FASTER SECURE MATRIX MULTIPLICATION USING THE STRASSEN ALGORITHM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

DİLEK ÖNER ŞİMŞEK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
CRYPTOGRAPHY

AUGUST 2023

Approval of the thesis:

FASTER SECURE MATRIX MULTIPLICATION USING THE STRASSEN ALGORITHM

submitted by **DİLEK ÖNER ŞİMŞEK** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Cryptography Department, Middle East Technical University** by,

Prof. Dr. A. Sevtap Selçuk Kestel _____
Dean, Graduate School of **Applied Mathematics**

Assoc. Prof. Dr. Oğuz Yayla _____
Head of Department, **Cryptography**

Prof. Dr. Ersan Akyıldız _____
Supervisor, **Cryptography, METU**

Prof. Dr. Murat Cenk _____
Co-supervisor, **Cryptography, METU**

Examining Committee Members:

Prof. Dr. Ersan Akyıldız
Cryptography, METU _____

Prof. Dr. Ferruh Özbudak
Faculty of Engineering and Natural Sciences, Sabancı University _____

Assoc. Prof. Dr. Ali Doğanaksoy
Mathematics, METU _____

Assoc. Prof. Dr. Oğuz Yayla
Cryptography, METU _____

Assoc. Prof. Dr. Fatih Sulak
Mathematics, Atılım University _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: DİLEK ÖNER ŞİMŞEK

Signature :

ABSTRACT

FASTER SECURE MATRIX MULTIPLICATION USING THE STRASSEN ALGORITHM

ŞİMŞEK, DİLEK ÖNER
Ph.D., Department of Cryptography
Supervisor : Prof. Dr. Ersan Akyıldız
Co-Supervisor : Prof. Dr. Murat Cenk

August 2023, 73 pages

In the cloud, private data is open to security problems. Homomorphic encryption offers a solution for addressing privacy concerns by enabling calculations on encrypted data. These computations may be statistical analysis or machine learning algorithms. Matrix multiplication also can be applied to the encrypted data. This is called secure matrix multiplication. Secure matrix multiplication is one of the primary operations in many applications, including statistical analysis and machine learning algorithms. Assume that A and B are two square matrices; the secure matrix multiplication algorithm creates packed polynomials for each matrix, encrypts them, and multiplies homomorphically. After decryption and unpacking, we find the product matrix $A \times B$. It is hard to do homomorphic multiplication for large matrices since the degree of the packed polynomials and the encryption parameters increase. In addition, for large matrices, homomorphic encryption operations become inefficient. In this thesis, we propose using the Strassen algorithm after dividing matrices into subblocks and applying a secure matrix multiplication algorithm to multiply two matrices securely to improve performance. The Strassen algorithm reduces the number of homomorphic multiplications from eight to seven in one-level recursion. Moreover, the homomorphic encryptions and multiplications are performed more efficiently because of a decrease in the degree of polynomials. We implement the Strassen algorithm for secure matrix multiplication using Fan and Vercauteren (FV) and Brakerski, Gentry, and

Vaikuntanathan (BGV) homomorphic encryption algorithms. The implementation results for dimensions between 8 and 128 with different submatrix sizes demonstrate significant improvements. To illustrate, when the FV homomorphic encryption is used for 128×128 matrix with the submatrix size of two, the algorithm is 47% faster than the standard secure block matrix multiplication method. Similarly, when using the BGV algorithm and considering a 128×128 matrix with a submatrix size of two, the algorithm is 44% faster than the standard secure block matrix multiplication. So, our implementation highlights the benefits of using the Strassen method for secure matrix multiplication within the homomorphic encryption framework, emphasizing its potential to improve performance, especially for bigger dimensions after dividing matrices into subblocks. Moreover, as an application, we calculate the multiple linear regression of encrypted data using the Strassen secure block matrix algorithm and compare results with the standard secure block matrix method. According to the results, when the matrix dimension is 128, and the submatrix size is two, computing the multiple linear regression with Strassen's secure block matrix multiplication algorithm is 47% faster than the standard secure block matrix multiplication algorithm.

Keywords: Secure Matrix Multiplication, The Strassen Algorithm, Homomorphic Encryption.

ÖZ

STRASSEN ALGORİTMASI İLE DAHA HIZLI GÜVENLİ MATRİS ÇARPIMI

ŞİMŞEK, DİLEK ÖNER
Doktora, Kriptografi Bölümü
Tez Yöneticisi : Prof. Dr. Ersan Akyıldız
Ortak Tez Yöneticisi : Prof. Dr. Murat Cenk

Ağustos 2023, 73 sayfa

Bulutta, kişisel veriler güvenlik sorunlarına açıktır. Homomorfik şifreleme, şifrelenmiş veriler üzerinde hesaplamalara olanak sağlayarak gizlilik endişelerini gidermeye yönelik bir çözüm sunar. Bu hesaplamalar istatistiksel analiz veya makine öğrenme algoritmaları olabilir. Matris çarpımı şifrelenmiş verilere de uygulanabilir. Buna güvenli matris çarpımı denir. Güvenli matris çarpımı, istatistiksel analiz ve makine öğrenimi algoritmaları da dahil olmak üzere birçok uygulamadaki temel işlemlerden biridir. A ve B'nin iki kare matris olduğunu varsayalım; güvenli matris çarpım algoritması, her matris için sıkıştırılmış polinomlar oluşturur, bunları şifreler ve homomorfik olarak çarpar. Şifreyi çözdükten ve sıkıştırılmış polinomu açtıktan sonra $A \times B$ çarpım matrisini bulabiliriz. Büyük matrisler için homomorfik çarpım yapmak, sıkıştırılmış polinomların derecesinin ve şifreleme parametrelerinin artması nedeniyle zordur. Ayrıca büyük matrisler için homomorfik şifreleme işlemleri verimsiz hale gelir. Bu tezde, matrisleri alt bloklara böldükten sonra, güvenli matris çarpım algoritmasını uygulamaktayız ve iki matrisi güvenli bir şekilde çarpmak için performansı artırmak amacıyla Strassen algoritmasını kullanmayı önermekteyiz. Strassen algoritması bir seviye özyineleme kullanıldığında homomorfik şifreleme sayısını sekizden yediye düşürmektedir. Ayrıca, homomorfik şifreleme ve çarpmalar polinomların derecesindeki azalmadan dolayı daha verimli uygulanmaktadır. Güvenli matris çarpımı için Fan ve Vercauteren (FV) ve Brakerski, Gentry ve Vaikuntanathan (BGV) homomorfik şifreleme algoritmalarını uygulamaktayız. 8 ile 128 arasındaki farklı matris ve alt matris

boyutları için uygulama sonuçları önemli gelişmeler göstermektedir. Örneğin, FV homomorfik şifreleme kullanıldığında 128×128 boyutlu matris için alt matris boyutu iki olduğunda Strassen güvenli blok matris çarpımı algoritması standart blok matris çarpımına göre $\%47$ daha hızlıdır. Ayrıca, benzer şekilde, BGV algoritması kullanıldığında alt matris boyutu iki olan 128×128 boyutlu bir matris düşünüldüğünde, algoritma standart güvenli matris çarpımına kıyasla $\%44$ daha hızlıdır. Bu nedenle, uygulamamız homomorfik şifreleme çerçevesinde güvenli matris çarpımı için Strassen yöntemini kullanmanın faydalarının altını çizerek, özellikle daha büyük boyutlar için matrisleri alt bloklara böldükten sonra performansı artırma potansiyelini vurgulamaktadır. Ayrıca, bir uygulama olarak, Strassen güvenli blok matris algoritmasını kullanarak şifrelenmiş verilerin çoklu doğrusal regresyonunu hesaplamakta ve sonuçları standart blok matris yöntemiyle karşılaştırmaktayız. Elde edilen sonuçlara göre, matris boyutu 128 ve alt matris boyutu iki olduğunda, Strassen'in güvenli blok matris çarpma algoritması ile çoklu doğrusal regresyon hesaplaması, standart güvenli blok matris çarpma algoritmasına göre $\%47$ daha hızlıdır.

Anahtar Kelimeler: Güvenli Matris Çarpımı, Strassen Algoritması, Homomorfik Şifreleme.

To My Family

ACKNOWLEDGMENTS

I would like to express my deepest thanks to Professor Dr. Murat Cenk, my thesis supervisor, for his essential assistance, courage, enthusiastic encouragement, and helpful advice during the development and production of this thesis. His unwavering commitment and eagerness to impart his knowledge have influenced my studies. Additionally, I would like to thank Professor Dr. Ersan Akyildiz for his helpful advice, intelligent talks, and contributions. Their contributions have considerably improved this work.

I especially appreciate my parents, Mehmet Ali and Emel Öner, for their everlasting love, support, and inspiration throughout my academic career. Their confidence in me has given me ongoing motivation. I also want to sincerely thank my sisters Hülya, Leyla, Ayla, and Seda for their steadfast support and inspiration.

I am also grateful to my husband, Associate Professor Tuncay Şimşek, for his immense understanding, patience, and stable support.

Finally, I want to sincerely thank my beautiful, lovely daughter, Dila, for bringing immense joy to my life and inspiring me to pursue my dreams. Her presence has been a constant reminder of the importance of balance, resilience, and determination. She is the best gift life has given me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

F_q	Finite field with q elements
Z_q	the ring of integers modulo q, where q is a prime number or a power of a prime number.
Z_q^n	the n-dimensional vector space over the ring $\mathbb{Z}_q$.
$f(x)$	a monic irreducible polynomial of degree d
R	$\mathbb{Z}[x]/(f(x))$
R_t	the message space for some integer $t > 1$
R_q	the set of polynomials in R with coefficients in $\mathbb{Z}_q$
Δ	$\lfloor q/t \rfloor$
δ_R	expansion factor of R
$\lfloor x \rfloor$	rounding down
$\lceil x \rceil$	rounding up
$\lfloor x \rceil$	rounding to the nearest integer
FHE	Fully Homomorphic Encryption
TFHE	Fast Fully Homomorphic Encryption over the Torus
SMM	Secure Matrix Multiplication Algorithm
SIMD	Single Instruction Multiple Data
(B)FV	Fan and Vercauteren Homomorphic Encryption Algorithm
BGV	Brakerski, Gentry, and Vaikuntanathan Homomorphic Encryption Algorithm
BV	Brakerski and Vaikuntanathan Homomorphic Encryption Algorithm
BGN	Boneh Goh Nissim Homomorphic Encryption Algorithm
CKKS	Cheon, Kim, Kim, and Song Homomorphic Encryption Algorithm
CRT	Chinese Remainder Theorem

CHAPTER 1

INTRODUCTION

In the cloud, calculations of private data, including medical and financial data, are vulnerable, and confidential data can be leaked. So, while performing computations of confidential data on the cloud, we need to encrypt them to provide security. These computations may be statistical functions like mean, standard deviation, or machine learning algorithms such as regression, logistic regression, and decision trees. One technique that allows users to carry out calculations on encrypted data in the cloud is homomorphic encryption. Security of fully homomorphic encryption algorithms primarily depends on challenging problems like the shortest vector problem, the discrete Gaussian sampling challenge, learning with error (LWE), and the ring learning with error (RLWE) problems. While both LWE and RLWE problems can serve as the basis for the hardness assumption in a Fully Homomorphic Encryption (FHE) scheme, RLWE demonstrates better performance. The RLWE scheme offers the advantage of reduced key sizes compared to the LWE scheme. BGV [5], B/FV [18], TFHE [13], and CKKS [12] are the most widely adopted and practical FHE schemes based on the RLWE problem. In this thesis, we use the BGV and B/FV algorithms for homomorphic operations in the secure matrix multiplication algorithm. According to [14], the BGV scheme requires a larger parameter set than the FV scheme when using a small plaintext modulus and a given circuit depth. However, when the plaintext modulus is of medium or large size, BGV performs better than FV.

Several homomorphic libraries have been published to implement fully homomorphic encryption algorithms, such as SEAL [11], HElib [22], and PALASIDE [41]. PALISADE is a DARPA and IARPA-funded project. It was built by a team at the

New Jersey Institute of Technology. The Cryptography and Privacy Research Group in Microsoft developed SEAL. Shai Halevi and Victor Shoup proposed HElib. In [8], a table lists popular FHE libraries with their authors, support schemes, and the language they are written in. SEAL, HElib, and PALASIDE libraries implement the BFV scheme. In addition, SEAL and HElib both implement CKKS and BGV. In this work, we use the SEAL library for homomorphic calculations of the BFV and BGV schemes.

For many applications, like statistical analysis or machine learning algorithms, matrix multiplication is one of the most essential operations. Secure matrix multiplication is also an essential operation in these domains. The secure matrix multiplication algorithm comprises a packing method and a homomorphic encryption algorithm such as FV or BGV. Packing methods are generally used to enhance the performance of the homomorphic calculations by reducing the number of homomorphic operations. The process of secure matrix multiplication is as follows: the matrices A and B are packed, and one polynomial is obtained for each matrix, a homomorphic encryption algorithm encrypts these polynomials, and encrypted polynomials are homomorphically multiplied. So, the result is still polynomial. However, we can again translate it to a matrix using unpacking and decryption. So, the result is a product matrix C. Yasuda et al. proposed new packing methods to increase efficiency and decrease the size of encrypted data, which can be used for somewhat homomorphic encryption schemes based on polynomial rings $R = \mathbb{Z}[x]/(x^n + 1)$ [39], [40]. Later, Duong et al. [16] generalized Yasuda et al.'s packing method for the secure inner product. Their calculation technique is a variant of Yasuda et al.'s [39] packing method. They implemented their algorithms and gave a comparison with previous approaches. Mishra et al. [26] generalized Duong et al.'s packing algorithm for more than one matrix multiplication over the BGV scheme. Since Duong's method needs large parameters for successful decryption, Mishra et al. proposed block matrix multiplication and the Chinese remain theorem to eliminate this problem. They implemented their method and compared it with the matrix-vector multiplication over HElib. Lastly, Wang et al. [36] introduced a column-order matrix encoding methodology, which requires a smaller parameter, to enhance Mishra et al.'s matrix encoding method.

Mishra et al. [27] proposed dividing matrices into subblocks since Duong's method

needs large parameters for successful decryption and to enhance performance, especially for large matrix sizes. However, additional improvements are needed. In this thesis, we increase the performance of Duong et al.'s secure matrix multiplication algorithm [16] by using the Strassen matrix multiplication algorithm [34] after dividing matrices into subblocks, and we name it Strassen's secure block matrix multiplication algorithm. We show that Strassen's secure block matrix multiplication algorithm performs better even with small dimensions. Using the Strassen algorithm, the number of homomorphic multiplications decreases from eight to seven in one-level recursion. We use the BFV and BGV homomorphic encryptions with the Strassen secure block matrix multiplication algorithm and compare results. Both BGV and BFV are lattice-based, quantum-secure, fully homomorphic encryption algorithms. When we use the FV algorithm in Strassen's secure block matrix multiplication, for 128×128 matrix, with the submatrix size is 16, the algorithm is 23% faster, while the submatrix size is two, the algorithm is 47% faster compared to the standard secure block matrix multiplication. In the case of the BGV algorithm, when considering a 128×128 matrix with a submatrix size of two, the Strassen algorithm is 44% faster than compared to the standard secure matrix multiplication, and for 96×96 matrix, when the submatrix size is three the algorithm is 42% faster than the standard secure block matrix multiplication. Overall, our implementation highlights the benefits of using the Strassen algorithm to secure matrix multiplication within the homomorphic encryption framework, emphasizing its potential to improve performance, particularly for larger dimensions. Moreover, dividing matrices into submatrices makes improvements to the parameters. We can multiply larger-sized matrices with smaller polynomial modulus degrees. Since we divide matrices into subblocks, the degree of the packed polynomials decreases, which also affects the cost. In addition, as an application, we compute the multiple linear regression of encrypted data using the Strassen secure block matrix multiplication and compare the results with the standard secure block matrix multiplication. According to the results, when the matrix dimension is 128, and the submatrix size is two, calculating the multiple linear regression with Strassen's secure block matrix multiplication algorithm is 47% faster than the standard secure block matrix multiplication algorithm.

The remaining sections are arranged as follows: Chapter 2 explains homomorphic

encryption and RLWE-based homomorphic encryption algorithms used in the thesis, namely the BGV and FV. Chapter 3 discusses the standard secure matrix multiplication, the secure matrix multiplication, and the standard secure block matrix multiplication algorithms. Chapter 4 presents the Strassen matrix multiplication, Strassen's secure block matrix multiplication algorithms, and our improvements, including the implementation results of Strassen's and standard secure block matrix multiplication algorithms. In Chapter 5, applications of Strassen's secure block matrix multiplication are discussed, and the multiple linear regression of encrypted data is implemented and compared with the standard secure block matrix multiplication algorithm. Lastly, Chapter 6 concludes the study.

CHAPTER 2

PRELIMINARIES

2.1 Homomorphic Encryption

Users can perform encrypted data calculations using homomorphic encryption while maintaining its function and format. This property of homomorphic encryption enables the protection of private information on the cloud. The use of homomorphic encryption enables mathematical operations on the ciphertext instead of the data. Let E be an encryption function, a, and b be the messages, homomorphic multiplication is $E(a) \otimes E(b) = E(a \times b)$. So, one can multiply two messages without knowing a and b.

According to the number of permitted calculations on the encrypted data, homomorphic encryptions are split into three subgroups: somewhat, partially, and fully homomorphic encryptions. Partially homomorphic encryption allows a single type of operation to be repeated indefinitely. Partially homomorphic systems include RSA [31], ElGamal [17], and Paillier [29]. Paillier is an additive homomorphic algorithm, and RSA and ElGamal are multiplicative algorithms. Some types of operations can be performed a certain number of times with somewhat homomorphic encryption. BGN [3], BGV and FV are examples of a somewhat homomorphic encryption scheme. Fully homomorphic encryption allows for an endless amount of operations to be carried out. Gentry's algorithm [20] is an example of a fully homomorphic encryption system. Fully homomorphic encryption algorithms are divided into four main subgroups: ideal lattice-based, over integers, (R)LWE-based, and NTRU-like.

The LWE is a generalization of the parity learning problem, which was introduced by

Oded Regev in 2005 [30]. Assume s is a secret vector in $\mathbb{Z}_q^n$, a is a uniform vector generated from an oracle in $\mathbb{Z}_q^n$, the noise e which has a standard deviation of αq and is distributed normally in $\mathbb{Z}_q^n$. The oracle produces the value $(a, \langle a, s\rangle + e \bmod q)$. The same s and new a and e are used for the next iteration of this process. The goal is to find s.

In the RLWE problem, the vectors of integers are replaced with elements from the ring of integers $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$. The use of RLWE is more efficient compared to LWE because for a single LWE sample $(a, b = \langle a, s\rangle/q + e)$, a new vector $a \in \mathbb{Z}_q^n$ is required. On the other hand, a single RLWE sample $(a, b = as/q + e \bmod R)$ only requires a ring element a, but it yields a full vector b with n coordinates.

Brakerski and Vaikuntanathan proposed the first partially homomorphic encryption method based on RLWE [7]. To make their approach fully homomorphic, Brakerski and Vaikuntanathan employed Gentry's blueprint squashing and bootstrapping technique [20]. Later, in [6], they introduced relinearization and modulus switching techniques and removed the squashing step. In the paper by Brakerski, Gentry, and Vaikuntanathan [5], an improvement was made upon Brakerski and Vaikuntanathan's scheme. They created the relinearization and modulus switching procedures, introduced a universal approach that can be used for both LWE and RLWE and made fully homomorphic encryption (FHE) possible without bootstrapping. Another contribution by Brakerski [4] established a scale-invariant method that replaces modulus switching. Additionally, Brakerski's approach was developed by Fan and Vercauteren [18] using ring learning with error setting, increasing algorithm efficiency. This thesis uses the BGV and FV algorithms for homomorphic encryption and multiplication.

The development of fully homomorphic encryption can be grouped into four generations according to the construction. The first generation FHE is Gentry's original scheme [20], which used ideal lattices. After, Dijk et al. [35] generated another algorithm based on integer arithmetic. However, since first-generation algorithms are inefficient because of rapid noise growth, they are not preferred in practical applications. The second-generation FHE schemes are based on the RLWE problem. Many encryption schemes used today are products of second-generation homomorphic encryptions. First, somewhat homomorphic encryption is constructed, then converted

to fully homomorphic using bootstrapping or modulus switching. Second-generation FHE started in 2011 with the BGV algorithm [5]. Later, an NTRU-based scheme called LTV [23] and BFV [18] followed it. In the third-generation FHE, relinearization steps are removed to do multiplication. GSW [21], FHEW [15] and TFHE [13] are third generation algorithms. The GSW scheme introduces the approximate eigenvector method, which provides an alternative approach for performing homomorphic operations and eliminates the need for key and modulus switching techniques. With this innovative method, the error growth brought on by homomorphic multiplications is drastically reduced to a small polynomial factor. Cheon, Kim, Kim, and Song [12] introduced the fourth generation of FHE algorithms in 2017. They proposed an approach for developing a leveled homomorphic encryption system for approximate arithmetic numbers and an open-source library formerly known as HEAAN. Fourth-generation schemes share similarities with second-generation schemes, but the key distinction is that fourth-generation schemes are approximate, allowing for considerably faster computation. The fourth generation incorporates the message space into a complex hyperplane, and the error introduced during encryption becomes a part of the approximation error that naturally arises during computations involving real-valued numbers. Second-generation schemes like BGV and B/FV are well-suited for modular exact arithmetic in finite fields. They offer efficient packing, enabling SIMD (Single Instruction Multiple Data) instructions to process computations over integer vectors (batching). These schemes are beneficial when processing large arrays of numbers simultaneously. However, they are unsuited for circuits requiring bootstrapping (such as circuits with significant multiplicative depth) or implementing non-linear functions. In cases where bit-wise operations and computations expressed as boolean circuits are predominant, it is recommended to utilize third-generation schemes such as TFHE, as they offer superior performance compared to earlier schemes [2]. However, it is important to note that TFHE has a limitation regarding the lack of support for CRT packing (batching). As a result, when simultaneous processing of large amounts of data is required, TFHE outperforms previous approaches. The fourth-generation scheme CKKS represents is the most suitable choice for arithmetic involving real numbers [25].

2.2 Homomorphic Encryption Algorithms

In this section, the BGV and B/FV algorithms are explained. We use both algorithms for secure matrix multiplication in the implementation. BGV and BFV are lattice-based, quantum-secure, fully homomorphic encryption algorithms. Also, they are both supported by the SEAL library.

2.2.1 Brakerski, Gentry and Vaikuntanathan Scheme (BGV)

The BGV scheme can be built upon both LWE [30] and RLWE problems [24], with the RLWE-based approach being more efficient. This algorithm incorporates key switching and modulus reduction techniques. Relinearization reduces the number of elements to two after multiplication, making the ciphertext decryptable using the secret key. The modulus switching method makes the algorithm fully homomorphic, and optimization is achieved through bootstrapping. This method converts a ciphertext $c_0 \in R_q^2$ into a ciphertext $c_1 \in R_p^2$, where decrypting c_1 still yields the message m. In addition, the modulus switching controls the multiplication noise. The key switching technique generalizes the relinearization method.

Let n be a power of 2, q be a positive odd integer modulus, and χ be an error distribution over R, where $R = \mathbb{Z}[x]/(x^n + 1)$. The length of the elements output by χ is constrained by B. The smallest possible value for B is chosen for security. BGV homomorphic encryption algorithm is as follows:

- Key Generation: λ is the security parameter, and s is the secret element $s \in \chi$ and set the secret key $k = (1, s) \in R_q^2$. $u \in R_q$ is generated randomly, e is a random error in χ. The output is secret and public keys $pk = (p_0, p_1) = (us + 2e, -u)$. Note that, $\langle pk, k \rangle = 2e$.
- Encryption: Let, the public key $pk \in R_q^2$ and the message $ms \in R_2$, convert ms into a vector $ms = (m, 0) \in R_q^2$ and choose random values $r, e_0, e_1 \in \chi$. Output the ciphertext $c = m + 2(e_0, e_1) + pk \cdot r$. Specifically, $c = (c_0, c_1) = (m + 2e_0 + p_0 r, 2e_1 - ur) \in R_q^2$.
- Decryption: Given the secret key $k \in R_q^2$ and the ciphertext $c \in R_q^2$, decryption

is $\langle c, k \rangle$.

$c_0 + c_1 k = m + 2e_0 + 2e_1 k + 2er$ and the output is:

$(m + 2(e_0 + e_1 k + er)$mod q $)$ mod $2 = m$.

Decryption works correctly since e, e_0, e_1, and k are small enough.

2.2.1.1 Homomorphic Addition and Multiplication

Adding components one by one is called homomorphic addition. If the resulting error falls inside the modulus q, then decryption is successful. The two input ciphertexts, which encrypt the messages m_0 and m_1, respectively, are assessed using the secret key as:

$$v = c_0 + c_1 \cdot k = m_0 + tn(\text{mod } q) \text{ and } v' = c'_0 + c'_1 \cdot k = m_1 + tn'(\text{mod } q).$$

The generated ciphertext is evaluated as follows;

$$v_{add} = v + v' = (m_0 + m_1) + t(n + n')(\text{mod } q).$$

The new noise is $n_{add} \approx n + n'$.

Due to the disregard for potential overflows modulo t in $m_0 + m_1$, this calculation is only an approximation.

Homomorphic multiplication is as follows:

$$\begin{aligned} &\langle c, k \rangle \cdot \langle c', k \rangle = (c_0 + c_1 k)(c'_0 + c'_1 k) \\ &= c_0 c'_0 + (c_0 c'_1 + c_1 c'_0)k + c_1 c'_1 k^2 \\ &= ct_0 + ct_1 k + ct_2 k^2. \end{aligned}$$

Hence, the extended ciphertext (ct_0, ct_1, ct_2) can be decrypted using the extended secret key $(1, k, k^2)$. However, each multiplication increases the size of the decryption key. To decrease the decryption key size, a key-switching technique is employed. Converting the ciphertext term $ct_2 k^2$ to $\overline{c_0} + \overline{c_1} k$ is the core idea behind this technique using the encryption of k^2 under k. Indeed, $Enc_k(k^2) = (\gamma, -\delta)$, where $(\gamma, -\delta) = (k^2 + 2e + urk, 2e_1 - ur) \approx (k^2 + \delta k, -\delta)$.

Thus, $k^2 \approx \gamma - \delta k$ and the extended ciphertext $ct_0 + ct_1 k + ct_2 k^2$ becomes a standard ciphertext with the encryption keys $\tilde{c_0} + \tilde{c_1} k$ which protects the identical plaintext.

The modulus switching approach transforms a ciphertext $c \in R_q^2$ into a ciphertext $c' \in R_p^2$, where decrypting c' still yields the same message m. Let $c = (c_0, c_1) \in R_q^2$ be a ciphertext, and consider c' to be the vector closest (using the ℓ-norm) to $(p/q) \cdot c$ such that $c' \equiv c \pmod 2$. It is important to note that for some $s \in \mathbb{Z}$, we have $[\langle c, k \rangle]_q = \langle c, k \rangle - sq$.

2.2.2 Fan and Vercauteren's Scheme(FV)

In the FV algorithm [18], the polynomial ring is defined as $R = \mathbb{Z}[x]/(f(x))$, where $f(x) \in \mathbb{Z}[x]$ is a polynomial of degree d that is monic irreducible. In practice, a common choice is to use a cyclotomic polynomial $m(x)$, the minimal polynomial of the primitive m-th roots of unity. The popular option is $f(x) = x^d + 1$ with $d = 2^k$. Let $a \in R$, and the coefficients of an element $a \in R$ are referred to as a_i. For an integer $q > 1$, $\mathbb{Z}[x]$ represents the set of integers $(-q/2, q/2]$, and R_q refers to the collection of polynomials in R with coefficients in $\mathbb{Z}_q$. The notation $[a]_q$ represents $a \mod q$. $\lfloor x \rceil$ represents rounding to the nearest integer, while $\lfloor x \rfloor$ and $\lceil x \rceil$ indicate rounding down and up, respectively. The FV algorithm employs two variants of relinearization: the first version generalizes the key switching in the BGV scheme [5], and the second version uses a method similar to modulus switching [7]. The FV algorithm also has a simpler decryption circuit compared to other somewhat homomorphic encryption algorithms. Let λ be the security parameter.

The FV algorithm consists of the following components [18]:

- $R = \mathbb{Z}[x]/(x^n + 1)$, where n is a power of two, and R_t is the message space for some integer $t > 1$. Set $\Delta = \lfloor q/t \rfloor$.
- SecretKeyGen(λ): A secret key $k \in R_q$ is sampled from a noise distribution χ.
- PublicKeyGen(sk): Compute $(p_0, p_1) = (-(a \cdot k + e) \mod q, a) \in R_q^2$, where a is sampled from R_q, and e is sampled from χ. The secret key is represented as k and is used in the computation.
- Encrypt(pk, m): Given a message $m \in R_t$, sample $u, y, z \leftarrow \chi$. The ciphertexts are $(ct_1, ct_2) = (p_0 \cdot u + y + \Delta \cdot m \mod q, p_1 \cdot u + z \mod q)$.

- Decryption: Compute $\left[\left\lfloor \frac{t\cdot[ct_1+ct_2\cdot k]_q}{q} \right\rceil\right]_t$. In other words, calculate $ct_1 + ct_2 \cdot k$, multiply by t/q, and round to the closest integer modulo t.

Note that there are two polynomial multiplications $M(n)$ with at most degree n in the encryption step in R_q. To demonstrate the validity of decryption for correctly encrypted ciphertexts, the following lemma is presented [18].

Lemma 1. *Assume that g is noise contained in the ciphertext and $||\chi|| < B$, we have*

$$[ct_1 + ct_2 \cdot k]_q = \Delta \cdot m + g$$

with $||g|| \leq 2 \cdot \delta_R \cdot B^2 + B$. This implies that for $2 \cdot \delta_R \cdot B^2 + B < \Delta/2$ decryption works correctly, where expansion factor $\rho_R = max\frac{||a\cdot b||_\infty}{||a||_\infty||b||_\infty} : a, b \in R$.

2.2.2.1 Homomorphic Addition and Multiplication

With homomorphic addition and multiplication, we can execute operations on encrypted data. After the decryption, the result is the same as it would have been done concerning unencrypted data.

Homomorphic addition of the FV algorithm on encrypted data is as follows: Assume that ct_i for $i = 1, 2$ are two ciphertexts, with $[ct_i(k)]_q = \Delta \cdot m_i + g_i$, then

$$[ct_1(k) + ct_2(k)]_q = \Delta \cdot [m_1 + m_2]_t + g_1 + g_2 - \epsilon \cdot t \cdot v$$

where $\epsilon = q/t - \Delta = q/t - \lfloor q/t \rfloor < 1$ and $m_1 + m_2 = [m_1 + m_2]_t + t \cdot r$. $||v|| \leq 1$, This indicates that the noise in the sum has increased additively by t. So, we have

$$c_{add}(ct_1, ct_2) = ([ct_1[0] + ct_2[0]]_q, [ct_1[1] + ct_2[1]]_q).$$

The process of homomorphic multiplication involves two phases. The first is multiplying two polynomials $ct_1(x)$ and $ct_2(x)$ and scaling by t/q. The second step is relinearization, which is used for the problem of ciphertext consisting of three-ring elements instead of two.

The following steps are basic multiplication. Let

$$ct_i = \Delta \cdot m_i + g_i + q \cdot v_i,$$

When we multiply ct_1 and ct_2:

$$\begin{aligned}(ct_1 \cdot ct_2)(k) &= (\Delta \cdot m_1 + g_1 + q \cdot v_1) \cdot (\Delta \cdot m_2 + g_2 + q \cdot v_2) \\ =&\Delta^2 \cdot m_1 \cdot m_2 + \Delta \cdot (m_1 \cdot g_2 + m_2 \cdot g_1) + q \cdot (g_1 \cdot v_2 + g_2 \cdot v_1) \\ &+ g_1 \cdot g_2 + q \cdot \Delta \cdot (m_1 \cdot v_2 + m_2 \cdot v_1) + q^2 \cdot v_1 \cdot v_2.\end{aligned}$$

We must recover a ciphertext that encrypts $[m_1 \cdot m_2]_t$. So, the equation is divided by q/t to prevent rounding errors. Let $ct_1(x) \cdot ct_2(x) = c_0 + c_1 \cdot x + c_2 \cdot x^2$

$$\tfrac{t}{q} \cdot (ct_1 \cdot ct_2)(k) = \lfloor t \cdot c_0/q \rceil + \lfloor t \cdot c_1/q \rceil \cdot k + \lfloor t \cdot c_2/q \rceil \cdot k^2 + v_\alpha$$

where

$$v_\alpha = (\lfloor t \cdot c_0/q \rceil - t \cdot c_0/q) + (\lfloor t \cdot c_1/q \rceil - t \cdot c_1/q) \cdot k + (\lfloor t \cdot c_2/q \rceil - t \cdot c_2/q) \cdot k^2$$

by the triangle inequality and this equation $||a - \lfloor a \rceil||_\infty \leq 1/2$ for all real numbers a, we obtain an approximation error r_a of size:

$$||v_a||_\infty < (\delta_R \cdot ||k||_\infty + 1)^2/2.$$

This concept demonstrates a homomorphic multiplication operation in which the ciphertexts ct_1 and ct_2, each consisting of two elements, are multiplied to yield a result with three elements:

$$c_{basicmult}(ct_1, ct_2) = \tfrac{t}{q} \cdot (ct_1 \cdot ct_2)(k) \text{ [19].}$$

Now that both plaintexts have been multiplied, we have a ciphertext that protects them. Relinearization is used to decrease the number of elements obtained in basic multiplication. In the FV algorithm, there are two versions of relinearization. The first is the generalization of key switching from BGV [5]. The second one is a form of modulus switching that changes modulo q to modulo $p \cdot q$. In basic multiplication,

we have:

$$c_0 = \left[\left\lfloor \frac{t\cdot(ct_1[0]\cdot ct_2[0]}{q} \right\rceil\right]_q$$
$$c_1 = \left[\left\lfloor \frac{t\cdot(ct_1[0]\cdot ct_2[1]+ct_1[1]\cdot ct_2[0])}{q} \right\rceil\right]_q$$
$$c_2 = \left[\left\lfloor \frac{t\cdot ct_1[1]\cdot ct_2[1]}{q} \right\rceil\right]_q.$$

While calculating c_0, c_1 and c_2 there are four polynomial multiplications $4M(n)$ in R_q.

Version 1: Write c_2 in base T, $c_2 = \sum_{i=0}^{\ell} c_2^{(i)} T^i$ with $c_2^i \in R_T$. For $\ell = \lfloor log_T(q) \rfloor$, sample $a_i \leftarrow R_q$, $e_i \leftarrow \chi$ and compute the relinearization key for $i = 0, 1, \ldots, \ell$:

$$rlk[i][j] = ([-(a_i \cdot k + e_i) + T^i \cdot k^2]_q, a_i), j = 0, 1$$

Compute c_0' and c_1' as:

$$c_0' = \left[c_0 + \sum_{i=0}^{\ell} rlk[i][0] \cdot c_2^{(i)}\right]_q$$
$$c_1' = \left[c_1 + \sum_{i=0}^{\ell} rlk[i][1] \cdot c_2^{(i)}\right]_q$$

return (c_0', c_1'). This is the result of homomorphic multiplication.

There are $\ell + 1$ polynomial multiplications while obtaining c_0', c_1', and the relinearization key. So, the cost of homomorphic multiplication is $3\ell + 7M(n)$.

Version 2: Sample $a \leftarrow R_{p.q}$, $e \leftarrow \chi'$ compute the relinearization key:

$$rlk = ([-(a \cdot k + e) + p \cdot k^2]_{p.q}, a).$$

Compute the result of the homomorphic multiplication as:

$$(c_{2,0}, c_{2,1}) = \left(\left[\left\lfloor \frac{c_2\cdot rlk[0]}{p} \right\rceil\right]_q, \left\lfloor \frac{c_2\cdot rlk[1]}{p} \right\rceil\right]_q\right)$$

return $(c_{2,0}, c_{2,1})$.

CHAPTER 3

SECURE MATRIX MULTIPLICATION

In homomorphic encryption, the packing methods make the computations efficient by allowing batching or reducing the number of homomorphic operations. For example, Lauter et al. [28] proposed a way to encode integers in a ciphertext to calculate their sums and products effectively. There are other works that use different packing methods for SIMD. To illustrate, Smart and Vercauteren [32] suggested a packing technique for SIMD using the Chinese Remainder Theorem. Their technique enables communications to be encrypted as a single ciphertext.

The secure matrix multiplication algorithm comprises a packing method and a homomorphic encryption algorithm such as FV or BGV. The process of secure matrix multiplication is as follows: first, the matrices A and B are packed, and one polynomial is obtained for each matrix. A homomorphic encryption algorithm encrypts these polynomials. Lastly, encrypted polynomials are homomorphically multiplied. So, the result is still polynomial. However, one can again translate it to a matrix using Theorem 1. So, the result is a product matrix C.

This section briefly defines the matrix multiplication algorithms. In section 3.1, we explain multiplying two matrices homomorphically in a standard way. The secure matrix multiplication algorithm is expressed in section 3.2. Lastly, in 3.3, the standard secure block matrix method is described.

3.1 The Standard Secure Matrix Multiplication

If we do not use a packing method, we encrypt all elements of the matrices A and B and then multiply them homomorphically in the standard secure matrix multiplication method. Let A and B be two $m \times m$ matrices with size m and $C = A \cdot B$,

$$A = \begin{bmatrix} a_{0,0} & \dots & a_{0,m-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,m-1} \end{bmatrix} \rightarrow \begin{bmatrix} enc(a_{0,0}) & \dots & enc(a_{0,m-1}) \\ \vdots & \ddots & \vdots \\ enc(a_{m-1,0}) & \dots & enc(a_{m-1,m-1}) \end{bmatrix}$$

$$B = \begin{bmatrix} b_{0,0} & \dots & b_{0,m-1} \\ \vdots & \ddots & \vdots \\ b_{m-1,0} & \dots & b_{m-1,m-1} \end{bmatrix} \rightarrow \begin{bmatrix} enc(b_{0,0}) & \dots & enc(b_{0,m-1}) \\ \vdots & \ddots & \vdots \\ enc(b_{m-1,0}) & \dots & enc(b_{m-1,m-1}) \end{bmatrix}.$$

The entries c_{ij} of the product matrix $C_{m\times m}$ for $0 \leq i, j \leq m-1$ of A and B can be computed as:

$$c_{ij} = \sum_{k=0}^{m-1} enc(a_{ik}) enc(b_{kj}).$$

The standard method needs m^3 homomorphic multiplications, $(m-1)m^2$ homomorphic additions, and $2m^2$ encryptions.

3.2 Secure Matrix Multiplication Algorithm

To enhance performance and minimize the size of encrypted data, Yasuda et al. presented new packing techniques that can be applied to somewhat homomorphic encryption algorithms based on polynomial rings $R = \mathbb{Z}[x]/(x^n + 1)$ [39], [40]. Later, Duong et al. [16] generalized Yasuda et al.'s packing method for the secure inner product. Their calculation method is a variant of Yasuda et al.'s [39] packing method. Building upon the work of Yasuda et al. [39], [40] on secure inner product computation, Doung et al. [16] extended the idea to enable matrix multiplication. In addition, they provided a revolutionary method for encoding a matrix in a single ciphertext, which reduces the size of the ciphertext and the computing cost to a single homomorphic multiplication for matrix multiplication operations. There are two methods

for packing. The first version is for small matrix entries, and the second is for large matrix entries. In this thesis, we use the packing method for small matrix entries.

Let Y and Z be two matrices of size $m \times m$ with integer entries, representing the matrices to be multiplied. Let $Y_1, \ldots, Y_m$ are the row vectors of Y, and $Z_1^T, \ldots, Z_m^T$ denote the column vectors of Z. To calculate the matrix multiplication $Y \cdot Z$, it is necessary to compute the inner products $\langle Y_j, Z_i^T \rangle$ for $j, i = 1, \ldots, m$. The packing of each row $Y_j = (y_{j,0}, \ldots, y_{j,m-1})$ and each column $Z_i^T = (z_{i,0}, \ldots, z_{i,m-1})$ can be achieved as follows:

$$\begin{aligned} pm^{(1)}(Y_j) &= \textstyle\sum_{a=0}^{m-1} y_{j,a} x^a, \\ pm^{(2)}(Z_i^T) &= -\textstyle\sum_{b=0}^{m-1} Z_{i,b} x^{n-b}. \end{aligned}$$

$pm(Y_j)$ and $pm(Z_i^T)$ are the polynomials representing the packed form of the row vectors Y_j and column vectors Z_i. Using the matrices Y and Z shown above as a reference, we may define the following two polynomials:

$$\begin{aligned} &Pol^{(1)}(Y) = pm(Y_1) + \ldots + pm(Y_m) x^{m(m-1)} \\ &= \textstyle\sum_{j=1}^{m} pm(Y_j) x^{(j-1)m}, \\ &Pol^{(2)}(Z) = pm(Z_1^T) + \ldots + pm(Z_m^T) x^{m^2(m-1)} \\ &= \textstyle\sum_{i=1}^{m} pm(Z_i^T) x^{(i-1)m^2}. \end{aligned} \tag{3.1}$$

Encrypt two types of polynomials:

$$\begin{aligned} ct^{(i)}(Y) &= Enc(Pol^1(Y), pk), \ i = 1, 2. \\ ct^{(i)}(Z) &= Enc(Pol^2(Z), pk), \ i = 1, 2. \end{aligned} \tag{3.2}$$

This method only has one homomorphic multiplication and two encryptions of degree $m^2 - 1$ (for polynomial Y) and $2m^3 - m^2$ (for polynomial Z). However, since the reduction polynomial is $x^n + 1$, the degree of the polynomial Z can be maximum $n - 1$.

The following theorem [27] ensures that decryption yields the product of matrices. The parameter n generates polynomial Z, so increasing m leads to bigger n and more polynomial degrees of Y and Z.

Theorem 1. *Assume $n \geq m^3$, $ct_2 = ct^{(i)}(Y) * ct^{(i)}(Z)$, let k is secret key and $Dec(ct_2, k) \in R_t$ show the outcome of its decryption. If decryption works as intended*

for ct_2, then the inner product $\langle Y_j, Z_i^T \rangle$ modulo t corresponds to the coefficient of $x^{(i-1)m^2+(j-1)m}$ in $Dec(ct_2, k)$ for every $1 \leq j, i$.

This is accurate since $Dec(ct, k) = Pol(Y) \times Pol(Z) = \sum_{j=1}^{m} \sum_{i=1}^{m} pm(Y_j) \times pm(Z_i^T)\ x^{(i-1)m^2+(j-1)m}$. The term of degree $(i-1)m^2 + (j-1)m$ in $Dec(ct, k)$ exactly corresponds to the term of degree $(i-1)m^2+(j-1)m$ in $pm(Y_j) \times pm(Z_i^T) \cdot x^{(i-1)m^2+(j-1)m}$ in the polynomial ring R. As a result, the coefficient of $x^{(i-1)m^2+(j-1)m}$ in $Dec(ct, k)$ provides the internal product $\langle Y_j, Z_i^T \rangle$.

Now, we give an example of how the packing works on two submatrices of size two.

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}, B = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}, A \times B = \begin{pmatrix} 4 & 2 \\ 2 & 1 \end{pmatrix} \bmod 5.$$

Let $n = 8$, for matrix A we have $1 + 2x$ and $2 + 3x$ for each row and the final polynomial is $(1+2x)+(2+3x)(x^2)$. For matrix B, we have $-2x^8-x^7$ and $-x^8-3x^7$ for each column. The final polynomial for B is $(-2x^8 - x^7) + (-x^8 - 3x^7)(x^4)$.

Using the above submatrices, we now explain how the decryption and unpacking work using Theorem 1. Let $t = 5$, and the reduction polynomial is $f(x) = x^8 + 1$. Assume that, after the homomorphic multiplication and decryption of submatrices A and B, we obtain the polynomial $2x^7+x^6+3x^5+2x^4+x^3+2x^2+x+4$. According to Theorem1, the coefficient of x^0, which is 4, is equal to the first row and the first column of the product matrix. Similarly, the coefficient of x^4, which is two, is equal to the first row and the second column of the product matrix, the coefficient of x^2, which is two, is equivalent to the second row and the first column of the product matrix, and the coefficient of x^6 which is one is equal to the second row and the second column of the product matrix mod t. Hence, we find the product matrix from encrypted polynomials after this process.

3.3 Standard Secure Block Matrix Multiplication Method

We divide matrices into smaller submatrices in the standard secure block matrix multiplication method. Then, we obtain packed polynomials and encrypt them using the FV or BGV algorithm. When we divide matrices into subblocks with dimension

$\frac{m}{2} \times \frac{m}{2}$, after packing the degree of polynomial A_{ij} becomes $\frac{m^2}{4} - 1$, and polynomial B_{ij} becomes $\frac{m^3}{4} - \frac{m^2}{4}$. Lastly, we homomorphically multiply and add encrypted polynomials to calculate each matrix entry of C. We decrypt the encrypted entries of matrix C_{ij}, so we have four decryptions. After decryption, the polynomial coefficients are the product of the matrices A and B according to Theorem 1.

Assume that A_{ij}, B_{ij} and C_{ij} are submatrices of size $\frac{m}{2}$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \text{ and } C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{3.3}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.$$

Let n be the degree of polynomials and $T(n)$ be the complexity of matrix multiplication. We have the recursive expression $T(n) = 8T(\frac{n}{2}) + 4(\frac{n}{2})^2$.

We have eight homomorphic multiplications and four homomorphic additions in the standard secure block matrix multiplication. We pack matrices as in (3.1) and encrypt as in (3.2) so the degree of the polynomials after encryption and homomorphic multiplication can be maximum $n - 1$ since all the operations are in $R = \mathbb{Z}[x]/(x^n + 1)$. So the cost of homomorphic multiplication is $(3\ell + 7)M(n)$ for $n = m^3$ where $\ell = \lfloor log_T(q) \rfloor$ in the FV algorithm. When we divide matrices into subblocks, since the number of homomorphic multiplication is eight, the cost of standard secure block matrix multiplication becomes $(24\ell + 56)M(m^3/8)$.

Figure 3.1 shows the secure block matrix multiplication process. Alice and Bob first divide matrices into subblocks and obtain packed polynomials. Second, they encrypt their packed polynomials and send them to the cloud. So, their data are encrypted in the cloud. The homomorphic matrix multiplication process is calculated in the cloud. If someone wants to compute matrix multiplication, she downloads the encrypted matrix product and finds the matrix multiplication $P \times Q$ by decrypting it.

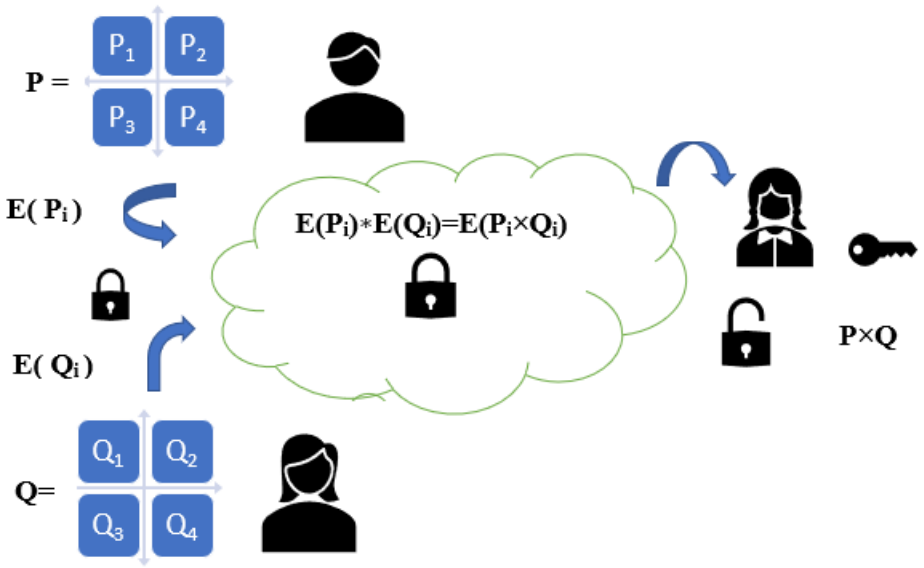

Figure 3.1: Secure Block Matrix Multiplication

CHAPTER 4

FASTER SECURE MATRIX MULTIPLICATION

It is hard to do homomorphic multiplications for large matrices since the cost increases significantly. So, for large dimensions, it is better to divide matrices into smaller blocks for efficiency. Mishra et al. [27] used smaller block matrices for secure matrix multiplication. Let P be a m-dimensional square matrix. Considering submatrix size M that is $m = b \cdot M$ for $b \in \mathbb{Z}$, we have b^2 sub-matrices with size $M \times M$. Instead of multiplying m-sized matrices, we multiply M-sized matrices, changing the cost of matrix multiplication to $b^3T(m)$. In this thesis, we propose Strassen's secure block matrix multiplication method to obtain improvements in the performance of the secure matrix multiplication algorithm.

In section 4.1, we describe Strassen's matrix multiplication algorithm. In section 4.2, we define Strassen's secure block matrix multiplication method. Lastly, in section 4.3, we explain the implementation results, which compare Strassen's secure block matrix multiplication with the standard secure block matrix multiplication.

4.1 Strassen's Matrix Multiplication Algorithm

Volker Strassen proposed the Strassen matrix multiplication algorithm in 1968 [34]. It is a recursive algorithm. Strassen's matrix multiplication is faster than the standard matrix multiplication, especially for large matrices. The computation cost of Strassen's algorithm is $O(m^{2.81})$, whereas the cost of the standard matrix multiplication algorithm is $O(m^3)$ where m is the matrix dimension. Assume P and Q are two square matrices with dimension m, which should be the power of two, and let

$C = P \cdot Q$. Strassen sets seven matrices $M_1, .., M_7$ with size $m/2$ as follows:

$$
\begin{aligned}
E_1 &= (P_{11} + P_{22})(Q_{11} + Q_{22}), \\
E_2 &= (P_{21} + P_{22})Q_{11}, \\
E_3 &= P_{11}(Q_{12} - Q_{22}), \\
E_4 &= P_{22}(Q_{21} - Q_{11}), \\
E_5 &= (P_{11} + P_{12})Q_{22}, \\
E_6 &= (P_{21} - P_{11})(Q_{11} + Q_{12}), \\
E_7 &= (P_{12} - P_{22})(Q_{21} + Q_{22}).
\end{aligned}
$$

There are seven multiplications and 18 additions in the Strassen algorithm. The product matrix $C_{m \times m}$ is calculated by using the above matrices:

$$
C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} E_1 + E_4 - E_5 + E_7 & E_3 + E_5 \\ E_2 + E_4 & E_1 - E_2 + E_3 + E_6 \end{pmatrix}.
$$

The Strassen algorithm keeps dividing the matrices into submatrices until they reach a crossover point where the submatrices are of a size 2^r, a size where the standard matrix multiplication is preferred. The algorithm is then used $k - r$ times recursively. Assume that the matrices are square and contain 2^k elements, the submatrices are of size $m_0 = 2^r$, the number of multiplications is $7^{k-r}8^r$, and the number of additions and subtractions are $4^r(2^r + 5)7^{k-r} - 6 \cdot 4^k$ when we apply the algorithm recursively until the crossover point is reached. In this thesis, we choose two as a crossover point, so we don't use the Strassen algorithm when the cipher matrix dimension is two.

The time complexity of the Strassen algorithm is:

$T(1) = 1, T(n) = 7T(n/2) + 18(n^2)$.

When the dimension is 2^k, the computational complexity becomes:

$$\begin{aligned}
T(n) &= 7T\left(\frac{2^k}{2}\right) + 18\left(\frac{2^k}{2}\right)^2 \\
&= 7T(2^{k-1}) + 18(2^{k-1})^2 \\
&= 7(7T(2^{k-2}) + 18(2^{k-2})^2) + 18(2^{k-1})^2 \\
&= 7^2T(2^{k-2}) + 7 \times 18(2^{k-2})^2 + 18(2^{k-1})^2 \\
&= 7^2(7T(2^{k-3}) + 18(2^{k-3})^2) + 7 \times 18(2^{k-2})^2 + 18(2^{k-1})^2 \\
&= 7^3T(2^{k-3}) + 7^2 \times 18(2^{k-3})^2 + 7 \times 18(2^{k-2})^2 + 18(2^{k-1})^2 \\
&\;\;\vdots \\
&= \sum_{i=0}^{k-1} 7^i \times 18(2^{k-i-1})^2 \\
&= 18(2^k)^2 \sum_{i=0}^{k-1} \frac{7^i}{(2^{i+1})^2} \\
&= \frac{9}{2}(2^k)^2 \sum_{i=0}^{k=1} \left(\frac{7}{4}\right)^i \\
&= \frac{9}{2} \times (2^k)^2 \times \frac{1 - (\frac{7}{4})^k}{1 - \frac{7}{4}} \\
&= 6 \times 7^k - 6 \times 4^k.
\end{aligned}$$

So we obtain $T(n) = 6n^{2.81} - 6n^2$, $n > 1$ and $T(1) = 1$.

It should also be noted that there are some further improvements to Strassen's algorithm. However, those improvements are enhancements of the additions. Winograd found 15 additions, a Strassen-like additively optimum algorithm [37] whose arithmetic complexity is $6n^{2,81} - 5n^2$ for $n = 2^k$ and [37]$(3.73n^{2.81} - 5n^2)$ for $n = 8 \cdot 2^k$ [37]. Cenk and Hasan [9] proposed a method that reduces the complexity of Winograd's variant to $(5n^{2.81} + 0.5n^{2.59} + 2n^{2.32} - 6.5n^2)$ for $n = 2^k$. They also showed that the total arithmetic complexity could be improved to $(3.55n^{2.81} + 0.148n^{2.59} + 1,02n^{2.32} - 6.5n^2)$ for $n = 8 \cdot 2^k$ [9]. After that, those works are the latest results on the additive complexity of Strassen-like multiplications [9]. Note that since we are mainly focused on the multiplications in homomorphic settings, we show improvements using the Strassen algorithm.

In the next section, we show the use of Strassen's matrix multiplication in the secure block matrix multiplication approach. Since the multiplications are much more costly than the additions in homomorphic encryption, we obtain significant improvements

over the classical block matrix multiplication.

4.2 Strassen's Secure Block Matrix Multiplication Method

In this method, we divide matrices into submatrices as in (3.3), and we obtain packed polynomials for each submatrix using (3.1). After that, we encrypt them with the FV or BGV algorithm. $M_1, M_2, ..M_7$ are calculated by homomorphic multiplications and additions of encrypted polynomials. The product matrix entries of C are computed with calculated M_i's. As can be seen, there are seven homomorphic multiplications and eighteen homomorphic additions in this algorithm. After homomorphic multiplication and addition of the packed polynomials, we obtain encrypted matrix entries of C. We decrypt C_{ij}'s, so there are four decryptions. According to Theorem 1, after decryption, the coefficients of the polynomial are equal to the entries of the product matrix $A \times B$.

Assuming the dimension of the matrices A and B is $m = 2^k$, n should be more than or equal to m^3 for decryption to work correctly. However, when we divide matrices into submatrices of size $\frac{m}{2}$, the smallest value of n decreases to $\frac{n}{8}$. We can divide matrices into subblocks with dimensions $\frac{m}{4}$ and $\frac{m}{8}$, then the smallest value of n becomes $\frac{n}{2^6}$ and $\frac{n}{2^9}$, respectively. In other words, since we divide matrices into subblocks, the degree of the packed polynomials and the other parameters like n, which is the degree of reduction polynomial, decreases. So this decrease affects the cost.

In Strassen's method, although the number of homomorphic addition is more than classical block matrix multiplication, its cost is insignificant since homomorphic addition is just polynomial addition in secure matrix multiplication.

Now, we give an example to observe the improvements. Let A and B be 8×8 matrices; if we recursively apply the Strassen algorithm twice, the submatrix size will be two. We start from 8×8 matrices and reach 2×2 matrices by splitting the initial matrix by two recursively. Then we obtain a polynomial for all 2×2 matrices and encrypt them using the secure matrix multiplication algorithm. The following

matrices show the starting 8×8 matrices and the smaller sizes matrices after splitting.

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,7} \\ a_{1,0} & a_{1,1} & \dots & a_{1,7} \\ \vdots & \vdots & & \vdots \\ a_{6,0} & a_{6,1} & \dots & a_{6,7} \\ a_{7,0} & a_{7,1} & \dots & a_{7,7} \end{bmatrix}, B \begin{bmatrix} b_{0,0} & b_{0,1} & \dots & b_{0,7} \\ b_{1,0} & b_{1,1} & \dots & b_{1,7} \\ \vdots & \vdots & & \vdots \\ b_{6,0} & b_{6,1} & \dots & b_{6,7} \\ b_{7,0} & b_{7,1} & \dots & b_{7,7} \end{bmatrix}$$

$$A_{11} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}, B_{11} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

$$A'_{11} = \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}, B'_{11} = \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix}$$

A and B are 8×8 matrix, A_{11} and B_{11} are 4×4, and A'_{11} and B'_{11} are 2×2 matrices. Since we obtain polynomials from 2×2 matrices we have four polynomials for matrices A_{11} and B_{11} as follows:

$$A_{11} = \begin{bmatrix} P_1 & P_2 \\ P_3 & P_4 \end{bmatrix}, \; B_{11} = \begin{bmatrix} Q_1 & Q_2 \\ Q_3 & Q_4 \end{bmatrix}.$$

For matrix A and B, we have 16 polynomials. Then, we encrypt these polynomials. Now, we can call matrix A and B as ciphertext matrix.

$$A = \begin{bmatrix} P_1 & P_2 & P_5 & P_6 \\ P_3 & P_4 & P_7 & P_8 \\ P_9 & P_{10} & P_{13} & P_{14} \\ P_{11} & P_{12} & P_{15} & P_{16} \end{bmatrix}, \; B = \begin{bmatrix} Q_1 & Q_2 & Q_5 & Q_6 \\ Q_3 & Q_4 & Q_7 & Q_8 \\ Q_9 & Q_{10} & Q_{13} & Q_{14} \\ Q_{11} & Q_{12} & Q_{15} & Q_{16} \end{bmatrix}.$$

Assume we apply the Strassen algorithm recursively twice to multiply matrix A and B; for a 2×2 matrix, the standard matrix multiplication method is used since the crossover point is two. Hence, we have eight polynomial multiplications when the matrix dimension is two. In the second round, we apply the Strassen multiplication, so we have seven multiplications. In total, we have $7 \cdot 8 = 56$ homomorphic polynomial multiplications. Let the matrices contain 2^k elements, and the submatrices are of

size $m_0 = 2^r$. When the crossover point is reached, the number of homomorphic multiplications is $7^{k-1-r}8^r$, and the number of homomorphic additions and subtractions is $4^r(2^r + 5)7^{k-1-r} - 6 \cdot 4^k - 1$.

Since $n = m^3$, and there are seven homomorphic multiplications in one level, the Strassen secure block matrix multiplication cost becomes $(21\ell + 49)M(m^3/8)$. In the second level of Strassen's secure block matrix multiplication, the cost is $49(3\ell + 7)M(m^3/64)$. While in the second level of standard secure block matrix multiplication, the cost is $64(3\ell + 7)M(\frac{m^3}{64})$. So, for c recursive calls, the cost is $7^c(3\ell + 7)M(\frac{m^3}{8^c})$.

4.3 Implementation Results

We use Visual Studio 2022 and MSVC for C++ implementation, Microsoft SEAL version 4.0 for homomorphic encryption and multiplication, and CMake 3.23.1 for building and installing SEAL. SEAL is an open-source homomorphic encryption library with an MIT license. The implementation ran on Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz (8 CPUs), 1.2GHz, 8 GB RAM. In the implementation step, first, we divide $m \times m$ matrices into $M \times M$ matrices, so $m = b \times M$. As a result, we obtain b^2 $M \times M$ matrices. For securely multiplying matrices, we turn these submatrices into polynomials by the secure matrix multiplication algorithm and encrypt these polynomials with the BFV and BGV algorithms. After encryption, we homomorphically multiply these polynomials by standard and Strassen's method and obtain product polynomials. Then, we decrypt using Theorem 1 and obtain the matrix form of the product matrix. The calculation time of all these processes is specified in Table 4.1 and 4.2 as microseconds.

Three encryption parameters must be set for implementing the BFV and BGV algorithms: polynomial modulus degree, coefficient modulus, and plaintext modulus. The polynomial modulus degree $(x^n + 1)$ is the first variable, which must be a positive power of two. Polynomial modulus n represents the degree of a power-of-two cyclotomic polynomial. If the polynomial modulus degree gets large, the ciphertext sizes also get more extensive, making operations slower but enabling more advanced en-

crypted calculations. For the polynomial modulus degree, we set 4096 and 8192. The next component is the ciphertext coefficient modulus, a big integer of several prime values, each with a maximum bit size of 60. These prime numbers are all represented by instances of the modulus class, and together, they make up the ciphertext coefficient modulus. The sum of the bit-lengths of its prime factors is the bit-length of the coefficient modulus. When the coefficient modulus gets large, the noise budget increases. Thus, encrypted computation capabilities rise. The coefficient modulus depends on the polynomial modulus. While the polynomial modulus degree is 4096, the coefficient modulus bit length can be maximum 109, and while the polynomial modulus degree is 8192, the bit length of the coefficient modulus can be maximum 218 bits. Lastly, the plaintext modulus (modulus t) controls the size of the plaintext data type and how much noise budget is used during multiplications. Hence, minimizing the plaintext data type is crucial for the greatest speed. We take default values for the plaintext modulus and ciphertext coefficient modulus.

The other parameter is the crossover point. The Strassen algorithm is recursive. When the matrices are small enough, we switch the Strassen algorithm to conventional matrix multiplication. The crossover point between the two algorithms is the k when recursive Strassen's algorithm is stopped and switched to conventional matrix multiplication. We choose two as a crossover point. Hence, when the submatrix size is $m/2$, we don't use the Strassen algorithm for multiplication.

Table 4.1 compares the standard secure block matrix multiplication and Strassen's secure block matrix multiplication algorithm results for different m (matrix dimension) and M (submatrix dimension). The drawbacks of Strassen's algorithm are the first is recursion stack uses up more memory, and the second is recursive calls increase latency. Strassen's algorithm generally shows a significant performance increase compared to the standard algorithm when dimensions are large. In our case, the Strassen algorithm is better even dimensions are small, and the performance is much better when m is bigger and M is smaller.

To illustrate, when we use the FV algorithm in Strassen's secure block matrix multiplication for 128×128 matrix, with the submatrix size is 16, the algorithm is 23% faster, while the submatrix size is two, the algorithm is 47% faster compared to the

standard secure block matrix multiplication. The algorithm is the fastest when the submatrix size is two, and when submatrices become larger, the results get closer to the standard secure block matrix multiplication. However, the time duration increases when the submatrix sizes get smaller since many computations exist calculating recursive operations. We write the total period of encryption of matrices A and B in the encryption column. The total decryption period of Strassen's and standard secure block matrix multiplication in microseconds are written in the decryption column. In Strassen's multiplication column, we write the time period of multiplication of matrices A and B using Strassen's secure block matrix multiplication method, and in the Standard multiplication column, we write the time period of matrix multiplication using the standard secure block matrix multiplication method. The proportion column is calculated by subtracting Strassen's secure block matrix multiplication time from standard secure block matrix multiplication time over standard secure block matrix multiplication time.

In Table 4.2, we use the BGV algorithm for encryption. The best performance increase is when the submatrix size is two. However, the duration of the Strassen algorithm gets closer to the standard algorithm when the submatrix sizes increase. Also, notably, when considering a 128×128 matrix with a submatrix size of two, the algorithm is 44% faster than compared to the standard secure matrix multiplication, and for 96×96 matrix, when the submatrix size is three Strassen's secure block matrix multiplication algorithm is 42% faster than the standard secure block matrix multiplication. Overall, our implementation showcases the advantages of utilizing the Strassen algorithm for secure matrix multiplication within the homomorphic encryption framework, highlighting its potential to enhance performance, especially for larger dimensions after dividing matrices into subblocks.

Note that the ciphertext matrix size must be a power of 2^k since we divide it recursively. In addition, the matrices must be square to use secure matrix multiplication. Moreover, SEAL does not support polynomial multiplications with its encoding and decoding. When we encode the polynomial with the SEAL encoder, it does not do polynomial multiplication but mutual multiplication. So, we need to write a new encoding and decoding method for polynomial multiplication, which turns the packed polynomials into plaintext of SEAL to encrypt with the encryptor. However, even if

we use our encoding and decoding, SEAL permits polynomial multiplication for a particular degree. After packing and multiplications, the coefficients of the polynomials increase a lot, exceeding the value allowed by SEAL. So, we use small random numbers up to five for matrix entries. When matrix entries get large, the algorithm works for smaller matrix dimensions. Also, since the polynomial modulus degree is 4096 and 8192 and $n \geq m^3$, the submatrix size can be maximum 16. For submatrix size 16, we use 8192; for other submatrix sizes, we use 4096 as polynomial modulus degree. Using Strassen's secure block matrix multiplication, we can compute homomorphic multiplication of 128×128 matrices with small polynomial modulus degrees such as 4096 and 8192.

In Table 4.1, we use the BFV algorithm and in Table 4.2, we use the BGV algorithm for homomorphic operations. Although the proportions are similar, the increase in the speed of Strassen's secure block matrix multiplication algorithm in the BFV algorithm is better than BGV according to the standard secure block matrix multiplication. However, when we compare BGV and BFV, we can see that the BGV algorithm is faster than BFV, and the operations take longer in the BFV algorithm. Details of the comparison are shown in Table 4.1 and 4.2.

Table 4.1: Comparison of Secure Matrix Multiplication Algorithms by Using the BFV Algorithm

m	M	Standard SMM	Strassen's SMM	Encryption	Decryption	%
8	2	288576	261567	58047	28786	9
16	2	2148165	1693277	206050	37217	21
	4	286433	259889	51756	9374	9
32	2	16965719	11704723	770501	102955	31
	4	2301354	1774490	206921	45720	22
	8	283861	249750	500538	17982	12
64	2	142247862	86628801	3030896	423107	39
	4	17320323	12283974	837624	134258	29
	8	2182583	1743846	2069277	92620	20
	16	1191886	1066338	52477306	146378	10
96	3	141529104	85224711	3031700	407875	39
	6	17354993	11944239	1729807	161514	31
	12	2199365	1736270	27752006	174312	21
128	2	1138692212	603107625	12049110	1512013	47
	4	140523224	85083122	3100359	449711	39
	8	79455365	49104099	5829112	364905	38
	16	9785657	7449247	106194823	616361	23

Table 4.2: Comparison of Secure Matrix Multiplication Algorithms by Using the BGV Algorithm

m	M	Naive SMM	Strassen's SMM	Encryption	Decryption	%
8	2	138718	125962	76571	6186	9
16	2	922668	751666	226507	24466	18
	4	139616	122902	61176	6971	11
32	2	6903482	5017813	781921	98408	27
	4	888173	736762	229499	26072	17
	8	133741	121326	528559	18986	9
64	2	56977988	36128309	3108642	346691	36
	4	6997816	5039658	800733	119649	27
	8	926054	765145	2091253	67747	17
	16	552866	498686	54255136	134462	9
96	3	65892161	37929429	3339790	362700	42
	6	7126283	4988186	1770325	148037	30
	12	922274	735972	27848397	199316	20
128	2	458697614	254024929	12793617	1333375	44
	4	59803178	36908125	3305789	801433	38
	8	7166994	5261394	7537535	273482	26
	16	495806	3458151	103951922	745102	23

CHAPTER 5

APPLICATIONS

Secure matrix multiplication can be used for various statistical calculations such as correlation and covariance matrices, principal component analysis, and linear regression. Assume we have fingerprint data and want to develop a fingerprint recognition system by computing correlation and covariance matrices. We can compute correlation and covariance matrices of encrypted data by using Strassen's secure block matrix multiplication algorithm securely and efficiently. Principal component analysis is generally used for image processing. Linear regression shows the relationship between independent and dependent variables. In this chapter, we compute the multiple linear regression using Strassen's secure block matrix multiplication algorithm and compare results with the standard secure block matrix method. We also explain the computation of correlation and covariance matrices using the Strassen secure block matrix multiplication algorithm.

Assume we have square data matrices X and Y and want to calculate multiple linear regression. However, the entries of the data matrices are private, so we need to encrypt them before calculating regression. Encrypted calculations can be done via homomorphic encryption, but homomorphic encryption is costly and slow, especially for large matrices. So, we propose using Strassen's secure block matrix multiplication algorithm for computing multiple linear regression of private information in the cloud.

Regression is used for the estimation of relationships between dependent (Y) and non-dependent (X) variables;

$$Y_i = \beta_0 + \beta_1 x_i + \epsilon_i, i = 1, \ldots, n$$

$$\beta = (X'X)^{-1}X'Y$$

Using the packing method and Strassen's secure block matrix multiplication algorithms, the regression can be calculated more efficiently as follows: Let X and Y be data matrices and X' be the transpose matrix of X. Here, the X matrix expresses independent, and the Y matrix expresses dependent variables. Note that X and Y must be square to apply secure matrix multiplication. If it is not, we can use padding. Then, we divide matrices into subblocks, obtain packed polynomials, and encrypt them using the FV algorithm. Following that, the data owner uploads encrypted polynomials to the cloud. In the cloud, the homomorphic multiplication of $X'X$ and $X'Y$ are calculated using the Strassen algorithm. Then the analyst downloads encrypted $X'X$ and $X'Y$. Lastly, she decrypts $X'X$ and $X'Y$, takes inverse of $X'X$, and computes $(X'X)^{-1}X'Y$. The process is shown in Figure 5.1. In this thesis, we compute the multiple linear regression using the standard secure block matrix method and Strassen's secure block matrix multiplication method and compare results in Table 5.1.

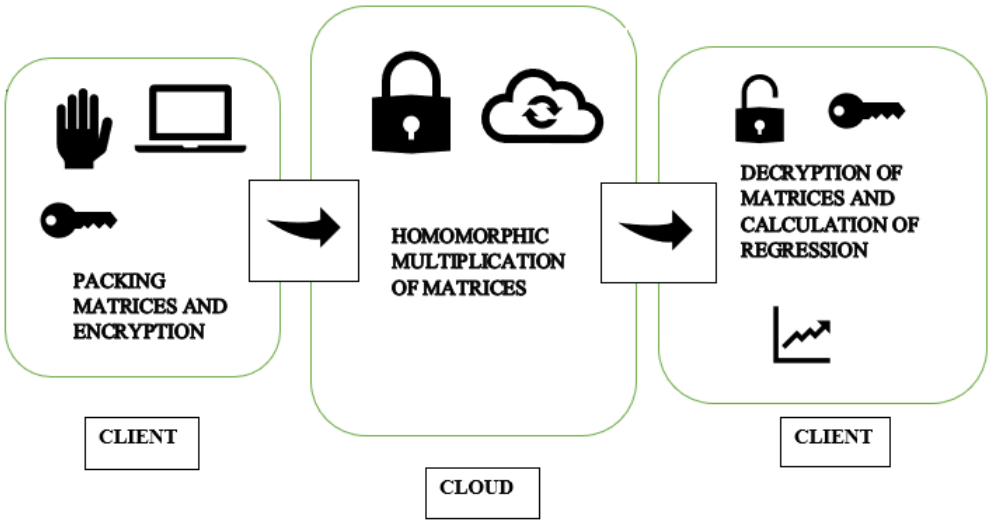

Figure 5.1: Computation of Regression of Encrypted Data

Table 5.1: Computation of the Multiple Linear Regression Using Secure Block Matrix Multiplication Algorithms

m	M	Standard SMM	Strassen's SMM	Encryption	Decryption	%
8	2	562486	487182	168345	24426	13
16	2	4386417	3500752	576529	101018	20
	4	553838	509893	147599	31801	8
32	2	34899366	25038698	2258967	376138	28
	4	4454449	3459270	578863	108157	22
	8	557789	512252	1470984	54896	8
64	2	286271943	174416203	9050268	1478617	39
	4	36019351	24915574	2383218	401679	30
	8	4293289	3408799	5973084	183900	20
	16	2470189	2210057	155043555	330506	10
96	3	286441116	176145616	9166740	1565176	38
	6	36815794	24541236	6806761	500030	33
	12	4593124	3489726	86496541	527532	24
128	2	2369341609	1240023497	36888436	10346659	47
	4	284413513	170662687	9308044	1641099	39
	8	36021962	25152327	23346880	729509	30

According to Table 5.1, when the matrix dimension is 128, and the submatrix size is two, calculating the multiple linear regression with Strassen's secure block matrix multiplication algorithm is 47% faster than the standard secure block matrix multiplication algorithm. We use 8192 as the polynomial modulus degree when the submatrix size is 16 and 4096 for other submatrix sizes. The matrix entries are random numbers up to five. Since $n \geq m^3$, the submatrix sizes can be maximum 16. Using the packing method and dividing matrices into subblocks, we can securely compute the multiple linear regression of 128×128 data matrix. If we did not use block matrix multiplication, we could not multiply such big matrices with small polynomial modulus degrees such as 4096 or 8192. We take two for the crossover point. m is main matrix and M is submatrix dimension. In the standard secure matrix multiplication part, we divide X and X' into submatrices, obtain packed polynomials, encrypt these polynomials with the BFV algorithm, and multiply homomorphically with the standard secure block matrix multiplication. So, in the standard secure block matrix multiplication column, we multiply XX' and $X'Y$ in the standard way and write time duration. Similarly, in Strassen's column, we multiply XX' and $X'Y$ in

Strassen's secure block matrix multiplication method and write the time period. In the encryption column, the packed polynomials of X, X', and Y are encrypted for both Strassen's and standard algorithms, and the total encryption time is written. In the decryption column, we compute the total time period of decryption of XX' and $X'Y$ in Strassen's and standard method.

There are other works that apply homomorphic encryption with linear regression. For example, Akavia et al. [1] presents a new protocol that allows linear regression on packed data utilizing the SIMD in the two-server. Chen and Zheng [10] also use the SEAL library for linear regression of encrypted data. However, they don't use any packing method. Stornes [33] also computes linear regression with BGV on PALASIDE. She does not use any packing or block matrix method either.

Another application of Strassen's secure block matrix multiplication is correlation and covariance matrices. Assume X is the data matrix, X^T is the transpose matrix of X, $\overline{X}$ is mean vector of each column of X, $\overline{X}^T$ is the transpose of mean vector $\overline{X}$, D is the diagonal matrix, $S_{m\times m}$ is the covariance matrix and $R_{m\times m}$ is the correlation matrix, $SSCP$ is the sum of square and cross product matrix;

$$X = \begin{bmatrix} x_{1,1} & \dots & x_{1,m} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \dots & x_{m,m} \end{bmatrix} \text{ and } \overline{X} = \begin{bmatrix} \overline{x}_1 \\ \vdots \\ \overline{x}_m \end{bmatrix}$$

The encrypted $SSCP$ matrix can be computed as follows:

$$SSCP = X \cdot X^T - m\overline{X} \cdot \overline{X}^T$$
$$S_{m\times m} = \tfrac{1}{(m-1)} \cdot SSCP$$

After calculating $SSCP$, we decrypt it and compute the covariance matrix S and the correlation matrix R.

$$D_{m\times m} = \begin{bmatrix} s_{11} & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & s_{mm} \end{bmatrix}, D^{-1/2}_{m\times m} = \begin{bmatrix} 1/\sqrt{s_{11}} & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & 1/\sqrt{s_{mm}} \end{bmatrix}$$

$$R_{m\times m} = D^{-1/2}_{m\times m} \cdot S_{m\times m} \cdot D^{-1/2}_{m\times m}$$

The $SSCP$ matrix is symmetric, and it is used for the calculation of the covariance matrix. The diagonal elements of the covariance matrix show the variance of each variable, and the off-diagonal elements show the variables' covariance. The correlation matrix, similar to the covariance matrix, reveals the relationships of the variables with each other. Unlike the covariance matrix, the size of the numbers gains importance in addition to the signs in the correlation matrix. In other words, the number values in the covariance matrix cannot be interpreted. Still, looking at the sign, whether the ratio between the variables is inverse or correct is decided. The correlation matrix gives a value between -1 and 1 for each 2-vector relationship. The two data vectors whose correlation is being tested have a strong direct correlation if this value is near 1 and a strong inverse ratio if it is close to -1. No linear link exists between the data if the correlation value is close to zero.

Assume we have biometric data such as fingerprint that has binary values in encrypted form, and we want to compute the correlation matrix for the fingerprint recognization system. To calculate the correlation matrix, we need to find the transpose of the square data matrix X. After that, we compute the means of every column and find the mean vector $\overline{X}$ and the transpose of the mean vector $\overline{X}^T$. Before packing, we divide X, X^T, $\overline{X}$ and $\overline{X}^T$ into subblocks. However, the matrices must be square for packing and dividing subblocks, so we use padding and fill with the zeroes of $\overline{X}$ and $\overline{X}^T$ to convert them into square matrices. Then, we pack X, X^T, $\overline{X}$ and $\overline{X}^T$ by using the binary packing method in (3.1), encrypt and homomorphically multiply with a homomorphic encryption algorithm and obtain $SSCP$. Since rational numbers appear when calculating matrices, the CKKS algorithm must be used for encryption and multiplication. For the calculation of the correlation and covariance matrices, we need to decrypt $SSCP$; otherwise, the number of homomorphic multiplication increases. In the paper [39], Yasuda et al. computed correlation and covariance between two variables X and Y using their secure inner product method. Here X and Y are $m-$sized vectors. Using the secure inner product, one can find the encrypted vectors' sum, mean, standard deviation, and variance. Also, in [38], Wu and Haven found the covariance of encrypted values by an encoding technique based on Smart and Vercauteren's packing method [32]. They combined the Chinese remainder theorem and batching.

CHAPTER 6

CONCLUSIONS

In this thesis, we show that using the Strassen matrix multiplication algorithm enhances the performance of secure matrix multiplication. The Strassen algorithm reduces the number of homomorphic multiplications from eight to seven when one-level recursion is used. Strassen's algorithm provides better performance results for secure matrix multiplication, even with small dimensions, since the degree of the polynomials in the homomorphic encryption decreases with the size of submatrices. The implementation results for dimensions between 8 and 128 with different submatrix sizes demonstrate significant improvements. For example, when we use the FV algorithm in Strassen's secure block matrix multiplication for 128×128 matrix, with the submatrix size of 16, the algorithm is 23% faster, while the submatrix size is two, the algorithm is 47% faster compared to the standard secure block matrix multiplication. We also use the BGV algorithm for encryption, and the best performance increase is when the submatrix size is two. However, the duration of the Strassen algorithm gets closer to the standard algorithm as the submatrix sizes increase. Also, notably, when considering a 128×128 matrix with a submatrix size of two, the algorithm is 44% faster than compared to the standard secure matrix multiplication, and for 96×96 matrix, when the submatrix size is three Strassen's secure block matrix multiplication algorithm is 42% faster than the standard secure block matrix multiplication. If we did not use block matrix multiplication, we could not calculate such big matrix dimensions with small polynomial modulus degrees such as 4096 or 8192. Overall, our implementation showcases the advantages of utilizing the Strassen algorithm for secure matrix multiplication within the homomorphic encryption framework, highlighting its potential to enhance performance, especially for larger dimensions. As

an application, we compute the multiple linear regression of encrypted data using the Strassen secure block matrix algorithm and compare results with the standard secure block matrix method. According to the results, when the matrix dimension is 128, and the submatrix size is two, computing the multiple linear regression with Strassen's secure block matrix multiplication algorithm is 47% faster than the standard secure block matrix multiplication algorithm.

REFERENCES

[1] A. Akavia, H. Shaul, M. Weiss, and Z. Yakhini, Linear-regression on packed encrypted data in the two-server model, pp. 21–32, 2019.

[2] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, Homomorphic encryption standard, Cryptology ePrint Archive, Paper 2019/939, 2019.

[3] D. Boneh, E.-J. Goh, and K. Nissim, Evaluating 2-dnf formulas on ciphertexts, in *Theory of Cryptography*, pp. 325–341, Springer Berlin Heidelberg, 2005.

[4] Z. Brakerski, Fully homomorphic encryption without modulus switching from classical gapsvp", booktitle="advances in cryptology – crypto 2012, pp. 868–886, Springer Berlin Heidelberg, 2012.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, (leveled) fully homomorphic encryption without bootstrapping, in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, p. 309–325, Association for Computing Machinery, 2012.

[6] Z. Brakerski and V. Vaikuntanathan, Efficient fully homomorphic encryption from (standard) lwe, in *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pp. 97–106, 2011.

[7] Z. Brakerski and V. Vaikuntanathan, Fully homomorphic encryption from ring-lwe and security for key dependent messages, in *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference*, volume 6841 of *Lecture Notes in Computer Science*, p. 501, Springer, 2011.

[8] A. Carey, On the explanation and implementation of three open-source fully homomorphic encryption libraries, 2020.

[9] M. Cenk and M. A. Hasan, On the arithmetic complexity of strassen-like matrix multiplications, J. Symb. Comput., 80(P2), p. 484–501, 2017.

[10] B. Chen and X. Zheng, Implementing linear regression with homomorphic encryption, Procedia Computer Science, 202, pp. 324–329, 2022.

[11] H. Chen, K. Laine, and R. Player, Simple encrypted arithmetic library - seal v2.1, pp. 3–18, 04 2017, ISBN 978-3-319-70277-3.

[12] J. H. Cheon, A. Kim, M. Kim, and Y. Song, Homomorphic encryption for arithmetic of approximate numbers, in *Advances in Cryptology – ASIACRYPT 2017*, pp. 409–437, Springer International Publishing, 2017.

[13] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds, in *Advances in Cryptology – ASIACRYPT 2016*, pp. 3–33, Springer Berlin Heidelberg, 2016.

[14] A. Costache, K. Laine, and R. Player, *Evaluating the Effectiveness of Heuristic Worst-Case Noise Analysis in FHE*, pp. 546–565, 2020.

[15] L. Ducas and D. Micciancio, Fhew: Bootstrapping homomorphic encryption in less than a second, Cryptology ePrint Archive, Paper 2014/816, 2014.

[16] D. Dung, P. Mishra, and M. Yasuda, Efficient secure matrix multiplication over lwe-based homomorphic encryption, Tatra Mountains Mathematical Publications, 67, 2016.

[17] T. ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, in G. R. Blakley and D. Chaum, editors, *Advances in Cryptology*, pp. 10–18, Springer Berlin Heidelberg, 1985.

[18] J. Fan and F. Vercauteren, Somewhat practical fully homomorphic encryption, Cryptology ePrint Archive, Report 2012/144, 2012.

[19] P. Fauzi, On fully homomorphic encryption, 2012.

[20] C. Gentry, A fully homomorphic encryption scheme, September 2009.

[21] C. Gentry, A. Sahai, and B. Waters, Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based, Cryptology ePrint Archive, Paper 2013/340, 2013.

[22] S. Halevi and V. Shoup, Design and implementation of helib: a homomorphic encryption library, Cryptology ePrint Archive, Paper 2020/1481, 2020.

[23] A. Lopez-Alt, E. Tromer, and V. Vaikuntanathan, On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption, Cryptology ePrint Archive, Paper 2013/094, 2013.

[24] V. Lyubashevsky, C. Peikert, and O. Regev, On ideal lattices and learning with errors over rings, in *Advances in Cryptology – EUROCRYPT 2010*, pp. 1–23, Springer Berlin Heidelberg, 2010.

[25] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. Fitzek, and N. Aaraj, Survey on fully homomorphic encryption, theory, and applications, Cryptology ePrint Archive, Paper 2022/1602, 2022.

[26] 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*, pp. 320–330, Springer International Publishing, Cham, 2017.

[27] P. K. Mishra, D. Rathee, D. H. Duong, and M. Yasuda, Fast secure matrix multiplications over ring-based homomorphic encryption, Information Security Journal: A Global Perspective, 30(4), pp. 219–234, 2021.

[28] M. Naehrig, K. Lauter, and V. Vaikuntanathan, Can homomorphic encryption be practical?, in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, p. 113–124, Association for Computing Machinery, New York, NY, USA, 2011.

[29] P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in *Advances in Cryptology — EUROCRYPT '99*, pp. 223–238, Springer Berlin Heidelberg, 1999.

[30] O. Regev, On lattices, learning with errors, random linear codes, and cryptography, in *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, p. 84–93, Association for Computing Machinery, 2005.

[31] R. L. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Commun. ACM, 21(2), p. 120–126, 1978.

[32] N. P. Smart and F. Vercauteren, Fully homomorphic encryption with relatively small key and ciphertext sizes, in *Public Key Cryptography – PKC 2010*, pp. 420–443, Springer Berlin Heidelberg, 2010.

[33] A. M. Stornes, Cryptographically private linear regression, 2021.

[34] V. Strassen, Gaussian elimination is not optimal, Numerische Mathematik, 13, pp. 354–356, 1969.

[35] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, Fully homomorphic encryption over the integers, Cryptology ePrint Archive, Paper 2009/616, 2009.

[36] S. Wang and H. Huang, Secure outsourced computation of multiple matrix multiplication based on fully homomorphic encryption, KSII Transactions on Internet and Information Systems, 13(11), pp. 5616–5630, 2019.

[37] S. Winograd, On multiplication of 22 matrices, Linear Algebra and its Applications, pp. 381–388, 1971, ISSN 0024-3795.

[38] D. J. Wu, Using homomorphic encryption for large scale statistical analysis, 2012.

[39] M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba, New packing method in somewhat homomorphic encryption and its applications, Security and Communication Networks, 8(13), pp. 2194–2213, 2015.

[40] M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba, Secure statistical analysis using rlwe-based homomorphic encryption, 2015.

[41] G. W. R. Yuriy Polyakov, Kurt Rohloff and D. Cousins, Palisade lattice cryptography library user manual (v1.11.8), 2022.

APPENDIX A

SOURCE CODE OF STRASSEN'S SECURE BLOCK MATRIX MULTIPLICATION ALGORITHM AND THE MULTIPLE LINEAR REGRESSION OF ENCRYPTED DATA

```
// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT license.

#include <Eigen/Dense>
#include <chrono>
#include <regex>
#include "examples.h"
#ifdef _WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif


#define DD_RUN_STRASSEM_MULT_TEST 1
#define DD_RUN_REGRESSION 0

using namespace std;
using namespace seal;


using dd_unit = int64_t;
using dd_unit_float = double;

constexpr int DD_MAX_RANDOM_NUMBER = 4;

using dd_inside_matrix = vector<dd_unit>;
typedef vector<dd_inside_matrix> dd_matrix;
typedef vector<dd_matrix> dd_sub_inside_matrices;
typedef vector<dd_sub_inside_matrices> dd_sub_matrices;

using dd_inside_matrix_f = vector<dd_unit_float>;
```

```
33  typedef vector<dd_inside_matrix_f> dd_matrix_f;
34
35  typedef vector<Ciphertext> dd_inside_chipher;
36  typedef vector<dd_inside_chipher> dd_chipher_matrix;
37
38  typedef vector<dd_unit> dd_polynom;
39  typedef vector<dd_polynom> dd_polynomial_inside_matrix;
40  typedef vector<dd_polynomial_inside_matrix> dd_polynomial_matrix;
41
42  long long fib(int f);
43  void warmup(int time);
44  void getMultResult(dd_polynomial_matrix& polinomial, dd_matrix&
    ↪  matrix, int div, int i_dim, int dim, int n);
45  void print_col_m(dd_matrix_f m, int d);
46  void print_col_m(dd_matrix m, int d);
47  dd_matrix_f toFloatMatrix(const dd_matrix& matrix);
48  Eigen::MatrixXd toEigen(const dd_matrix& matrix);
49  dd_matrix_f toFtMatrix(const Eigen::MatrixXd& eigenMatrix);
50  dd_matrix_f inverseMatrix(const dd_matrix& matrix);
51
52  dd_matrix multiply_m(const dd_matrix& matrix, dd_unit constant);
53  dd_matrix_f multiply_m(const dd_matrix_f& A, const dd_matrix_f& B);
54  dd_matrix_f multiply_m(const dd_matrix_f& matrix, dd_unit_float
    ↪  constant);
55  dd_matrix multiply_m(const dd_matrix& A, const dd_matrix& B);
56  dd_matrix transpose_square(dd_matrix matrix);
57
58  void strassen_splitt(dd_chipher_matrix& A, dd_chipher_matrix& B,
    ↪  size_t roww, size_t coll, size_t d);
59  void strassen_joinn(dd_chipher_matrix& A, dd_chipher_matrix& B,
    ↪  size_t roww, size_t coll, size_t d);
60  void strassen_addd(Evaluator& evaluator, dd_chipher_matrix& A,
    ↪  dd_chipher_matrix& B, dd_chipher_matrix& C, size_t d);
61  void strassen_subtractt(
62      Evaluator& evaluator, dd_chipher_matrix& A, dd_chipher_matrix&
    ↪  B, dd_chipher_matrix& C, size_t d);
63  void strassen_ciph_matrix_mult(
64      Evaluator& evaluator, RelinKeys relin_keys, dd_chipher_matrix&
    ↪  A, dd_chipher_matrix& B, dd_chipher_matrix& C,
65      size_t dim, int crossover);
66
67  void seperate_line();
68  void print_m(dd_matrix m, int d);
69  void print_m(dd_matrix_f m, int d);
70  void print_v(dd_inside_matrix v, int d);
71  void set_matrix(dd_matrix& m, int d);
72  void set_col_matrix(dd_matrix& matrix, int n);
73  void split(dd_matrix& A, dd_matrix& B, int roww, int coll, int d);
74  void sub_matrix(dd_matrix& A, dd_matrix& out_matrix, int row, int
    ↪  col, int div);
```

```
75
76  void print_polynom(dd_polynom& p1, size_t dim);
77  void mult_polynom(dd_polynom& p1, dd_polynom& p2, dd_polynom& pout,
    ↪  size_t dim);
78  void generate_polynom(dd_matrix& matrix, dd_polynom& p_out, size_t
    ↪  poly_dim, size_t dim);
79  void generate_polynom(dd_matrix& matrix, dd_polynom& p_out, size_t
    ↪  poly_dim, size_t dim, int n);
80  void add_polynom(dd_polynom& p1, dd_polynom& p2, size_t dim);
81
82  void normal_multiply(dd_matrix& A, dd_matrix& B, dd_matrix& C, int
    ↪  d);
83
84  void standart_ciph_matrix_mult(
85      Evaluator& evaluator, RelinKeys relin_keys, dd_chipher_matrix&
    ↪  chipher_matrix_A,
86      dd_chipher_matrix& chipher_matrix_B, dd_chipher_matrix&
    ↪  chipher_matrix_out, int dim);
87
88  vector<string> dd_split_str(const string& str, const string& delim);
89  void dd_encode(dd_polynom& p, size_t dim, std::string& str);
90  void dd_decode(std::string& text, dd_polynom& pout);
91
92  void create_chipher_matrix(
93      Encryptor& encryptor, dd_sub_matrices& sub_matrices,
    ↪  dd_chipher_matrix& chipher_out, size_t poly_dim, size_t i_dim,
94      size_t div);
95  void create_chipher_matrix(
96      Encryptor& encryptor, dd_sub_matrices& sub_matrices,
    ↪  dd_chipher_matrix& chipher_out, size_t poly_dim, size_t i_dim,
97      size_t div, int n);
98
99  void create_polynomial_result(
100     Decryptor& decryptor, dd_chipher_matrix& chipher_matrix,
    ↪  dd_polynomial_matrix& out_poly_matrix, size_t poly_dim,
101     size_t div);
102
103 void dd_create_sub_result(dd_polynom& polynom, dd_matrix&
    ↪  out_matrix, size_t matrix_dim, int n);
104
105 void example_bfv_basics(
106     int main_matrix_dim, int chipher_matrix_dim, size_t
    ↪  poly_modulus_degree_dd, int crossover_dd, int warmupTime)
107 {
108 #if DD_RUN_REGRESSION
109     {
110         dd_unit dim = main_matrix_dim;
111         dd_inside_matrix inside(dim);
112
113         dd_matrix mA(dim, inside);
```

```
114         dd_matrix my(dim, inside);
115 
116         set_matrix(mA, dim);
117         set_col_matrix(my, dim);
118         auto mA_T = transpose_square(mA);
119 
120         std::cout << "X: \n";
121         print_m(mA, dim);
122         std::cout << "X': \n";
123         print_m(mA_T, dim);
124         std::cout << "y: \n";
125         print_m(my, dim);
126 
127         auto xtrans_x = multiply_m(mA_T, mA);
128         std::cout << "X'X: \n";
129         print_m(xtrans_x, dim);
130 
131         auto xtrans_y = multiply_m(mA_T, my);
132         std::cout << "X'y: \n";
133         print_col_m(xtrans_y, dim);
134 
135         auto xtrans_x_inv = inverseMatrix(xtrans_x);
136         // std::cout << "(X'X)^-1: \n";
137         // print_m(xtrans_x_inv, dim);
138 
139         auto xtrans_y_f = toFloatMatrix(xtrans_y);
140         // std::cout << "xtrans_y_f: \n";
141         // print_m(xtrans_y_f, dim);
142 
143         auto result = multiply_m(xtrans_x_inv, xtrans_y_f);
144         std::cout << "result: \n";
145         print_col_m(result, dim);
146 
147         std::cout << "================= warming up
    ↪  ================== \n";
148         warmup(warmupTime);
149         std::cout << "=================  finished
    ↪  ================== \n";
150 
151         int div = chipher_matrix_dim;
152         int i_dim = dim / div;
153         int n = i_dim * i_dim * i_dim;
154         int poly_dim = n * n;
155         size_t poly_modulus_degree = poly_modulus_degree_dd;
156         int crossover = crossover_dd;
157         dd_inside_matrix inside_dim(i_dim);
158 
159         dd_matrix inside_matix(i_dim, inside_dim);
160         dd_sub_inside_matrices sub_inside_matrices(div,
    ↪  inside_matix);
```

```
161         dd_sub_matrices sub_matrices_A(div, sub_inside_matrices);
162         dd_sub_matrices sub_matrices_Trans(div,
    ↪  sub_inside_matrices);
163         dd_sub_matrices sub_matrices_y(div, sub_inside_matrices);
164 
165 
166         for (int m = 0; m < div; ++m)
167             for (int n = 0; n < div; ++n)
168                 sub_matrix(mA, sub_matrices_A[m][n], i_dim + i_dim *
    ↪  m, i_dim + i_dim * n, i_dim);
169         for (int m = 0; m < div; ++m)
170             for (int n = 0; n < div; ++n)
171                 sub_matrix(mA_T, sub_matrices_Trans[m][n], i_dim +
    ↪  i_dim * m, i_dim + i_dim * n, i_dim);
172         for (int m = 0; m < div; ++m)
173             for (int n = 0; n < div; ++n)
174                 sub_matrix(my, sub_matrices_y[m][n], i_dim + i_dim *
    ↪  m, i_dim + i_dim * n, i_dim);
175 
176         EncryptionParameters parms(scheme_type::bfv);
177         parms.set_poly_modulus_degree(poly_modulus_degree);
178 
    ↪  parms.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree));
179 
    ↪  parms.set_plain_modulus(PlainModulus::Batching(poly_modulus_degree,
    ↪  20));
180         SEALContext context(parms);
181         KeyGenerator keygen(context);
182         SecretKey secret_key = keygen.secret_key();
183         PublicKey public_key;
184         keygen.create_public_key(public_key);
185         RelinKeys relin_keys;
186         keygen.create_relin_keys(relin_keys);
187         Encryptor encryptor(context, public_key);
188         Evaluator evaluator(context);
189         Decryptor decryptor(context, secret_key);
190 
191         dd_inside_chipher inside_chipher(div);
192 
193         dd_chipher_matrix chipher_matrix_A(div, inside_chipher);
194         dd_chipher_matrix chipher_matrix_Trans(div, inside_chipher);
195         dd_chipher_matrix chipher_matrix_y(div, inside_chipher);
196 
197         dd_chipher_matrix sn_chipher_matrix_A(div, inside_chipher);
198         dd_chipher_matrix sn_chipher_matrix_Trans(div,
    ↪  inside_chipher);
199         dd_chipher_matrix sn_chipher_matrix_y(div, inside_chipher);
200 
201         dd_chipher_matrix chipher_matrix_standart_ATransA(div,
    ↪  inside_chipher);
```

```
202         dd_chipher_matrix chipher_matrix_standart_ATrans_y(div,
    ↪ inside_chipher);
203 
204         dd_chipher_matrix chipher_matrix_strassen_ATransA(div,
    ↪ inside_chipher);
205         dd_chipher_matrix chipher_matrix_strassen_ATrans_y(div,
    ↪ inside_chipher);
206 
207         std::cout << "\n\nEncryption ";
208         chrono::high_resolution_clock::time_point bas3 =
    ↪ chrono::high_resolution_clock::now();
209         create_chipher_matrix(encryptor, sub_matrices_Trans,
    ↪ chipher_matrix_Trans, poly_dim, i_dim, div);
210         create_chipher_matrix(encryptor, sub_matrices_A,
    ↪ chipher_matrix_A, poly_dim, i_dim, div, n);
211         create_chipher_matrix(encryptor, sub_matrices_y,
    ↪ chipher_matrix_y, poly_dim, i_dim, div, n);
212 
213         create_chipher_matrix(encryptor, sub_matrices_Trans,
    ↪ sn_chipher_matrix_Trans, poly_dim, i_dim, div);
214         create_chipher_matrix(encryptor, sub_matrices_A,
    ↪ sn_chipher_matrix_A, poly_dim, i_dim, div, n);
215         create_chipher_matrix(encryptor, sub_matrices_y,
    ↪ sn_chipher_matrix_y, poly_dim, i_dim, div, n);
216         chrono::high_resolution_clock::time_point son3 =
    ↪ chrono::high_resolution_clock::now();
217         auto duration3 =
    ↪ chrono::duration_cast<chrono::microseconds>(son3 -
    ↪ bas3).count();
218         std::cout << "took: " << duration3 << " microseconds." <<
    ↪ std::endl;
219         {
220             std::cout << "\nStandart multiplication ";
221             chrono::high_resolution_clock::time_point bas1 =
    ↪ chrono::high_resolution_clock::now();
222             standart_ciph_matrix_mult(
223                 evaluator, relin_keys, chipher_matrix_Trans,
    ↪ chipher_matrix_A, chipher_matrix_standart_ATransA, div);
224             standart_ciph_matrix_mult(
225                 evaluator, relin_keys, chipher_matrix_Trans,
    ↪ chipher_matrix_y, chipher_matrix_standart_ATrans_y, div);
226             chrono::high_resolution_clock::time_point son1 =
    ↪ chrono::high_resolution_clock::now();
227             auto duration1 =
    ↪ chrono::duration_cast<chrono::microseconds>(son1 -
    ↪ bas1).count();
228             std::cout << "took: " << duration1 << " microseconds."
    ↪ << std::endl;
229         }
230         {
```

```
231             std::cout << "\Strassen multiplication ";
232             chrono::high_resolution_clock::time_point bas1 =
    ↪  chrono::high_resolution_clock::now();
233             strassen_ciph_matrix_mult(
234                 evaluator, relin_keys, sn_chipher_matrix_Trans,
    ↪  sn_chipher_matrix_A, chipher_matrix_strassen_ATransA,
235                 div, crossover);
236             strassen_ciph_matrix_mult(
237                 evaluator, relin_keys, sn_chipher_matrix_Trans,
    ↪  sn_chipher_matrix_y, chipher_matrix_strassen_ATrans_y,
238                 div, crossover);
239             chrono::high_resolution_clock::time_point son1 =
    ↪  chrono::high_resolution_clock::now();
240             auto duration1 =
    ↪  chrono::duration_cast<chrono::microseconds>(son1 -
    ↪  bas1).count();
241             std::cout << "took: " << duration1 << " microseconds."
    ↪  << std::endl;
242         }
243
244         dd_polynom inside_polynom(poly_dim);
245         dd_polynomial_inside_matrix p_inside_matrix(div,
    ↪  inside_polynom);
246
247         dd_polynomial_matrix p_standart_ATransA(div,
    ↪  p_inside_matrix);
248         dd_polynomial_matrix p_strassen_ATransA(div,
    ↪  p_inside_matrix);
249
250         dd_polynomial_matrix p_standart_ATrans_y(div,
    ↪  p_inside_matrix);
251         dd_polynomial_matrix p_strassen_ATrans_y(div,
    ↪  p_inside_matrix);
252
253         std::cout << "\nDecryption ";
254         chrono::high_resolution_clock::time_point bas4 =
    ↪  chrono::high_resolution_clock::now();
255         create_polynomial_result(
256             decryptor, chipher_matrix_standart_ATransA,
    ↪  p_standart_ATransA, poly_modulus_degree, div);
257         create_polynomial_result(
258             decryptor, chipher_matrix_strassen_ATransA,
    ↪  p_strassen_ATransA, poly_modulus_degree, div);
259
260         create_polynomial_result(
261             decryptor, chipher_matrix_standart_ATrans_y,
    ↪  p_standart_ATrans_y, poly_modulus_degree, div);
262         create_polynomial_result(
263             decryptor, chipher_matrix_strassen_ATrans_y,
    ↪  p_strassen_ATrans_y, poly_modulus_degree, div);
```

```
264         chrono::high_resolution_clock::time_point son4 =
    ↪ chrono::high_resolution_clock::now();
265         auto duration4 =
    ↪ chrono::duration_cast<chrono::microseconds>(son4 -
    ↪ bas4).count();
266         std::cout << "took: " << duration4 << " microseconds.\n\n"
    ↪ << std::endl;
267
268         dd_matrix standart_mult_result_ATransA(dim, inside);
269         dd_matrix strassen_mult_result_ATransA(dim, inside);
270
271         dd_matrix standart_mult_result_ATrans_y(dim, inside);
272         dd_matrix strassen_mult_result_ATrans_y(dim, inside);
273
274         getMultResult(p_standart_ATransA,
    ↪ standart_mult_result_ATransA, div, i_dim, dim, n);
275         getMultResult(p_strassen_ATransA,
    ↪ strassen_mult_result_ATransA, div, i_dim, dim, n);
276
277         getMultResult(p_standart_ATrans_y,
    ↪ standart_mult_result_ATrans_y, div, i_dim, dim, n);
278         getMultResult(p_strassen_ATrans_y,
    ↪ strassen_mult_result_ATrans_y, div, i_dim, dim, n);
279
280         std::cout << "standart_mult_result_X'X: \n";
281         print_m(standart_mult_result_ATransA, dim);
282
283         std::cout << "strassen_mult_result_X'X: \n";
284         print_m(strassen_mult_result_ATransA, dim);
285
286         std::cout << "standart_mult_result_X'y: \n";
287         print_col_m(standart_mult_result_ATrans_y, dim);
288
289         std::cout << "strassen_mult_result_X'y: \n";
290         print_col_m(strassen_mult_result_ATrans_y, dim);
291
292         auto standart_mult_result_ATransA_inv =
    ↪ inverseMatrix(standart_mult_result_ATransA);
293         // std::cout << "(X'X)^-1: \n";
294         // print_m(xtrans_x_inv, dim);
295
296         auto standart_mult_result_ATrans_y_f =
    ↪ toFloatMatrix(standart_mult_result_ATrans_y);
297         // std::cout << "xtrans_y_f: \n";
298         // print_m(xtrans_y_f, dim);
299
300         auto result_enc =
    ↪ multiply_m(standart_mult_result_ATransA_inv,
    ↪ standart_mult_result_ATrans_y_f);
301         std::cout << "result_enc: \n";
```

```
302             print_col_m(result_enc, dim);
303             std::cout << "result: \n";
304             print_col_m(result, dim);
305         }
306     #endif // FT_RUN_REGRESSION
307
308     #if DD_RUN_STRASSEM_MULT_TEST
309         {
310             int dim = main_matrix_dim;
311             int div = chipher_matrix_dim;
312             int i_dim = dim / div;
313
314             int n = i_dim * i_dim * i_dim;
315             int poly_dim = n * n;
316
317             size_t poly_modulus_degree = poly_modulus_degree_dd;
318             int crossover = crossover_dd;
319
320             dd_inside_matrix inside(dim);
321
322             dd_inside_matrix inside_dim(i_dim);
323
324             dd_matrix mA(dim, inside);
325             dd_matrix mB(dim, inside);
326             dd_matrix mC(dim, inside);
327
328             set_matrix(mA, dim);
329             Sleep(1000);
330             set_matrix(mB, dim);
331
332             // mA[0][0] = 1;
333             // mA[1][0] = 2;
334             // mA[0][1] = 2;
335             // mA[1][1] = 3;
336
337             // mB[0][0] = 2;
338             // mB[1][0] = 1;
339             // mB[0][1] = 1;
340             // mB[1][1] = 3;
341
342             normal_multiply(mA, mB, mC, dim);
343
344             print_m(mA, dim);
345             print_m(mB, dim);
346             print_m(mC, dim);
347
348             dd_matrix inside_matix(i_dim, inside_dim);
349             dd_sub_inside_matrices sub_inside_matrices(div,
    ↪  inside_matix);
350             dd_sub_matrices sub_matrices_A(div, sub_inside_matrices);
```

```
351         dd_sub_matrices sub_matrices_B(div, sub_inside_matrices);
352         for (int i = 0; i < div; ++i)
353             for (int k = 0; k < div; ++k)
354                 sub_matrix(mA, sub_matrices_A[i][k], i_dim + i_dim *
    ↪  i, i_dim + i_dim * k, i_dim);
355         for (int i = 0; i < div; ++i)
356             for (int k = 0; k < div; ++k)
357                 sub_matrix(mB, sub_matrices_B[i][k], i_dim + i_dim *
    ↪  i, i_dim + i_dim * k, i_dim);
358 
359         EncryptionParameters parms(scheme_type::bfv);
360         parms.set_poly_modulus_degree(poly_modulus_degree);
361 
    ↪  parms.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree));
362 
    ↪  parms.set_plain_modulus(PlainModulus::Batching(poly_modulus_degree,
    ↪  20));
363         SEALContext context(parms);
364         KeyGenerator keygen(context);
365         SecretKey secret_key = keygen.secret_key();
366         PublicKey public_key;
367         keygen.create_public_key(public_key);
368         RelinKeys relin_keys;
369         keygen.create_relin_keys(relin_keys);
370         Encryptor encryptor(context, public_key);
371         Evaluator evaluator(context);
372         Decryptor decryptor(context, secret_key);
373 
374         dd_inside_chipher inside_chipher(div);
375         dd_chipher_matrix chipher_matrix_A(div, inside_chipher);
376         dd_chipher_matrix chipher_matrix_B(div, inside_chipher);
377         dd_chipher_matrix chipher_matrix_standart_Out(div,
    ↪  inside_chipher);
378         dd_chipher_matrix chipher_matrix_strassen_Out(div,
    ↪  inside_chipher);
379 
380         std::cout << "\n\nEncryption ";
381         chrono::high_resolution_clock::time_point start3 =
    ↪  chrono::high_resolution_clock::now();
382         create_chipher_matrix(encryptor, sub_matrices_A,
    ↪  chipher_matrix_A, poly_dim, i_dim, div);
383         create_chipher_matrix(encryptor, sub_matrices_B,
    ↪  chipher_matrix_B, poly_dim, i_dim, div, n);
384         chrono::high_resolution_clock::time_point end3 =
    ↪  chrono::high_resolution_clock::now();
385         auto duration3 =
    ↪  chrono::duration_cast<chrono::microseconds>(end3 -
    ↪  start3).count();
386         std::cout << "took: " << duration3 << " microseconds." <<
    ↪  std::endl;
```

```
387
388         std::cout << "\nStandart multiplication ";
389         chrono::high_resolution_clock::time_point bas1 =
    ↪ chrono::high_resolution_clock::now();
390         standart_ciph_matrix_mult(
391             evaluator, relin_keys, chipher_matrix_A,
    ↪ chipher_matrix_B, chipher_matrix_standart_Out, div);
392         chrono::high_resolution_clock::time_point son1 =
    ↪ chrono::high_resolution_clock::now();
393         auto duration1 =
    ↪ chrono::duration_cast<chrono::microseconds>(son1 -
    ↪ bas1).count();
394         std::cout << "took: " << duration1 << " microseconds." <<
    ↪ std::endl;
395
396         std::cout << "\nStrassen multiplication ";
397         chrono::high_resolution_clock::time_point bas2 =
    ↪ chrono::high_resolution_clock::now();
398         strassen_ciph_matrix_mult(
399             evaluator, relin_keys, chipher_matrix_A,
    ↪ chipher_matrix_B, chipher_matrix_strassen_Out, div, crossover);
400         chrono::high_resolution_clock::time_point son2 =
    ↪ chrono::high_resolution_clock::now();
401         auto duration2 =
    ↪ chrono::duration_cast<chrono::microseconds>(son2 -
    ↪ bas2).count();
402         std::cout << "took: " << duration2 << " microseconds." <<
    ↪ std::endl;
403
404         dd_polynom inside_polynom(poly_dim);
405         dd_polynomial_inside_matrix p_inside_matrix(div,
    ↪ inside_polynom);
406         dd_polynomial_matrix p_standart_matrix(div,
    ↪ p_inside_matrix);
407         dd_polynomial_matrix p_strassen_matrix(div,
    ↪ p_inside_matrix);
408
409         std::cout << "\nDecryption ";
410         chrono::high_resolution_clock::time_point start4 =
    ↪ chrono::high_resolution_clock::now();
411         create_polynomial_result(decryptor,
    ↪ chipher_matrix_standart_Out, p_standart_matrix,
    ↪ poly_modulus_degree, div);
412         chrono::high_resolution_clock::time_point end4 =
    ↪ chrono::high_resolution_clock::now();
413         auto duration4 =
    ↪ chrono::duration_cast<chrono::microseconds>(end4 -
    ↪ start4).count();
414         std::cout << "took: " << duration4 << " microseconds.\n\n"
    ↪ << std::endl;
```

```
415         create_polynomial_result(decryptor,
    ↪  chipher_matrix_strassen_Out, p_strassen_matrix,
    ↪  poly_modulus_degree, div);
416
417         dd_matrix standart_mult_result(dim, inside);
418         dd_matrix strassen_mult_result(dim, inside);
419
420         for (int o = 0; o < div; ++o)
421             for (int k = 0; k < div; ++k)
422             {
423                 dd_matrix standart_m(i_dim, inside);
424                 dd_matrix strassen_m(i_dim, inside);
425
426                 dd_create_sub_result(p_standart_matrix[o][k],
    ↪  standart_m, i_dim, n);
427                 dd_create_sub_result(p_strassen_matrix[o][k],
    ↪  strassen_m, i_dim, n);
428                 // print_m(mResult, i_dim);
429                 for (size_t r = 0; r < i_dim; r++)
430                     for (size_t p = 0; p < i_dim; p++)
431                     {
432                         standart_mult_result[o * i_dim + r][k *
    ↪  i_dim + p] = standart_m[r][p];
433                         strassen_mult_result[o * i_dim + r][k *
    ↪  i_dim + p] = strassen_m[r][p];
434                     }
435             }
436
437         print_m(standart_mult_result, dim);
438         print_m(strassen_mult_result, dim);
439     }
440 #endif // FT_RUN_STRASSEM_MULT_TEST
441 }
442
443
444 void getMultResult(dd_polynomial_matrix& polinomial, dd_matrix&
    ↪  matrix, int div, int i_dim, int dim, int n)
445 {
446     dd_inside_matrix inside(dim);
447     for (int u = 0; u < div; ++u)
448         for (int t = 0; t < div; ++t)
449         {
450             dd_matrix standart_m(i_dim, inside);
451             dd_matrix strassen_m(i_dim, inside);
452
453             dd_create_sub_result(polinomial[u][t], standart_m,
    ↪  i_dim, n);
454             for (size_t c = 0; c < i_dim; c++)
455                 for (size_t p = 0; p < i_dim; p++)
456                 {
```

```
                    matrix[u * i_dim + c][t * i_dim + p] =
    ↪  standart_m[c][p];
                }
            }
}

long long fib(int f)
{
    if (f <= 1)
        return f;
    else
        return fib(f - 1) + fib(f - 2);
}

void warmup(int time)
{
    auto bas = std::chrono::high_resolution_clock::now();
    auto son = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> elapsed = son - bas;

    // This will run for approximately 5 minutes
    while (elapsed.count() < time)
    {
        fib(30);

        son = std::chrono::high_resolution_clock::now();
        elapsed = son - bas;
    }
}

dd_matrix_f inverseMatrix(const dd_matrix& matrix)
{
    // convert dd_matrix to Eigen::MatrixXd
    Eigen::MatrixXd eigenMatrix = toEigen(matrix);

    // compute the inverse
    Eigen::MatrixXd eigenMatrixInv = eigenMatrix.inverse();

    // convert Eigen::MatrixXd back to dd_matrix
    auto invMatrix = toFtMatrix(eigenMatrixInv);

    return invMatrix;
}

Eigen::MatrixXd toEigen(const dd_matrix& matrix)
{
    int n = matrix.size();

    Eigen::MatrixXd eigenMatrix(n, n);
    for (int s = 0; s < n; s++)
```

```
506             for (int e = 0; e < n; e++)
507                 eigenMatrix(s, e) = static_cast<double>(matrix[s][e]);
508 
509     return eigenMatrix;
510 }
511 
512 dd_matrix_f toFtMatrix(const Eigen::MatrixXd& eigenMatrix)
513 {
514     int n = eigenMatrix.rows();
515 
516     dd_matrix_f matrix(n, std::vector<dd_unit_float>(n));
517     for (int f = 0; f < n; f++)
518         for (int g = 0; g < n; g++)
519             matrix[f][g] = static_cast<dd_unit_float>(eigenMatrix(f,
    ↪  g));
520 
521     return matrix;
522 }
523 
524 dd_matrix_f toFloatMatrix(const dd_matrix& matrix)
525 {
526     auto n = matrix.size();
527 
528     dd_matrix_f matrix_f(n, std::vector<dd_unit_float>(n));
529     for (int u = 0; u < n; u++)
530         for (int r = 0; r < n; r++)
531             matrix_f[u][r] =
    ↪  static_cast<dd_unit_float>(matrix[u][r]);
532 
533     return matrix_f;
534 }
535 
536 
537 dd_matrix multiply_m(const dd_matrix& matrix, dd_unit constant)
538 {
539     size_t rowws = matrix.size();
540     size_t colls = matrix[0].size();
541 
542     dd_matrix result(rowws, dd_inside_matrix(colls));
543 
544     for (size_t z = 0; z < rowws; ++z)
545     {
546         for (size_t y = 0; y < colls; ++y)
547         {
548             result[z][y] = matrix[z][y] * constant;
549         }
550     }
551 
552     return result;
553 }
```

```
554
555 dd_matrix_f multiply_m(const dd_matrix_f& matrix, dd_unit_float
    ↪ constant)
556 {
557     size_t rowws = matrix.size();
558     size_t colls = matrix[0].size();
559
560     dd_matrix_f result(rowws, dd_inside_matrix_f(colls));
561
562     for (size_t p = 0; p < rowws; ++p)
563     {
564         for (size_t d = 0; d < colls; ++d)
565         {
566             result[p][d] = matrix[p][d] * constant;
567         }
568     }
569
570     return result;
571 }
572
573 dd_matrix multiply_m(const dd_matrix& A, const dd_matrix& B)
574 {
575     if (A[0].size() != B.size())
576     {
577         throw std::invalid_argument("Number of columns in A should
    ↪ be equal to number of rows in B");
578     }
579
580     size_t A_rowws = A.size();
581     size_t A_colls = A[0].size();
582     size_t B_colls = B[0].size();
583
584     // Initialize the result matrix with zeros
585     dd_matrix result(A_rowws, dd_inside_matrix(B_colls, 0));
586
587     for (size_t u = 0; u < A_rowws; ++u)
588     {
589         for (size_t v = 0; v < B_colls; ++v)
590         {
591             for (size_t y = 0; y < A_colls; ++y)
592             {
593                 result[u][v] += A[u][y] * B[y][v];
594             }
595         }
596     }
597
598     return result;
599 }
600
601 dd_matrix_f multiply_m(const dd_matrix_f& A, const dd_matrix_f& B)
```

```
602 {
603     if (A[0].size() != B.size())
604     {
605         throw std::invalid_argument("Number of columns in A should
    ↪  be equal to number of rows in B");
606     }
607 
608     size_t A_rowws = A.size();
609     size_t A_colls = A[0].size();
610     size_t B_colls = B[0].size();
611 
612     // Initialize the result matrix with zeros
613     dd_matrix_f result(A_rowws, dd_inside_matrix_f(B_colls, 0));
614 
615     for (size_t a = 0; a < A_rowws; ++a)
616     {
617         for (size_t b = 0; b < B_colls; ++b)
618         {
619             for (size_t c = 0; c < A_colls; ++c)
620             {
621                 result[a][b] += A[a][c] * B[c][b];
622             }
623         }
624     }
625 
626     return result;
627 }
628 
629 dd_matrix transpose_square(dd_matrix matrix)
630 {
631     size_t n = matrix.size();
632     dd_matrix transpose(n, dd_inside_matrix(n));
633 
634     for (size_t u = 0; u < n; ++u)
635     {
636         for (size_t v = 0; v < n; ++v)
637         {
638             transpose[v][u] = matrix[u][v];
639         }
640     }
641 
642     return transpose;
643 }
644 
645 void strassen_splitt(dd_chipher_matrix& A, dd_chipher_matrix& B,
    ↪  size_t roww, size_t coll, size_t d)
646 {
647     for (size_t u1 = 0, u2 = roww; u1 < d; u1++, u2++)
648         for (size_t v1 = 0, v2 = coll; v1 < d; v1++, v2++)
649             B[u1][v1] = A[u2][v2];
```

```
650 }
651
652 void strassen_joinn(dd_chipher_matrix& A, dd_chipher_matrix& B,
    ↪ size_t roww, size_t coll, size_t d)
653 {
654     for (int p1 = 0, p2 = roww; p1 < d; p1++, p2++)
655         for (int r1 = 0, r2 = coll; r1 < d; r1++, r2++)
656             B[p2][r2] = A[p1][r1];
657 }
658
659 void strassen_addd(Evaluator& evaluator, dd_chipher_matrix& A,
    ↪ dd_chipher_matrix& B, dd_chipher_matrix& C, size_t d)
660 {
661     for (int r = 0; r < d; r++)
662         for (int p = 0; p < d; p++)
663         {
664             Ciphertext result_matrix;
665             evaluator.add(A[r][p], B[r][p], result_matrix);
666
667             C[r][p] = result_matrix;
668         }
669 }
670
671 void strassen_subtractt(Evaluator& evaluator, dd_chipher_matrix& A,
    ↪ dd_chipher_matrix& B, dd_chipher_matrix& C, size_t d)
672 {
673     for (int i = 0; i < d; i++)
674         for (int j = 0; j < d; j++)
675         {
676             Ciphertext result_matrix;
677             evaluator.sub(A[i][j], B[i][j], result_matrix);
678
679             C[i][j] = result_matrix;
680         }
681 }
682
683 void strassen_ciph_matrix_mult(
684     Evaluator& evaluator, RelinKeys relin_keys, dd_chipher_matrix&
    ↪ A, dd_chipher_matrix& B, dd_chipher_matrix& C,
685     size_t dim, int crossover)
686 {
687     // if the dimension is not divisible by two, add padding
688     if (dim % 2 != 0)
689     {
690         dim++;
691         A.resize(dim);
692         B.resize(dim);
693         C.resize(dim);
694
695         // loop to resize the inside of the matrices
```

```
696         for (int i = 0; i < dim; i++)
697         {
698             A[i].resize(dim);
699             B[i].resize(dim);
700             C[i].resize(dim);
701         }
702     }
703 
704     // if we are at the crossover point, call the normal
    ↪ multiplication
705     if (dim <= crossover)
706     {
707         standart_ciph_matrix_mult(evaluator, relin_keys, A, B, C,
    ↪ dim);
708         return;
709     }
710     else
711     {
712         // setting the dimension of the new matrices
713         size_t neew_dd = dim / 2;
714 
715         dd_inside_chipher inside(neew_dd);
716 
717         // initialize the submatices
718         dd_chipher_matrix A11(neew_dd, inside);
719         dd_chipher_matrix A12(neew_dd, inside);
720         dd_chipher_matrix A21(neew_dd, inside);
721         dd_chipher_matrix A22(neew_dd, inside);
722         dd_chipher_matrix B11(neew_dd, inside);
723         dd_chipher_matrix B12(neew_dd, inside);
724         dd_chipher_matrix B21(neew_dd, inside);
725         dd_chipher_matrix B22(neew_dd, inside);
726         dd_chipher_matrix C11(neew_dd, inside);
727         dd_chipher_matrix C12(neew_dd, inside);
728         dd_chipher_matrix C21(neew_dd, inside);
729         dd_chipher_matrix C22(neew_dd, inside);
730 
731         // split matrices A and B in 4 submatrices
732         strassen_splitt(A, A11, 0, 0, neew_dd);
733         strassen_splitt(A, A12, 0, neew_dd, neew_dd);
734         strassen_splitt(A, A21, neew_dd, 0, neew_dd);
735         strassen_splitt(A, A22, neew_dd, neew_dd, neew_dd);
736         strassen_splitt(B, B11, 0, 0, neew_dd);
737         strassen_splitt(B, B12, 0, neew_dd, neew_dd);
738         strassen_splitt(B, B21, neew_dd, 0, neew_dd);
739         strassen_splitt(B, B22, neew_dd, neew_dd, neew_dd);
740 
741         // matrices to store results from arithmetic operations
742         dd_chipher_matrix result1(neew_dd, inside);
743         dd_chipher_matrix result2(neew_dd, inside);
```

```
744
745         // calculate M1
746         strassen_addd(evaluator, A11, A22, result1, neew_dd);
747         strassen_addd(evaluator, B11, B22, result2, neew_dd);
748         dd_chipher_matrix M1(neew_dd, inside);
749         strassen_ciph_matrix_mult(evaluator, relin_keys, result1,
    ↪  result2, M1, neew_dd, crossover);
750
751         // calculate M2
752         strassen_addd(evaluator, A21, A22, result1, neew_dd);
753         dd_chipher_matrix M2(neew_dd, inside);
754         strassen_ciph_matrix_mult(evaluator, relin_keys, result1,
    ↪  B11, M2, neew_dd, crossover);
755
756         // calculate M3
757         strassen_subtractt(evaluator, B12, B22, result2, neew_dd);
758         dd_chipher_matrix M3(neew_dd, inside);
759         strassen_ciph_matrix_mult(evaluator, relin_keys, A11,
    ↪  result2, M3, neew_dd, crossover);
760
761         // calculate M4
762         strassen_subtractt(evaluator, B21, B11, result2, neew_dd);
763         dd_chipher_matrix M4(neew_dd, inside);
764         strassen_ciph_matrix_mult(evaluator, relin_keys, A22,
    ↪  result2, M4, neew_dd, crossover);
765
766         // calculate M5
767         strassen_addd(evaluator, A11, A12, result1, neew_dd);
768         dd_chipher_matrix M5(neew_dd, inside);
769         strassen_ciph_matrix_mult(evaluator, relin_keys, result1,
    ↪  B22, M5, neew_dd, crossover);
770
771         // calculate M6
772         strassen_subtractt(evaluator, A21, A11, result1, neew_dd);
773         strassen_addd(evaluator, B11, B12, result2, neew_dd);
774         dd_chipher_matrix M6(neew_dd, inside);
775         strassen_ciph_matrix_mult(evaluator, relin_keys, result1,
    ↪  result2, M6, neew_dd, crossover);
776
777         // calculate M7
778         strassen_subtractt(evaluator, A12, A22, result1, neew_dd);
779         strassen_addd(evaluator, B21, B22, result2, neew_dd);
780         dd_chipher_matrix M7(neew_dd, inside);
781         strassen_ciph_matrix_mult(evaluator, relin_keys, result1,
    ↪  result2, M7, neew_dd, crossover);
782
783         // calculating C11
784         strassen_addd(evaluator, M1, M4, result1, neew_dd);
785         strassen_addd(evaluator, result1, M7, result2, neew_dd);
786         strassen_subtractt(evaluator, result2, M5, C11, neew_dd);
```

```
787
788         // calculating C12
789         strassen_addd(evaluator, M3, M5, C12, neew_dd);
790
791         // calculating C21
792         strassen_addd(evaluator, M2, M4, C21, neew_dd);
793
794         // calculating C22
795         strassen_subtractt(evaluator, M1, M2, result1, neew_dd);
796         strassen_addd(evaluator, M3, M6, result2, neew_dd);
797         strassen_addd(evaluator, result1, result2, C22, neew_dd);
798
799         // add the resulting matrices in one matrix
800         strassen_joinn(C11, C, 0, 0, neew_dd);
801         strassen_joinn(C12, C, 0, neew_dd, neew_dd);
802         strassen_joinn(C21, C, neew_dd, 0, neew_dd);
803         strassen_joinn(C22, C, neew_dd, neew_dd, neew_dd);
804     }
805 }
806
807 void dd_create_sub_result(dd_polynom& polynom, dd_matrix&
    ↪  out_matrix, size_t matrix_dim, int n)
808 {
809     for (size_t a = 0; a < matrix_dim; a++)
810         for (size_t b = 0; b < matrix_dim; b++)
811             out_matrix[a][b] = -polynom[a * matrix_dim + b *
    ↪  matrix_dim * matrix_dim + n];
812 }
813
814 void normal_multiply(dd_matrix& A, dd_matrix& B, dd_matrix& C, int
    ↪  d)
815 {
816     for (int a = 0; a < d; a++)
817     {
818         for (int b = 0; b < d; b++)
819         {
820             for (int c = 0; c < d; c++)
821             {
822                 C[a][b] += A[a][c] * B[c][b];
823             }
824         }
825     }
826 }
827
828 void standart_ciph_matrix_mult(
829     Evaluator& evaluator, RelinKeys relin_keys, dd_chipher_matrix&
    ↪  chipher_matrix_A,
830     dd_chipher_matrix& chipher_matrix_B, dd_chipher_matrix&
    ↪  chipher_matrix_out, int dim)
831 {
```

```
832         for (int u = 0; u < dim; u++)
833         {
834             for (int v = 0; v < dim; v++)
835             {
836                 for (int y = 0; y < dim; y++)
837                 {
838                     Ciphertext result_matrix;
839                     evaluator.multiply(chipher_matrix_A[u][y],
    ↪   chipher_matrix_B[y][v], result_matrix);
840                     evaluator.relinearize_inplace(result_matrix,
    ↪   relin_keys);
841                     if (y == 0)
842                         chipher_matrix_out[u][v] = result_matrix;
843                     else
844                         evaluator.add_inplace(chipher_matrix_out[u][v],
    ↪   result_matrix);
845 
846                     // chipher_matrix_out[i][j] +=
    ↪   chipher_matrix_A[i][k] * chipher_matrix_B[k][j];
847                 }
848             }
849         }
850     }
851 
852 
853     vector<string> dd_split_str(const string& str, const string& delim)
854     {
855         vector<string> tokenss;
856         size_t prevv = 0, poss = 0;
857         do
858         {
859             poss = str.find(delim, prevv);
860             if (poss == string::npos)
861                 poss = str.length();
862             string tokenn = str.substr(prevv, poss - prevv);
863             if (!tokenn.empty())
864                 tokenss.push_back(tokenn);
865             prevv = poss + delim.length();
866         } while (poss < str.length() && prevv < str.length());
867         return tokenss;
868     }
869 
870     void dd_decode(std::string& text, dd_polynom& pout)
871     {
872         auto exprs = dd_split_str(text, " + ");
873         for (auto e : exprs)
874         {
875             auto atoms = dd_split_str(e, "x^");
876             if (atoms.size() > 1)
877             {
```

```
878             dd_unit val;
879             std::stringstream stream;
880             stream << std::hex << atoms[0];
881             stream >> val;
882 
883             if (val > 516'096)
884                 val -= 1'032'193;
885 
886             auto index = std::stoi(atoms[1]);
887             pout[index] = val;
888         }
889         else if (atoms.size() == 1)
890         {
891             dd_unit val;
892             std::stringstream stream;
893             stream << std::hex << atoms[0];
894             stream >> val;
895 
896             if (val > 516'096)
897                 val -= 1'032'193;
898 
899             pout[0] = val;
900         }
901         else
902             throw out_of_range("ft_decode parsing error");
903     }
904 }
905 
906 void dd_encode(dd_polynom& p, size_t dim, std::string& str)
907 {
908     for (int i = dim - 1; i >= 0; i--)
909     {
910         if (p[i] == 0)
911             continue;
912         if (p[i] > 516'096)
913             throw std::out_of_range("value bigger than 516096");
914 
915         auto value = p[i];
916         if (p[i] < 0)
917             value = 1'032'193 + p[i];
918         std::stringstream stream_h;
919         stream_h << std::hex << (uint64_t)value;
920         stream_h << "x^";
921         std::string result(stream_h.str());
922         result += std::to_string(i);
923         if (i != 0)
924             result += " + ";
925         str += result;
926     }
927 }
```

```
928
929 void create_polynomial_result(
930     Decryptor& decryptor, dd_chipher_matrix& chipher_matrix,
    ↪  dd_polynomial_matrix& out_poly_matrix, size_t poly_dim,
931     size_t div)
932 {
933     for (int a = 0; a < div; ++a)
934         for (int b = 0; b < div; ++b)
935         {
936             Plaintext decrypt_result;
937             // std::cout << "    + noise budget in ciph_matrix_mult:
    ↪  "
938             //           <<
    ↪  decryptor.invariant_noise_budget(chipher_matrix[i][k]) << "
    ↪  bits" << std::endl;
939             decryptor.decrypt(chipher_matrix[a][b], decrypt_result);
940             dd_polynom pout(poly_dim, 0);
941             dd_decode(decrypt_result.to_string(), pout);
942             out_poly_matrix[a][b] = pout;
943         }
944 }
945
946 void create_chipher_matrix(
947     Encryptor& encryptor, dd_sub_matrices& sub_matrices,
    ↪  dd_chipher_matrix& chipher_out, size_t poly_dim, size_t i_dim,
948     size_t div)
949 {
950     for (int z = 0; z < div; ++z)
951         for (int k = 0; k < div; ++k)
952         {
953             dd_polynom polynom_out(poly_dim);
954             generate_polynom(sub_matrices[z][k], polynom_out,
    ↪  poly_dim, i_dim);
955
956             string str;
957             dd_encode(polynom_out, poly_dim, str);
958             Plaintext plain_matrix(str);
959
960             Ciphertext x_encrypted;
961             encryptor.encrypt(plain_matrix, x_encrypted);
962             chipher_out[z][k] = x_encrypted;
963         }
964 }
965
966 void create_chipher_matrix(
967     Encryptor& encryptor, dd_sub_matrices& sub_matrices,
    ↪  dd_chipher_matrix& chipher_out, size_t poly_dim, size_t i_dim,
968     size_t div, int n)
969 {
970     for (int t = 0; t < div; ++t)
```

```
971             for (int u = 0; u < div; ++u)
972             {
973                 dd_polynom polynom_out(poly_dim);
974                 generate_polynom(sub_matrices[t][u], polynom_out,
    ↪  poly_dim, i_dim, n);
975
976                 std::string str;
977                 dd_encode(polynom_out, poly_dim, str);
978                 Plaintext plain_matrix(str);
979
980                 Ciphertext x_encrypted;
981                 encryptor.encrypt(plain_matrix, x_encrypted);
982                 chipher_out[t][u] = x_encrypted;
983             }
984     }
985
986     void generate_polynom(dd_matrix& matrix, dd_polynom& p_out, size_t
    ↪  poly_dim, size_t dim, int n)
987     {
988         for (int i = 0; i < dim; ++i)
989         {
990             dd_polynom outer(poly_dim);
991             dd_polynom inner(poly_dim);
992             dd_polynom resultp(poly_dim);
993
994             outer[i * dim * dim] = 1;
995             for (int k = 0; k < dim; ++k)
996                 inner[n - k] -= matrix[k][i];
997
998             mult_polynom(outer, inner, resultp, poly_dim);
999             add_polynom(p_out, resultp, poly_dim);
1000        }
1001    }
1002
1003    void generate_polynom(dd_matrix& matrix, dd_polynom& p_out, size_t
    ↪  poly_dim, size_t dim)
1004    {
1005        for (int i = 0; i < dim; ++i)
1006        {
1007            dd_polynom outer(poly_dim);
1008            dd_polynom inner(poly_dim);
1009            dd_polynom resultp(poly_dim);
1010
1011            outer[i * dim] = 1;
1012            for (int o = 0; o < dim; ++o)
1013                inner[o] += matrix[i][o];
1014
1015            mult_polynom(outer, inner, resultp, poly_dim);
1016            add_polynom(p_out, resultp, poly_dim);
1017        }
```

```
1018 }
1019
1020 void mult_polynom(dd_polynom& p1, dd_polynom& p2, dd_polynom& pout,
     ↪  size_t dim)
1021 {
1022     for (int i = 0; i < dim; i++)
1023     {
1024         if (p1[i] == 0)
1025             continue;
1026         for (int k = 0; k < dim; k++)
1027         {
1028             if (p2[k] == 0 || i + k >= dim)
1029                 continue;
1030             pout[i + k] = p1[i] * p2[k];
1031         }
1032     }
1033 }
1034
1035 void add_polynom(dd_polynom& p1, dd_polynom& p2, size_t dim)
1036 {
1037     for (size_t o = 0; o < dim; o++)
1038         p1[o] += p2[o];
1039 }
1040
1041
1042
1043 void print_polynom(dd_polynom& p, size_t dim)
1044 {
1045     for (int k = dim - 1; k >= 0; --k)
1046     {
1047         if (p[k] == 0)
1048             continue;
1049         std::cout << p[k] << "x^" << k << "\n";
1050     }
1051     seperate_line();
1052 }
1053
1054 void set_matrix(dd_matrix& matrix, int n)
1055 {
1056     // Create a random number generator
1057     std::random_device rd;
1058     std::mt19937 gen(rd());
1059     std::uniform_int_distribution<> distr(0, DD_MAX_RANDOM_NUMBER);
1060
1061     // Resize the matrix to the desired size
1062     matrix.resize(n, dd_inside_matrix(n));
1063
1064     // Fill the matrix with random integers between 0 and 10
1065     for (int h = 0; h < n; h++)
1066         for (int g = 0; g < n; g++)
```

```
            matrix[h][g] = distr(gen);
}

void set_col_matrix(dd_matrix& matrix, int n)
{
    // Create a random number generator
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distr(0, DD_MAX_RANDOM_NUMBER);

    // Resize the matrix to the desired size
    matrix.resize(n, dd_inside_matrix(n, 0));

    // Fill the first column with random integers between 0 and 10
    for (int z = 0; z < n; z++)
        matrix[z][0] = distr(gen);
}

void sub_matrix(dd_matrix& A, dd_matrix& out_matrix, int row, int
↪   col, int div)
{
    for (int g = 0; g < div; ++g)
        for (int h = 0; h < div; ++h)
            out_matrix[g][h] = A[row - div + g][col - div + h];
}

void seperate_line()
{
    std::cout << "-----------------------------------------------" <<
↪   std::endl;
}

double roundToDigits(double num, int digits)
{
    double factor = std::pow(10.0, digits);
    return std::round(num * factor) / factor;
}

void print_m(dd_matrix m, int d)
{
    for (int s = 0; s < d; s++)
    {
        for (int t = 0; t < d; t++)
        {
            std::cout << "\t" << roundToDigits(m[s][t], 3);
        }
        std::cout << std::endl;
    }
    std::cout << "-----------------------------------------------" <<
↪   std::endl;
```

```
1114 }
1115
1116 void print_m(dd_matrix_f m, int d)
1117 {
1118     for (int z = 0; z < d; z++)
1119     {
1120         for (int y = 0; y < d; y++)
1121         {
1122             std::cout << "\t    " << roundToDigits(m[z][y], 7);
1123         }
1124         std::cout << std::endl;
1125     }
1126     std::cout << "-----------------------------------------------" <<
     ↪  std::endl;
1127 }
1128
1129 void print_col_m(dd_matrix_f m, int d)
1130 {
1131     for (int u = 0; u < d; u++)
1132     {
1133         std::cout << "\t" << roundToDigits(m[u][0], 3);
1134         std::cout << std::endl;
1135     }
1136     std::cout << "-----------------------------------------------" <<
     ↪  std::endl;
1137 }
1138
1139 void print_col_m(dd_matrix m, int d)
1140 {
1141     for (int r = 0; r < d; r++)
1142     {
1143         std::cout << "\t" << roundToDigits(m[r][0], 3);
1144         std::cout << std::endl;
1145     }
1146     std::cout << "-----------------------------------------------" <<
     ↪  std::endl;
1147 }
1148
1149 void print_v(dd_inside_matrix v, int d)
1150 {
1151     for (int a = 0; a < d; a++)
1152     {
1153         std::cout << "\t" << v[a];
1154     }
1155     std::cout << std::endl;
1156     std::cout << "-----------------------------------------------" <<
     ↪  std::endl;
1157 }
1158
1159 void split(dd_matrix& A, dd_matrix& B, int roww, int coll, int d)
```

```
{
    for (int u1 = 0, u2 = roww; u1 < d; u1++, u2++)
    {
        for (int v1 = 0, v2 = coll; v1 < d; v1++, v2++)
        {
            B[u1][v1] = A[u2][v2];
        }
    }
}
```

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Öner Şimşek, Dilek
Nationality: Turkish (TC)

EDUCATION

Degree	Institution	Year of Graduation
M.Sc., Cryptography	METU	2011
B.S., Statistics, and Comp. Sc.	Baskent University	2009
High School	Gölbasi Anotalian High Sch.	2005

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
2012-	Social Security Institute	Social Security Specialist

PUBLICATIONS

Dilek Öner Şimşek, Overview of Triage Systems and Determining the Factors Affecting Emergency Department References in Turkey by Logistic Regression, 2018, Social Security Journal