Powerformer: Efficient Privacy-Preserving Transformer with Batch Rectifier-Power Max Function and Optimized Homomorphic Attention

Dongjin Park
Chung-Ang University

Eunsang Lee*
Sejong University

Joon-Woo Lee*
Chung-Ang University

Abstract

We propose an efficient non-interactive privacy-preserving Transformer inference architecture called **Powerformer**. Since softmax is a non-algebraic operation, previous studies have attempted to modify it to be HE-friendly, but these methods have encountered issues with accuracy degradation or prolonged execution times due to the use of multiple bootstrappings. We propose replacing softmax with a new ReLU-based function called the *Batch Rectifier-Power max* (BRP-max) function without any unstable approximation methods, which outperforms even original BERT performance within BERT-Large model while requiring fewer levels, allowing it to operate with only a single bootstrapping. We also present a matrix multiplication algorithms specialized for attention block that reduce the number of key-switchings by 35% to 91% compared to existing state-of-the-art methods. We design clear end-to-end HE-based implementation for private Transformer model, and our implementation of Powerformer on the BERT-tiny model using RNS-CKKS takes 503 seconds on a single-threaded CPU, and to the best of our knowledge, this is the first end-to-end non-interactive Transformer implementation using HE.

1 Introduction

The Transformer model is one of the most prominent technologies in recent times, with the Transformer-based language model, ChatGPT, pushing the boundaries of traditional AI capabilities by maximizing the potential of large language models. It enables natural interactions with humans, bringing about significant innovation. However, ChatGPT currently operates on a Machine Learning as a Service (MLaaS) model, where the client sends their data to the server, which then performs inference using its model and returns the results to the client. This process exposes the client's data to the server, raising significant privacy concerns. Due to these issues, security-conscious companies are hesitant to use ChatGPT, fearing the potential leakage of corporate data, which ultimately limits the full utilization of the powerful Transformer-based models. To address privacy issues in an MLaaS environment, Privacy-Preserving Machine Learning (PPML) using HE to perform inference on encrypted data has been actively researched [3, 7, 13, 16, 19, 27]. With the increasing focus on Transformer models, there has also been a rise in research on secure inference for Transformers [4,6,9,21,23,28,29]. PPML research is divided into interactive PPML, where inference is performed through communication between the client and server, and non-interactive PPML, where the server performs inference on encrypted data without communication.

Interactive PPML [6,9,13,21,23,24,27] primarily employs approaches using HE and Multi-Party Computation (MPC), which offer the advantage of relatively fast execution times. However, when targeting complex models like Transformers, this approach can result in communication costs exceeding tens of gigabytes, which can be a constraint in environments with insufficient communication bandwidth. In contrast, non-interactive PPML involves only the server performing computations on the client's encrypted data without any communication, with the client simply sending their encrypted data and receiving the encrypted final results. Due to the advantage of not requiring online communication, many studies have adopted the non-interactive approach [3, 7, 10, 16, 19, 28, 29]. This paper focuses on non-interactive privacy-preserving Transformer inference. RNS-CKKS [5] is a good choice of HE for non-interactive Transformer inference because it offers the advantage of performing a large number of real-number operations quickly. Two major challenges in implementing efficient non-interactive privacy-preserving Transformer inference on RNS-CKKS are the softmax operation and the matrix multiplication required for attention. The most critical issue in HE-based Transformer research is that there is still no paper that fully addresses a clear end-to-end implementation of an HE-based Transformer model that operates stably in practice.

*Co-corresponding authors:
eslee3209@sejong.ac.kr, jwlee2815@cau.ac.kr

Softmax Function The softmax function can be calculated using the following equation, where the subtraction of $x_{\max}$ is included for numerical stability [15].

$$y_i = \frac{\exp(x_i - x_{max})}{\sum_{j=0}^{m-1} \exp(x_j - x_{max}))} \quad (1)$$

The two most common approaches for applying the softmax function in HE are replacing the softmax function with a polynomial function or accurately approximating the softmax function itself. The first approach replaces the softmax function with a polynomial and fine-tunes the model, potentially improving time performance in HE-based implementations, but a key issue is the accuracy drop when the polynomial fails to effectively replace softmax in the neural network. For example, the replacement method for softmax function proposed by Zimerman et al. [29] significantly reduces accuracy, as shown in the experimental results provided in Section 2.1. The second approach focuses on providing a fast and accurate approximation. For example, the most recent research, such as [28], approximate the softmax function using Taylor series and Goldschmidt polynomial approximation methods. Due to the high instability of the exponential function, it is difficult to perform accurate calculations on RNS-CKKS. As a result, the use of the subtraction x_{max} for stabilizing becomes unavoidable, which in turn requires a large number of comparison operations, leading to significant computational overhead during secure inference.

Matrix Multiplication All papers dealing with HE-based private inference do not fully present end-to-end matrix multiplication methods for the entire Transformer network. For example, previous studies in this research domain include NEXUS [28] and the study by Zimerman et al [29]. NEXUS proposes a ciphertext-plaintext multiplication algorithm for computing the query, key, and value matrices Q_i, K_i, V_i in multi-head attention, as well as ciphertext-ciphertext multiplication for the computation of $Q_i K_i^T$. However, homomorphic algorithms for subsequent multiplications, such as the multiplication with V_i or the output weight matrix W^O, cannot be found in NEXUS. Additionally, NEXUS employs a variety of packing methods, such as component-wise packing, row-wise packing, column-wise packing, and diagonal-wise packing. However, it lacks a detailed explanation of how the packing structure changes during inference to enable non-interactive computation, where definitions of the aforementioned packing methods are provided in Section B.2. On the other hand, Zimerman et al.'s work [29] does not provide detailed implementation specifics, such as homomorphic matrix multiplication algorithms, nor does it offer code for implementation in the encrypted state, making it very difficult to reproduce their non-interactive implementation and analyze its efficiency. Therefore, it is essential for future research on HE-based Transformer implementation to propose a matrix multiplication method specifically suitable for HE-based private Transformer inference and to provide end-to-end implementation source code of this approach.

Although the entire HE-based implementation of the Transformer network is not fully addressed, several matrix multiplication methods for an attention block were proposed in HE-MPC hybrid implementation. However, the computational amount for this matrix multiplication was not optimized, or there are many intermediate ciphertexts due to the non-compact implementation, causing a significant amount of the bootstrapping amount. For example, BOLT [23] proposed a homomorphic matrix multiplication algorithm utilizing column-major packing, and while the plaintext-ciphertext multiplication, which requires $2\sqrt{d}$ key switchings, is fairly efficient the ciphertext-ciphertext matrix multiplication still requires as many as $d \log d + 2d$ key switchings, which is more inefficient implementation compared to Jiang et al.'s well-known homomorphic matrix multiplication method [10]. On the other hand, the row-wise and column-wise packing methods used in NEXUS [28] are inefficient, as they leave many slots unused, and our analysis indicates that the use of these packing methods requires a large number of bootstrapping operations. In addition, setting aside the number of bootstrapping operations, implementing multi-head attention using the matrix multiplication algorithm proposed by NEXUS requires at least $6d^2$ key switchings. This imposes an excessive number of key switchings for the Transformer model we are targeting, where $d = 128$. While Jiang et al. proposed an efficient general ciphertext-ciphertext matrix multiplication algorithm, this method needs to be specialized for the attention block, which has intricate matrix operation structures.

1.1 Our Contributions

We propose an efficient end-to-end private HE-based Transformer implementation, called **PowerFormer** by addressing two important research points in HE-based implementation.

- We introduce a new set of softmax replacement functions, the *Batch Rectifier-Power max* (BRPmax) function set, which surpasses existing softmax replacement methods. This novel approach achieves high numerical stability, almost the same classification accuracy with original BERT model, and efficient inference runtime performance through our new training method.
- We propose a matrix multiplication operation optimized for multi-head attention, presenting an end-to-end implementation method that minimizes both the number of key-switching operations in multi-head attention and the number of bootstrapping operations.

Batch Rectifier-Power Max (BRPmax) Method We propose approximating the softmax function as $\mathrm{ReLU}(x + c)^p / R_d$ for some c, p, and R_d, where constants c, p are fixed. With adapted batch method from batch normalization, the

running denominator R_d is learned during training, and it is treated as a constant during inference, allowing the operation to be performed on encrypted data by multiplying by the constant $1/R_d$. The ReLU function is precisely approximated using the method by Lee et al. [17,18]. We numerically demonstrate that the Powered ReLU (Rectifier-Power) function effectively substitutes the exponential function. We also show that batch methods in distillation learning resolve dynamic scaling issues, a common challenge in PPML.

Optimized Homomorphic Attention We further optimize the matrix-matrix multiplication algorithm by Jiang et al. [10]. For plaintext-ciphertext matrix multiplication, we integrate several steps of plaintext-ciphertext multiplication from the ciphertext-ciphertext matrix multiplication algorithm into a single baby-step giant-step summation, which requires 85% to 91% fewer key switchings compared to [10]. We also propose a blockwise matrix multiplication algorithm optimized for multiplying the query matrix and key matrix, which requires 35% fewer key switchings compared to [10].

End-to-End HE-based Transformer Implementation We present the first end-to-end non-interactive secure Transformer inference architecture Powerformer. This architecture replaces non-linear operations, such as softmax, GELU, and layer normalization, with polynomial approximations, while leveraging our efficient matrix multiplication algorithm. Although our focus is on the BERT-tiny model, the architecture can be easily extended to other models, such as BERT-medium and BERT-base.

We have fully implemented our architecture Powerformer end-to-end using the RNS-CKKS library, Lattigo [1]. Unlike NEXUS, which provided results only for individual components executed on encrypted data, we implemented the entire process and present the complete results. Our implementation takes 503 seconds on a single CPU thread, which is at least 39× faster than NEXUS based on single-thread performance. Given that this is a single-thread implementation, we expect the runtime to be reduced by tens to hundreds of times with GPU or hardware acceleration [12,14].

1.2 Outline and Notations

Chapter 2 introduces the limitations of existing research that replaces the softmax function in the Transformer model and describes our BRPmax method. Chapter 3 proposes the matrix multiplication techniques specialized for efficiently implementing the attention block and compares them with existing HE-based attention block implementations. Chapter 4 discusses additional considerations for the private Transformer model and clearly details how to harmoniously combine the aforementioned techniques to implement the Transformer model end-to-end. Chapter 5 presents experimental results that demonstrate the superiority of these techniques and shows the specific performance of our encrypted implementation. Chapter 6 concludes the paper. Appendix A covers the detailed relationship between this paper and other key prior works on private Transformer models, and Appendix B explains the preliminaries necessary to understand this paper, including the Transformer model, the RNS-CKKS FHE scheme, and the homomorphic matrix multiplication method.

In this paper, we use the following notations. $\rho(v;r)$ denotes the vector $v \in \mathbb{R}^n$ that has been cyclically shifted to the left by r positions. $\log(\cdot)$ refers to the logarithm with base 2. For $a \times b$ matrix X, $X_{i,j}$ implies the element indexed by row i and column j of the matrix X. Any notations in the main text that are not defined within the main body may be defined in Appendix B.

2 Batch Rectifier-Power-max Method

One of the most time-consuming operations in Transformer is the softmax operation, because of the division operation and comparison operations in this function. Division with ciphertext in HE is challenging and typically performed using approximation algorithms, such as Goldschmidt's algorithm, that utilize addition and multiplication. The wider the range of values to be divided, the more computationally intensive the operation becomes. To reduce this range, the largest value among the inputs to softmax is subtracted before performing the softmax computation. This is also done in the conventional Transformer because of the numerical stability. However, calculating this maximum value itself also requires significant computation. For these reasons, the softmax operation is inefficient when performed on HE. In this section, we propose two methods for eliminating the division operation and the comparison operation in Transformer inference.

2.1 Limitation of Previous Softmax Implementation

In Zhang et al. [28], a method was proposed to eliminate comparison operations by setting the a_{max} value as a constant during calculations. This approach is based on the observation that the a_{max} value does not necessarily have to be the largest value in the vector for the result to remain unaffected. Instead, they subtract a constant larger than the actual a_{max}, ensuring all values become non-positive. However, Zhang et al. did not specify which constant should be used, making the method ambiguous and practically difficult to apply in RNS-CKKS. Since the maximum value of the inputs to the softmax function varies across different data points and dimensions, a conservative constant, such as the global maximum observed during training, must be used during inference to ensure valid operations across all inputs.

For example, in the RTE task of the GLUE benchmark, the global maximum value of the softmax inputs observed during training was 63.5. If this value is considered as a_{max} during

Table 1: Performance Down of $ReLU(x)/\sqrt{L}$ on BERT model (%).

	BERT-Tiny	BERT-Medium	BERT-Base
RTE	5.42	10.47	17.33
CoLA	0.05	17.28	58.8
MRPC	1.96	14.46	16.91
STS-B	10.28	34.1	2.06
QNLI	0.99	4.36	54.81
Average	3.74	16.13	29.98

computation, and an input during inference has all elements $a_i < 25$ for all i, then $exp(a_i - a_{max}) < e^{25-63.5} = 2^{-55.5}$. Given that NEXUS uses a scaling factor $\Delta = 2^{50}$, this results in $\Delta \cdot m < 1$, causing the message to vanish. Since the scale of the softmax inputs varies significantly across dimensions and data, vectors where all elements are less than 25 occur frequently, making meaningful computation difficult and leading to significant accuracy loss in the overall softmax operation. Therefore, treating a_{max} as a global constant has many practical limitations due to the stability of the exponential function and the numerical capacity constraints of RNS-CKKS.

It is challenging to remove the comparison operations in softmax using this method, so it is necessary to compute the maximum for each input while it is still encrypted. However, performing the maximum operation on encrypted data requires executing $\log L$ sequential comparison operations. For $L = 128$, this would necessitate seven maximum operations. Since a precise maximum operation consumes significant depth, bootstrapping must be performed between each maximum operation. Therefore, bootstrapping would need to be used six times throughout the computation, making the process highly inefficient. To bypass these operations, Zimerman et al. replaced the softmax function with $\mathrm{ReLU}(x)/\sqrt{L}$ and conducted fine-tuning to perform inference.

While this approach works to some extent with lightweight models, it leads to significant performance degradation with larger models. Table 1 shows the performance degradation observed within BERT models of various sizes. When applied to the BERT-Tiny model with 2 layers and 128 hidden dimensions, there was a reduction in accuracy of approximately 3.7%. However, with the BERT-Medium model, which has 8 layers and 512 hidden dimensions, the accuracy dropped by around 16%, and for the BERT-Base model with 12 layers and 768 hidden dimensions, the accuracy decreased by 30%. Given the nature of Transformers, which are primarily used in large-scale models, this approach is impractical for practical use.

2.2 Batch Method in Softmax

We propose a method similar to batch normalization to eliminate the division operation with ciphertext. Batch normalization is a commonly used technique that normalizes inputs using the mean and standard deviation of each batch before applying the activation function, thereby enhancing training stability and speed. A characteristic of this method is that, during inference, normalization is not performed using the mean and variance of the test data within the same computation block. Instead, the moving average of the batch-wise mean and variance from the training dataset, known as the running mean and running variance, is used for normalization. This is because it is not possible to calculate the batch-wise mean and variance during inference since operations are not performed in batches. Despite performing normalization using the training dataset, this method is known to achieve good performance in many tasks.

Inspired by batch normalization, a similar approach to batch normalization can be applied to the softmax operation. During inference, it is difficult to perform division using values from the test data due to the use of HE. Instead, a value obtained from the training dataset is used for division. We will call this value the *running denominator*. Since the running denominator is derived from the training dataset and is owned by the server, it can be stored in plaintext. Dividing by this value is equivalent to simple scalar multiplication, which eliminates the need for heavy approximation methods like Goldschmidt's algorithm, providing significant computational advantages. Therefore, it is important to devise a method that maintains high accuracy while using this approach.

In batch normalization, the running mean and running variance are computed during the training process by continuously calculating the moving average or exponential weighted average of the means and variances of all values within each batch for the corresponding layer. This approach minimizes fluctuations caused by varying values in each batch, thereby achieving stable running means and running variances. This methodology can be analogously applied to determine the running denominator of the softmax function. However, it is imperative to recognize that the denominator of the softmax function serves a different purpose compared to the running mean and running variance. The denominator of the softmax function normalizes each value to lie between 0 and 1, thereby enhancing the numerical stability of the Transformer model. Consequently, the batch method should be designed to preserve this numerical stability when applied to the softmax function.

Within a single batch, there are multiple vectors subjected to the softmax operation. Let the batch size be denoted as B, the number of heads as h, the sequence length as L, and the embedding dimension as d_m. Each input matrix $X_i (i = 0, \cdots, B-1)$ in the batch yields a query matrix Q_{ij} and a key matrix $K_{ij} (j = 0, \cdots, h-1)$ for each head, which have dimensions $L \times d_m/h$, with $d_m/h = d_k$. The inner products of these rows produce $T_{ij} = Q_{ij} K_{ij}^T / \sqrt{d_k}$, resulting in matrices of size $L \times L$. The rows of these matrices contain inner product values indicating the similarity between each query

and various keys. The softmax function is applied to the rows of each T_{ij}. Denote each row of T_{ij} as $T_{ijk} = (t_{ijkl})_l$, where $(k = 0, \cdots, L-1, l = 0, \cdots, L-1)$. Thus, the number of vectors subjected to the softmax operation per batch is $B \cdot h \cdot L$ and the values to consider when calculating the running denominator are:

$$D_{ijk} = \sum_{l=0}^{L-1} e^{t_{ijkl}}$$

If the technique used in batch normalization is directly applied to compute the running denominator, it would involve calculating the moving average values across batches , $MovingAverage(D_{ijk})$. To enable faster ciphertext computation, we need to use a constant form of the Running Denominator. However, since the current values are not uniform, they cannot be converted into a constant, and thus an additional function is required. We applied the maximum operation over the batch, $MovingAverage(\max_{i,j,k} D_{ijk})$. Instead of averaging the D_{ijk} values, employing the maximum value within the batch, $\max_{i,j,k} D_{ijk}$ as the running denominator would result in dividing by a larger value during inference on test data, generally reducing the values to below 1 and thereby enhancing numerical stability.

2.3 Rectifier-Power Function

Another reason why the softmax function requires a significant amount of computation time on encrypted data is the comparison operation. Due to the large scale differences in the exponential function, to compute accurately, it is necessary to calculate the maximum value of each row in the input matrix and subtract this maximum from each value. Since this maximum operation is a non-arithmetic operation, a considerable amount of computation time and deep computational depth are required to accurately approximate it. Additionally, because the maximum operation needs to be performed sequentially $\log L$ times, a substantial amount of bootstrapping operations is also involved. Therefore, using functions with significant scale differences, like the exponential function, is unsuitable for computations on HE. On the other hand, one of the reasons why Transformers can achieve good performance is that the softmax function effectively emphasizes important tokens. Thus, the scale difference of the softmax function based on the input could impact the performance of the Transformer.

We consider using another function instead of the exponential function. This function should resemble the shape of the exponential function but with less variation in function values. This led us to devise the Rectifier-Power(RP) function:

$$\mathrm{RP}_{p,c}(x) = \mathrm{ReLU}(x+c)^p$$

The ReLU function yields zero for negative values, eliminating the influence of negative inputs. This aligns with the property of the exponential function, which produces very small values for negative inputs, effectively minimizing their impact. For positive values, a polynomial function x^p is used. It does not exhibit excessively large differences in function values for different inputs as the exponential function does. The offset c preserves information about some negative values close to zero, which is characteristic of the exponential function. This prevents the loss of information that occurs when the higher-order terms of k reduce the range below 1, and it also helps avoid excessive focus on large values, maintaining a stable and balanced attention distribution overall.

In $\mathrm{RP}_{p,c}(x)$, the setting of p is task-dependent. In situations where it is important to focus on specific words or phrases, such as text classification, a higher p value may be advantageous, whereas tasks that require consideration of various contexts, such as translation or text generation, may benefit from a lower p value. This is a unique advantage of the RP Function, which the conventional softmax does not possess. The setting of c follows the value of p and it is recommended to use an offset of similar magnitude to p. An excessively large offset for a small p could hinder the creation of balanced attention by preventing focus on specific values, whereas an excessively small offset for a large p could lead to unbalanced attention, causing excessive focus on certain values.

2.4 Training and Inference Method

We use knowledge distillation to fine-tune the model after replacing the softmax function with the Rectifier-Power max (RPmax) function. Knowledge distillation is a commonly used technique that transfers the knowledge of a well-trained teacher model to a student model through an additional learning step. This technique allows for reducing the size and complexity of the model while maintaining performance. It typically involves applying a loss function such as mean square error(MSE) to the logits of the teacher model, or applying a loss function like cross entropy(CE) to the probability distribution of the teacher model. This approach enables the generation of a more simplified HE-friendly student model, which is well-suited for HE inference tasks, as it replaces time-consuming ciphertext operations with plaintext training performed in advance. [20], [22] demonstrate the efficiency of knowledge distillation in Transformer inference. MPCFormer achieved 97% of BERT-base performance, while SECFormer further improved BERT-base and BERT-large performance by 3.4% and 24.7%, respectively, compared to MPCFormer.

We follow the knowledge distillation method proposed in [11]. Using the fine-tuned original BERT model for each downstream task as the teacher, distillation was performed by applying loss functions to only four key positions, rather than to all positions of the model. These positions are (1) the embedding layer, (2) the attention matrix in each Transformer layer, (3) the hidden states after each Transformer layer, and (4) the final prediction layer. The distillation is conducted in two steps. First, MSE is applied to positions (1), (2), and (3)

Table 2: Comparison between Batch Max Sum and Batch Max

		accuracy (%)	# epochs
RTE	Batch Sum Max	68.59	309
	Batch Max	68.95	103
CoLA	Batch Sum Max	59.36	222
	Batch Max	60.33	100
MRPC	Batch Sum Max	86.27	307
	Batch Max	86.52	69
STS-B	Batch Sum Max	88.71	145
	Batch Max	89.40	95
QNLI	Batch Sum Max	91.18	87
	Batch Max	91.52	51
Average	Batch Sum Max	78.82	214
	Batch Max	79.35	83.6

for the initial training phase, and then Soft CE is applied to position (4) in the second training phase. Through this consistent distillation method, we can effectively verify the utility of the student model. In all future experiments, we will use knowledge distillation as the default method for validation.

We made modifications to the existing running denominator for batch training. We created a new running denominator in the form of $D_{ijkl} = \max e^{t_{ijkl}}$, and we will refer to the previous running denominator as 'batch sum max' and the new form as 'batch max.' The difference in how the denominator is calculated in softmax during the knowledge distillation process affects both the model's convergence speed and its final performance. The batch sum max method uses the sum of the last dimension as the denominator, which means that changes in individual values have a significant impact on the overall denominator. This approach makes the model highly sensitive to frequently changing values during training, leading to difficulties in distinguishing between important and less important information. Ultimately, this increases the complexity of learning and can slow down convergence. In contrast, the batch max method uses only the largest value in the batch as the denominator, minimizing the influence of frequently changing values and enabling more stable learning. This method allows the model to focus on important information during training, resulting in faster convergence and higher accuracy. Table 2 illustrates the impact of the batch method on final accuracy and convergence speed under the same conditions.

Now, our training method and inference method is explained, which effectively reduce inference latency on encrypted data while maintaining accuracy nearly comparable to the original model by utilizing both the batch method and the RP function. The original BERT model is used as the teacher model, while the BERT model that replaces the softmax function with the following RPmax function is used as the student model.

$$\mathrm{RPmax}_{p,c}(T=[t_{ijkl}]) = \left[\frac{\mathrm{RP}_{p,c}(t_{ijkl})}{\max_{i,j,k,l}\mathrm{RP}_{p,c}(t_{ijkl})}\right]_{ijkl},$$

where T refers to a four-dimensional batch as defined in Section 2.2. Unlike the traditional softmax, the $\mathrm{RPmax}_{k,c}$ function uses the maximum value across the entire batch as the denominator, and thus the input is not a single row but the entire batch. During training, we extract $R = \max_{i,j,k,l}\mathrm{RP}_{p,c}(t_{ijkl})$ for each batch and continuously update it by taking a moving average with the R value obtained from the previous batch. After performing this for several epochs, let the result of the moving average be R_d.

When performing inference, we replace the softmax function with the following BRPmax function and compute with the BERT model in an encrypted state.

$$\mathrm{BRPmax}_{p,c}(T=[t_{ijkl}]) = \left[\frac{\mathrm{RP}_{p,c}(t_{ijkl})}{R_d}\right]$$

Note that the BRPmax function, unlike RPmax, is a component-wise operation. Therefore, when performing the actual computation, you can use the SIMD property to perform the ReLU^p operation once in the encrypted state and then perform plaintext multiplication with $1/R_d$. The ReLU^p operation can be effectively computed using the method by Lee et al. without any maximum functions, and since there is no need to use bootstrapping between operations, this significantly reduces the overall computation time.

3 Matrix Operations for Transformer

3.1 Constant Rectangle Matrix Multiplication

In the first part of multi-head attention, for each head j, the weight matrices for query, key, and value, denoted as $W_{Q_i}^{(j)}, W_{K_i}^{(j)}, W_{V_i}^{(j)} \in \mathbb{R}^{L\times d_k}$, are multiplied by X on the right to obtain $Q_i^{(j)}, K_i^{(j)}, V_i^{(j)}$. The BERT-tiny model we are targeting has parameters $L = d_m = 128$ and $d_k = d_m/h = d_m/2$. In this section, we focus on optimizing the homomorphic multiplication of a $d\times d$ matrix and a $d\times\ell$ matrix, where we have $d|\ell$. The case where $d = 128$ and $\ell = 64$ corresponds to the BERT-tiny model. The rectangular matrix multiplication formula from Jiang et al. is as follows:

$$A\cdot W = \sum_{j=0}^{d/\ell-1}\left(\sum_{i=0}^{\ell-1}(\phi^i\circ\sigma(A))\odot(\psi^i\circ\tau(\bar{W}))\right)_{j\ell:j(\ell+1)} \tag{2}$$

$\bar{W}\in\mathbb{R}^{d\times d}$ refers to a square matrix created by horizontally replicating the matrix $W\in\mathbb{R}^{d\times\ell}$ with d/ℓ times. Based on this formula, the inner summation operation is performed in the HE setting as follows.

$$\sum_{i=0}^{\ell-1}(U^{\phi})^i U^{\sigma} a\odot(U^{\psi})^i U^{\tau}\bar{w}$$

We note that, unlike in the scenario of Jiang et al., the weight matrix is not in an encrypted state and can therefore be freely packed. Therefore, the server can precompute $w^{(i)} = (U^{\psi})^i U^{\tau} \bar{w}$, and then the following operation can be performed:

$$\sum_{i=0}^{\ell-1} (U^{\phi})^i U^{\sigma} a \odot w^{(i)} \tag{3}$$

In the case of $a^{(0)} = U^{\sigma} a$, the operation can be performed in HE as follows:

$$U^{\sigma} a = \sum_{k=-d+1}^{d-1} u_k^{\sigma} \odot Rot(a;k) \tag{4}$$

Additionally, we have

$$(U^{\phi})^i U^{\sigma} a = v_i \odot Rot(U^{\sigma} a; i) + v_i' \odot Rot(U^{\sigma} a; i-d), \tag{5}$$

where v_i and v_i' are defined as follows:

$$v_i[j] = \begin{cases} 1 & \text{if } 0 \le [j]_d < d-i \\ 0 & \text{otherwise,} \end{cases} \quad v_i'[j] = \begin{cases} 1 & \text{if } d-i \le [j]_d < d \\ 0 & \text{otherwise.} \end{cases}$$

By substituting equation 4 into 5, and then substituting the result into 3 and simplifying, we obtain the following equation.

$$\begin{aligned}
&\sum_{i=0}^{\ell-1} (w^{(i)} \odot v_i) \odot Rot\left(\sum_{k=-d+1}^{d-1} u_k^{\sigma} \odot Rot(a;k); i \right) \\
&\quad + \sum_{i=0}^{\ell-1} (w^{(i)} \odot v_i') \odot Rot\left(\sum_{k=-d+1}^{d-1} u_k^{\sigma} \odot Rot(a;k); i-d \right) \\
&= \sum_{i=0}^{\ell-1} \sum_{k=-d+1}^{d-1} (w^{(i)} \odot v_i \odot \rho(u_k^{\sigma}; i)) \odot Rot(a; k+i) \\
&\quad + \sum_{i=0}^{\ell-1} \sum_{k=-d+1}^{d-1} (w^{(i)} \odot v_i' \odot \rho(u_k^{\sigma}; i-d)) \odot Rot(a; k+i-d) \\
&= \sum_{j=-2d+1}^{d+\ell-2} s_j \odot Rot(a; j),
\end{aligned}$$

where s_j refers to the plaintext vector composed of $w^{(i)}, v_i, \rho(u_k^{\sigma}; i), \rho(u_k^{\sigma}; i-d)$ that results from simplifying the above equations. Now, the baby-step giant-step method can be used to reduce the number of rotations to approximately $2\sqrt{3d+\ell-2}$ as follows.

$$\sum_{0 \le j < N_2} Rot\Bigg(\sum_{0 \le i < N_1} \rho(s_{N_1 j + i - 2d + 1}; 2d - 1 - N_1 j) \odot Rot(a; i) \; ; N_1 j - 2d + 1 \Bigg), \tag{6}$$

where N_1 and N_2 are customizable parameters determining the number of rotations satisfying $N_1 N_2 = 3d + \ell - 2$. In this context, the inner rotations' summation operations can be further optimized using the double hoisting technique, which reduces the computational load. To maximize the effect of hoisting, N_1 should be set larger than N_2.

For the outer summation in the formula 2, a rotation and sum operation is performed where each row is divided into units of ℓ elements, summed, and then followed by a zeroing out process. This operation follows the exact steps of Jiang et al. Afterward, a rotation and sum operation is performed on the resulting matrix again to replicate the same matrix, ensuring that the square matrix is packed with repeated matrices corresponding to the number of heads. For query matrix, for instance, $\bar{Q}_i^{(j)} = [Q_i^{(j)} | Q_i^{(j)}] \in \mathbb{R}^{d \times d}$ will be derived as the resulting matrix, where $Q_i^{(j)}$ is the resultant query matrix having dimension $\mathbb{R}^{d \times d/2}$.

Additionally, we note that the same input matrix X is used for the matrix multiplications in the computation of $Q_i^{(j)}$, $K_i^{(j)}$, and $V_i^{(j)}$ in multi-head attention. This allows the baby step process in equation 6, i.e., the computation of $Rot(a;i)$, to be performed only once across the three matrix multiplications, thereby further reducing the number of rotations. The process described above constitutes our proposed algorithm, which we name ConstRecMatrixMult.

3.2 Constant Matrix Multiplication

At the final stage of multi-head attention, a $L \times L$ matrix A is multiplied by an $L \times L$ constant matrix W^O. We consider the case of multiplication of a $d \times d$ ciphertext matrix and a $d \times d$ plaintext matrix, which corresponds to the multiplication with W^O when $d = L$. Since W^O is a $d \times d$ square matrix, the situation is slightly different from above subsection. The multiplication of the square matrix W^O with A can be represented as follows:

$$A \cdot W^O = \sum_{i=0}^{d-1} (\phi^i \circ \sigma(A)) \odot (\psi^i \circ \tau(W^O)) \tag{7}$$

Using this formula, if we apply the equation in 3, it can be similarly applied as follows:

$$\sum_{i=0}^{d-1} (U^{\phi})^i U^{\sigma} a \odot w^{(i)}$$

The subsequent process proceeds similarly to the inner summation process in the above subsection. This results in approximately $4\sqrt{4d-2}$ rotations being performed. Since the outer summation is not required, it can be omitted. The process described so far is the operation of our proposed algorithm, which we refer to as ConstMatrixMult.

3.3 Blockwise Matrix Multiplication

In Sections 3.1 and 3.2, we discussed homomorphic algorithms for multiplication with weight matrices. In this subsec-

tion, we discuss a homomorphic algorithm for computation of $Q_i^{(j)} K_i^{(j)T}$ in Equation 11. This computation is divided into two parts: performing the transpose operation on $K_i^{(j)T}$ to obtain $K_i^{(j)T}$ and then multiplying $Q_i^{(j)T}$ and $K_i^{(j)T}$ to obtain each $Q_i^{(j)} K_i^{(j)T}$. We use the transpose operation as described in Jiang et al.'s method, and thus, we only discuss the multiplication of $Q_i^{(j)}$ and $K_i^{(j)T}$. We note that $Q_i^{(j)} \in \mathbb{R}^{d \times d/2}$ and $K_i^{(j)T} \in \mathbb{R}^{d/2 \times d}$ for $d = L$. In this operation, since the resulting matrix is $\mathbb{R}^{d \times d}$, one might trivially consider zero-padding each $Q_i^{(j)}$ and $K_i^{(j)T}$ to treat them as $d \times d$ matrices and perform the matrix operations using Jiang et al.'s method. However, we note that the dimension of the vectors actually being multiplied is $d/2$, and we propose a method to further optimize this computation.

First, let's split the $Q_i^{(j)}$ matrix vertically into two square matrices, denoted as $Q_{i,0}^{(j)}$ and $Q_{i,1}^{(j)} \in \mathbb{R}^{d/2 \times d/2}$. Similarly, the $K_i^{(j)}$ matrix can be split vertically into $K_{i,0}^{(j)}$ and $K_{i,1}^{(j)} \in \mathbb{R}^{d/2 \times d/2}$. Then, since $Q_i^{(j)} = \left[\begin{array}{c} Q_{i,0}^{(j)} \\ \hline Q_{i,1}^{(j)} \end{array}\right]$ and $K_i^{(j)T} = \left[K_{i,0}^{(j)T} | K_{i,1}^{(j)T}\right]$, $Q_i^{(j)} K_i^{(j)T}$ can be expressed as follows.

$$Q_i^{(j)} K_i^{(j)T} = \left[\begin{array}{c|c} Q_{i,0}^{(j)} K_{i,0}^{(j)T} & Q_{i,0}^{(j)} K_{i,1}^{(j)T} \\ \hline Q_{i,1}^{(j)} K_{i,0}^{(j)T} & Q_{i,1}^{(j)} K_{i,1}^{(j)T} \end{array}\right]$$

Since we derived square matrices by vertically copying the same matrix when multiplying by the weight matrix, $\bar{Q}_i^{(j)}$ and $\bar{K}_i^{(j)T}$ have the following forms.

$$\bar{Q}_i^{(j)} = [Q_i^{(j)} | Q_i^{(j)}] = \left[\begin{array}{c|c} Q_{i,0}^{(j)} & Q_{i,0}^{(j)} \\ \hline Q_{i,1}^{(j)} & Q_{i,1}^{(j)} \end{array}\right], \bar{K}_i^{(j)T} = \left[\begin{array}{c} K_i^{(j)T} \\ \hline K_i^{(j)T} \end{array}\right] = \left[\begin{array}{c|c} K_{i,0}^{(j)T} & K_{i,1}^{(j)T} \\ \hline K_{i,0}^{(j)T} & K_{i,1}^{(j)T} \end{array}\right]$$

In this case, $Q_i^{(j)} K_i^{(j)T}$ can be calculated by performing block-wise matrix multiplication between each of the block matrices. In other words, for $A = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right]$, $B = \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$, let us define $A \boxdot B = \left[\begin{array}{c|c} A_{00}B_{00} & A_{01}B_{01} \\ \hline A_{10}B_{10} & A_{11}B_{11} \end{array}\right]$. Then the following equation holds.

$$Q_i^{(j)} K_i^{(j)T} = \bar{Q}_i^{(j)} \boxdot \bar{K}_i^{(j)T}$$

Now, we design an efficient homomorphic operation for $A \boxdot B$. The $A \boxdot B$ operation can be transformed as follows.

$$A \boxdot B = \sum_{k=0}^{d/2-1} (\tilde{\phi}^k \circ \tilde{\sigma}(A)) \odot (\tilde{\psi}^k \circ \tilde{\tau}(B))$$

Here, for each transformation f ($f = \sigma, \tau, \phi, \psi$), $\tilde{f}$ is defined as follows.

$$\tilde{f}(A) = \left[\begin{array}{c|c} f(A_{00}) & f(A_{01}) \\ \hline f(A_{10}) & f(A_{11}) \end{array}\right]$$

The key factor in determining the computation amount at this stage is the number of non-zero diagonals in the matrix representation for each transformation. Specifically, we need to consider the representations $U^{\tilde{\sigma}}, U^{\tilde{\tau}}, \tilde{V}, \tilde{W}$ for $\tilde{\sigma}, \tilde{\tau}, \tilde{\phi}, \tilde{\psi}$. The number of non-zero diagonals in $U^{\tilde{\sigma}}$ is the same as the number of non-zero diagonals in U^{σ} for a $d/2 \times d/2$ matrix, which is $2 \cdot d/2 - 1 = d - 1$. The number of non-zero diagonals in $U^{\tilde{\tau}}$ is approximately twice that of the $U^{\tilde{\tau}}$ matrix for a $d/2 \times d/2$ matrix. This is because, in τ, only a single rotation can achieve a cyclic shift in one column, while in $\tilde{\tau}$, two types of rotations are required to rotate a single column. As a result, the number of non-zero diagonals in $U^{\tilde{\tau}}$ is the same as in $U^{\tilde{\sigma}}$, which is $2 \cdot d/2 - 1 = d - 1$. $\tilde{V}$ represents a blockwise column shifting operation and has two non-zero diagonals. $\tilde{W}$ represents a blockwise row shifting operation and, unlike W, which has one non-zero diagonal, has two non-zero diagonals. The reason is similar to that for $\tilde{\tau}$.

If the $A \boxdot B$ operation is performed using zero-padding on A and B and optimized with a baby-step giant-step approach, approximately $2\sqrt{2d} + 2\sqrt{d} + 3d = 3d + (2\sqrt{2} + 2)\sqrt{d}$ rotations are required, which is significantly less than the $2\sqrt{d} + 2\sqrt{d} + 4 \cdot d/2 = 2d + 4\sqrt{d}$ in Jiang et al.'s method. As demonstrated by the experimental results in Section 5.2, this difference is substantial in practice, and thus we employ the method of performing the $\bar{Q}_i^{(j)} \boxdot \bar{K}_i^{(j)T}$ operation to compute $Q_i^{(j)} K_i^{(j)T}$. The process described so far is the operation of the proposed algorithm, which we refer to as CIPHERTEXTMATRIXMULTBLOCK.

3.4 Parallel Homomorphic Algorithms for Matrix Computations

Jiang et al. [10] proposed a method for performing two homomorphic matrix multiplications simultaneously using parallel row-major packing. We also adopt this method to adapt the proposed matrix multiplication algorithms, which are compatible with row-major packing, to work with parallel row-major packing. This conversion process involves only the following two steps:

- Double the steps for all rotations.
- Replace the constant vectors used in plaintext-ciphertext multiplication for each of the two matrix multiplication algorithms with a new vector that alternates the values from the two original vectors. Specifically, replace the corresponding constant vectors $a = [a_0, a_1, \cdots, a_{n/2-1}]$ and $b = [b_0, b_1, \cdots, b_{n/2-1}]$ with $[a_0, b_0, a_1, b_1, \cdots, a_{n/2-1}, b_{n/2-1}]$.

For example, consider the case of modifying the CONSTMATRIXMULT algorithm to obtain the ciphertext A_1W_1/A_2W_2 when given a ciphertext in which two matrices A_1 and A_2 are packed in the form of (A_1/A_2). Recall that

the ConstMatrixMult algorithm involves computing a double summation as shown in Equation 6. Let the vector r_m for the matrix multiplication with W_1 be denoted as $r_m^{(1)}$, and the vector r_m for the matrix multiplication with W_2 be denoted as $r_m^{(2)}$. To modify the ConstMatrixMult algorithm to be compatible with parallel packing, simply replace the vector r_m in Equation 6 with the vector $r_m^{(1)}/r_m^{(2)}$, and replace the rotation steps i and $N_1 j - 2d + 1$ with $2i$ and $2(N_1 j - 2d + 1)$, respectively. We refer to the algorithms ConstRecMatrixMult, Transpose, CiphertextMatrixMultBlock, CiphertextRecMatrixMult, and ConstMatrixMult, each modified to be compatible with parallel row-major packing, as ParConstRecMatrixMult, ParTranspose, ParCiphertextMatrixMultBlock, ParCiphertextRecMatrixMult, and ParConstMatrixMult, respectively.

Additionally, we propose a method to perform homomorphic operations for matrix multiplication in the feed-forward network more efficiently by utilizing the parallel packing structure. Consider the case of multiplying a 128x128 matrix Y by a matrix $W_{F1} \in \mathbb{R}^{128\times 512}$. Let us denote the four submatrices that compose W_{F1} as $W_{F1}^{(1)}$, $W_{F1}^{(2)}$, $W_{F1}^{(3)}$, and $W_{F1}^{(4)}$, such that $W_{F1} = [W_{F1}^{(1)}|W_{F1}^{(2)}|W_{F1}^{(3)}|W_{F1}^{(4)}]$. Given the ciphertext (Y/Y) as input, we then perform ConstMatrixMult for $W_{F1}^{(1)}$ and $W_{F1}^{(2)}$, and ConstMatrixMult for $W_{F1}^{(3)}$ and $W_{F1}^{(4)}$ in parallel (i.e., by executing ParConstMatrixMult), thereby computing $(T_1/T_2) = (YW_{F1}^{(1)}/YW_{F1}^{(2)})$ and $(T_3/T_4) = (YW_{F1}^{(3)}/YW_{F1}^{(4)})$. Next, consider the case of multiplying the ciphertexts (T_1/T_2) and (T_3/T_4), which have passed through the activation function, by the second weight matrix. Let us denote the four submatrices that compose the matrix W_{F2} as $W_{F2}^{(1)}$, $W_{F2}^{(2)}$, $W_{F2}^{(3)}$, and $W_{F2}^{(4)}$, such that $W_{F2}^T = [W_{F2}^{(1)T}|W_{F2}^{(2)T}|W_{F2}^{(3)T}|W_{F2}^{(4)T}]^T$. Then, by performing the ConstMatrixMult algorithm twice in parallel (i.e., executing ParConstMatrixMult), we compute $(T_1W_{F2}^{(1)}/T_2W_{F2}^{(2)})$ and $(T_3W_{F2}^{(3)}/T_4W_{F2}^{(4)})$. What we aim to obtain is $Z = \sum_{j=0}^{3} T_j W_{F2}^{(j)}$. This can be achieved by adding the two ciphertexts to obtain the ciphertext ct and then proceeding with the following steps. $ct \leftarrow ct + Rot(ct; 1)$
$ct \leftarrow ct \odot (1_{d\times d}/0_{d\times d})$
$ct \leftarrow ct + Rot(ct; -1)$

3.5 Microbenchmarks

In this section, we compare the performance of the proposed homomorphic matrix algorithm with other existing algorithms. One of the effective metrics for estimating the complexity is the number of key-switching operations. Therefore, we compare the number of key-switching operations used by each algorithm by counting the number of non-scalar multiplications, rotations, and substitutions [28] that involve key-switching.

Table 3: Comparison of the number of key switchings across different homomorphic matrix multiplication algorithms. The specific number of key-switchings for the parameters $d = 128$ and $\ell = 64$ are also provided.

		#key-switch	
		equation	value
Case 1	NEXUS	$2d\ell$	16384
	Jiang et al.	$4\ell + 2\sqrt{2d} + 2\sqrt{d} + \log(d/\ell)$	312
	BOLT	$2\sqrt{d}$	23
	Powerformer	$2\sqrt{3d+\ell} + \log(d/\ell)$	44
Case 2	NEXUS	d^2	16384
	Jiang et al.	$4d + 2\sqrt{2d} + 2\sqrt{d}$	567
	BOLT	$d\log d + 2d$	1152
	Powerformer	$5d/2 + 4\sqrt{d}$	366
Case 4	NEXUS	$2d^2$	32768
	Jiang et al.	$4d + 2\sqrt{2d} + 2\sqrt{d}$	567
	BOLT	$2\sqrt{d}$	23
	Powerformer	$4\sqrt{d}$	46
MHA	NEXUS	$\geq 6d^2$	$\geq$ 98304
	Jiang et al.	$27d/2 + 12\sqrt{2d} + 12\sqrt{d}$	2380
	BOLT	$3d\log d + 4d + 8\sqrt{d}$	3291
	Powerformer	$4d + 3\sqrt{14d} + 8\sqrt{d} + 2\sqrt{2d}$	853

Table 3 presents the microbenchmarks for the homomorphic matrix algorithms. In this table, the number of key-switching operations is based on the BERT-tiny model we are targeting, with values of $d = 128$ and $\ell = 64$.

The matrix multiplication operations required for multi-head attention can be categorized into four cases:

- Case 1: Calculation of Q_i, K_i, V_i as shown in Equation 10, which corresponds to the multiplication of a $d \times d$ ciphertext matrix and a $d \times \ell$ plaintext matrix.
- Case 2: Calculation of $Q_i K_i^T$ as shown in Equation 11, which corresponds to the multiplication of a $d \times d/2$ ciphertext matrix and a $d/2 \times d$ ciphertext matrix.
- Case 3: Calculation of multiplication by V_i as shown in Equation 12, which corresponds to the multiplication of a $d \times d$ ciphertext matrix and a $d \times d/2$ ciphertext matrix.
- Case 4: Calculation of multiplication by W^O, which corresponds to the multiplication of a $d \times d$ ciphertext matrix and a $d \times d$ plaintext matrix.

In Case 3, we use the matrix multiplication technique proposed by Jiang et al. [10], while for Cases 1, 2, and 4, we propose new optimized algorithms. Table 3 compares the number of key switchings for Powerformer with other algorithms for Cases 1, 2, and 4, and also provides the specific number of key switchings for BERT-tiny ($d = 128, \ell = 64$).

It is important to note that the packing methods compatible with each algorithm differ. Both Jiang et al. and Powerformer

use row-major packing, which allows for a direct performance comparison in each case. As shown in Table 3, Powerformer outperforms the technique by Jiang et al. in all cases. On the other hand, NEXUS uses a combination of component-wise, row-wise, and column-wise packing, while BOLT employs both column-major packing and diagonal-major packing. One effective way to fairly compare the number of key switchings between Powerformer and NEXUS or BOLT, which use different packing methods, is to compare the total number of key switchings consumed in multi-head attention. In this comparison, we exclude the key switchings required for softmax, as our focus is on comparing matrix multiplication algorithms. It can be observed that Powerformer uses the fewest key switchings in multi-head attention. Since NEXUS does not provide a specific matrix multiplication algorithm for Case 3, we excluded Case 3 when calculating the number of key switchings used in multi-head attention in NEXUS. Nonetheless, NEXUS still consumes the most key switchings.

4 Implementation of Homomorphic Transformer Encoder

In this section, we propose a homomorphic Transformer encoder architecture. We target the Bert-tiny model as described in Figure 3. Our goal is to present an architecture for a homomorphic Transformer encoder that can be implemented in RNS-CKKS, utilizing the proposed RP function and optimized homomorphic matrix algorithms.

4.1 Parallel Row-Major Packing

Since the embedding dimension d and sequence length L of the BERT-tiny model we are targeting are both 128, the dimension of the input matrix X is 128×128, consisting of a total of 2^{14} real numbers. First, we need to decide how to pack the input matrix X into a ciphertext in the form of a 1-dimensional vector. As shown in Table 3, since the proposed matrix multiplication algorithms perform multi-head attention the fastest while being compatible with row-major packing, it is desirable to adopt row-major packing. However, since we set the polynomial modulus degree N to 2^{16}, the number of slots n allocated to one ciphertext is 2^{15}, which is twice the number of elements in X. Thus, if we use row-major packing, extra slots will be left unused.

One way to pack this input matrix into a ciphertext is to pack the values of X in the first half of the ciphertext using row-major packing and fill the remaining half with zeros. However, this approach is inefficient as it wastes half of the ciphertext slots. Therefore, we use parallel row-major packing [10] to pack the input matrix X as (X/X), ensuring that all intermediate matrices of the Transformer inference are always packed into ciphertexts using parallel row-major packing. This allows two matrix multiplications to be performed in parallel, reducing the computation time. Specifically, we note that the multi-head attention in the BERT-tiny model has 2 heads. Originally, the ConstRecMatrixMult, Transpose, CiphertextMatrixMultBlock, and CiphertextRecMatrixMult algorithms need to be executed twice in total—once for each head—but by using parallel row-major packing, they can be executed in parallel just once.

Additionally, the row-major packing we use has the advantage of compactly storing data by utilizing all slots of the ciphertext, which can significantly reduce the number of bootstrappings required. In PowerTransformer, all bootstrappings are performed on a single ciphertext. For example, if we assume bootstrapping is performed right after layer normalization, the size of the intermediate matrix is 128×128. In NEXUS, which uses row-wise or column-wise packing, a total of 128 bootstrappings would be required, whereas PowerTransformer requires only a single bootstrapping, making it $128\times$ more efficient.

Table 4: Comparison between GELU + BRPmax, ReLU + BRPmax, and Original BERT (%)

	GELU + RPmax	ReLU + RPmax	Original
CoLA	60.33	59.33	58.80
MRPC	86.52	86.27	85.54
QNLI	91.52	90.99	91.10
RTE	68.95	70.40	70.04
STS-B	89.40	89.24	89.82
Average	79.35	79.25	79.06

4.2 GELU and Layer Normalization

The GELU function in the BERT model is a nonlinear function, and as such, several studies have explored approximating GELU [6, 21, 28]. In NEXUS [28], GELU is approximated using a piecewise polynomial, which requires computing an approximation polynomial for the sign function three times. We simply replace GELU with a ReLU approximation polynomial [18], which requires only one computation of the approximation polynomial for the sign function, resulting in a shorter computation time compared to NEXUS. According to Table 4, the ReLU used with RPmax also outperforms the original model with GELU and original softmax function, while showing an acceptable performance loss of 0.1% compared to the model with GELU and RPmax. Therefore, to gain a time advantage, we used ReLU in the Feed Forward Network. We also confirmed through experiments that replacing ReLU does not lead to any loss in accuracy.

We approximate layer normalization in a manner similar to [28]. The nonlinear function $y = \frac{1}{\sqrt{x}}$ is approximated using an iterative algorithm based on Newton's method [25]. Also, we numerically confirm that when applying both the algorithm based on Newton's method and the precise ReLU approximation, there is no change in accuracy in plaintext.

4.3 Workflow of the Proposed Architecture

Now, based on the components described so far, we will outline the workflow of the proposed homomorphic Transformer architecture.

- First, the client encrypts the input matrix X using the parallel row-major packing. This ciphertext stores the data in the form of $ct_x = (X/X)$. The client then sends this ciphertext to the server, and from that point onward, after which the server takes over the computations.
- The PARCONSTRECMATRIXMULT algorithm is executed three times on the input ciphertext in the form of (X/X) to compute $([Q_{i_0}|Q_{i_0}]/[Q_{i_1}|Q_{i_1}])$, $([K_{i_0}|K_{i_0}]/[K_{i_1}|K_{i_1}])$, and $([V_{i_0}|V_{i_0}]/[V_{i_1}|V_{i_1}])$.
- Using the ciphertext $([K_{i_0}|K_{i_0}]/[K_{i_1}|K_{i_1}])$ as input, the PARTRANSPOSE algorithm is performed to compute $([K_{i_0}|K_{i_0}]^T/[K_{i_1}|K_{i_1}]^T)$.
- The PARCIPHERTEXTMATRIXMULTBLOCK algorithm is performed on the two ciphertexts $([Q_{i_0}|Q_{i_0}]/[Q_{i_1}|Q_{i_1}])$ and $([K_{i_0}|K_{i_0}]^T/[K_{i_1}|K_{i_1}]^T)$ to compute $(Q_{i_0}K_{i_0}^T/Q_{i_1}K_{i_1}^T)$.
- The PARCIPHERTEXTRECMATRIXMULT algorithm is performed on the two ciphertexts $(Q_{i_0}K_{i_0}^T/Q_{i_1}K_{i_1}^T)$ and $([V_{i_0}|V_{i_0}]/[V_{i_1}|V_{i_1}])$, followed by a component-wise approximate softmax operation. This results in the following ciphertext: $[\text{BRPmax}(Q_{i_0}K_{i_0}^T)V_{i_0}|\text{BRPmax}(Q_{i_0}K_{i_0}^T)V_{i_0}]/$ $[\text{BRPmax}(Q_{i_1}K_{i_1}^T)V_{i_1}|\text{BRPmax}(Q_{i_1}K_{i_1}^T)V_{i_1}]$.
- Let the current ciphertext be denoted as ct. To perform concatenation on this ciphertext, the following operation is performed:
  $ct \leftarrow ct \odot ([1_{d\times d/2}|0_{d\times d/2}]/[0_{d\times d/2}|0_{d\times d/2}]) + Rot(ct \odot ([0_{d\times d/2}|0_{d\times d/2}]/[1_{d\times d/2}|0_{d\times d/2}]); -(d-1))$
  $ct \leftarrow ct + Rot(ct; -1)$
  This results in the following ciphertext:
  $[\text{BRPmax}(Q_{i_0}K_{i_0}^T)V_{i_0}|\text{BRPmax}(Q_{i_1}K_{i_1}^T)V_{i_1}]/$
  $[\text{BRPmax}(Q_{i_0}K_{i_0}^T)V_{i_0}|\text{BRPmax}(Q_{i_1}K_{i_1}^T)V_{i_1}]$.
- The PARCONSTMATRIXMULT algorithm is executed to obtain the following result:
  $([\text{BRPmax}(Q_{i_0}K_{i_0}^T)V_{i_0}|\text{BRPmax}(Q_{i_1}K_{i_1}^T)V_{i_1}]W^O/$
  $[\text{BRPmax}(Q_{i_0}K_{i_0}^T)V_{i_0}|\text{BRPmax}(Q_{i_1}K_{i_1}^T)V_{i_1}]W^O)$.
- An approximate layer normalization is performed, which we will denote as (Z/Z).
- Using the PARCONSTMATRIXMULT algorithm twice on the input (Z/Z), we compute the following results, respectively. $(T_1/T_2) = (ZW_1/ZW_2)$
  $(T_3/T_4) = (ZW_3/ZW_4)$.
- An approximate ReLU is applied to each of the two ciphertexts.
  $(T_1/T_2) \leftarrow (ReLU(T_1)/ReLU(T_2))$
  $(T_3/T_4) \leftarrow (ReLU(T_3)/ReLU(T_4))$
- The PARCONSTMATRIXMULT algorithm is applied to the two inputs (T_1/T_2) and (T_3/T_4), resulting in (T_1V_1/T_2V_2) and (T_3V_3/T_4V_4), respectively. These are then added together to compute $(T_1V_1 + T_3V_3/T_2V_2 + T_4V_4)$. After that, the following steps are performed.
  $ct \leftarrow ct + Rot(ct; 1)$
  $ct \leftarrow ct \odot (1_{d\times d}/0_{d\times d})$
  $ct \leftarrow ct + Rot(ct; -1)$
  This finally results in $(T_1V_1 + T_2V_2 + T_3V_3 + T_4V_4/T_1V_1 + T_2V_2 + T_3V_3 + T_4V_4)$, which is the output after applying the feed-forward network.
- Perform approximate layer normalization.

The above steps represent the workflow for a single encoder layer, and repeating this process twice will complete the computation for the entire BERT-tiny model. After the server completes the computations, it sends the ciphertext, where the matrix Y is packed in the form of (Y/Y), back to the client. The client can then decrypt this to obtain the result matrix Y.

5 Experimental Results

5.1 BRPmax Performance Evaluation

We used a learning rate of 3×10^{-5} for fine-tuning in downstream tasks, and a learning rate of 1×10^{-5} for distillation learning. In all settings, we applied a batch size of 32, AdamW optimizer with a weight decay of 1×10^{-2}, and early stopping after 10 rounds based on the validation set. All models were evaluated on five tasks from the GLUE benchmark: RTE, CoLA, MRPC, STS-B, and QNLI, to compare their performance.

The effectiveness of BRPmax was validated through the comparison in Table 5. We experiment with BERT models replacing the softmax function with each following function used in the previous works and our proposed models. After fine-tuning each modified model, we compare the accuracy and the number of epochs used for each task.

- **2Quad**: The $\text{2Quad}(\{x_i\}) = \frac{(x_i+c)^2}{\sum_j (x_j+c)^2}$ function used in SecFormer [22]
- **ZBD+ & Distill**: The $\text{ReLU}(\{x_i\})/\sqrt{L}$ function used by Zimerman et al [29] (Although distillation was not used in Zimerman et al.'s paper, we use distillation in this experiment.)
- **BRPmax$_{1,0}$**: The BRPmax function with basic parameters $p = 1$ and $c = 0$
- **BRPmax$_{5,5}$**: The BRPmax function with parameters $p = 5$ and $c = 5$

Figure 1 focuses on comparing the average accuracy difference with the original BERT-base model and average epochs

in the fine-tuning for each modified model. Table 6 is the result of the comparison of BRPmax and original BERT in the BERT-Base and BERT-Large to demonstrate the performance of BPRmax method in the larger model.

Table 5 and Figure 1 show that replacing the function with the 2Quad function resulted in the lowest performance, with a noticeable decrease in accuracy. While the method of Zimerman et al., when used with distillation, mitigated the significant accuracy drop seen in Table 1 to some extent, there was still a meaningful decrease in accuracy. The $\text{BRPmax}_{1,0}$ method is similar to Zimerman et al.'s approach in that it uses the ReLU function, except that it differs in applying the batch method. Even by applying the batch method alone, we can see overall better performance. Additionally, the $\text{BRPmax}_{5,5}$ function, which applies the power of ReLU and adjusts the offset accordingly, shows a significant increase in accuracy on average. The number of epochs required for fine-tuning is also noticeably reduced when using BRPmax, indicating that the computational load for model fine-tuning is also improved. The $\text{BRPmax}_{p,c}$ model is the only distillation method that outperforms the original model and consumes the fewest epochs to converge. Additionally, since the parameter p, which can be tuned task-wise, has not been fully optimized, there is room for further performance improvements.

Table 6 presents the efficiency of BRPmax in both BERT-base and BERT-large models. Transformer models are primarily used in large-scale LLMs, and increasingly larger models are being developed. Given this trend, models used for inference need to perform well even with deeper architectures. The BRPmax function also surpasses the performance of the original model in the BERT-large model, which has 336M parameters.

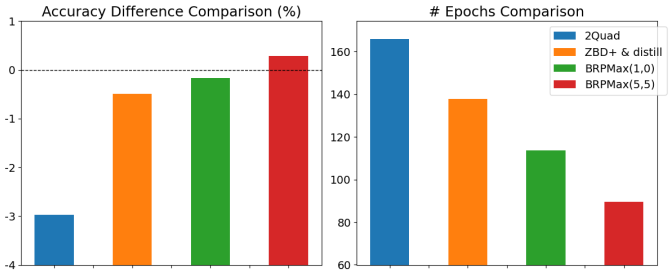

Figure 1: Average Accuracy Difference with Original BERT and Epochs comparison

5.2 Evaluation of Homomorphic Matrix Multiplication

In sections 5.2 and 5.3, we provide experimental results on RNS-CKKS. We use a polynomial degree of $N = 2^{16}$ and $n = 2^{15}$ slots. The sparse secret encapsulation method [2] is employed. The Hamming weight of the main secret key is set to 192, and the Hamming weight of the ephemeral secret is set to 32. The default scaling factor is 2^{45}, and 55-bit primes are used for both the special modulus and base modulus. The number of special moduli is 3. The CoeffsToSlots and SlotsToCoeffs procedures are performed with three levels using the level collapsing technique. The approximation polynomial degrees for the cosine function and inverse sine function are 63 and 1, respectively. The number of double-angle formulas is 2. The available number of levels is 16. The total modulus used is $55+45\times16+55\times3+55\times14 = 1,710$ bits, which achieves 128-bit security under the sparse secret encapsulation method. The simulation is conducted using Lattigo [1], a representative RNS-CKKS library, on AMD Ryzen Threadripper PRO 5995WX at 2.7GHz with 512 GB RAM, running the Ubuntu 20.04 operating system.

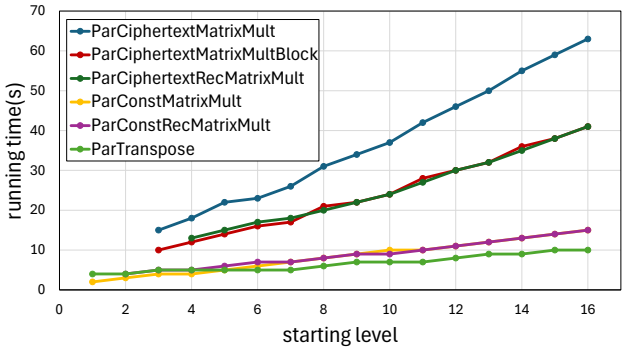

Figure 2: Runtimes of various homomorphic algorithms for matrix multiplication and Matrix Transposition at different levels.

Figure 2 compares the runtimes of the matrix operation algorithms from [10] and the proposed algorithms at various starting levels. We used the same parameters and also consistently used the Lattigo library to ensure a fair comparison.

For matrix transposition, we use the ParTranspose algorithm, which is identical to the matrix transposition algorithm presented in [10]. For matrix multiplication, while [10] uses the ParCiphertextMatrixMult algorithm, we use the ParCiphertextMatrixMultBlock, ParCiphertextRecMatrixMult, ParConstMatrixMult, and ParConstRecMatrixMult algorithms depending on the situation. Figure 2 shows that using the proposed algorithms significantly reduces the runtimes for matrix multiplication.

Meanwhile, one notable observation in Figure 2 is that the runtime of the algorithms increases almost linearly with each level. This is because, in the RNS-CKKS algorithm, the number of RNS moduli used increases linearly with the levels. Therefore, we designed the homomorphic Transformer architecture so that the matrix multiplication algorithms would operate at the lowest possible level.

Table 5: Comparison of accuracy(%) and epochs across distill methods in BERT-Base

	RTE		CoLA		MRPC		STS-B		QNLI		Average	
	accs	epochs	accs	epochs	accs	epochs	accs	epochs	accs	epochs	accs	epochs
2Quad	64.62	257	55.24	100	82.11	182	87.54	183	90.90	78	76.08	160
ZBD+ & Distill	70.04	184	57.53	100	85.54	134	88.72	196	91.01	45	78.57	131.8
$\text{BRPmax}_{1,0}$	70.04	143	58.55	100	86.03	108	88.83	88	91.03	99	78.89	107.6
$\text{BRPmax}_{5,5}$	68.95	103	60.33	100	86.52	69	89.40	95	91.52	51	79.35	83.6
Original BERT	70.04	-	58.80	-	85.54	-	89.82	-	91.10	-	79.06	-

Table 6: Comparison of BRPmax and Original BERT in BERT-Base and BERT-Large (%)

		BRPmax	Original	difference
CoLA	BERT-Base	60.33	58.80	1.53
	BERT-Large	64.07	64.33	-0.26
MRPC	BERT-Base	86.52	85.54	0.98
	BERT-Large	88.24	87.25	0.98
QNLI	BERT-Base	91.52	91.10	0.42
	BERT-Large	92.49	91.73	0.77
RTE	BERT-Base	68.95	70.04	-1.08
	BERT-Large	74.73	75.45	-0.72
STS-B	BERT-Base	89.40	89.82	-0.41
	BERT-Large	90.42	90.53	-0.12
Average	BERT-Base	79.35	79.06	0.29
	BERT-Large	81.99	81.86	0.13

5.3 Evaluation of the proposed Bert-Tiny Architecture

We implemented the homomorphic Transformer architecture of the Bert-tiny model on RNS-CKKS. We use the STSB (Semantic Textual Similarity Benchmark) from the GLUE tasks as our dataset. Table 7 shows the runtime for each component of the proposed Transformer architecture Powerformer for the Bert-Tiny model. The total inference time is 503 seconds. The approximate ReLU entry excludes the time spent performing ReLU operations during approximate softmax. As shown in Table 7, bootstrapping operations dominate the overall runtime. The relatively lower runtime of the homomorphic matrix algorithms compared to bootstrapping is due to the architecture being optimized to perform homomorphic matrix algorithms at the lowest possible level.

Additionally, Table 7 also provides a comparison with NEXUS. NEXUS does not provide an end-to-end implementation, so Table 7 do not include their end-to-end results. However, in the case of BERT-Tiny, NEXUS involves seven bootstrapping stages, and each stage requires 128 bootstrappings, resulting in a total of $7 \times 128 = 864$ bootstrappings. Thus, we can estimate the total bootstrapping time. It should be noted that bootstrapping in the SEAL library [26] used by NEXUS is slower than that in Lattigo [1]. Therefore, for a fair comparison, we estimated the bootstrapping time using the same library, Lattigo, and the same parameters as ours.

Table 7: Comparison of the execution time per component between the NEXUS for the Bert-Tiny model and the proposed Powerformer architecture. The asterisk(*) indicates an estimated value.

component		#call	time(s)
Powerformer	ParConstRecMatrixMult	6	49
	ParTranspose	2	11
	ParCiphertextMatrixMultBlock	2	28
	bootstrapping	11	243
	approximate ReLU	4	23
	approximate softmax	2	13
	ParCiphertextRecMatrixMult	2	34
	approximate layer normalization	4	23
	ParConstMatrixMult	10	79
	total		**503**
NEXUS	bootstrapping	896	19793*
	total		$\geq$19793*

Even excluding the execution times of matrix multiplication algorithms, softmax, and other operations in NEXUS, it is evident that our implementation is significantly faster.

One notable point is that we achieved 503 seconds using a single CPU thread, which could be reduced by tens to hundreds of times with the use of GPU accelerators or hardware accelerators [12, 14]. The NEXUS paper utilizes multi-threading, but to be accelerated by a GPU or hardware accelerator, it must eventually be implemented as a single thread before conversion.

6 Conclusion

We proposed an efficient non-interactive privacy-preserving Transformer inference architecture. We suggested replacing softmax with the proposed BRPmax function, which maintains accuracy while requiring small computation amount. We presented matrix-matrix multiplication algorithms specialized for attention operation that reduce the number of key-switchings significantly. Our implementation of Powerformer on the BERT-tiny model using RNS-CKKS scheme completed inference in 503 seconds on a single-threaded CPU, marking a significant milestone as the first end-to-end non-interactive transformer implementation using HE.

References

[1] Lattigo v5. Online: https://github.com/tuneinsight/lattigo, Nov. 2023. EPFL-LDS, Tune Insight SA.

[2] Bossuat, J.-P., Troncoso-Pastoriza, J., and Hubaux, J.-P. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In *International Conference on Applied Cryptography and Network Security* (2022), Springer, pp. 521–541.

[3] Brutzkus, A., Gilad-Bachrach, R., and Elisha, O. Low latency privacy preserving inference. In *International Conference on Machine Learning* (2019), PMLR, pp. 812–821.

[4] Chen, T., Bao, H., Huang, S., Dong, L., Jiao, B., Jiang, D., Zhou, H., Li, J., and Wei, F. The-x: Privacy-preserving transformer inference with homomorphic encryption. In *Findings of the Association for Computational Linguistics: ACL 2022* (2022), pp. 3510–3520.

[5] Cheon, J. H., Han, K., Kim, A., Kim, M., and Song, Y. A full rns variant of approximate homomorphic encryption. In *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25* (2019), Springer, pp. 347–368.

[6] Dong, Y., Lu, W.-j., Zheng, Y., Wu, H., Zhao, D., Tan, J., Huang, Z., Hong, C., Wei, T., and Chen, W. Puma: Secure inference of llama-7b in five minutes. *arXiv preprint arXiv:2307.12533* (2023).

[7] Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., and Wernsing, J. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning* (2016), PMLR, pp. 201–210.

[8] Halevi, S., and Shoup, V. Algorithms in helib. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34* (2014), Springer, pp. 554–571.

[9] Hao, M., Li, H., Chen, H., Xing, P., Xu, G., and Zhang, T. Iron: Private inference on transformers. *Advances in neural information processing systems 35* (2022), 15718–15731.

[10] Jiang, X., Kim, M., Lauter, K., and Song, Y. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security* (2018), pp. 1209–1222.

[11] Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351 2019* (2019).

[12] Jung, W., Kim, S., Ahn, J. H., Cheon, J. H., and Lee, Y. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 114–148.

[13] Juvekar, C., Vaikuntanathan, V., and Chandrakasan, A. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX security symposium (USENIX security 18)* (2018), pp. 1651–1669.

[14] Kim, S., Kim, J., Kim, M. J., Jung, W., Kim, J., Rhu, M., and Ahn, J. H. Bts: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th annual international symposium on computer architecture* (2022), pp. 711–725.

[15] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436–444.

[16] Lee, E., Lee, J.-W., Lee, J., Kim, Y.-S., Kim, Y., No, J.-S., and Choi, W. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *International Conference on Machine Learning* (2022), PMLR, pp. 12403–12422.

[17] Lee, E., Lee, J.-W., No, J.-S., and Kim, Y.-S. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE Transactions on Dependable and Secure Computing 19*, 6 (2021), 3711–3727.

[18] Lee, J., Lee, E., Lee, J.-W., Kim, Y., Kim, Y.-S., and No, J.-S. Precise approximation of convolutional neural networks for homomorphically encrypted data. *IEEE Access 11* (2023), 62062–62076.

[19] Lee, J.-W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.-S., et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *iEEE Access 10* (2022), 30039–30054.

[20] Li, D., Wang, H., Shao, R., Guo, H., Xing, E. P., and Zhang, H. MPCFORMER: fast, performant and provate transformer inference with MPC. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023* (2023).

[21] Lu, W.-j., Huang, Z., Gu, Z., Li, J., Liu, J., Ren, K., Hong, C., Wei, T., and Chen, W. Bumblebee: Secure two-party inference framework for large transformers. *Cryptology ePrint Archive* (2023).

[22] Luo, J., Zhang, Y., Zhang, Z., Zhang, J., Mu, X., Wang, H., Yu, Y., and Xu, Z. SecFormer: Fast and accurate privacy-preserving inference for transformer models via SMPC. In *Findings of the Association for Computational Linguistics ACL 2024* (2024).

[23] Pang, Q., Zhu, J., Möllering, H., Zheng, W., and Schneider, T. Bolt: Privacy-preserving, accurate and efficient inference for transformers. *Cryptology ePrint Archive* (2023).

[24] Park, J., Kim, M. J., Jung, W., and Ahn, J. H. Aespa: Accuracy preserving low-degree polynomial activation for fast private inference. *arXiv preprint arXiv:2201.06699* (2022).

[25] Qu, H., and Xu, G. Improvements of homomorphic secure evaluation of inverse square root. In *International Conference on Information and Communications Security* (2023), Springer, pp. 110–127.

[26] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, Jan. 2023. Microsoft Research, Redmond, WA.

[27] Srinivasan, W. Z., Akshayaram, P., and Ada, P. R. Delphi: A cryptographic inference service for neural networks. In *Proc. 29th USENIX secur. symp* (2019), vol. 3.

[28] Zhang, J., Liu, J., Yang, X., Wang, Y., Chen, K., Hou, X., Ren, K., and Yang, X. Secure transformer inference made non-interactive. *Cryptology ePrint Archive* (2024).

[29] Zimerman, I., Baruch, M., Drucker, N., Ezov, G., Soceanu, O., and Wolf, L. Converting transformers to polynomial form for secure inference over homomorphic encryption. *arXiv preprint arXiv:2311.08610* (2023).

A Related Works

Softmax The softmax operation in Equation 1 requires exponential function and division operations, which are not supported by HE. Therefore, several studies have attempted to replace these operations with polynomial approximations [4, 28, 29].

Zimerman et al. [29] approximate the softmax function using the following equation, where L is the length of the sequence:

$$\frac{\mathrm{ReLU}(x)}{\sqrt{L}} \tag{8}$$

They approximate the ReLU operation using the method by Lee et al. [18]. However, as mentioned in Chapter 3, actual

experiments show that this approach can lead to the significant accuracy loss or even failure to converge.

Another method for approximating softmax uses the following equation [4]:

$$S(x_i) = x_i \cdot T\left(\sum_j \mathrm{ReLU}\left(\left(\frac{x_j}{2}+1\right)^3\right)\right) \tag{9}$$

Here, T is a 3-layer neural network that includes two activation functions. Assuming that each activation function requires, on average, one bootstrapping, this method is expected to need at least three bootstrappings. In contrast, our proposed method requires only one bootstrapping.

As previously mentioned, [28] replaces x_{max} with an appropriate constant during inference when performing Equation 1. This replacement constant must be predetermined as a sufficiently large value. However, this can cause the $exp(x_i - x_{max})$ value to become very small, resulting in extremely low values that resemble noise in RNS-CKKS, which uses fixed-point arithmetic, potentially causing numerical issues. Our proposed method is safe from such numerical issues.

In interactive PPML studies [9, 23], softmax is typically performed using MPC techniques. However, this approach does not align with our goal of performing softmax non-interactively without communication.

Homomorphic Matrix Multiplication Transformer inference on encrypted data requires both ciphertext-plaintext matrix multiplication and ciphertext-ciphertext matrix multiplication. Since RNS-CKKS supports element-wise operations and rotations on one-dimensional vectors, it is necessary to utilize these features to implement matrix multiplication.

Jiang et al. [10] implement ciphertext-ciphertext matrix multiplication using matrix permutations and element-wise multiplications to compute the product of two matrices A and B. Matrix permutations can be expressed as matrix-vector multiplications, which are performed using the method by Halevi and Shoup [8]. While we adopt this idea, Transformers also require plaintext-ciphertext matrix multiplication, for which we further optimize the algorithm. Additionally, we propose a blockwise multiplication algorithm optimized for the $d \times d/2$ and $d/2 \times d$ matrix multiplications required in the implementation of this Transformer architecture.

The ciphertext-plaintext matrix multiplication described by NEXUS [28] requires communication between the client and server, which does not align with the non-interactive nature of most ciphertext-plaintext matrix multiplications in Transformers beyond the initial one. Furthermore, when using the ciphertext-ciphertext matrix multiplication method proposed by NEXUS to multiply the query matrix and key matrix, it requires $(d/2)^2$ rotations, which is far less efficient compared to the $2d + 4\sqrt{d}$ rotations required by our proposed method.

The authors of [4, 29] did not provide an algorithm for matrix multiplication. In interactive PPML, it is common to implement matrix multiplication using MPC, but in BOLT [23], matrix multiplication is performed on encrypted data using HE, which can be compatible with non-interactive PPML. While the ciphertext-plaintext matrix multiplication proposed in BOLT is efficient, the ciphertext-ciphertext matrix multiplication requires $d \log d$ rotations. Considering the use of $d = 128$, this is less efficient compared to our proposed method, which requires $2d + 4\sqrt{d}$.

Non-Interactive Transformer Inference Architecture As previously mentioned, research on secure Transformer inference is divided into studies that use interactive approaches [4, 6, 9, 20, 21, 23] and those that use non-interactive approaches [28, 29]. Interactive methods generally offer faster execution times but require significant communication. In contrast, non-interactive methods allow the server to perform the entire inference process on encrypted data without online communication, making them advantageous in environments with limited communication capabilities. Although non-interactive methods may take longer to execute, they can be accelerated by tens to hundreds of times using GPUs or hardware accelerators.

Additionally, [29] does not provide specific implementation details, such as matrix multiplication methods, and while [28] emphasizes its non-interactive nature, the plaintext-ciphertext matrix multiplication used in that study requires communication, which compromises the non-interactive property when executing architectures with multiple encoder layers. Therefore, our work represents the first fully non-interactive secure Transformer architecture, and we plan to release our end-to-end source code publicly.

B Preliminaries

B.1 Transformer

In this paper, we focus on homomorphically implementing a Transformer-based model, BERT (Bidirectional Encoder Representations from Transformers) using the RNS-CKKS scheme. The BERT-tiny model consists of two identical encoder blocks, where each encoder block sequentially performs multi-head attention, layer normalization, feed-forward network, and layer normalization.

First, the input sentence is tokenized, and each token undergoes an embedding process to become a fixed-size vector. After embedding, we obtain the $L \times d_m$ matrix X, which serves as the input to the first encoder block. The multi-head attention mechanism has h heads, and for each head, the query, key, and value matrices are computed by multiplying the input matrix X with the corresponding weight matrices. If the query, key, and value weight matrices for head j ($j = 0, 1, \cdots, h-1$) are denoted as $W_{Q_i}^{(j)}$, $W_{K_i}^{(j)}$, and $W_{V_i}^{(j)} \in \mathbb{R}^{d_m \times d_m/h}$, respectively, the following matrix multiplications need to be performed:

$$Q_i^{(j)} = XW_{Q_i}^{(j)}, K_i^{(j)} = XW_{K_i}^{(j)}, V_i^{(j)} = XW_{V_i}^{(j)}. \quad (10)$$

For each head, the following $L \times L$ matrix is computed:

$$\frac{Q_i^{(j)} K_i^{(j)T}}{\sqrt{d/2}} \quad (11)$$

Next, apply softmax and multiply by $V_i^{(j)}$ to obtain the following $L \times d_m/h$ matrix:

$$Y_j = softmax\left(\frac{Q_i^{(j)} K_i^{(j)T}}{\sqrt{d/2}}\right) V_i^{(j)}. \quad (12)$$

The Y_j matrices for the multiple heads are concatenated horizontally to form the $L \times d_m$ matrix $[Y_0|Y_1|\cdots|Y_{h-1}]$. After that, the weight matrix W^O is multiplied on the right, and according to the skip connection, matrix X is added, resulting in $[Y_0|Y_1|\cdots|Y_{h-1}]W^O + X$, which completes the multi-head attention process.

Next, layer normalization is performed to obtain the matrix Y. In the subsequent feed-forward network, the weight matrix $W_{F1} \in \mathbb{R}^{d_m \times d_h}$ is first multiplied to obtain YW_{F1}, followed by applying GELU and then multiplying by the second weight matrix $W_{F2} \in \mathbb{R}^{d_h \times d_m}$ on the right. After that, layer normalization is performed. The process described so far constitutes one encoder layer, and the BERT model repeats this encoder layer several times with the same structure, though with different weight parameters. In this paper, our homomorphic implementation focuses on the BERT-tiny model, which has parameters $d_m = L = 128$, $h = 2$, and $d_h = 512$. Figure 3 shows the architecture of one encoder block in the BERT-tiny model.

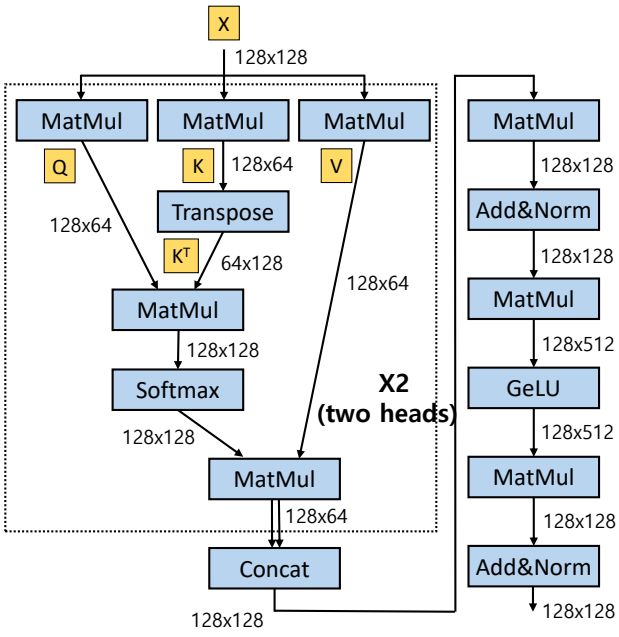

Figure 3: Overview of one encoder block of BERT-tiny Transformer architecture.

B.2 RNS-CKKS Fully HE

RNS-CKKS is a fully HE scheme that supports fixed-point arithmetic on encrypted data. In the RNS-CKKS scheme, a ciphertext is represented as a pair $(b,a) \in R_Q^2$ with respect to some large integer Q, where $R_Q = \mathbb{Z}_Q[X]/(X^N+1)$. A real vector $u \in \mathbb{R}^{N/2}$ is encrypted into a single ciphertext, and we denote the number of slots $N/2$ as n. Homomorphic operations are performed simultaneously on each slot in parallel. If the ciphertext of a vector $u \in \mathbb{R}^n$ is simply denoted as $[u]$, then homomorphic addition, scalar multiplication, non-scalar multiplication, and rotation can be described as follows:

- $[u] \oplus [v] = [u+v]$
- $[u] \otimes v = u \otimes [v] = [u \odot v]$
- $[u] \boxtimes [v] = [u \odot v]$
- Rot($[u]$;r) = $[\rho(u;r)]$,

where $u \odot v$ denotes Hadamard multiplication, i.e., component-wise multiplication, and $\langle u \rangle_r$ denotes the vector u that has been cyclically shifted to the left by r positions. For simplicity, the RNS-CKKS approximation errors have been omitted from the above equations.

Each ciphertext has a parameter ℓ called the level, which represents the number of consecutive homomorphic multiplications it can perform. With each multiplication operation, the level decreases by one, and once it reaches zero, no further homomorphic multiplications can be performed. At this point, a bootstrapping process must be performed to restore the ciphertext to a higher level.

The $m \times m$ matrix $A = (a_{ij})$ should be packed into a one-dimensional vector to be stored in a ciphertext, and there are various methods for matrix packing. In row-wise packing [28], each row of the matrix is packed into a single ciphertext as follows: $\{Enc(a_{i,1}, a_{i,2}, \cdots, a_{i,m})\}_{i=1}^m$. Similarly, in column-wise packing [28], the matrix is packed as follows: $\{Enc(a_{1,j}, a_{2,j}, \cdots, a_{m,j})\}_{j=1}^m$. In row-major packing [10, 13], all rows of the matrix are packed into a single ciphertext by concatenating all row vectors in order as follows: $Enc(a_{1,1}, a_{1,2}, \cdots, a_{1,m}, a_{2,1}, \cdots, a_{m,m})$. Similarly, in column-major packing [23], the matrix is packed as follows: $Enc(a_{1,1}, a_{2,1}, \cdots, a_{m,1}, a_{1,2}, \cdots, a_{m,m})$. In diagonal-major packing [23], all diagonals of the matrix are packed into a single ciphertext by concatenating all diagonal elements in order as follows: $Enc(a_{1,1}, a_{2,2}, \cdots, a_{m,m}, a_{1,2}, a_{2,3} \cdots, a_{m-1,m}, a_{m,1}, \cdots)$.

B.3 Homomorphic Matrix Operations

Jiang et al. [10] proposed an efficient method for performing homomorphic matrix operations. Their idea involves representing matrix multiplication as a combination of matrix permutations and Hadamard multiplication, both of which are suitable for homomorphic evaluation. In [10], the following matrix permutations for a $d \times d$ matrix $A = [A_{i,j}]$ are defined:

$$\sigma(A)_{ij} = A_{i,i+j}, \quad \tau(A)_{ij} = A_{i+j,j},$$
$$\phi(A)_{ij} = A_{i,j+1}, \quad \psi(A)_{ij} = A_{i+1,j}.$$

Using these permutations, the matrix multiplication can be expressed as follows:

$$A \cdot B = \sum_{k=0}^{d-1} (\phi^k \circ \sigma(A)) \odot (\psi^k \circ \tau(B)) \tag{13}$$

We assume that each matrix is packed into a vector of size d^2 using a row-major packing. Applying a permutation to each matrix A can be considered equivalent to performing a certain linear transform on each packed vector $\mathbf{a} \in \mathbb{R}^{d^2}$. Then, the linear Transformer corresponding to the permutation f ($f = \sigma, \tau, \phi, \psi$) is equivalent to multiplying by a matrix $U^f \in \mathbb{R}^{d^2 \times d^2}$, and thus we have $f(A)_{ij} = (U^f \cdot \mathbf{a})_{d \cdot i+j}$. Then, the equation corresponding to Equation 13 becomes as follows:

$$\sum_{k=0}^{d-1} ((U^\phi)^k U^\sigma \mathbf{a}) \odot (U^\psi)^k U^\tau \mathbf{b}) \tag{14}$$

Here, the matrix-vector multiplication $U^f \cdot \mathbf{a}$ can be computed using the equation $U^f \cdot \mathbf{a} = \sum_k (u_k^f \odot \rho(\mathbf{a}; t_k))$ for some finite sequence $\mathbf{t} = (t_k) \in (-d^2/2, d^2/2]$, as suggested by Halevi and Shoup [8]. Here, u_k^f refers to the k-th diagonal of the matrix U^f. We will estimate the computation time for the homomorphic matrix multiplication based on the number of key switching operations, where key switching corresponds to both rotations and non-scalar multiplications. The number of rotations is equal to the number of non-zero components in $\mathbf{t}$, which corresponds to the number of nonzero diagonals in the matrix U^f, denoted as $N(U^f)$. For each permutation, the number of nonzero diagonals is $N(U^\sigma) = 2d-1$, $N(U^\tau) = d$, $N(U^\phi) = 2$, and $N(U^\psi) = 1$. Applying the Baby-step Giant-step algorithm for σ and τ requires approximately $2\sqrt{N(U^f)}$ rotations. For ϕ and ψ, 2 and 1 rotations are needed, respectively. Using this approach to perform the operation in 13, approximately $(2\sqrt{2}+1)\sqrt{d}+3d$ rotations and d ciphertext multiplications are required.

A similar algorithm can be used to perform the transpose operation. The permutation matrix U^t for the transpose operation $t(A) = A^T$ has $2d-1$ nonzero diagonals, requiring approximately $2\sqrt{2d}$ rotations. Since the transpose operation follows Jiang et al.'s method, the detailed algorithm can be found in [10]. In our paper, this algorithm is referred to as CIPHERTEXTMATRIXMULT.