Copyright ⓒ 2019 KSII

Secure Outsourced Computation of Multiple Matrix Multiplication Based on Fully Homomorphic Encryption

Shufang Wang, Hai Huang*
School of Information, Zhejiang Sci-Tech University
Hangzhou, 310018, China
[e-mail: shufangwang95@163.com, haihuang1005@gmail.com]
*Corresponding author: Hai Huang

Received January 1, 2019; revised April 2, 2019; accepted July 1, 2019; published November 30, 2019

Abstract

Fully homomorphic encryption allows a third-party to perform arbitrary computation over encrypted data and is especially suitable for secure outsourced computation. This paper investigates secure outsourced computation of multiple matrix multiplication based on fully homomorphic encryption. Our work significantly improves the latest Mishra et al.'s work. We improve Mishra et al.'s matrix encoding method by introducing a column-order matrix encoding method which requires smaller parameter. This enables us to develop a binary multiplication method for multiple matrix multiplication, which multiplies pairwise two adjacent matrices in the tree structure instead of Mishra et al.'s sequential matrix multiplication from left to right. The binary multiplication method results in a logarithmic-depth circuit, thus is much more efficient than the sequential matrix multiplication method with linear-depth circuit. Experimental results show that for the product of ten 32×32 (64×64) square matrices our method takes only several thousand seconds while Mishra et al.'s method will take about tens of thousands of years which is astonishingly impractical. In addition, we further generalize our result from square matrix to non-square matrix. Experimental results show that the binary multiplication method and the classical dynamic programming method have a similar performance for ten non-square matrices multiplication.

Keywords: Secure outsourced computation, secure multiple matrix multiplication, computation over encrypted data, fully homomorphic encryption

This research was supported by Zhejiang Provincial Natural Science Foundation of China under Grant No. LY16F020025 and Zhejiang Sci-Tech University Graduate Dissertation Development Fund No 2018-XWLWPY-M-06-03.

1. Introduction

Cloud computing service allows users to outsource data processing and storage tasks to cloud platform. However, storing data on cloud server somewhere could pose a severe threat to users' privacy as cloud managers could be curious. The issues of privacy in cloud computing will probably lead to a number of privacy concerns and hinder the popularity of cloud computing. A promising solution to address these concerns is fully homomorphic encryption (FHE) which enables to perform arbitrary computations over encrypted data without decrypting it first.

With fully homomorphic encryption a user can encrypt their private data locally and send the ciphertexts to cloud platform which performs the computations on encrypted data and sends back the result in the form of ciphertext to the user. Afterward, the user decrypts the result with high certainty that no one else knows their private data. Fully homomorphic encryption is a very powerful tool for outsourcing computations on confidential data and has become increasingly popular in cloud computing security.

The first fully homomorphic encryption scheme was proposed by Gentry et al. [1] in 2009. Since then, fully homomorphic encryption has been rapidly developed and a number of improved schemes have been proposed [2-7]. On the other hand, fully homomorphic encryption has been used to build a variety of outsourced computation applications, e.g., secure data statistics and machine learning [8-11].

Matrix multiplication is a fundamental and time consuming operation in many higher level computations applications. An improvement in matrix multiplication will lead to a significant improvement in the performance of the higher level applications. Halevi et al. [12] proposed three different matrix encoding methods for matrix-vector multiplication based on single instruction multiple data (SIMD) technique [13], i.e., row-order, column-order, diagonal-order. Duong et al. [14] proposed a new matrix encoding method for secure matrix multiplication. Recently, Rathee et al. [15] proposed a new matrix encoding method based on hypercube structure and Jiang et al. [16] proposed a new matrix encoding method based on SIMD technique.

All the methods above are only for secure multiplication for two matrices and there is little work investigating secure multiple (n>2) matrix multiplication. The only work we are aware of that investigated secure multiple matrix multiplication was proposed by Mishra et al. [17], which is an extension of Duong et al. 's [14] two matrices multiplication method. Let A_1 ,A_2,…,A_n be square matrices with size of $m \times m$. In order to support the multiple matrix multiplication, they define the different encoding methods for each matrix A_i $\{i=1,\ldots,n\}$ respectively and the next matrix requires larger parameter than the previous one. However, such a large parameter makes homomorphic multiplication more slow. Thus, their method will become impractical asymptotically as the number of matrices involved increases.

In this paper, our main aim is to improve Mishra et al.'s [17] work for further efficiency. Our contributions are as follows.

First, we extend Halevi et al.'s [12] column-order matrix encoding from matrix/vector multiplication into matrix-matrix multiplication. Compared to Mishra et al. 's encoding method, the main advantage of the column-order encoding method is that homomorphic multiplication of two matrices will lead to the third one which is also in the form of column-order encoding. This way, all the n matrices will be encoded with the fixed-size parameter and thus our solution is much more efficient asymptotically.

Second, Mishra et al. make use of sequential multiplication to calculate the product of multiple matrices, which calculates the product of *n* matrices one by one from left to right. We introduce a new method called binary multiplication, which multiplies pairwise two adjacent matrices in the tree structure. Compared to Mishra et al.' sequential multiplication, our approach has lower multiplicative circuit depth and thus will be much more efficient. Further, we optimize our method by multi-thread technique. Experimental results show that our method takes 2860.57 seconds for the product of ten 32×32 matrices and 10772.2 seconds for ten 64×64 matrices. Comparatively, Mishra et al.'s gave experimental results only for the product of three 32×32 matrices and 64×64 matrices respectively. According to their own estimate, Mishra et al.'s method will take about 21924468 years for ten 32×32 matrix and about 159923135 years for ten 64×64 matrices which is astonishingly impractical. Thus, our method is significantly faster than Mishra et al.'s method.

Third, we further generalize our result from square matrix to non-square matrix multiplication. For multiple non-square matrix multiplication, we additionally introduce the classical dynamic programming technique to calculate the product of ten non-square matrices. Experimental results show that the binary multiplication method and the dynamic programming method have a similar performance for multiple non-square matrix multiplication. Specifically, they take 5995.16 seconds and 5046.81 seconds for the product of some set of ten non-square matrices respectively.

2. Related Work

Some related works [18-22] focus on verifiable secure outsourcing of two matrix computation. These solutions exploit specific properties of matrix multiplication and design special protocols for secure outsourced matrix multiplication. Hopefully, these custom solutions are more efficient than that based on fully homomorphic encryption. However, a disadvantage is that each protocol must be designed, and proved secure, which are error-prone. Moreover, none of the protocols above investigates secure outsourced computation of multiple matrix multiplication.

Secure multiparty computation [23] is another general framework for secure outsourced computation. However, this paradigm requires either significantly high communication overhead between the client and the cloud server or assuming the existing of the two-server [24] which is vulnerable to the collusion attack.

3. Preliminaries

3.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption is an encryption method that allows anyone to compute an arbitrary function *f* on an encryption of x, without knowledge of the private key. As a result, one obtains an encryption of *f(x)*.

Definition 1. The fully homomorphic encryption scheme consists of four procedures $\varepsilon = (KeyGen, Encryt, Decrypt, Evaluate)$:

1. $(pk, sk) \leftarrow KeyGen(1^{\lambda})$: It takes a security parameter λ as an input and outputs a public key *pk* and a secret key *sk*.
2. $c \leftarrow Enc(pk, m)$: It takes a public key and a plaintext, outputs a ciphertext *c*.
3. $m \leftarrow Dec(sk, c)$: It takes a private key and a ciphertext, outputs a plaintext *m*.

4. $c_f \leftarrow Eval(pk, f, c_1, ..., c_n)$: It takes the public key, a function $f : P^n \to P$, and a set of n ciphertexts $(c_1, ..., c_n)$ which is the encryption $(m_1, ..., m_n)$ and outputs a ciphertext c_f which is the encryption of $f(m_1, ..., m_n)$.

3.2 BGV

BGV scheme [4] and its variants [5-6] are defined over ring-LWE of the form $\mathrm{A} = Z[x]/\Phi_m(X)$ where $\Phi_m(X)$ is the m'th cyclotomic polynomial. The ciphertext space is set to be $\mathrm{A}_q := \mathrm{A}/q\mathrm{A}$ for an odd integer modulus q. A BGV-type scheme has a chain of moduli, $q_0 < q_1 < \ldots < q_{L-1}$, where freshly encrypted ciphertexts are defined over largest modulus A_{L-1}. Ciphertexts defined over A_{qi} are called level-i ciphertexts.

The plaintext space for BGV scheme is the ring $\mathrm{A}_p = \mathrm{A}/p\mathrm{A}$, where p is a prime. A salient feature of BGV scheme and its variants is that it supports single instruction multiple data (SIMD) parallel operations [13]. Under modulo p, the cyclotomic polynomial $\Phi_m(x)$ can be factorized into l distinct irreducible polynomials such that $\Phi_m(x) = \prod_{i=1}^{l} F_i(x) \bmod p$, each with degree $d = \Phi(m)/l$. Each factor corresponds to a plaintext slot and the following isomorphism (equation 1) holds.

$$\mathrm{A}_p \cong Z_p[x]/F_1(x) \otimes \ldots \otimes Z_p[x]/F_l(x) \cong F_{p^d} \otimes \ldots \otimes F_{p^d} \tag{1}$$

By the polynomial CRT, the polynomial $a \in \mathrm{A}_p$ decomposes into l slots $(a_i)_{i=1}^{l} \in (F_{p^d})^l$. Thus, we can pack l messages into a single plaintext polynomial and perform l additions or multiplications at the cost of just a single operation. Assume that $a = CRT((a_i)_{i=1}^{l})$ and $b = CRT((b_i)_{i=1}^{l})$, we have the following equation 2.

$$\begin{cases} CRT^{-1}(a + b \bmod (p, \Phi_m)) = (a_i + b_i \bmod (p, F_i))_{i=1}^{l} \\ CRT^{-1}(a \cdot b \bmod (p, \Phi_m)) = (a_i \cdot b_i \bmod (p, F_i))_{i=1}^{l} \end{cases} \tag{2}$$

Also, it is possible to rotate or permute the underlying plaintext slots in a batched vector by applying automorphism mappings of the form $\kappa : a(X) \to a(X^k)$ where $k \in Z_m^* / < p >$.

3.3 Dynamic Programming for Multiple Non-square Matrix Multiplication

Assume that there are n non-square matrices $\mathrm{A}_1, \mathrm{A}_2, \ldots, \mathrm{A}_n$ with size $p_0 \times p_1$, $p_1 \times p_2$, $p_2 \times p_3$, …, $p_{n-1} \times p_n$ and the goal is to calculate the product of n matrices. As matrix multiplication is associative, no matter how a product of $\mathrm{A}_1 \times \mathrm{A}_2 \ldots \times \mathrm{A}_n$ is parenthesized, the result obtained will remain the same. However, the order in which the product is parenthesized has a signification impact on the computational overhead of a product of n matrices. Dynamic programming [25] is an optimization method for solving a complex problem by breaking it down into simpler subproblems and can be used to determine the optimal parenthesization of a product of n matrices.

Let $m(i, j)$ denote the minimum number of multiplications for computing $\mathrm{A}_i \times \mathrm{A}_{i+1} \ldots \times \mathrm{A}_j$. A recursive formula is defined as follows (equation 3).

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k \le j-1} \{ m(i,k) + m(k+1,j) + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases} \quad (3)$$

As there are many overlapping subprobelms within this recursive formula, a direct recursive algorithm will result in an exponential time complexity. Instead, one can solve this recursive formula efficiently in either of two ways.

Top-down approach with memorization: Before trying to solve a sub-problem, we first check memory table to see if the solution has already been stored. If a solution has been stored, we just use it directly without computation, otherwise we solve the sub-problem and add its solution into the table.

Bottom-up approach: We can reformulate the problem in a bottom-up fashion. Solving the sub-problems first and use their solutions to build the solutions to bigger problems.

4. Mishra et al.'s Secure Multiple Matrix Multiplication Method

Let A be a $m \times m$ matrix. For each row $A_i = (a_{i1}, \ldots, a_{im})$ of A, they define two polynomials in $R = \mathbb{Z}[x]/(x^n + 1)$ as follows (equation 4).

$$\begin{cases} pm_{m,3}^{(1)}(A_i) = \sum_{u=1}^{m} a_{iu} x^{u-1} \\ pm_{m,3}^{(2)}(A_i) = -\sum_{u=1}^{m} a_{iu} x^{n-(u-1)m^2-m+1} \end{cases} \quad (4)$$

Let A,B,C be three matrices with size of $m \times m$. Mishra et al. [17] define three types of polynomial in R for three matrices A,B,C as follows (equation 5).

$$\begin{cases} pol_{m,3}^{(1)}(\mathrm{A}) = \sum_{i=1}^{m} pm_{m,3}^{(1)}(A_i) x^{(i-1)m} \\ pol_{m,3}^{(2)}(\mathrm{B}) = \sum_{j=1}^{m} pm_{m,3}^{(1)}(\overline{B}_j^T) x^{(j-1)m^2} \\ pol_{m,3}^{(3)}(\mathrm{C}) = \sum_{k=1}^{m} pm_{m,3}^{(2)}(C_k^T) x^{(k-1)m^3} \end{cases} \quad (5)$$

Where $B_j^T = (b_{1j}, \ldots, b_{mj})$ and C_k^T are the j^{th} and the k^{th} columns of B and C respectively, and $\overline{B}_j^T = (b_{mj}, \ldots, b_{1j})$. Define three types of packed ciphertext for a matrix A to be $ct^{(i)}(\mathrm{A}) := Enc(pol_{m,3}^{(i)}(\mathrm{A}), pk)$ for i=1,2,3.

Theorem 1 [17,Theorem 3]: Assume $n \ge m^4$. Let $ct = ct^{(1)}(\mathrm{A}) * ct^{(2)}(\mathrm{B}) * ct^{(3)}(\mathrm{C})$ and let $Dec(ct, sk) \in R_t$ denote its decryption result. Then for each $i,k \in \{1, \ldots, m\}$, the $(i,k)^{th}$ entry of the matrix A×B×C is the coefficient of $x^{(i-1)m+(k-1)m^3}$ in $Dec(ct, sk)$.

An advantage of their encoding method is that whole matrix is encoded into one ciphertext, thus the multiplication of two matrices requires only one homomorphic multiplication. However, the drawback of Mishra et al.'s scheme is that each matrix requires different encoding method and the next matrix requires a factor of m larger parameter to encode than the previous one. This method for three matrices is about 80 $\sim$ 100 times slower than two matrices case. They estimated [17] that as the number of matrices increase, the running time will be at least about 80 times slower. Thus, their method will become impractical asymptotically as the number of matrix increases.

5. Our Secure Multiple Matrix Multiplication Schemes

5.1 A Naive Method

A naive method for secure multiple matrix multiplication is to encrypt each entry in each matrix by one ciphertext as shown in **Fig. 1**, then use traditional matrix multiplication in ciphertext domain as shown in **Fig. 2** This simple solution is able to multiply multiple matrices with fixed-size parameter.

$$\begin{bmatrix} enc(a_{1,1}) & . \ . \ . & enc(a_{1,m}) \\ . & . & . \\ . & . & . \\ . & . & . \\ enc(a_{m,1}) & . \ . \ . & enc(a_{m,m}) \end{bmatrix} \xleftarrow{pk} \begin{bmatrix} a_{1,1} & . \ . \ . & a_{1,m} \\ . & . & . \\ . & . & . \\ . & . & . \\ a_{m,1} & . \ . \ . & a_{m,m} \end{bmatrix}$$

Fig. 1. Encrypt each entry in each matrix

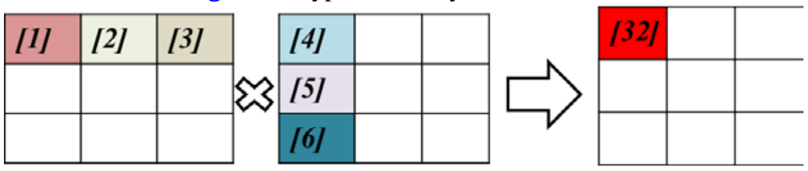

Fig. 2. Traditional matrix multiplication in ciphertext

Obviously, the main drawback of this naive method is that it requires one ciphertext for each entry of a matrix.This results in m^2 ciphertext for each matrix and O(m^3) operations for two matrix multiplication, which require a lot of time and space depending on the size of the input matrix.

5.2 Secure Column-order Matrix multiplication

Halevi et al. [12] proposed three different matrix encoding methods for matrix-vector multiplication, i.e., row-order, column-order, diagonal-order. We adopt column-order method and generalize it to matrix-matrix multiplication. Assume that A,B are two $m\times m$ matrices and C=A×B. We can write A,B in the form of column order by equation 6

$$\mathrm{A} = (\boldsymbol{a}_1 | \ldots | \boldsymbol{a}_m), \mathrm{B} = (\boldsymbol{b}_1 | \ldots | \boldsymbol{b}_m) \tag{6}$$

where both $\boldsymbol{a}_i = (a_{1i}, a_{2i}, \ldots, a_{mi}), \boldsymbol{b}_i = (b_{1i}, b_{2i}, \ldots, b_{mi})\{i = 1,2,\ldots,m\}$ are m dimensional column vectors. Now we rewrite C as equation 7

$$\mathrm{C} = (\sum_{i=1}^{i=m} b_{i1}\boldsymbol{a}_i | \ldots | \sum_{i=1}^{i=m} b_{im}\boldsymbol{a}_i) \tag{7}$$

Now we transform the column-order matrix multiplication method above into the ciphertext domain. Assume that ct(A),ct(B) are ciphertexts of two matrices A,B, which are encrypted column-wise as equation 8

$$ct(\mathrm{A}) = (u_1 | \ldots | u_m), ct(\mathrm{B}) = (v_1 | \ldots | v_m) \tag{8}$$

where u_i, v_i is the encryption of $\boldsymbol{a}_i, \boldsymbol{b}_i$ respectively by SIMD technique. In order to perform matrix multiplication homomorphically, we first apply replicate operation [12] to each column of ct(B) obtaining the m^2 ciphertexts $v_{i1}, v_{i2}, ..., v_{im}\{i = 1,2,...,m\}$ such that $v_{ij}\{i, j = 1,2,...,m\}$ is the encryption of $\boldsymbol{b}_i[j] = b_{ji}$ in all positions. Now we have equation 9.

$$ct(\mathrm{C}) = ct(\mathrm{A}) \times ct(\mathrm{B}) = (\sum_{i=1}^{i=m} u_i v_{1i} \mid ... \mid \sum_{i=1}^{i=m} u_i v_{mi}) \tag{9}$$

Our method for secure column-order matrix multiplication is defined in **Algorithm 1**.

Algorithm 1: Secure-Matrix-Multiplication(ct(A),ct(B))

```
Input: ct(A),ct(B)        // The ciphertexts of matrices A,B
Output: ct(AB)            //The ciphertext of matrix AB
for $j$=1 to $m$
    for $i$=1 to $m$
        ct(temp)=ct(A)[$i$] ×replicate(ct(B)[$j$],$i$)
        ct(C)[$j$]= ct(C)[$j$]+ct(temp)
        end for
    end for
ct(AB)←ct(C)
return ct(AB)
```

The following example shown in **Fig. 3** demonstrates how the secure matrix multiplication is performed.

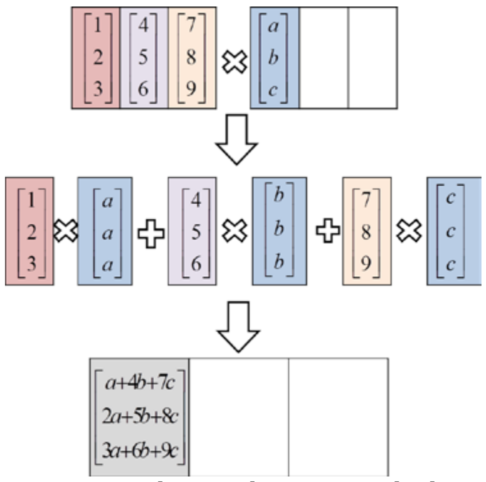

Fig. 3. Secure column-order matrix multiplication

Note that the main advantage of the column order encoding method is that the resulting ciphertext ct(C) is also in column order which enables to keep on multiplying ct(C) with next encrypted matrix homomorphically in the same way as above. As all matrices can are encoded in column order, we can select a fixed-size parameter for all matrices. Thus, our encoding method is much more efficient compared to Mishra et al.'s encoding method which requires a factor of 80 times larger paramerters as the number of matrix increases.

5.3 Binary Multiplication Method for Secure Multiple Matrix Multiplication

With the column order encoding technique, a natural method to calculate the product of n matrices $A_1, A_2, \ldots, A_n$ is to multiply them sequentially as above. However, the sequential multiplication method creates a circuit with $d(C_{smult}) = n \times d(C_{2\text{-}mult})$ multiplicative depth, where $d(C_{2\text{-}mult})$ denotes the circuit depth of two-matrix multiplication. We propose a better method called binary multiplication method multiplying two adjacent matrices pairwise in a tree structure shown in Fig. 4. The product of n matrices creates a circuit with $\lceil \log_2 n \rceil$ multiplicative depth. Therefore, $d(C_{bmult}) = \lceil \log_2 n \rceil \times d(C_{2\text{-}mult})$. The reduced circuit depth allows much slower noise growth and thus enable us to select smaller parameters in the underlying fully homomorphic encryption scheme resulting in a greater efficiency. Our technique for binary multiplication method for secure multiple matrix multiplication is defined in Algorithm 2.

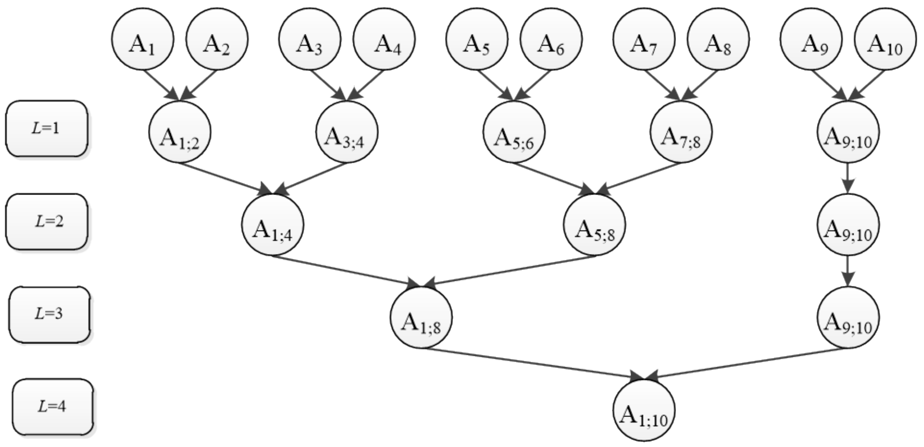

Fig. 4. Binary multiplication method for secure multiple matrix multiplication

Algorithm 2: Binary-Multiplication (ct(A_1), ct(A_2),…, ct(A_n))

```
Input: B=(ct(A_1), ct(A_2),…, ct(A_n))   //B stores a vector of encrypted matrices
Output: ct(A_1×A_2…×A_n)                  //The final result of multiple matrix multiplication
N=n
for L=1 to ⌈log_2 n⌉
    m=N                        //m is the number of matrices involved in level L
    N=(m+1)/2                  //N is the number of matrices involved in level L+1
    for i=1 to N
        if(m%2==1&&i==N)       //If m is odd, no multiplication is required for the last matrix
            B_i=B_{i*2}
        else
            B_i = Secure-Matrix-Multiplication(B_{i*2-1},B_{i*2})
    end for
end for
return B_1                     //The final result is stored in first element B_1
```

5.4 Secure Multiple Non-square Matrix Multiplication

If input matrices are non-square, we could further apply the dynamic programming method to find the most efficient way to perform the product of multiple matrices. Assume that matrix A_i with size of $p_{i-1} \times p_i$ (i=1,2,…,n). Given a sequence of ($p_0, p_1, \ldots, p_n$), the dynamic programming algorithm with bottom-up approach [25] as shown in **Algorithm 3** outputs $\{s_{i,j}\}_{1<=i<=n, 1<=j<=n}$ which stores optimal parenthesized location for a product of $A_i \times A_{i+1} \ldots \times A_j$.

Alogorithm 3: Matrix-Chain-Order ($p_0, p_1, \ldots, p_n$) [25]

```
Input: $p_0,p_1,\ldots,p_n$   //$p_{i-1}, p_i$ are the row and column dimensions of matrix $A_i$
Output: $\{s_{i,j}\}_{1<=i<=n, 1<=j<=n}$  //The optimal parenthesized location for $A_i \times A_{i+1} \ldots \times A_j$
For $i$=1 to $n$
$m[i,i] = 0$            //Initialization
for $r$=2 to $n$                //$r$ is the length of subchain of matrix
  for $i$=1 to $n$-$r$+1
    $j=i+r-1$
    $m[i,j]$=MAXINT   //$m$ stores the minimum value of multiplication for $A_i \times A_{i+1} \ldots \times A_j$
    for $k=i$ to $j$-1
      min = $m[i, k] + m[k + 1, j] + p_{i-1} \times p_k \times p_j$
      if min<$m[i,j]$
        {
        $m[i,j]$=min
        $s_{i,j} = k$    //The optimal parenthesized location for $A_i \times A_{i+1} \ldots \times A_j$
        }
    end for
  end for
end for
```

Given $\{s_{i,j}\}_{1<=i<=n, 1<=j<=n}$ and a vector of encrypted matrices (ct(A_1), ct(A_2),…, ct(A_n)), the **Algorithm 4** outputs the final result ct($A_1 \times A_2 \ldots \times A_n$).

Algorithm 4: Secure-Multiple-Matrix-Multiplication(s,i,j,B)

```
Input:
-$\{s_{i,j}\}_{1<=i<=n, 1<=j<=n}$   //$\{s_{i,j}\}$ stores the optimal parenthesized location for $A_i \times A_{i+1} \ldots \times A_j$.
-$i$=1  //$i$ is the index of first matrix
-$j=n$  //$j$ is the index of last matrix
-B=(ct($A_1$), ct($A_2$),…, ct($A_n$))  //B stores the vector of encrypted matrices
Output: ct($A_1 \times A_2 \ldots \times A_n$)     //The final result of multiple matrix multiplication
if($i$==$j$)
  return $B_i$
else
{
  $T_1$=Secure-Multiple-Matrix-Multiplication (s, $i$, $s_{i,j}$, B)
```

```
    $T_2$=Secure-Multiple-Matrix-Multiplication (s, $s_{i,j}$+1, $j$, B)
    T = Secure-Matrix-Multiplication ($T_1$, $T_2$)
    return T
}
```

6. Implementation and comparison

In this section, we implement our algorithms in Section 5 and compare them with Mishra et al.'s method. Our experiments run on an Intel® Xeon® Gold 6148 CPU with 2.40 GHz and 503G RAM, using HElib library [26] in C++ programs for the implementation of BGV scheme and its variants. We basically need to select the following parameters.
-*L*: the number of moduli for leveled ciphertext spaces
-*p*: a prime plaintext modulus
-*k*: the security level
-*slot*: the number of slots

Targeting *k*=80-bits of security and selecting an appropriate depth parameter *L* and plaintext modulus *p*=257, we get the results in **Table 1** and **Table 2**, which show the performances for multiple 32×32 (*slot*>=32) and 64×64 (*slot*>=64) matrix multiplication.

Mishra et al. [17] implemented their method for two and three 32×32 (64×64) matrices multiplication in Intel Core i7-4790 CPU with 3.60 GHz and 8.00GB RAM in C programs. As shown in **Table 1** and **Table 2**, for the product of two or three 32×32 (64×64) matrices, Mishra et al.'s method is more efficient than ours. However, when *n*>=4, our method significantly outperforms Mishra et al.'s method.

They estimated [17] that as the number of matrices increase, the running time will be at least about 80 times slower. Thus, a simple calculation shows that for ten 32×32 (64×64) matrices multiplication, their method will take about 21924468 (159923135) years which is an astronomical figure. In comparison, our method with multi-thread optimization takes only 2860.57 (10772.2) seconds for ten 32×32 (64×64) matrices multiplication.

In addition, our method also outperforms the naive method which takes 19003.3 seconds for ten 32×32 matrices multiplication as shown in **Table 1**. For ten 64×64 matrices multiplication, the program based on the naive method runs out of memory when *n*>=5 as shown in **Table 2**.

Table 1. Secure multiple 32×32 matrix multiplication (*k*=80,*p*=257)

Number of matrices	2(s)	3(s)	4(s)	5	6	7	8	9	10
Mishra et al.[17]	0.297	32.969	2637.52*	58.611h *	4688.92h *	42.82y *	3425.698 y*	274055.8 58y*	2192446 8.63y*
Naive Method	*L*=3; 418.398	*L*=5; 1926.43	*L*=5; 2021.51	*L*=7; 6298.53s	*L*=7; 6409.88s	*L*=7; 6410.65s	*L*=7; 7318.12s	*L*=9; 17848.2s	*L*=9; 19003.3s
Sequential Multiplication	*L*=3; 58.449	*L*=5; 167.65	*L*=9; 1162.09	*L*=11; 2416.2s	*L*=15; 5593.28s	*L*=16; 7683.87s	*L*=19; 9926.29s	*L*=21; 13579.6s	*L*=25; 22179.3s
Our method (nomthread)	*L*=3; 58.449	*L*=5; 198.446	*L*=5; 281.162	*L*=9; 1933.67s	*L*=9; 2274.8s	*L*=9; 2787.49s	*L*=9; 3225.01s	*L*=11; 6464.03s	*L*=11; 7173.77s
Our method (mthread)	*L*=3; 58.449	*L*=5; 198.84	*L*=5; 203.626	*L*=9 1240.6s	*L*=9; 1290.82s	*L*=9; 1403.39s	*L*=9; 1410.64s	*L*=11; 2720.79s	*L*=11; 2860.57s

Naive method: A binary multiplication method for the naive encoding matrix with multi-thread optimization. Sequential multiplication: A simple sequential multiplication from left to right for column-order encoding matrix. Our method (nomthread): A binary multiplication method in a tree structure for column-order encoding matrix without multi-thread optimization. Our method (mthread): A binary multiplication method in a tree structure for column-order encoding matrix with multi-thread optimization. "*": The time is calculated based on their own estimate in [17]. s: second. h: hour. y: year.

Table 2. Secure multiple 64×64 matrix multiplication (k=80,p=257)

Number of matrices	2(s)	3(s)	4(s)	5	6	7	8	9	10
Mishra et al.[17]	2.391	240.485	19238.8 *	427.529 h*	1425.096 d*	312.35 y*	24987.99 y*	1999039.188y*	159923135.058y*
Naive method	L=3; 9703.5	L=6; 56630.9	L=6; 60053.2	#	#	#	#	#	#
Sequential Multiplication	L=4; 270.79	L=6; 1609.14	L=9; 4992.13	L=11; 10604.8s	L=15; 25276.8s	L=18; 30735.9s	L=20; 35087s	L=21; 61135.5s	L=26; 104518s
Our method (nomthread)	L=3; 270.792	L=6; 1800.48	L=6; 2290.22	L=9; 7242.18s	L=9; 9008.77s	L=9; 10265.2s	L=9; 12085s	L=11; 23913.9s	L=11; 27000.3s
Our method (mthread)	L=3; 270.792	L=6; 1744.22	L=6; 1738.44	L=9; 5411.52s	L=9; 5569.88s	L=9; 5757.94s	L=9; 6060.82s	L=11; 10828.6s	L=11; 10772.2s

Naive method: A binary multiplication method for the naive encoding matrix with multi-thread optimization. Sequential multiplication: A simple sequential multiplication from left to right for column-order encoding matrix. Our method (nomthread): A binary multiplication method in a tree structure for column-order encoding matrix without multi-thread optimization. Our method (mthread): A binary multiplication method in a tree structure for column-order encoding matrix with multi-thread optimization. "*": The time is calculated based on their own estimate in [17]. s: second. h: hour. d:day. y: year. #: The program runs out of memory.

Fig. 5 and **Fig. 6** illustrate pictorially the running time of all the above methods for ten 32×32 and 64×64 matrices multiplication respectively. As shown in these figures, the running time of Mishra et al. 's work for 32×32 (64×64) matrices multiplication radically increases when n>=4. For ten 64×64 matrices multiplication, the program for the naive method runs out of memory when n>=5, and thus no time is given for these cases.

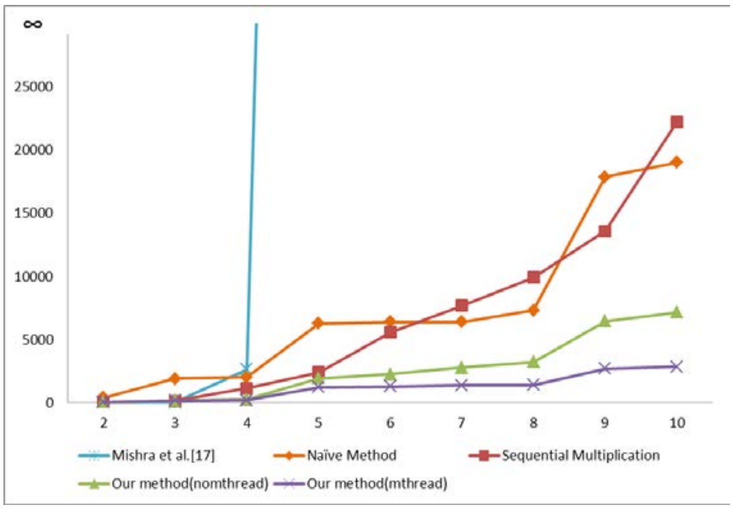

Fig. 5. Secure multiple 32×32 matrix multiplication

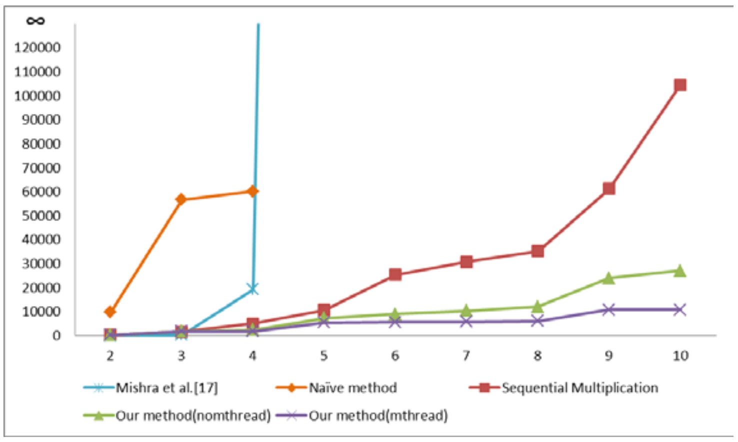

Fig. 6. Secure multiple 64×64 matrix multiplication

We also implement our algorithms for non-square matrix case. **Table 3** shows our experimental results for ten non-square $p_{i-1}\times p_i$ (i=1,2,…,10) matrices multiplication. We meticulously select a set of ten matrices of size p_0=40,p_1=30,…,p_{10}=25, which is suitable for the dynamic programming technique.

Table 3. Secure multiple non-square matrix multiplication

Dimensions of Matrices	40,30,35,15,60,5,70,10,50,20,25					
	L	Init(s)	Encrypt(s)	Homo-Eval(s)	Decrypt(s)	Total(s)
Naive method	9	2.254	129.375	#	10.048	#
Dmethod	15	5.837	11.114	5030.36	5.044	5046.81
Our method	11	3.639	7.965	5983.72	3.341	5995.16

Naive method: A binary multiplication method for the naive encoding matrix with multi-thread optimization. Dmethod: A dynamic programming method for column-order encoding matrix. Our method: A binary multiplication method in a tree structure for column-order encoding matrix with multi-thread optimization. Init: The time for setting up system parameters. Encrypt: The time for encrypting ten matrices. Decrypt: The time for decrypting ten matrices. Total: The time for all computations. Homo-Eval: The time for homomorphic computation of the product of ten non-square matrices. s: second. #: The program runs out of memory.

Fig. 7 illustrates pictorially the running time of all the sub-algorithms including initialization, encyption, decryption and homomorphic evaluation etc. As can been seen in the figure, our binary multiplication method enjoys similar performance to the dynamic programming method while the program based on the naive method runs out of memory.

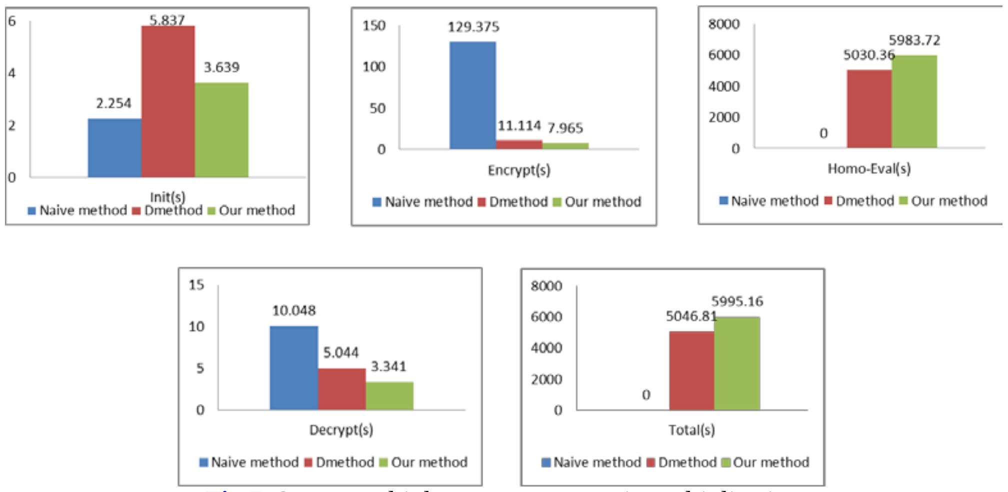

Fig 7. Secure multiple non-square matrix multiplication

7. Conclusion

This paper investigates secure outsourced computation of multiple matrix multiplication based on fully homomorphic encryption. Our work radically improves the latest Mishra et al.'s method.

First, we propose a column-order matrix encoding method extending Halevi et al.'s work. Our encoding method requires only fixed-size parameter, compared to Mishra et al.'s encoding which requires huge secure parameter. Second, we introduce a new method called binary multiplication for multiple matrix multiplication. Experimental results show that our method takes only thousands seconds while Mishra et al.'s method will takes tens of thousands of years for the product of ten matrices. Third, we further generalize our result from square matrix to non-square matrix multiplication. Experimental results show that binary multiplication method and dynamic programming method have a similar performance for multiple non-square matrix multiplication.

A possible direction for future work is to combine other matrix encoding methods, e.g., Rathee et al.'s hypercube structure and Jiang et al. 's method, into our framework to see if we can further improve efficiency of secure multiple matrix multiplication.

References

[1] Craig Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. of the forty-first annual ACM symposium on Theory of computing (STOC '09)*, pp. 169-178, 2009. Article (CrossRef Link)

[2] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. of EUROCRYPT*, pp. 24-43, 2010. Article (CrossRef Link)

[3] Zvika Brakerski and Vinod Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *Proc. of IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS 2011)*, pp. 97-106, October 22-25, 2011. Article (CrossRef Link)

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*, pp. 309-325, 2012. Article (CrossRef Link)

[5] Craig Gentry, Shai Halevi, and Nigel P. Smart, "Fully homomorphic encryption with polylog overhead," in *Proc. of the 31st Annual international conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT'12)*, pp. 465-482, 2012. Article (CrossRef Link)

[6] Gentry, C., Halevi, S., Smart, N.P, "Homomorphic evaluation of the aes circuit," in *Proc. of Advances in Cryptology-CRYPTO 2012*, pp. 850-867, 2012. Article (CrossRef Link)

[7] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptoticallyfaster, attribute-based," in *Proc. of Adv. Crypto 2013*, pp. 75–92, 2013. Article (CrossRef Link)

[8] D. Wu and J. Haven, "Using homomorphic encryption for large scale statistical analysis," *Technical Report*, 2012. Article (CrossRef Link)

[9] Wen-jie Lu, Shohei Kawasaki, Jun Sakuma, "Using Fully Homomorphic Encryption for Statistical Analysis of Categorical, Ordinal and Numerical Data," in *Proc. of 24th Annual Network and Distributed System Security Symposium*, 2017. Article (CrossRef Link)

[10] Graepel, T., Lauter, K., Naehrig, M., "Ml confidential: Machine learning on encrypted data," in *Proc. of the 15th International Conference on Information Security and Cryptology*, pp. 1-21, 2012. Article (CrossRef Link)

[11] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing, "CryptoNets: applying neural networks to encrypted data with high throughput and accuracy," in *Proc. of the 33rd International Conference on International Conference on Machine Learning*, vol. 48, pp. 201-210, 2016. Article (CrossRef Link)

[12] Halevi S, Shoup V, "Algorithms in Helib," in *Proc. of Advances in Cryptology-CRYPTO 2014*, pp. 554-571,2014. Article (CrossRef Link)

[13] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Des. Codes Cryptography*, vol 71, pp. 57-81, 2014. Article (CrossRef Link)

[14] Dung Hoang Duong, Pradeep Kumar Mishra, Masaya Yasuda, "Efficient Secure Matrix Multiplication Over LWE-Based Homomorphic Encryption," *Tatra Mountains Mathematical Publications*, vol 67, pp. 69-83, 2016. Article (CrossRef Link)

[15] Deevashwer Rathee, Pradeep Kumar Mishra, and Masaya Yasuda, "Faster PCA and Linear Regression through Hypercubes in Helib," in *Proc. of the 2018 Workshop on Privacy in the Electronic Society (WPES'18)*, pp. 42-53, 2018. Article (CrossRef Link)

[16] Xiaoqian Jiang, Miran Kim, Kristin Lauter and Yongsoo Song, "Secure Outsourced Matrix Computation and Application to Neural Networks," in *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, pp.1209-1222, 2018. Article (CrossRef Link)

[17] Mishra P.K., Duong D.H., Yasuda M, "Enhancement for Secure Multiple Matrix Multiplications over Ring-LWE Homomorphic Encryption," in *Proc. of Information Security Practice and Experience (ISPEC 2017)*, vol. 10701, pp. 320-333, 2017. Article (CrossRef Link)

[18] David Benjamin and Mikhail J. Atallah, "Private and Cheating-Free Outsourcing of Algebraic Computations," in *Proc. of the 2008 Sixth Annual Conference on Privacy, Security and Trust (PST '08)*, pp. 240-245, 2008. Article (CrossRef Link)

[19] Mikhail J. Atallah and Keith B. Frikken, "Securely outsourcing linear algebra computations," in *Proc. of the 5th ACM Symposium on Information, Computer and Communications Security* (*ASIACCS* '10), pp. 48-59, 2010. Article (CrossRef Link)

[20] Heriniaina F, Lei X, Huang T, et al, "Achieving security, robust cheating resistance, and high-efficiency for outsourcing large matrix multiplication computation to a malicious cloud," *Information Sciences*, vol. 280, pp. 205-217, 2014. Article (CrossRef Link)

[21] P. Mohassel, "Efficient and Secure Delegation of Linear Algebra," *IACR Cryptology ePrint Archive*, 2011. Article (CrossRef Link)

[22] D. Fiore and R. Gennaro, "Publicly verifiable delegation of large polynomials and matrix computations, with applications," in *Proc. of the 2012 ACM conference on Computer and communications security*, pp. 501–512, 2012. Article (CrossRef Link)

[23] Andrew Chi-Chih Yao, "How to generate and exchange secrets," in *Proc. of the 27th Annual Symposium on Foundations of Computer Science (FOCS '86)*, pp. 162-167, 1986. Article (CrossRef Link)

[24] P. Mohassel and Y. Zhang, “SecureML: A System for Scalable Privacy-Preserving Machine Learning,” in *Proc. of 2017 IEEE Symposium on Security and Privacy (SP)*, pp. 19-38, 2017. Article (CrossRef Link)
[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Third Edition (3rd ed.)*, The MIT Press, 2009. Article (CrossRef Link)
[26] Halevi, S., Shoup, V., “Design and implementation of a homomorphic-encryption library,” *Technical Report*, 2012. Article (CrossRef Link)

Shufang Wang is a postgraduate student at the Zhejiang Sci-Tec University, Hangzhou, China. She is interested in secure outsourced computation and fully homomorphic encryption.

Hai Huang is an Associate Professor at the Zhejiang Sci-Tech university, Hangzhou, China. He obtained his Ph.D. degree in Computer Science from Shanghai Jiaotong University, China in 2010. He is broadly interested in the general area of security, privacy, and applied cryptography.