $\ell - np$ slots.

Comparison with existing algorithms: Compared with algorithms designed specifically for square matrices [35]–[38], BMM-I offers greater flexibility in matrix dimensions. Despite the pairwise coprime constraints among (n, m, p), one can use padding with zeros to meet the requirements. Generally, BMM-I needs fewer padding positions. Compared with algorithms that support encrypted matrix multiplication of arbitrary dimensions [39], [40], as well as those facilitating encrypted approximate number matrix operations [33], [35]–[38], BMM-I requires the fewest number of multiplication depths and ciphertext rotations. While the algorithm in [38] requires a smaller number of ciphertext-ciphertext multiplications (Mul), our algorithm does not need plaintext-ciphertext multiplication (CMul); see Table I.

However, the condition $\ell > 2 \cdot \max\{mn, mp, np\}$ in BMM-I constrains the dimensions of matrices. For a set of CKKS parameters with $\ell = 2^{12}$, Jiang et al.'s algorithm [36] supports encrypted matrix multiplication of dimensions $(64, 64, 64)$, whereas BMM-I only supports dimensions of $(43, 45, 44)$, which is about $\sqrt{\ell/2}$. (Note that the algorithm in [38] only supports $(16, 16, 16)$, i.e., $\sqrt[3]{\ell}$, encrypted matrix multiplication under the same setting.) However, experimental results in Section VII show that BMM-I is still practical.

C. BMM-II (Encrypted Version of Algorithm 2)

Similarly, we propose an encrypted version of Algorithm 2 as BMM-II (Algorithm 6). We note that the discussion in Remark 1 also holds for BMM-II.

Algorithm 6 BMM-II

```
Input: ct.$\boldsymbol{a}$ and ct.$\boldsymbol{b}$, which are ciphertexts of the bicyclic encoding of matrices $\boldsymbol{A} \in \mathcal{R}^{n\times m}$ and $\boldsymbol{B} \in \mathcal{R}^{m\times p}$, respectively,
    where $(n, m, p)$ are pairwise coprime and the number of slots $\ell > n \cdot m \cdot p$.
Output: ct.$\boldsymbol{x}$, whose first $n \cdot p$ slots correspond to the bicyclic encoding of the resulting matrix $\boldsymbol{X} = \boldsymbol{AB} \in \mathcal{R}^{n\times p}$.
  1) Initialize ct.$\boldsymbol{x} \leftarrow$ Enc($\mathbf{0}$).
  2) Set ct.$\boldsymbol{a} \leftarrow$ Repeat(ct.$\boldsymbol{a}, p$), ct.$\boldsymbol{b} \leftarrow$ Repeat(ct.$\boldsymbol{a}, n$).
  3) Compute ct.$\boldsymbol{x} \leftarrow$ Mul(ct.$\boldsymbol{a}$, ct.$\boldsymbol{b}$).
  4) ct.$\boldsymbol{x} \leftarrow$ SegSum(ct.$\boldsymbol{x}, np$)
```

Proof of Theorem 1 item 1. In comparison to Algorithm 2 in plaintext, all steps of BMM-II are essentially the same. Therefore, the item 1 of Theorem 1 follows from Proposition 5, which also completes the proof of Theorem 1. □

Comparison with BMM-I: Compared with BMM-I, BMM-II imposes stricter constraints on matrix dimensions, requiring $\ell > nmp$. Given a fixed ℓ, this constraint limits the matrix dimensions that BMM-II can support. However, this stricter requirement ensures that we can always construct vectors of dimension nmp from the diagonal encoding of the two input matrices. The product of these two vectors encapsulates all the information of the matrix multiplication result, leading to an optimized result on the number of ciphertext operations.

Comparison with existing algorithms: Similar to BMM-II, both the Rizomiliotis-Triakosia (RT) algorithm [38] and Zheng et al.'s algorithm [30] have a restriction of $d = O(\sqrt[3]{N})$, where $d = \max(n, m, p)$ and N is the dimension of the ring used in the encryption algorithm. Compared to the RT algorithm, the number of Rots required by BMM-II is reduced from a linear function in d to a logarithmic function. Compared to Zheng et al.'s algorithm, the cost of BMM-II is considerable; theoretically, BMM-II and Zheng et al.'s algorithm are the two most cost-effective algorithms. However, because Zheng et al.'s algorithm relies on the tensor structure of the ring, it currently only supports the BGV scheme, while BMM-II can support any homomorphic encryption scheme that supports SIMD, including BGV, B/FV, CKKS.

However, we note that when computing high-dimensional matrix multiplication in a block-wise manner, these algorithms require more blocks than the other algorithms listed in Table I, which makes them not so practical as shown in Section VII. Next, we will discuss how to circumvent this obstacle to support encrypted matrix multiplication with higher dimensions.

V. Encrypted Matrices of High Dimensions

Assume that the matrix multiplication with dimensions (n, m, p) satisfies the condition that $\max(mn, mp, np) > \ell$. This implies that at least one matrix involved in the multiplication (either $\boldsymbol{A}$, $\boldsymbol{B}$, or $\boldsymbol{X}$) requires multiple ciphertexts for storage. Under this setting, traditional methods typically resort to block matrix multiplication. Besides, there exists another natural approach, called *segmented strategy* for handling high-dimensional encrypted matrices, in which any bicyclic encoding vector of matrices with dimension larger than ℓ may be divided into multiple vectors of dimension ℓ.

A. Block Matrix Multiplication

Let $\boldsymbol{A} \in \mathcal{R}^{n\times m}$ and $\boldsymbol{B} \in \mathcal{R}^{m\times p}$ be the two matrices to be multiplied. Assume that the number of slots ℓ supports a (n_0, m_0, p_0) matrix multiplication for packed ciphertexts with $m_0 = \max\{n_0, m_0, p_0\}$. For simplicity, we further assume that $n_1 = n/n_0 = m/m_0 = p/p_0$. Then both $\boldsymbol{A}$ and $\boldsymbol{B}$ can be split into $n_1 \times n_1$ blocks. To compute this (n_1, n_1, n_1) block matrix multiplication, one needs to compute n_1^{ω} matrix multiplications of dimension (n_0, m_0, p_0) with $2 < \omega < 3$, e.g., for Strassen algorithm [31], $\omega = \log 7 \approx 2.81$. We call the resulting algorithm the *block* version of BMM-I, which requires $m_0 n_1^{\omega}$ Muls and $2(m_0 + 1)n_1^{\omega}$ Rots. In fact, the block strategy applies to all matrix multiplication algorithms. In Table III, we summarize the cost of some of them, such as [30], [36], [38]. Note that $n_0 = m_0 = p_0 = \sqrt{\ell}$ for [36] and $n_0 = m_0 = p_0 = \sqrt[3]{\ell}$ for [38].

It appears that the block version of BMM-II and Zheng et al.'s algorithm [30] are faster than the others in Table III. However, as mentioned previously, these two algorithms have to deal with more blocks. In particular, n_1 for these two algorithms is approximately $d/\ell^{1/3}$, while for the others n_1 is about $d/\ell^{1/2}$, where $d = \max(n, m, p)$.

TABLE III
THE COST OF DIFFERENT BLOCK ALGORITHMS FOR HIGH DIMENSIONAL (n, m, p) ENCRYPTED MATRIX MULTIPLICATION WITH $d = \max\{n, m, p\}$.

Method	#Mul	#CMul	#Rot	Mult. depth
Block [36]	$\ell^{\frac{1-\omega}{2}} d^\omega$	$5\ell^{\frac{1-\omega}{2}} d^\omega$	$(3\ell^{\frac{1-\omega}{2}} + 5\ell^{\frac{1}{2}(\frac{1}{2}-\omega)})d^\omega$	1 Mul + 2 CMul
Block [38]	$\ell^{-\frac{\omega}{3}} d^\omega$	$2\ell^{\frac{1-\omega}{3}} d^\omega$	$(2\ell^{\frac{1-\omega}{3}} + \ell^{-\frac{1}{3}\omega} \log \ell)d^\omega$	1 Mul + 1 CMul
Block [30]†	n_1^ω	$2n_1^\omega$	$2 \log N \cdot n_1^\omega$	1 Mul + 1 CMul
Block BMM-I	$m_0 n_1^\omega$	0	$2(m_0 + 1)n_1^\omega$	1 Mul
Block BMM-II ‡	n_1^ω	0	$3 \log m_0 \cdot n_1^\omega$	1 Mul

† N is the ring dimension of the BGV scheme.
‡ m_0 is a power-of-two integer.

For the block version of BMM-I, if we assume that $n_0 \approx m_0 \approx p_0 \approx \sqrt{\ell/2}$ and $d = \max\{n, m, p\}$, then #Mul $\le (\ell/2)^{\frac{1-\omega}{2}} d^\omega$, and #Rot $\le 2(\ell/2)^{(1-\omega)/2} d^\omega$, which seems worse than that of algorithms in [36], [38]. However, the advantages of the block version of BMM-I include: it requires only one multiplicative depth, and it needs no CMul. Experiments in Section VII-D show that the block version of BMM-I performs well in practice.

B. Segmented Matrix Multiplication

The bicyclic encoding introduced in Section III allows us to segment the bicyclic encoding vector of matrices, thereby supporting high-dimensional encrypted matrix multiplication following Algorithm 1 exactly. To facilitate this, it is necessary to introduce a fundamental operation called LongRot, used to rotate the segmented vectors of a high-dimensional vector.

1) The LongRot *algorithm:* Given $\boldsymbol{a} \in \mathcal{R}^d$ with $d > \ell$, the LongRot operation implements the following functionality:

- Construct $\underline{\boldsymbol{a}} = (\boldsymbol{a}, \dots, \boldsymbol{a}) \in \mathcal{R}^{t \cdot d}$, repeating t times of $\boldsymbol{a}$;
- Rotate the vector $\underline{\boldsymbol{a}}$ to the left by k positions, resulting in $\underline{\boldsymbol{a}}' = \rho_k(\underline{\boldsymbol{a}})$;
- Select the first τ elements of $\underline{\boldsymbol{a}}'$ and divide them into $\left\lceil \frac{\tau}{\ell} \right\rceil$ groups, each containing ℓ elements, possibly zero-padding for the last one.

Clearly, this functionality is designed to construct the ciphertexts of $\underline{\boldsymbol{a}}_i$ and $\underline{\boldsymbol{b}}_i$ from the ciphertexts of $\boldsymbol{a}$ and $\boldsymbol{b}$ in Step 2 of Algorithm 1, respectively. We omit the detailed description here since this involves only some tedious control structures. See Appendix A for details.

Proposition 10. *The* LongRot *algorithm correctly computes the above functionality within* $\left\lceil \frac{\tau}{\ell} \right\rceil$ Rot*s,* $2\left\lceil \frac{\tau}{\ell} \right\rceil + \frac{\tau}{d} + 1$ CMul*s, and one* CMul *multiplicative depth.*

Algorithm 7 LongRot

Input: Ciphertexts $(\mathsf{ct}.\boldsymbol{a}_i)_{0 \le i < w}$ for $\boldsymbol{a} = (\boldsymbol{a}_0, \boldsymbol{a}_1, \dots, \boldsymbol{a}_{w-2}, \boldsymbol{a}_{w-1}) \in \mathcal{R}^d$ with $\boldsymbol{a}_{w-1} \in \mathcal{R}^z$ and $\boldsymbol{a}_i \in \mathcal{R}^\ell$ for $i = 0, 1, \dots, w-2$ (i.e., $d = (w-1)\ell + z$ with $0 \le z < \ell$, where ℓ is the number of slots), the number of repeated times t, the number of positions to be rotated $k \in [0, d)$, the number of selected elements τ.
Output: Ciphertexts $(\mathsf{ct}.\underline{\boldsymbol{a}}_i')_{0 \le i < \lceil \frac{\tau}{\ell} \rceil}$, i.e., $\left\lceil \frac{\tau}{\ell} \right\rceil$ ciphertexts corresponding to the first τ elements of $\underline{\boldsymbol{a}}'$.

2) BMM-III (Segmented version of BMM-II): Now, we present an algorithm (BMM-III) for high-dimensional encrypted matrix multiplication under bicyclic encoding. It is essentially a direct translation of Algorithm 1 into its encrypted version. The only difference lies in Step 3a, where the rotation step size is adjusted from $-jn$ to $(m-j)n$. These two are evidently equivalent and meet the requirement of LongRot.

Proof of Theorem 2. From the structure of BMM-III, it is easy to see that the required multiplicative depth is one Mul and one CMul. It follows from Proposition 10 that Step 3a and 3b requires at most $2\left\lceil \frac{np}{\ell} \right\rceil + \frac{p}{m} + 1$ and $2\left\lceil \frac{np}{\ell} \right\rceil + \frac{n}{m} + 1$ CMuls, respectively, and both costs at most $\left\lceil \frac{np}{\ell} \right\rceil$ Rots. Therefore, totally, it requires at most $2m \cdot \left\lceil \frac{np}{\ell} \right\rceil$ Rots and $(4 \cdot \left\lceil \frac{np}{\ell} \right\rceil + 2)m + n + p$ CMuls. Step 3c costs $\left\lceil \frac{np}{\ell} \right\rceil$ Muls. Thus, the total number of Muls is bounded by $m \cdot \left\lceil \frac{np}{\ell} \right\rceil$. ☐

Algorithm 8 BMM-III

```
Input: Ciphertexts $(\mathsf{ct}.\boldsymbol{a}_i)_{i<\lceil mn/\ell \rceil}$ for the bicyclic encoding of $\boldsymbol{A} \in \mathcal{R}^{n \times m}$ and ciphertexts $(\mathsf{ct}.\boldsymbol{b}_i)_{i<\lceil mp/\ell \rceil}$ for the bicyclic
    encoding of $\boldsymbol{B} \in \mathcal{R}^{m \times p}$, where $(n, m, p)$ are pairwise coprime.
Output: Ciphertexts $(\mathsf{ct}.\boldsymbol{x}_i)_{0 \le i<\lceil np/\ell \rceil}$ for the bicyclic encoding of the resulting matrix $\boldsymbol{X} \in \mathcal{R}^{n \times p}$.
1) For $i = 0, 1, \dots, \lceil np/\ell \rceil - 1$ initialize $\mathsf{ct}.\boldsymbol{x}_i \leftarrow \mathsf{Enc}(\mathbf{0})$.
2) Compute the smallest positive integer $r$ such that $p \mid (r \cdot m - n)$ and $r \cdot m - n > 0$.
3) For $j = 0, 1, \dots, m-1$ do the following:
  a) $(\mathsf{ct}.\underline{\boldsymbol{a}}_i')_i \leftarrow \mathsf{LongRot}((\mathsf{ct}.\underline{\boldsymbol{a}}_i)_{i<\lceil \frac{mn}{\ell} \rceil}, \lceil \frac{p}{m} \rceil, (m-j)n, np)$.
  b) $(\mathsf{ct}.\underline{\boldsymbol{b}}_i')_i \leftarrow \mathsf{LongRot}((\mathsf{ct}.\underline{\boldsymbol{b}}_i)_{i<\lceil \frac{mp}{\ell} \rceil}, \lceil \frac{n}{m} \rceil, j(rm-n), np)$.
  c) Compute $\mathsf{ct}.\boldsymbol{x}_i \leftarrow \mathsf{Add}(\mathsf{ct}.\boldsymbol{x}_i, \mathsf{Mul}(\mathsf{ct}.\underline{\boldsymbol{a}}_i', \mathsf{ct}.\underline{\boldsymbol{b}}_i'))$ for $i = 0, 1, \dots, \lceil np/\ell \rceil - 1$.
```

Comparison: For square encrypted matrix multiplication with dimension d, BMM-III requires $O(d^3)$ ciphertext operations. Therefore, in an asymptotic sense, the number of ciphertext operations required by BMM-I is greater than those required by the block version algorithms in Section V-A. However, experiments in Section VII demonstrate that when $d \le 1024$, BMM-III has a distinct advantage over those block version algorithms. The reason is that most of the block version algorithms are originally recursive. It is well known that the efficiency of recursive algorithms is not that fast. Usually, one can rewrite a recursive algorithm as a loop algorithm. However, in the case of matrix multiplication over encrypted data, such a rewriting will increase the required multiplicative depth regarding the recursion depth, which is unacceptable.

TABLE IV
THE COST OF BMM-III FOR HIGH-DIMENSIONAL RECTANGULAR MATRIX MULTIPLICATION

Size	Method	#Mul	#CMul	#Rot	Mult. depth
$(n, n, \ell/n)$	[14]	n	$5n$	$3n + \frac{n^2}{\ell} + \frac{n\sqrt{n}}{\sqrt{\ell}} - 1$	1 Mul + 2 CMul
	BMM-III†	n	$7n + \frac{\ell}{n}$	$2n$	1 Mul + 1 CMul
$(n, \ell/n, \ell/n)$	[14]	$\frac{\ell}{n}$	$\frac{5\ell}{n}$	$\frac{3\ell}{n} + \frac{6\sqrt{\ell}}{n\sqrt{n}} + \frac{2\ell}{n^2} - 2$	1 Mul + 2 CMul
	BMM-III†	$\frac{\ell}{n}$	$\frac{7\ell}{n} + n$	$\frac{2\ell}{n}$	1 Mul + 1 CMul
$(\ell/n, n, \ell/n)$	[14]	$\frac{\ell}{n}$	$\frac{5\ell}{n}$	$\frac{3\ell}{n} + \frac{6\sqrt{\ell}}{n\sqrt{n}} + \frac{\ell}{n^2} - 1$	1 Mul + 2 CMul
	BMM-III†	$\frac{\ell}{n}$	$\frac{6\ell}{n} + 2n$	$\frac{2\ell}{n}$	1 Mul + 1 CMul

† For BMM-III, we should make the dimensions pairwise coprime.

Additionally, BMM-III supports encrypted matrix multiplication of flexible dimensions. In the literature, Huang et al. [14] investigated encrypted matrix multiplication for high-dimensional rectangular matrices with different shapes, including matrix multiplication of dimensions $(n, n, \ell/n)$, $(n, \ell/n, \ell/n)$, and $(\ell/n, n, \ell/n)$, where ℓ is the number of slots. All of these shapes cost similar ciphertext operations. Compared with theirs in Table IV shows that BMM-III reduces the number of Rots by a factor $\frac{1}{3}$ and saves one CMul depth, at a cost of a bit more CMuls.

VI. ANOTHER APPLICATION OF SEGMENTED STRATEGY

In this section, we consider applying the segmented strategy to the algorithm by Lu et al. [33], which is not under bicyclic encoding. This algorithm is no longer the best for matrix multiplication with smaller dimensions, as shown in Table I. However, combining several optimizations and observations, the segmented version of Lu et al.'s algorithm (SegLKS) requires the fewest number of ciphertext rotations in theory among all currently known encrypted matrix multiplication algorithms for high-dimensional matrices.

A. Lu et al.'s Algorithm

The algorithm in [33] for encrypted matrix multiplication relies on the Replicate operation. The function of $\mathsf{Replicate}_i(\mathsf{ct}.\boldsymbol{a})$ is to transform a ciphertext of $\boldsymbol{a} = (a_0, \dots, a_{\ell-1})$ into a ciphertext of $(a_i, a_i, \dots, a_i)$. A Replicate can be finished within one CMul plus $\log \ell$ Rots and Adds. Lu et al.'s algorithm supports matrix multiplication of any dimension. For computing an (n, m, p) matrix multiplication $\boldsymbol{X} = \boldsymbol{A}\boldsymbol{B}$ in encrypted form, the algorithm encodes all matrices row by row.

When $\max(m, p) < \ell$, the number of ciphertexts in Lu et al.'s algorithm corresponding to $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{X}$ are n, m and n, respectively. It requires mn Muls and CMuls, and $mn \log p$ Rots. Note that only $\log \ell$ key-switching keys are enough for Lu et al.'s algorithm, which is used to replicate each entry of $\boldsymbol{A}$. In addition, since the matrices are encoded by rows, it naturally supports the segmented method.

Algorithm 9 SegLKS (Segmented variant of [33])

```
Input: Ciphertexts $(\mathsf{ct}.\boldsymbol{a}_{i,j})_{0\le i<n, 0\le j<\lceil m/\ell\rceil}$ for $\boldsymbol{A}\in\mathcal{R}^{n\times m}$ and ciphertexts $(\mathsf{ct}.\boldsymbol{b}_{i,j})_{0\le i<m, 0\le j<\lceil p/\ell\rceil}$ for $\boldsymbol{B}\in\mathcal{R}^{m\times p}$, where
    $\mathsf{ct}.\boldsymbol{a}_{i,j}$ is the ciphertext corresponding to the $j$-th segment of the $i$-th row of $\boldsymbol{a}$, similar for $\mathsf{ct}.\boldsymbol{b}_{i,j}$.
Output: Ciphertexts $(\mathsf{ct}.\boldsymbol{x}_{i,j})_{0\le i<n, 0\le j<\lceil p/\ell\rceil}$ for the resulting matrix $\boldsymbol{X}\in\mathcal{R}^{n\times p}$.
1) For $i = 0$ to $n-1$ do
    a) For $j = 0$ to $\lceil p/\ell\rceil - 1$ do
        i) For $k = 0$ to $\lceil m/\ell\rceil$ do
            A) For $\iota = 0$ to $\ell - 1$ compute $\mathsf{ct}.\boldsymbol{x}_{i,j} \leftarrow \mathsf{Add}(\mathsf{ct}.\boldsymbol{x}_{i,j}, \mathsf{Mul}(\mathsf{Replicate}_\iota(\mathsf{ct}.\boldsymbol{a}_{i,k}), \mathsf{ct}.\boldsymbol{b}_{k\ell+\iota,j}))$.
2) Return $(\mathsf{ct}.\boldsymbol{x}_{i,j})_{0\le i<n, 0\le j<\lceil p/\ell\rceil}$.
```

B. SegLKS *(Segmented Lu et al.'s Algorithm)*

Now we consider matrix multiplication of high dimensions, where each row of $\boldsymbol{A}$ or $\boldsymbol{B}$ is encoded and encrypted as multiple ciphertexts in a segmented manner, say, $\min(m,p) > \ell$. The number of ciphertexts corresponding to $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{X}$ are $n\lceil \frac{m}{\ell}\rceil$, $m\lceil \frac{p}{\ell}\rceil$ and $n\lceil \frac{p}{\ell}\rceil$, respectively. Further, this algorithm requires $mn\lceil \frac{p}{\ell}\rceil$ Muls and CMuls, and $mn\log p$ Rots. However, we can optimize the algorithm further.

1) On encoding (O1): Since the cost of SegLKS heavily relies on a factor nm, when $nm \gg mp$, it is costly. If this is the case, we can encode the matrix by columns, or, equivalently, consider the matrix multiplication in transpose $\boldsymbol{X}^{\mathrm{T}} = \boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}$. So, one can always assume that $n \le p$.

2) On the number of ciphertexts (O2): For matrix $\boldsymbol{A}\in\mathcal{R}^{n\times m}$, since it only involves the Replicate operation, it can be encoded and encrypted into fewer ciphertexts without affecting efficiency. Indeed, it can be encrypted in a row-wise manner into $\lceil \frac{nm}{\ell}\rceil$ ciphertexts. For instance, if $mn < \ell$, this reduces the number of ciphertexts for $\boldsymbol{A}$ from n to 1.

3) On the number of Rots *(O3):* For multiplying with the ciphertexts of each row of $\boldsymbol{B}$, one must replicate each element of $\boldsymbol{A}$ to a vector of dimension p theoretically. However, all segments are the same. So it does not need $mn\log p$ Rots, but only needs $mn\log\ell$ Rots. This observation shows that the required Rots is independent of p.

4) On Replicate *the same ciphertext (O4):* In SegLKS, we need to replicate $\boldsymbol{a} = (a_0,\ldots,a_{m-1})\in\mathcal{R}^m$ to $(a_i,a_i,\ldots,a_i)\in\mathcal{R}^\ell$ for $i = 0,\ldots,m-1$. This costs $m\log\ell$ Rots and m CMuls. However, to obtain the same results, one may first group $\boldsymbol{a}$ into groups with each group κ elements and repeat each group ℓ/κ times. Then, for each repeated ciphertext, replicate the κ elements. For instance, assume that our goal is to obtain ciphertexts of $(i,\ldots,i)\in\mathcal{R}^{16}$ from $\boldsymbol{a} = (1,2,3,4)\in\mathcal{R}^4$ for $i = 1,2,3,4$. First, we select $(1,2)\in\mathcal{R}^2$ and repeat it 8 times to obtain $(1,2,\ldots,1,2)$, which costs 1 CMuls and $\log(\ell/\kappa) = 3$ Rots. Then we can obtain $(1,0,\ldots,1,0)$ and $(0,2,\ldots,0,2)$ by 2 CMuls. Then we can obtain $(1,\ldots,1)$ and $(2,\ldots,2)$ by 2 Rots. With this optimization, we can reduce the number of Rots from 16 to 10, at a cost of m/κ more CMuls and one more CMul depth. So, to replicate all elements of $\boldsymbol{A}$, the required Rots and CMuls are bounded by $nm\left(\frac{\log(\ell/\kappa)}{\kappa} + \log\kappa\right)$ and $nm(1+1/\kappa)$ respectively. The minimum number of Rots achieves when κ satisfies $\kappa e^{\kappa-1} = \ell$.

TABLE V
THE COST OF SEGMENTED ALGORITHMS FOR HIGH DIMENSIONAL ENCRYPTED (n,m,p) MATRIX MULTIPLICATION WITH $n \le p$.

Method	#Mul	#CMul	#Rot	#Mult. depth
BMM-III	$m\left\lceil\frac{np}{\ell}\right\rceil$	$4m\left\lceil\frac{np}{\ell}\right\rceil$ $+2(n+p+2m)$	$2m\left\lceil\frac{np}{\ell}\right\rceil$	1 Mul + 1 CMul
SegLKS/O3	$mn\left\lceil\frac{p}{\ell}\right\rceil$	mn	$mn\log\ell$	1 Mul + 1 CMul
SegLKS/O4	$mn\left\lceil\frac{p}{\ell}\right\rceil$	$nm(1+\frac{1}{\kappa})$	$nm(\frac{\log\frac{\ell}{\kappa}}{\kappa} + \log\kappa)$	1 Mul + 2 CMul

Summary: We summarize in Table V the segmented algorithms. If $\left\lceil\frac{np}{\ell}\right\rceil$ is small, say a constant, the efficiency of BMM-III is relevant since the required number of operations is only linear in m. In the case of $n,m \ll p$, the advantage of SegLKS is evident, as the required number of Rots and CMuls is independent of p. Furthermore, we also note that the number of Rots required by SegLKS is even less than that of the state-of-the-art algorithm [30], which needs $\ell^{-\frac{2}{3}}d^2\log d$ Rots with $d = \max\{n,m,p\}$, asymptotically more than $\log\ell\cdot d^2$ (SegLKS/O3) when d tends to infinity.

VII. Implementation and Evaluation

In this section, we first introduce our implementation with more details, followed by a comprehensive experimental study of the involved algorithms to evaluate their performance.

A. Implementation

To evaluate the performance of the algorithms introduced in this paper, we implement them all (BMM-I, BMM-II, BMM-III, SegLKS) and the algorithms of [36], [38] by using the CKKS scheme [6] implemented in Microsoft's open-source SEAL [15]. We also implement the naïve (textbook) and Strassen block version of the algorithms in [36], BMM-I, and BMM-II. Our implementation is open-sourced at https://github.com/hangenba/bicyclic_mat_mul.

In SEAL, the key-switching keys for Rot_k are generated only for $k = 2^i$ by default. When a Rot_k operation is involved for a non-power-of-two k, the task can be accomplished using consecutive rotations, e.g., $\mathsf{Rot}_3(\mathsf{ct}.\boldsymbol{x}) = \mathsf{Rot}_1(\mathsf{Rot}_2(\mathsf{ct}.\boldsymbol{x}))$. This is the main reason for the time-consuming nature of Rot. For efficiency, we pre-generate all the necessary switching keys in our implementation. Although this needs additional time and storage overhead for key generation, according to our test with BMM-III for $N = 8192$, it saves about 15% of computing time. Once these keys are generated, they can be reused in subsequent operations.

There are many fast algorithms for matrix multiplication in plain, e.g., [31], [53]. Almost all of these algorithms are recursive. It is well known that a recursive program can always be rewritten as an iterative loop. However, in the context of ciphertext computation, such as encrypted Strassen's matrix computation, the multiplication depth of the rewritten version may depend on the recursion depth, which will slow down the algorithm significantly. In contrast, the original recursive algorithm might only require a single Mul depth (possibly plus one or two CMul depths), although it is memory-consuming and hard to optimize further.

B. Setup

All experiments are run on a *single* thread (no parallelization) of a Desktop with an Intel Core i9-12900K at 3.2 GHz and 32 GB memory. We evaluate encrypted matrix multiplication algorithms with different dimensions:

- Small (almost) square: all matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{X}$ can be packed into one ciphertext. This setting is to evaluate BMM-I, BMM-II, and algorithms in [36], [38].
- Large (almost) square: all matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{X}$ have to be packed into several ciphertexts. This setting is to evaluate BMM-III, together with the block version of Jiang et al.'s algorithm [36] and BMM-I.
- Rectangular: Some of $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{X}$ are rectangular. This setting is to compare the performance of BMM-III, SegLKS and the state-of-the-art algorithm [14].

Recall that the ciphertext of CKKS is $\mathbb{Z}[X]/\langle X^N + 1, q\rangle$, and there is a scale factor Δ in CKKS related to the precision of the computed results. In our tests, all matrix entries are set by $\mathtt{pow(-1, i+j) * rand()/pow(2, 30)}$, roughly in the interval $(-2, 2)$. The degree of ciphertext space and ciphertext modulus may vary depending on algorithms. In particular, we always set the bit-size of q as $50 + L \cdot \log\Delta + 60$, where L is the number of multiplicative depth (Mul and CMul) required by the corresponding algorithm (see, e.g., Table I).

Such a setting always achieves at least 128-bit security level according to the latest lattice estimator [54]. We also note that multiple test runs show that in this setup, the maximal absolute errors of the computed results are always less than 10^{-2}. All timings include encryption, computation, and decryption.

C. Matrices with Small Dimension

For CKKS-based encrypted matrix multiplication with small dimension (each of the matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{X} = \boldsymbol{AB}$ can be encoded and encrypted into a single ciphertext), we compare BMM-I and BMM-II with the algorithms in [36] and [38].

When $N = 8192$, the Rizomiliotis-Triakosia (RT) algorithm [38] supports square matrix multiplication of dimension $\sqrt[3]{N/2} = 16$. As indicated in Table VI, BMM-I achieves a 1.5x speedup under the same parameter settings as the RT algorithm. However, due to the lower multiplicative depth required by BMM-I, it can finish the computation with a smaller ciphertext modulus. Under this setting, compared with the RT algorithm, BMM-I eventually achieves a 2.4x speedup. In the same parameter setting, BMM-II supports a $(15, 16, 17)$ matrix multiplication, which performs the best among these algorithms, achieving a 16.6x speedup. When $N = 16384$ or $N = 32768$, the RT algorithm does not have corresponding matrix multiplication, whereas both BMM-I and BMM-II have. More precisely, the RT algorithm can support matrix multiplication of other dimensions at a cost of more ciphertext operations. The reason is that the rotation operation Rot is designed for vectors of dimension ℓ. If the dimension is not exact ℓ, then one Rot can be finished with two Rots, two CMuls and one Add. The same reason holds for Jiang et al.'s algorithm when $N = 16384$ in Table VII.

We test three different N in Table VII. For the same N, BMM-I demonstrates a clear advantage (achieving a 4.4x speedup at least) compared with Jiang et al.'s algorithm [36], though the matrix dimensions it supports are only about $\sqrt{1/2} \approx 70\%$ of that supported by Jiang et al.'s algorithm.

In fact, the matrix dimension supported by all these algorithms are constrained by the number of plaintext slots ℓ. For instance, for $N = 8192$ ($\ell = 4096$), the algorithm in [36] can support square matrix multiplication of dimension $\sqrt{\ell} = 64$, the algorithm in [38] is limited to $\sqrt[3]{\ell} = 16$, and BMM-I (resp. BMM-II) can support matrix multiplication of dimensions $(43, 45, 44)$ (resp. $(15, 16, 17)$). However, the dimensions supported by BMM-I and BMM-II are quite flexible. For example,

TABLE VI
PERFORMANCE COMPARISON WITH THE RT ALGORITHM FOR SMALL-DIMENSIONAL MATRICES. $N = 8192$ AND $\log \Delta = 30$.

Method	$\log q$	Dimension	Time (ms)	Speedup
RT [38]	170	$(16, 16, 16)$	199	1.0x
BMM-I	170	$(16, 19, 17)$	130	1.5x
BMM-I	140	$(16, 19, 17)$	82	2.4x
BMM-II	140	$(15, 16, 17)$	**13**	16.6x

TABLE VII
PERFORMANCE COMPARISON WITH [36] FOR MATRICES WITH SMALL DIMENSION, WHERE $\log \Delta = 30$. "–" INDICATES THAT THE ALGORITHM DOES NOT SUPPORT MATRIX MULTIPLICATION FOR THE DIMENSIONS.

Method	$\log q$	$N = 8192$		$N = 16384$		$N = 32768$	
		Dim.	Time (ms)	Dim.	Time (ms)	Dim.	Time (ms)
[36]	200	$(64, 64, 64)$	1453	–	–	$(128, 128, 128)$	13526
BMM-I	140	$(43, 45, 44)$	284	$(61, 64, 63)$	1003	$(89, 91, 90)$	3059
BMM-II	140	$(15, 16, 17)$	13	$(21, 16, 23)$	31	$(31, 16, 33)$	48

when $N = 16384$, BMM-I can support matrix multiplication of dimensions $(61, 64, 63)$ or others, say $(29, 128, 31)$ with a runtime of 1940 ms when $\log q = 140$, while the other two algorithms do not support square matrix multiplication under this parameter setting.

Comparison with the algorithm in [30]: Since we did not implement our algorithms for the BGV scheme, we cannot directly compare the performance with Zheng et al.'s algorithm [30]. As analyzed previously, the performance of BMM-II should be a bit better than that of theirs. In particular, according to [30, Table 2], it is reported that their algorithm costs 1 Mul, 2 CMuls and 14 Rots for a $(16, 16, 16)$ matrix multiplication (their implementation costs about 119ms on our desktop for the first example given in [30, Tab. 3]), while BMM-II requires only 1 Mul and 13 Rots for a $(15, 16, 17)$ matrix multiplication in Table VII.

D. Large Square Matrix Multiplication

To evaluate the performance of algorithms for square matrices of high dimensions, we assume a scenario involving matrix multiplications with dimensions of 128, 256, 512 and 1024, respectively. For these tasks, we test different algorithms, including the naïve and Strassen version of block matrix multiplication, and the segmented BMM-III; see Tables VIII–X for details. Since it follows from Table VI that the algorithm in [38] is not comparable with BMM-I, we here only consider the Jiang et al.'s algorithm [36], BMM-I, and BMM-II as the base algorithms for the block version.

TABLE VIII
PERFORMANCE COMPARISON FOR $(128, 128, 128)$ MATRIX MULTIPLICATION. $\log \Delta = 30$ EXCEPT FOR BMM-III WITH $\log \Delta = 40$.

Method	$\log q$	$N = 8192$		$N = 32768$	
		Basic block	Time (s)	Basic block	Time (s)
Naïve block [36]	200	$(64, 64, 64)$	11.34	$(128, 128, 128)$	14.17
Strassen + [36]	200	$(64, 64, 64)$	10.34	$(128, 128, 128)$	14.17
Naïve block BMM-I	140	$(43, 45, 44)$	7.59	$(86, 89, 87)$	13.32
Strassen + BMM-I	140	$(32, 35, 33)$	8.31	$(64, 67, 65)$	11.99
Naïve block BMM-II	140	$(15, 16, 17)$	**4.40**	$(21, 32, 23)$	8.32
Strassen + BMM-II	140	$(11, 8, 9)$	84.89	$(17, 16, 19)$	127.69
BMM-III	190	$(128, 131, 129)$	11.05	$(128, 131, 129)$	41.60

1) Memory and the number of ciphertexts: In Table VIII-X, we only list the dimensions of the involved basic block, from which, together with the input dimensions, the number of blocks and hence the number of ciphertexts can be determined. For example, each matrix is split into 2×2 blocks for Naïve block [36] in Table VIII with $N = 8192$, and hence each matrix is encrypted as 4 ciphertexts; while for the naïve block BMM-II with the same N, the second matrix is split into 8×8 blocks, and hence encrypted as 64 ciphertexts. Although using BMM-I or BMM-II for high-dimensional matrix multiplication usually generates more ciphertexts than, e.g., Jiang et al.'s [36], which makes these algorithms require more memory than that of [36], those algorithms based on BMM-III do not have this disadvantage. For the last row in Table VIII as another example, each matrix is encrypted as 5 ciphertexts, only 1 more than that of [36]. Once again, we note that the timings include all for encryption, computation, and decryption.

2) Timings: From Table VIII, it is evident that for $(128, 128, 128)$ matrix multiplication, the most efficient is BMM-II using the naïve block method, while surprisingly, the Strassen version of BMM-II is the worst. In fact, this is consistent with the previous analysis, as although the basic version of BMM-II is theoretically the best among these algorithms, it supports the smallest matrix dimensions, resulting in more blocks and thus affecting the performance. Therefore, we will not consider the Strassen version of BMM-II for further tests.

TABLE IX
PERFORMANCE COMPARISON FOR $(256, 256, 256)$ MATRIX MULTIPLICATION. $\log \Delta = 30$ EXCEPT FOR ALGO. 8 WITH $\log \Delta = 40$.

Method	$\log q$	$N = 8192$		$N = 32768$	
		Basic block	Time (s)	Basic block	Time (s)
Naïve block [36]	200	$(64, 64, 64)$	89.33	$(128, 128, 128)$	112.01
Strassen + [36]	200	$(64, 64, 64)$	71.54	$(128, 128, 128)$	97.90
Naïve block BMM-I	140	$(43, 45, 44)$	59.57	$(86, 89, 87)$	73.35
Strassen + BMM-I	140	$(32, 35, 33)$	56.98	$(64, 67, 65)$	80.99
Naïve block BMM-I	140	$(15, 16, 17)$	76.60	$(21, 32, 23)$	76.19
BMM-III	190	$(256, 259, 257)$	**42.90**†	$(256, 259, 257)$	110.99

† We pre-generate the key-switching keys. Otherwise, it takes about 50s.

As indicated in Table IX, compared with the naïve block version, the Strassen block version does provide a speedup. In addition, the Strassen block version of BMM-I is more efficient than that of Jiang et al.'s. However, it should be noted that the Strassen block version of BMM-I requires more ciphertexts. For instance, when $N = 8192$, the Strassen block version of BMM-I needs 64 ciphertexts to store a matrix, while the Strassen block version of Jiang et al.'s algorithm requires only 16. Table IX also shows that the segmented algorithm BMM-III with $N = 8192$ outperforms all block algorithms, each matrix stored in 17 ciphertexts due to the coprime limitation of the dimensions. For example, it can finish a $(256, 259, 257)$ encrypted matrix multiplication within 42.90 seconds (2x faster than naïve block algorithm in [36]), of which 15.80 seconds are spent generating the switching keys required for Rot. Indeed, once these keys are generated, they can be reused, hence possibly further improving the efficiency for subsequent computations in practical applications.

TABLE X
PERFORMANCE COMPARISON FOR MATRIX MULTIPLICATION OF DIMENSION 512 AND 1024 WITH $N = 8192$.

Method	$\log q$	$(512, 512, 512)$		$(1024, 1024, 1024)$	
		Basic block	Time (s)	Basic block	Time (s)
Naïve block [36]	200	$(64, 64, 64)$	728	$(64, 64, 64)$	6028
Strassen + [36]	200	$(64, 64, 64)$	479	$(64, 64, 64)$	3514
Naïve block BMM-I	140	$(43, 45, 44)$	490	$(43, 45, 44)$	4240
Strassen + BMM-I	140	$(32, 35, 33)$	390	$(32, 35, 33)$	2757†
Naïve block BMM-II	140	$(15, 16, 17)$	1766	$(15, 16, 17)$	–
BMM-III	190	$(512, 515, 513)$	**181**†	$(1024, 1027, 1025)$	**1200**†

† By default, $\log \Delta = 30$ except for these with $\log \Delta = 40$.

Based on observations derived from Table VIII and IX, the setting $N = 8192$ performs better than $N = 32768$ for all tested algorithms. Thus, in tests with dimensions of 512 and 1024, we fix $N = 8192$. From Table X, the naïve block BMM-II perform the worst, since, again, more blocks slow down the speed. In addition, we have a similar observation to that from Table IX, indicating that BMM-III has the best performance among these algorithms. Specifically, for the task of $(1024, 1024, 1024)$ encrypted matrix multiplication, BMM-III is 5x faster than the naïve block version Jiang et al.'s algorithm [36]. By the way, We also test the performance of the naïve block BMM-III, which costs about 260s for the $(512, 512, 512)$ case with the basic block as $(256, 259, 257)$.

E. Rectangular Matrix Multiplication

Huang et al. in [14] investigated high-dimensional encrypted rectangular matrix multiplication for different shapes. As indicated in Table XI, for the different dimensions in [14], as the dimensions increase, the efficiency of BMM-III gradually outperforms that of Huang et al.'s algorithm, with a 2.6x speedup at most. Furthermore, BMM-III can be used to cases of even larger dimension. For instance, it can finish a $(8191, 8192, 15)$ encrypted matrix multiplication within 1452.16 seconds, while Huang et al.'s algorithm in its current setup does not support this instance.

Tables XII and XIII include two additional cases of rectangular encrypted matrix multiplication not discussed in [14]. In the first case (Table XII), matrix $\boldsymbol{A}$ is short and wide, while matrix $\boldsymbol{B}$ is tall and narrow, i.e., $n, p \ll m$. The naïve block version of BMM-I exhibits the best performance, about 4x faster than the corresponding version of Jiang et al.'s algorithm. Note thatBMM-III is a bit faster than the naïve block version of Jiang et al.'s, but slower than that of BMM-I. So, we omit its

TABLE XI
PERFORMANCE FOR RECTANGULAR MATRIX MULTIPLICATION (I).

Huang et al.'s algorithm [14]†		BMM-III‡			
Dimension	Time (s)	Dimension	KeyGen (s)	Total (s)	Speedup
$(256, 256, 16)$	**6.19**	$(256, 257, 17)$	16.15	23.60	
$(256, 16, 256)$	**6.23**	$(256, 17, 257)$	14.69	19.37	
$(1024, 1024, 16)$	108.22	$(1024, 1025, 17)$	14.68	**55.79**	1.9x
$(1024, 16, 1024)$	108.31	$(1024, 17, 1025)$	14.47	**43.64**	2.4x
$(2048, 2048, 8)$	218.09	$(2048, 2049, 11)$	14.49	**98.01**	2.2x
$(2048, 8, 2048)$	218.09	$(2049, 8, 2051)$	14.63	**81.09**	2.6x

† For the algorithm in [14], N is set as the same as theirs and $\log\Delta = 30$.
‡ For BMM-III, $N = 8192$ and $\log\Delta = 40$.

TABLE XII
PERFORMANCE (IN SEC.) FOR RECTANGULAR MATRIX MULTIPLICATION (II). $N = 8192$, $\log\Delta = 30$.

Dimension	Naïve block [36]	Naïve block BMM-I	Speedup
$(4, 1636, 5)$	36.92	**9.76**	3.7x
$(8, 3405, 9)$	78.58	**19.92**	3.9x
$(16, 6903, 17)$	157.00	**38.57**	4.0x
$(32, 13847, 33)$	317.31	**76.73**	4.1x

performance here. Note that the naïve block BMM-II applies to the tests in Table XII. It costs 0.20, 1.52, 23.02, and 513.07 seconds, respectively.

In Table XIII, we report experimental results for another case, i.e., $n \approx m \ll p$ and $mn < \ell$ (for this case the LongRot algorithm and hence the BMM-III algorithm do not work), for which SegLKS demonstrates the best performance. This is because the dimensions (n, m) of the matrix $\boldsymbol{A}$ are relatively small, and the number of Rots and CMuls required by SegLKS are independent of the number of columns p of the matrix $\boldsymbol{B}$ (as already indicated in Section VI-B). We implement all optimizations in Section VI-B. Experiments show that SegLKS with O4 is about 3x faster than that without O4. Thus, we only list the performance of SegLKS with O4 in Table XIII. It follows from Table XIII that the naïve block version of BMM-I is significantly faster than the naïve block version of Jiang et al.'s, but slower than SegLKS with O4 (in Table V $\kappa = 8$ is fixed for $\ell = 4096$). In particular, for the case of $(32, 33, 13847)$, SegLKS with O4 is about 38x faster than the naïve block version of [36].

TABLE XIII
PERFORMANCE (IN SEC.) FOR RECTANGULAR MATRIX MULTIPLICATION (III). $N = 8192$, $\log\Delta = 30$.

Dimension	Naïve block [36]	Naïve block BMM-I	SegLKS+O4
$(4, 5, 1636)$	36.31	**0.10**	0.25
$(8, 9, 3405)$	77.56	0.65	**0.63**
$(16, 17, 6903)$	157.32	4.96	**2.23**
$(32, 33, 13847)$	316.76	38.82	**8.24**

F. Comparison with most-recent algorithms

In a recent work [41], Zhu et al. presented four different algorithms for different matrix dimensions (square and rectangular). In a very recent work [42], Gao and Gao presented a homomorphic encryption scheme named GMS for encrypted matrix multiplication. The two works considered similar cases and designed and implemented algorithms in HElib. The code of the two works does not seem to be open-sourced. So we take the time from their paper directly. For a $(128, 128, 128)$ encrypted matrix multiplication, Zhu et al.'s algorithm costs about 200 seconds (with a 2.5GHz CPU), while Gao and Gao's algorithm takes about 102 seconds (with a 2.6 GHz CPU). Note that the naïve block version of BMM-II costs only 4.4 seconds (see Table VIII). Similar observations can be made for other square or rectangular cases as well. While such significant performance differences may be attributed to the use of different software libraries, different encryption schemes, or different parameters, the theoretical results in Table I show that BMM-I and BMM-II have their own advantages.

VIII. CONCLUSION

In this paper, we introduce bicyclic encoding, a novel method for matrix encoding. We design several new algorithms for encrypted matrix multiplication under bicyclic encoding, and investigate the block and segmented methods for handling high-dimensional matrices. In the context of CKKS, the following conclusions can be drawn from our theoretical analysis and comprehensive experimental study:

- For encrypted matrix multiplication of small dimensions, BMM-I or BMM-II is the optimal choice;
- When dealing with larger-scale encrypted square matrix multiplication, although theoretically, variants based on the Strassen block-wise strategy are faster, the practical performance is better with the segmented strategy (BMM-III);
- For those types of rectangular encrypted matrix multiplication discussed in [14], BMM-III shows better performance;
- For rectangular encrypted matrix multiplication with $n \approx p \ll m$, the naïve block BMM-I is effective;
- For rectangular encrypted matrix multiplication with $n \approx m \ll p$, SegLKS that combines Lu et al.'s algorithm [33] with the segmented strategy demonstrates exceptional performance.

From the perspective of practical application, the implementations in this paper can still be further optimized. For example, employing a multi-thread parallelization can yield additional acceleration; for block algorithms, the intermediate results after rotations can be reused, and hence further acceleration may be achieved; the hoisting and double-hoisting technique (e.g., [45]) may be used to accelerate our implementation further as well.

Furthermore, implementing these algorithms using BGV or B/FV schemes would significantly enhance integer matrix multiplication on encrypted data. Additionally, exploring other matrix operations beyond transpose and multiplication under bicyclic encoding are worth further investigation.

Finally, the error analysis for matrix multiplication on encrypted data is another intriguing direction, which is closely related to choosing parameters, e.g., the scale factor Δ for CKKS, or the plaintext modulus for BGV and B/FV.

Acknowledgments

We thank Zhicong Huang for sharing with us, together with Cheng Hong, Chenkai Weng, and Wen-jie Lu, their codes for [14], and Xiaopeng Zheng for discussion with us on the multiplicative depth of the encrypted Strassen algorithm. We would also like to thank an anonymous reviewer from ACM CCS '24 whose suggestions inspired us to design BMM-II, and we also thank other anonymous reviewers for their suggestions that helped present the paper more clearly.

References

[1] R. Rivest, L. Adleman, and M. Dertouzos, "On data banks and privacy homomorphisms," in *Foundations of Secure Computation*, R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, Eds. Atlanta: Academic Press, 1978, pp. 165–179.

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing (May 31 - June 2, 2009, Bethesda, USA)*, M. Mitzenmacher, Ed. New York: ACM, 2009, pp. 169–178, https://doi.org/10.1145/1536414.1536440.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory*, vol. 6, no. 3, pp. 13:1–13:36, 2014, https://doi.org/10.1145/2633600.

[4] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Advances in Cryptology – Proc CRYPTO 2012 (August 19–23, 2012, Santa Barbara, CA, USA)*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds. Heidelberg: Springer, 2012, vol. 7417, pp. 868–886, http://doi.org/10.1007/978-3-642-32009-5_50.

[5] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive https://eprint.iacr.org/2012/144, 2012.

[6] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proceedings of ASIACRYPT 2017 – 23rd International Conference on the Theory and Applications of Cryptology and Information Security (December 3-7, 2017, Hong Kong, China), Part I*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin, Eds. Heidelberg: Springer, 2017, vol. 10624, pp. 409–437, https://doi.org/10.1007/978-3-319-70694-8_15.

[7] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology - Proceedings of EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Part I (April 26-30, 2015, Sofia, Bulgaria)*, ser. Lecture Notes in Computer Science, E. Oswald and M. Fischlin, Eds. Heidelberg: Springer, 2015, vol. 9056, pp. 617–640, https://doi.org/10.1007/978-3-662-46800-5_24.

[8] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020, https://doi.org/10.1007/s00145-019-09319-x.

[9] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Designs, Codes and Cryptography*, vol. 71, no. 1, pp. 57–81, 2014, https://doi.org/10.1007/s10623-012-9720-4.

[10] S. Halevi and V. Shoup, "Bootstrapping for HElib," in *Advances in Cryptology – Proc EUROCRYPT 2015 (April 26–30, 2015, Sofia, Bulgaria), Part I*, ser. Lecture Notes in Computer Science, E. Oswald and M. Fischlin, Eds. Heidelberg: Springer, 2015, vol. 9056, pp. 641–670, https://doi.org/10.1007/978-3-662-46800-5_25.

[11] A. Kim, M. Deryabin, J. Eom, R. Choi, Y. Lee, W. Ghang, and D. Yoo, "General bootstrapping approach for RLWE-based homomorphic encryption," *IEEE Transactions on Computers*, pp. 1–13, 2023, https://doi.org/10.1109/TC.2023.3318405.

[12] F.-H. Liu and H. Wang, "Batch bootstrapping I: A new framework for SIMD bootstrapping in polynomial modulus," in *Advances in Cryptology – EUROCRYPT 2023 (Lyon, France, April 23-27, 2023)*, ser. Lecture Notes in Computer Science, C. Hazay and M. Stam, Eds. Cham: Springer, 2023, vol. 14006, pp. 321–352, https://doi.org/10.1007/978-3-031-30620-4_11.

[13] B. Xiang, J. Zhang, Y. Deng, Y. Dai, and D. Feng, "Fast blind rotation for bootstrapping FHEs," in *Advances in Cryptology – CRYPTO 2023 (Santa Barbara, USA, August 20–24, 2023)*, ser. LNCS, H. Handschuh and A. Lysyanskaya, Eds. Cham: Springer, 2023, vol. 14084, pp. 3–36, https://doi.org/10.1007/978-3-031-38551-3_1.

[14] Z. Huang, C. Hong, C. Weng, W.-j. Lu, and H. Qu, "More efficient secure matrix multiplication for unbalanced recommender systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 551–562, 2023, https://doi.org/10.1109/TDSC.2021.3139318.

[15] Microsoft, "Microsoft SEAL (release 4.1.1)," Accessed in July, 2023, https://github.com/microsoft/SEAL.

[16] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "MP2ML: A mixed-protocol machine learning framework for private inference," in *Proceedings of the 15th International Conference on Availability, Reliability and Security (Virtual Event, Ireland, August 25 - 28, 2020)*, M. Volkamer and C. Wressnegger, Eds. New York: ACM, 2020, pp. 14:1–10, https://doi.org/10.1145/3407023.3407045.

[17] S. Fu, Y. Yu, and M. Xu, "A secure algorithm for outsourcing matrix multiplication computation in the cloud," in *Proceedings of the Fifth ACM International Workshop on Security in Cloud Computing (Abu Dhabi, United Arab Emirates, 2 April, 2017)*, C. Wang and M. Kantarcioglu, Eds. New York: ACM, 2017, pp. 27–33, https://doi.org/10.1145/3055259.3055263.

[18] J.-G. Dumas, P. Lafourcade, J. Lopez Fenner, D. Lucas, J.-B. Orfila, C. Pernet, and M. Puys, “Secure multiparty matrix multiplication based on Strassen-Winograd algorithm,” in *Advances in Information and Computer Security – Proceedings of the 14th International Workshop on Security (Tokyo, Japan, August 28–30, 2019)*, ser. Lecture Notes in Computer Science, N. Attrapadung and T. Yagi, Eds. Cham: Springer, 2019, vol. 11689, pp. 67–88, https://doi.org/10.1007/978-3-030-26834-3_5.

[19] L. Zhao and L. Chen, “Sparse matrix masking-based non-interactive verifiable (outsourced) computation, revisited,” *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 6, pp. 1188–1206, 2020, https://doi.org/10.1109/TDSC.2018.2861699.

[20] H. Chen, M. Kim, I. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh, “Maliciously secure matrix multiplication with applications to private deep learning,” in *Advances in Cryptology – ASIACRYPT 2020 (Daejeon, South Korea, December 7–11, 2020)*, ser. Lecture Notes in Computer Science, S. Moriai and H. Wang, Eds. Cham: Springer, 2020, vol. 12493, pp. 31–59, https://doi.org/10.1007/978-3-030-64840-4_2.

[21] S. Balla and F. Koushanfar, “HELiKs: HE linear algebra kernels for secure inference,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (November 26 - 30, 2023, Copenhagen, Denmark)*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. New York: ACM, 2023, pp. 2306–2320, https://doi.org/10.1145/3576915.3623136.

[22] J. Zhu, S. Li, and J. Li, “Information-theoretically private matrix multiplication from MDS-coded storage,” *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1680–1695, 2023, https://doi.org/10.1109/TIFS.2023.3249565.

[23] R. Hiromasa, M. Abe, and T. Okamoto, “Packing messages and optimizing bootstrapping in GSW-FHE,” *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 99, no. 1, pp. 73–82, 2016, https://doi.org/10.1587/transfun.E99.A.73.

[24] Y. Bai, X. Shi, W. Wu, J. Chen, and Y. Feng, “seIMC: A GSW-based secure and efficient integer matrix computation scheme with implementation,” *IEEE Access*, vol. 8, no. 1, pp. 98 383–98 394, 2020, https://doi.org/10.1109/ACCESS.2020.2996000.

[25] C. Gentry, A. Sahai, and B. Waters, “Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based,” in *Advances in Cryptolog–Proc CRYPTO 2013 (August 18-22, 2013, Santa Barbara, CA, USA), Part I*, ser. Lecture Notes in Computer Science, R. Canetti and J. A. Garay, Eds. Heidelberg: Springer, 2013, vol. 8042, pp. 75–92, http://dx.doi.org/10.1007/978-3-642-40041-4_5.

[26] D. H. Duong, P. K. Mishra, and M. Yasuda, “Efficient secure matrix multiplication over LWE-based homomorphic encryption,” *Tatra Mountains Mathematical Publications*, vol. 67, no. 1, pp. 69–83, 2016, https://doi.org/10.1515/tmmp-2016-0031.

[27] M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba, “Secure statistical analysis using RLWE-based homomorphic encryption,” in *Information Security and Privacy – ACISP 2015 (Brisbane, Australia, June 29 – July 1, 2015)*, ser. Lecture Notes in Computer Science, E. Foo and D. Stebila, Eds. Cham: Springer, 2015, vol. 9144, pp. 471–487, https://doi.org/10.1007/978-3-319-19962-7_27.

[28] ——, “New packing method in somewhat homomorphic encryption and its applications,” *Security and Communication Networks*, vol. 8, no. 13, pp. 2194–2213, 2015, https://doi.org/10.1002/sec.1164.

[29] P. K. Mishra, D. H. Duong, and M. Yasuda, “Enhancement for secure multiple matrix multiplications over ring-LWE homomorphic encryption,” in *Information Security Practice and Experience (Melbourne, Australia, December 13–15, 2017)*, ser. Lecture Notes in Computer Science, J. K. Liu and P. Samarati, Eds. Cham: Springer, 2017, vol. 10701, pp. 320–330, https://doi.org/10.1007/978-3-319-72359-4_18.

[30] X. Zheng, H. Li, and D. Wang, “A new framework for fast homomorphic matrix multiplication,” Cryptology ePrint Archive, Paper 2023/1649, 2023, https://eprint.iacr.org/2023/1649.

[31] V. Strassen, “Gaussian elimination is not optimal,” *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969, https://doi.org/10.1007/BF02165411.

[32] S. Halevi and V. Shoup, “Algorithms in HElib,” in *Advances in Cryptology – CRYPTO 2014 (Santa Barbara, USA, August 17-21, 2014)*, ser. Lecture Notes in Computer Science, J. A. Garay and R. Gennaro, Eds. Heidelberg: Springer, 2014, vol. 8616, pp. 554–571, https://doi.org/10.1007/978-3-662-44371-2_31.

[33] W. Lu, S. Kawasaki, and J. Sakuma, “Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data,” in *NDSS 2017: 24th Annual Network and Distributed System Security Symposium (San Diego, USA, February 26–March 1, 2017)*. The Internet Society, 2017, https://doi.org/10.14722/ndss.2017.23119.

[34] S. Wang and H. Huang, “Secure outsourced computation of multiple matrix multiplication based on fully homomorphic encryption,” *KSII Transactions on Internet and Information Systems*, vol. 13, no. 11, pp. 5616–5630, 2019, https://doi.org/10.3837/tiis.2019.11.019.

[35] D. Rathee, P. K. Mishra, and M. Yasuda, “Faster PCA and linear regression through hypercubes in HElib,” in *WPES’18: Proceedings of the 2018 Workshop on Privacy in the Electronic Society (Toronto, Canada, 15 October 2018)*, D. Lie, M. Mannan, and A. Johnson, Eds. New York: ACM, 2018, pp. 42–53, https://doi.org/10.1145/3267323.3268952.

[36] X. Jiang, M. Kim, K. Lauter, and Y. Song, “Secure outsourced matrix computation and application to neural networks,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (October 15–19, 2018, Toronto, Canada)*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. New York: ACM, 2018, pp. 1209–1222, https://doi.org/10.1145/3243734.3243837.

[37] J. Jang, Y. Lee, A. Kim, B. Na, D. Yhee, B. Lee, J. H. Cheon, and S. Yoon, “Privacy-preserving deep sequential model with matrix homomorphic encryption,” in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (Nagasaki, Japan, 30 May 2022- 3 June 2022)*, Y. Suga, K. Sakurai, , X. Ding, and K. Sako, Eds. New York: ACM, 2022, p. 377–391, https://doi.org/10.1145/3488932.3523253.

[38] P. Rizomiliotis and A. Triakosia, “On matrix multiplication with homomorphic encryption,” in *Proceedings of the 2022 on Cloud Computing Security Workshop (Los Angeles, USA)*, F. Regazzoni and M. van Dijk, Eds. New York: ACM, 2022, pp. 53–61, https://doi.org/10.1145/3560810.3564267.

[39] J. Chiang, “Volley revolver: A novel matrix-encoding method for privacy-preserving neural networks (inference),” arXiv preprint arXiv:2201.12577, 2022, https://doi.org/10.48550/arXiv.2201.12577.

[40] H. Huang and H. Zong, “Secure matrix multiplication based on fully homomorphic encryption,” *The Journal of Supercomputing*, vol. 79, no. 5, pp. 5064–5085, 2023, https://doi.org/10.1007/s11227-022-04850-4.

[41] L. Zhu, Q. Hua, Y. Chen, and H. Jin, “Secure outsourced matrix multiplication with fully homomorphic encryption,” in *Computer Security – ESORICS 2023 (The Hague, The Netherlands, September 25–29, 2023)*, ser. Lecture Notes in Computer Science, G. Tsudik, M. Conti, K. Liang, and G. Smaragdakis, Eds. Cham: Springer, 2024, vol. 14344, pp. 249–269, https://doi.org/10.1007/978-3-031-50594-2_13.

[42] J. Gao and Y. Gao, “GMS: an efficient fully homomorphic encryption scheme for secure outsourced matrix multiplication,” *The Journal of Supercomputing*, vol. 80, pp. 26 435–26 461, 2024, https://doi.org/10.1007/s11227-024-06449-3.

[43] G. C. Fox, S. W. Otto, and A. J. G. Hey, “Matrix algorithms on a hypercube I: Matrix multiplication,” *Parallel computing*, vol. 4, no. 1, pp. 17–31, 1987, https://doi.org/10.1016/0167-8191(87)90060-3.

[44] M. Babenko, E. Golimblevskaia, A. Tchernykh, E. Shiriaev, T. Ermakova, L. B. Pulido-Gaytan, G. Valuev, A. Avetisyan, and L. A. Gagloeva, “A comparative study of secure outsourced matrix multiplication based on homomorphic encryption,” *Big Data and Cognitive Computing*, vol. 7, no. 2, 2023, https://doi.org/10.3390/bdcc7020084.

[45] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, “Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys,” in *Advances in Cryptology – EUROCRYPT 2021 (Zagreb, Croatia, October 17–21, 2021)*, ser. Lecture Notes in Computer Science, A. Canteaut and F.-X. Standaert, Eds. Cham: Springer, 2021, vol. 12696, pp. 587–617, https://doi.org/10.1007/978-3-030-77870-5_21.

[46] X. Ma, C. Ma, Y. Jiang, and C. Ge, “Improved privacy-preserving PCA using optimized homomorphic matrix multiplication,” *Computers & Security*, vol. 138, p. 103658, 2024, https://doi.org/10.1016/j.cose.2023.103658.

[47] N. Genise, C. Gentry, S. Halevi, B. Li, and D. Micciancio, “Homomorphic encryption for finite automata,” in *Advances in Cryptology – ASIACRYPT 2019*, ser. Lecture Notes in Computer Science, S. D. Galbraith and S. Moriai, Eds. Cham: Springer, 2019, vol. 11922, pp. 473–502, https://doi.org/10.1007/978-3-030-34621-8_17.

[48] D. Ö. Şimşek and M. Cenk, "Faster secure matrix multiplication with the BGV algorithm," in *Proceedings of the 16th International Conference on Information Security and Cryptology (October 18-19, 2023, Ankara, Turkey)*, A. A. Selçuk, Ş.. Sağiroğlu, Y. Oğuz, and C. Tezcan, Eds. Danvers: IEEE, 2023, pp. 1–5, https://doi.org/10.1109/ISCTrkiye61151.2023.10336104.

[49] J. Hu, F. Wang, and K. Chen, "Faster matrix approximate homomorphic encryption," *Computer Standards & Interfaces*, vol. 87, p. 103775, 2024, https://doi.org/10.1016/j.csi.2023.103775.

[50] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of ACM*, vol. 56, no. 6, pp. 34:1–40, 2009, https://doi.org/10.1145/1568318.1568324.

[51] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *Journal of ACM*, vol. 60, no. 6, pp. 43:1–35, 2013, https://doi.org/10.1145/2535925.

[52] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures*. San Mateo: Morgan Kaufmann Publishers, 1992.

[53] J. Alman and V. V. Williams, "A refined laser method and faster matrix multiplication," in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (January 10 – 13, 2021, Virtually)*, D. Marx, Ed. Philadelphia: SIAM, 2021, pp. 522–539, https://doi.org/10.1137/1.9781611976465.32.

[54] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015, https://doi.org/10.1515/jmc-2015-0016. Lattice Estimator: https://github.com/malb/lattice-estimator.

Appendix A
The LongRot Algorithm

Given $\boldsymbol{a} = (a_0, \dots, a_{d-1}) \in \mathcal{R}^d$, the LongRot operation implements the following functionality:

- Construct $\underline{\boldsymbol{a}} = (\boldsymbol{a}, \dots, \boldsymbol{a}) \in \mathcal{R}^{t \cdot d}$;
- Rotate the vector $\underline{\boldsymbol{a}}$ to the left by k positions, resulting in $\underline{\boldsymbol{a}}' = \rho_k(\underline{\boldsymbol{a}})$;
- Select the first τ elements of $\underline{\boldsymbol{a}}'$ and divide them into $\left\lceil \frac{\tau}{\ell} \right\rceil$ groups, each containing ℓ elements.

Recall $d = (w-1)\ell + z$ with $0 \le z < \ell$.

A. The Case of $w > 1$

TABLE XIV
An illustrative description of the plaintext version of LongRot.

$\boldsymbol{a} \in \mathcal{R}^d$	$\underbrace{\overbrace{a_0,\dots,a_{\ell-1}}^{\ell},\dots,\overbrace{a_{u\ell},\dots,a_{(u+1)\ell-1}}^{\ell},\dots,\overbrace{a_{(w-2)\ell},\dots,a_{(w-1)\ell-1}}^{\ell},\overbrace{a_{(w-1)\ell},\dots,a_{(w-1)\ell+z-1}}^{z}}_{d}$	$d = (w-1)\ell + z,\ z < \ell$
$j = 0$	$\overbrace{\underbrace{a_{u\ell+v},\dots,a_{(u+1)\ell-1}}_{\ell-v},\underbrace{a_{(u+1)\ell},\dots,a_{(u+1)\ell+v-1}}_{v},\dots,\underbrace{a_{(w-2)\ell+v},\dots,a_{(w-1)\ell-1}}_{\ell-v},a_{(w-1)\ell},\dots,a_{(w-1)\ell+z-1}}^{d-k}$	$k = u\ell + v,\ v < \ell$
$j = 1$	$\underbrace{a_0,\dots,a_{v_1-1}}_{v_1},\underbrace{a_{v_1},\dots,a_{\ell-1}}_{\ell-v_1},\underbrace{a_{\ell},\dots,a_{\ell+v_1-1}}_{v_1},\dots,\underbrace{a_{(w-2)\ell+v_1},\dots,a_{(w-1)\ell-1}}_{\ell-v_1},a_{(w-1)\ell},\dots,a_{(w-1)\ell+z-1}$	$\tau = \iota\ell + \kappa,\ \kappa < \ell$
$\vdots$	$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots$	$\tau - (d-k) = \gamma d + \delta$
$j = \gamma$	$\underbrace{a_0,\dots,a_{v_\gamma-1}}_{v_\gamma},\underbrace{a_{v_\gamma},\dots,a_{\ell-1}}_{\ell-v_\gamma},\underbrace{a_{\ell},\dots,a_{\ell+v_\gamma-1}}_{v_\gamma},\dots,\underbrace{a_{(w-2)\ell+v_\gamma},\dots,a_{(w-1)\ell-1}}_{\ell-v_\gamma},a_{(w-1)\ell},\dots,a_{(w-1)\ell+z-1}$	$0 \le \delta < d$
$j = \gamma + 1$	$\underbrace{a_0,\dots,a_{v_{\gamma+1}-1}}_{v_{\gamma+1}},\underbrace{a_{v_{\gamma+1}},\dots,a_{\ell-1}}_{\ell-v_{\gamma+1}},\dots,\underbrace{a_{(\alpha-1)\ell+v_{\gamma+1}},\dots,a_{\alpha\ell-1}}_{\ell-v_{\gamma+1}},a_{\alpha\ell},\dots,a_{\alpha\ell+\beta-1}$	$\delta = \alpha\ell + \beta,\ \beta < \ell$

Table XIV illustrates how LongRot works $w > 1$. We first introduce some notations:

- $k = u\ell + v$ with $0 \le v < \ell$,
- $\tau = \iota\ell + \kappa$ with $0 \le \kappa < \ell$,
- $\tau - (d-k) = \gamma d + \delta$ with $0 \le \delta < d$ if $\tau > d - k$,
- $\delta = \alpha\ell + \beta$ with $0 \le \beta < \ell$.

a) Case 1: $\tau \le d - k$: In this case, we only need to consider the row of $j = 0$ of Table XIV with the following two subcases.

If $\iota = 0$, LongRot outputs only one ciphertext, which is an encryption of $\boldsymbol{a}_0' = (a_{u\ell+v}, \dots, a_{u\ell+v+\tau-1})$. However, this ciphertext may involve two input ciphertexts of LongRot depending on if $\tau \le \ell - v$ or not. If $\iota > 0$, LongRot outputs $\iota + 1$ ciphertexts. For $i = 0, 1, \dots, \iota - 1$, the corresponding plaintexts are $\boldsymbol{a}_i' = (a_{(u+i)\ell+v}, \dots, a_{(u+i+1)\ell-1}, a_{(u+i+1)\ell}, \dots, a_{(u+i+1)\ell+v-1})$; and at last if $\kappa \le \ell - v$ then $\boldsymbol{a}_\iota' = (a_{(u+\iota)\ell+v}, \dots, a_{(u+\iota)\ell+v+\kappa-1})$, else $\boldsymbol{a}_\iota' = (a_{(u+\iota)\ell+v}, \dots, a_{(u+\iota+1)\ell-1}, a_{(u+\iota+1)\ell}, \dots, a_{(u+\iota+1)\ell+\kappa-\ell+v-1})$.

b) Case 2: $\tau > d - k$: In this case, we need to consider the full Table XIV. From Table XIV, we observe that v_i in each line is the step size of the rotation, which can be computed in advance. Let $v_0 = v$. Then for $j = 1, \dots, \gamma + 1$ define

$$v_j = \begin{cases} \ell - (z - v_{j-1}) & \text{if } z > v_{j-1}, \\ v_{j-1} - z & \text{if } z \le v_{j-1}. \end{cases}$$

We also define the last index of the ciphertexts at the end of each row in Table XIV: $i_0 = w + u + g(z, v_j) - 2$ and $i_j = w + g(z, v_j) - 2$ for $j = 1, \dots, \gamma + 1$, where $g(z, v_j) = 1$ if $z > v_j$ and 0 otherwise.

First, we consider the row $j = 0$ of Table XIV. When $z > v_0$, we have $\boldsymbol{a}_i' = (a_{(u+i)\ell+v_0}, \dots, a_{(u+i+1)\ell-1}, a_{(u+i+1)\ell}, \dots a_{(u+i+1)\ell+v_0-1})$ for $i = 0, 1, \dots, i_0 - 1 = w - u - 2$, and $\boldsymbol{a}_{i_0}' = (a_{(w-1)\ell+v_0}, \dots, a_{(w-1)\ell+z-1},\ a_0, \dots, a_{v_1-1})$. When $z \le v_0$, we have $\boldsymbol{a}_i' = (a_{(u+i)\ell+v_0}, \dots, a_{(u+i+1)\ell-1},\ a_{(u+i+1)\ell}, \dots, a_{(u+i+1)\ell+v_0-1})$ for $i = 0, 1, \dots, i_0 - 1 = w - u - 3$, and

$\boldsymbol{a}'_{i_0} = (a_{(w-2)\ell+v_0}, \ldots, a_{(w-1)\ell-1}, a_{(w-1)\ell}, \ldots, a_{(w-1)\ell+z-1}, a_0, \ldots, a_{v_1-1})$. Note that if $\gamma = 0$ and $\beta < v_1$ then the last v_1 entries $(a_0, \ldots, a_{v_1-1})$ of $\boldsymbol{a}'_{i_0}$ should be replaced by $(a_0, \ldots, a_{\beta-1})$. In addition, the current index of the last ciphertext is $h := i_0$.

For the j-th row of Table XIV with $j = 1, \ldots, \gamma - 1$, we have $\boldsymbol{a}'_{h+i+1} = (a_{i\ell+v_j}, \ldots, a_{(i+1)\ell-1}, a_{(i+1)\ell}, \ldots a_{(i+1)\ell+v_j-1})$ for $i = 0, 1, \ldots, i_j - 1$. If $z > v_j$ then $\boldsymbol{a}'_{h+i_j+1} = (a_{(w-1)\ell+v_j}, \ldots, a_{(w-1)\ell+z-1}, a_0, \ldots, a_{v_{j+1}-1})$ else

$$\boldsymbol{a}'_{h+i_j+1} = (a_{(w-2)\ell+v_j}, \ldots, a_{(w-1)\ell-1}, a_{(w-1)\ell}, \ldots, a_{(w-1)\ell+z-1}, a_0, \ldots, a_{v_{j+1}-1})$$

and update the index of the last ciphertext $h := h + i_j + 1$.

For the γ-th row, we have $\boldsymbol{a}'_{h+i+1} = (a_{i\ell+v_\gamma}, \ldots, a_{(i+1)\ell-1}, a_{(i+1)\ell}, \ldots, a_{(i+1)\ell+v_\gamma-1})$ for $i = 0, 1, \ldots, i_\gamma - 1$. Then update $h := h + i_\gamma + 1$. If $z > v_\gamma$, then the last ciphertext of the row γ is an encryption of $\boldsymbol{a}'_h$ with

$$\boldsymbol{a}'_h = (a_{(w-1)\ell+v_\gamma}, \ldots, a_{(w-1)\ell+z-1}, a_0, \ldots a_{\beta-1})$$

if $\tau < (h+1)\ell$ and $\beta \le v_{\gamma+1}$, and $\boldsymbol{a}'_h = (a_{(w-1)\ell+v_\gamma}, \ldots, a_{(w-1)\ell+z-1}, a_0, \ldots, a_{v_{\gamma+1}-1})$ otherwise. If $z \le v_\gamma$, then the last ciphertext of the row γ is an encryption of $\boldsymbol{a}'_h$ with $\boldsymbol{a}'_h = (a_{(w-2)\ell+v_\gamma}, \ldots, a_{(w-1)\ell-1}, a_{(w-1)\ell}, \ldots, a_{(w-1)\ell+z-1}, a_0, \ldots a_{\beta-1})$ if $\tau < (h+1)\ell$ and $\beta \le v_{\gamma+1}$, and otherwise $\boldsymbol{a}'_h = (a_{(w-2)\ell+v_\gamma}, \ldots, a_{(w-1)\ell-1}, a_{(w-1)\ell}, \ldots a_{(w-1)\ell+z-1}, a_0, \ldots, a_{v_{\gamma+1}-1})$.

Now we consider the row $j = \gamma + 1$ of Table XIV. First for $i = 0, 1, \ldots, \alpha - 2$, we have

$$\boldsymbol{a}'_{h+i+1} = (a_{i\ell+v_{\gamma+1}}, \ldots, a_{(i+1)\ell+v_{\gamma+1}-1}). \tag{6}$$

Then update $h := h + \max\{0, \alpha - 1\}$. (Indeed, if $\alpha = 0$, Eq. (6) will never be executed.) Further, if $\alpha = 0$ and $\beta > v_{\gamma+1}$ then update $h := h + 1$ and $\boldsymbol{a}'_h = (a_{v_{\gamma+1}}, \ldots, a_{\beta-1})$. If $\alpha > 0$ and $\beta > v_{\gamma+1}$ then

$$\boldsymbol{a}'_{h+1} = (a_{(\alpha-1)\ell+v_{\gamma+1}}, \ldots, a_{\alpha\ell-1}, a_{\alpha\ell}, \ldots, a_{\alpha\ell+v_{\gamma+1}-1})$$

and $\boldsymbol{a}'_{h+2} = (a_{\alpha\ell+v_{\gamma+1}}, \ldots, a_{\alpha\ell+\beta-1})$, and update $h := h + 2$. If $\alpha > 0$ and $\beta \le v_{\gamma+1}$ then update $h := h + 1$ and $\boldsymbol{a}'_h = (a_{(\alpha-1)\ell+v_{\gamma+1}}, \ldots, a_{\alpha\ell-1}, a_{\alpha\ell}, \ldots, a_{\alpha\ell+\beta-1})$. This completes the functionality in the plaintext domain.

We can easily translate the discussion in Section V-B into the encrypted version as the following algorithm.

Algorithm 4 (LongRot)

Input: Ciphertexts $(\mathsf{ct}.\boldsymbol{a}_i)_{0\le i<w}$ for $\boldsymbol{a} = (\boldsymbol{a}_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_{w-2}, \boldsymbol{a}_{w-1}) \in \mathcal{R}^d$ with $\boldsymbol{a}_{w-1} \in \mathcal{R}^z$ and $\boldsymbol{a}_i \in \mathcal{R}^\ell$ for $i = 0, 1, \ldots, w-2$ (i.e., $d = (w-1)\ell + z$ with $0 \le z < \ell$, where ℓ is the number of slots), the number of repeated times t, the number of positions to be rotated $k \in [0, d)$, the number of selected elements τ.

Output: Ciphertexts $(\mathsf{ct}.\underline{\boldsymbol{a}}'_i)_{0\le i<\lceil \frac{\tau}{\ell} \rceil}$, i.e., $\lceil \frac{\tau}{\ell} \rceil$ ciphertexts that encrypt the first τ elements of $\underline{\boldsymbol{a}}'$.

1) Compute two non-negative integers u and v such that $k = u\ell + v$ with $v < \ell$. Compute two non-negative integers γ and δ such that $\tau - (d-k) = \gamma d + \delta$ with $\delta < d$. Compute two non-negative integers α and β such that $\delta = \alpha\ell + \beta$ with $\beta < \ell$. Compute two non-negative integers ι and κ such that $\tau = \iota \cdot \ell + \kappa$ with $\kappa < \ell$. Set $v_0 := v$ and $h := 0$.
2) If $\tau \le d - k$ then do the following:
   a) If $\iota = 0$ and $\tau \le \ell - v$ then compute $\mathsf{ct}.\tilde{\boldsymbol{a}}_0 \leftarrow \mathsf{Rot}_v(\mathsf{ct}.\boldsymbol{a}_u)$ and $\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Sl}_{[0,\tau-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_0)$.
   b) If $\iota = 0$ and $\tau > \ell - v$ then compute $\mathsf{ct}.\tilde{\boldsymbol{a}}_0 \leftarrow \mathsf{Rot}_v(\mathsf{ct}.\boldsymbol{a}_u)$ and $\mathsf{ct}.\tilde{\boldsymbol{a}}_1 \leftarrow \mathsf{Rot}_{-(\ell-v)}(\mathsf{ct}.\boldsymbol{a}_{u+1})$ and compute
   $$\mathsf{ct}.\underline{\boldsymbol{a}}'_{h+1} \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_0), \mathsf{Sl}_{[\ell-v,\tau-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_1)).$$
   c) If $\iota > 0$ then do the following: For $i = 0, 1, \ldots, \iota$ compute
   $$\mathsf{ct}.\tilde{\boldsymbol{a}}_i \leftarrow \mathsf{Rot}_v(\mathsf{ct}.\boldsymbol{a}_{u+i}).$$
   For $i = 0, 1, \ldots, \iota-1$ compute $\mathsf{ct}.\underline{\boldsymbol{a}}'_i \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_i), \mathsf{Sl}_{[\ell-v,\ell-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i+1}))$. Set $h := h+\iota$. If $\kappa \le \ell-v$ then $\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Sl}_{[0,\kappa-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_\iota)$, else compute $\mathsf{ct}.\tilde{\boldsymbol{a}}_{\iota+1} \leftarrow \mathsf{Rot}_{-(\ell-v)}(\mathsf{ct}.\boldsymbol{a}_{u+\iota+1})$ and compute $\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_\iota), \mathsf{Sl}_{[\ell-v,\kappa-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{\iota+1}))$.
3) If $\tau > d - k$ then do the following:
   a) Set $i_0 := w - u + g(z, v_0) - 2$, where $g(x, y) = 1$ if $x > y$ and 0 otherwise.
   b) For $j = 1, 2, \ldots, \gamma + 1$ do: If $z > v_{j-1}$ then set $v_j := \ell - (z - v_{j-1})$ else set $v_j := v_{j-1} - z$; Set $i_j := w + g(z, v_j) - 2$.
   c) For $i = 0, 1, \ldots, i_0$ compute $\mathsf{ct}.\tilde{\boldsymbol{a}}_i \leftarrow \mathsf{Rot}_{v_0}(\mathsf{ct}.\boldsymbol{a}_{u+i})$. For $i = 0, 1, \ldots, i_0 - 1$ compute
   $$\mathsf{ct}.\underline{\boldsymbol{a}}'_i \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v_0-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_i), \mathsf{Sl}_{[\ell-v_0,\ell-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i+1})).$$
   Update $\mathsf{ct}.\tilde{\boldsymbol{a}}'_0 \leftarrow \mathsf{Rot}_{v_0-z}(\mathsf{ct}.\boldsymbol{a}_0)$ and $h := h + i_0$.
   d) If $z > v_0$: If $(i_0 + 1)\ell > \tau$ and $\beta \le v_1$ and $\gamma = 0$ then compute
   $$\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,z-v_0-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_0}), \mathsf{Sl}_{[z-v_0,z-v_0+\beta-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}'_0))$$
   else compute $\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,z-v_0-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_0}), \mathsf{Sl}_{[z-v_0,z-v_0+v_1-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}'_0))$. Set $\mathsf{ct}.\tilde{\boldsymbol{a}}_0 := \mathsf{ct}.\tilde{\boldsymbol{a}}'_0$.
   e) If $z \le v_0$: Compute $\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_0+1} \leftarrow \mathsf{Rot}_{-(\ell-v_0)}(\mathsf{ct}.\boldsymbol{a}_{w-1})$ and $\mathsf{ct}.\boldsymbol{t} \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v_0-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_0}), \mathsf{Sl}_{[\ell-v_0,\ell-v_0+z-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_0+1})$, and set $\mathsf{ct}.\tilde{\boldsymbol{a}}_0 := \mathsf{ct}.\tilde{\boldsymbol{a}}'_0$; If $(h+1)\ell > \tau$ and $\beta \le v_1$ and $\gamma = 0$ then compute
   $$\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Add}(\mathsf{ct}.\boldsymbol{t}, \mathsf{Sl}_{[\ell-v_0+z,\ell-v_0+z+\beta-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_0)),$$

else compute

$$\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Add}(\mathsf{ct}.\boldsymbol{t}, \mathsf{Sl}_{[\ell-v_0+z,\ell-v_0+z+v_1-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_0)).$$

f) For $j = 1, 2, \dots, \gamma$ do the following:
   i) For $i = 1, 2, \dots, i_j$, compute $\mathsf{ct}.\tilde{\boldsymbol{a}}_i \leftarrow \mathsf{Rot}_{v_j}(\mathsf{ct}.\boldsymbol{a}_i)$. For $i = 0, 1, \dots, i_j - 1$ compute

$$\mathsf{ct}.\underline{\boldsymbol{a}}'_{h+i+1} \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v_j-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_i), \mathsf{Sl}_{[\ell-v_j,\ell-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i+1})).$$

   Update $\mathsf{ct}.\tilde{\boldsymbol{a}}'_0 \leftarrow \mathsf{Rot}_{v_j-z}(\mathsf{ct}.\boldsymbol{a}_0)$ and $h := h + i_j + 1$.
   ii) If $z > v_j$: If $(h+1)\ell > \tau$ and $\beta \le v_1$ and $j = \gamma$ then compute $\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,z-v_0-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_j}), \mathsf{Sl}_{[z-v_0,z-v_0+\beta-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}'_0))$ else compute $\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,z-v_j-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_j}), \mathsf{Sl}_{[z-v_j,z-v_j+v_{j+1}-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}'_0))$. Set $\mathsf{ct}.\tilde{\boldsymbol{a}}_0 := \mathsf{ct}.\tilde{\boldsymbol{a}}'_0$.
   iii) If $z \le v_j$: Compute $\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_j+1} \leftarrow \mathsf{Rot}_{-(\ell-v_j)}(\mathsf{ct}.\boldsymbol{a}_{w-1})$ and $\mathsf{ct}.\boldsymbol{t} \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v_j-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_j}), \mathsf{Sl}_{[\ell-v_j,\ell-v_j+z-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i_j+1})$, and set $\mathsf{ct}.\tilde{\boldsymbol{a}}_0 := \mathsf{ct}.\tilde{\boldsymbol{a}}'_0$; If $(h+1)\ell > \tau$ and $\beta \le v_1$ and $j = \gamma$ then compute

$$\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Add}(\mathsf{ct}.\boldsymbol{t}, \mathsf{Sl}_{[\ell-v_j+z,\ell-v_j+z+\beta-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_0))$$

   else compute

$$\mathsf{ct}.\underline{\boldsymbol{a}}'_h \leftarrow \mathsf{Add}(\mathsf{ct}.\boldsymbol{t}, \mathsf{Sl}_{[\ell-v_j+z,\ell-v_j+z+v_{j+1}-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_0)).$$

g) For $i = 1, 2, \dots, \alpha - 1$ compute $\mathsf{ct}.\tilde{\boldsymbol{a}}_i \leftarrow \mathsf{Rot}_{v_{\gamma+1}}(\mathsf{ct}.\boldsymbol{a}_i)$. For $i = 0, 1, \dots, \alpha - 2$ compute

$$\mathsf{ct}.\underline{\boldsymbol{a}}'_{h+i+1} \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v_{\gamma+1}-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_i), \mathsf{Sl}_{[\ell-v_{\gamma+1},\ell-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{i+1})).$$

   Update $h := h + \max\{0, \alpha - 1\}$.
h) If $\alpha = 0$ and $\beta > v_{\gamma+1}$ then compute $\mathsf{ct}.\underline{\boldsymbol{a}}'_{h+1} \leftarrow \mathsf{Sl}_{[0,\beta-v_{\gamma+1}-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_0)$ and update $h := h + 1$.
i) If $\alpha > 0$ compute $\mathsf{ct}.\tilde{\boldsymbol{a}}_\alpha \leftarrow \mathsf{Rot}_{-(\ell-v_{\gamma+1})}(\mathsf{ct}.\boldsymbol{a}_\alpha)$ and do the following:
   i) If $\beta > v_{\gamma+1}$ compute $\mathsf{ct}.\underline{\boldsymbol{a}}'_{h+1} \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v_{\gamma+1}-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{\alpha-1}), \mathsf{Sl}_{[\ell-v_{\gamma+1},\ell-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_\alpha))$ and $\mathsf{ct}.\underline{\boldsymbol{a}}'_{h+2} \leftarrow \mathsf{Sl}_{[0,\beta-v_{\gamma+1}-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_\alpha)$, and update $h := h + 2$.
   ii) If $\beta \le v_{\gamma+1}$: $\mathsf{ct}.\underline{\boldsymbol{a}}'_{h+1} \leftarrow \mathsf{Add}(\mathsf{Sl}_{[0,\ell-v_{\gamma+1}-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_{\alpha-1}), \mathsf{Sl}_{[\ell-v_{\gamma+1},\ell-v_{\gamma+1}+\beta-1]}(\mathsf{ct}.\tilde{\boldsymbol{a}}_\alpha))$ and update $h := h + 1$.

4) Return $(\mathsf{ct}.\underline{\boldsymbol{a}}'_i)_{0\le i\le h}$. □

Proposition 10. The LongRot algorithm is correct, requires at most $\left\lceil \frac{\tau}{\ell} \right\rceil$ Rots, $2\left\lceil \frac{\tau}{\ell} \right\rceil + \frac{\tau}{d} + 1$ CMuls, and one CMul multiplicative depth.

Proof. After translating Algorithm 7 into its plaintext version, a straightforward verification shows that Algorithm 7 indeed accomplishes the aforementioned functionality at the beginning of Section V-B , thereby confirming its correctness. To analyze the cost, without loss of generality, we only consider the case of $\tau > d - k$ and $\alpha > 0$.

While it follows from the correctness that $h = \left\lceil \frac{\tau}{\ell} \right\rceil - 1$, it follows from the Algorithm that $h = \sum_{j=0}^{\gamma}(i_j + 1) + \alpha$ or $h = \sum_{j=0}^{\gamma}(i_j + 1) + \alpha + 1$. From Step 3c, 3(f)i, 3g and 3i, the number of Rots is $\sum_{j=0}^{\gamma}(i_j + 1) + \alpha + 1 \le h + 1 = \left\lceil \frac{\tau}{\ell} \right\rceil$. For the number of CMuls, we notice that the number of resulting ciphertexts is $\left\lceil \frac{\tau}{\ell} \right\rceil$ and each ciphertext is a sum of two selected ciphertexts, except for the $\gamma + 1 \le \frac{\tau}{d} + 1$ ciphertexts computed in Step 3e and 3(f)iii, where each ciphertext is a sum of three selected ciphertexts. □

Appendix B
Missing Proofs

Proof of Proposition 5. For the correctness of Algorithm 2, we need the following properties that are all implied by the pairwise coprimality of (n, m, p).

Lemma 11. *Suppose that the integers n, m, and p are pairwise coprime. Then for all $0 \le k < m$ and $0 \le i < np$, we have*
1) $[[knp + i]_{mn}]_n = [i]_n$.
2) $[[knp + i]_{mp}]_p = [i]_p$.
3) $[[knp + i]_{mn}]_m = [[knp + i]_{mp}]_m$.
4) For all $0 \le \ell < m$ with $k \ne \ell$, $[[knp + i]_{mn}]_m \ne [[\ell np + i]_{mn}]_m$.

Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be the bicyclic encoding of $\boldsymbol{A} \in \mathcal{R}^{n\times m}$ and $\boldsymbol{B} \in \mathcal{R}^{m\times p}$, respectively. Denote by $\underline{\boldsymbol{a}}$ and $\underline{\boldsymbol{b}}$ the resulting vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ in Step 2. Denote by $\underline{\boldsymbol{x}}$ the vector $\boldsymbol{x}$ after Step 3,and by $\overline{\boldsymbol{x}}$ the resulting vector $\boldsymbol{x}$ after Step 4. Let the (i, j)-entry of $\boldsymbol{A}$ and $\boldsymbol{B}$ be $A_{i,j}$ and $B_{i,j}$, respectively.

It follows from the bicyclic encoding and Step 2 that $\underline{a}_i = a_{[i]_{mn}} = A_{[[i]_{mn}]_n,[[i]_{mn}]_m}$ and $\underline{b}_i = b_{[i]_{mp}} = B_{[[i]_{mp}]_m,[[i]_{mp}]_p}$ for $0 \le i < nmp$. Now, the definition of the segmented sum of vector implies that for $0 \le i < np$,

$$\begin{aligned}\overline{x}_i &= \sum_{0\le k<m} \underline{x}_{knp+i} \\ &= \sum_{0\le k<m} \underline{a}_{knp+i} \cdot \underline{b}_{knp+i}\end{aligned}$$

$$
\begin{aligned}
&= \sum_{0 \le k < m} A_{[[knp+i]_{mn}]_n, [[knp+i]_{mn}]_m} B_{[[knp+i]_{mp}]_m, [[knp+i]_{mp}]_p} \\
&= \sum_{0 \le k < m} A_{[i]_n, [[knp+i]_{mn}]_m} \cdot B_{[[knp+i]_{mp}]_m, [i]_p} \qquad (7) \\
&= \sum_{0 \le j < m} A_{[i]_n, j} \cdot B_{j, [i]_p} \qquad (8) \\
&= X_{[i]_n, [i]_p},
\end{aligned}
$$

where Eq. (7) (resp. (8)) follows from the items 1 and 2 (resp. 3 and 4) of Lemma 11.

In Step 2, we need at most $\lceil \log p \rceil$ (resp. $\lceil \log n \rceil$) vector rotations to update $\boldsymbol{a}$ (resp. $\boldsymbol{b}$), and we need at most $\lceil \log m \rceil$ vector rotations to compute the segment-sum in Step 4. The only occurrence of component-wise vector product happens in Step 3, which completes the proof. □

Proof of Proposition 6. For $\boldsymbol{A} = (a_{i,j})_{0 \le i < n, 0 \le j < m}$, according to the definition of bicyclic encoding, we have $\boldsymbol{d}(\boldsymbol{A}) = (d_k)$ with $d_k = a_{[k]_n, [k]_m}$ for $0 \le k < n \cdot m$. For $\boldsymbol{A}^{\mathrm{T}} = (a'_{i,j})_{0 \le i < m, 0 \le j < n}$ with $a'_{i,j} = a_{j,i}$, we assume that $\boldsymbol{d}(\boldsymbol{A}^{\mathrm{T}}) = (d'_k)_{0 \le k < n \cdot m}$. Obviously, $d'_k = a'_{[k]_m, [k]_n} = a_{[k]_n, [k]_m} = d_k$ for $0 \le k < n \cdot m$. □