Secure Transformer Inference Made Non-interactive

Jiawen Zhang
Zhejiang University
kevinzh@zju.edu.cn

Jian Liu✉
Zhejiang University
jian.liu@zju.edu.cn

Xinpeng Yang
Zhejiang University
yangxinpeng@zju.edu.cn

Yinghao Wang
Zhejiang University
asternight@zju.edu.cn

Kejia Chen
Zhejiang University
chenkejia@zju.edu.cn

Xiaoyang Hou
Zhejiang University
xiaoyanghou@zju.edu.cn

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

Xiaohu Yang
Zhejiang University
yangxh@zju.edu.cn

ABSTRACT

Secure transformer inference has emerged as a prominent research topic following the proliferation of ChatGPT. Existing solutions are typically interactive, involving substantial communication load and numerous interaction rounds between the client and the server.

In this paper, we propose NEXUS, the first non-interactive protocol for secure transformer inference, where the client is only required to submit an encrypted input and await the encrypted result from the server. Central to NEXUS are two innovative techniques: SIMD ciphertext compression/decompression, and SIMD slots folding. Consequently, our approach achieves a speedup of 2.8× and a remarkable bandwidth reduction of 368.6×, compared to the state-of-the-art solution presented in S&P '24.

1 INTRODUCTION

Transformers, such as GPT [43] and BERT [16], have revolutionized the field of artificial intelligence. They excel in a wide range of applications such as language translation, content generation, and question answering. However, these applications often involve sensitive data, leading to growing concerns about user privacy. For example, OpenAI has developed ChatGPT as an online inference service, along with a remote API for developers, where users can easily access the services by submitting prompts or messages. While this approach is convenient, it poses significant privacy risks since users' submitted data may contain sensitive information.

Secure inference is a two-party cryptographic protocol enabling model inference to proceed in such a manner that the server $\mathcal{S}$ learns nothing about the input submitted by the clients Cs, and C learns nothing about $\mathcal{S}$'s model, except for the inference results. Most of such protocols are designed for convolutional neural networks (CNNs) [2, 27, 30, 36] and some recent works also support transformer-based models [10, 24, 26, 35, 38, 40].

It is noteworthy that many of these protocols are *interactive*, involving substantial communication load and numerous interaction rounds between C and $\mathcal{S}$. For example, the state-of-the-art solution for secure transformer inference, known as BOLT [40], documents 59.61GB of bandwidth comsumption and 10,509 interaction rounds for a single inference. Such a substantial communication overhead notably contributes to latency, especially in WAN configurations, and renders conventional hardware acceleration techniques such as GPUs or FPGAs ineffective.

We emphasize the critical importance of establishing secure inference as *non-interactive*, wherein C only needs to submit an encrypted input and await the encrypted result from $\mathcal{S}$. For scenarios demanding real-time responses, existing secure inference protocols, whether interactive or non-interactive, fail to meet the speed criteria. Nevertheless, non-interactive protocols show promise in meeting this criteria by leveraging hardware acceleration. In non-real-time scenarios such as financial planning and hospital diagnosis, where C can tolerate an extended response latency, non-interactive protocols are feasible, whereas interactive ones are not. This is because interactive protocols necessitate C's computing resources to remain engaged during the waiting period, impeding the execution of other tasks.

Our contribution. In this paper, we propose NEXUS, which, to the best of our knowledge, is the first non-interactive protocol for secure transformer inference. Our approach in designing NEXUS begins with C encrypting its input using RNS-CKKS fully homomorphic encryption (FHE), enabling $\mathcal{S}$ to execute the transformer on the FHE-encrypted data. The single instruction multiple data (SIMD) technique is applied to process $N = 2^{15}$ elements in a batch, and polynomial approximations are used to handle non-linear functions such as GELU, softmax, layer normalization and argmax. We remark that our approximation does *not* require any model retraining or fine-tuning. Additionally, we propose two novel and fundamental techniques to enhance the efficiency of this basic solution:

- **SIMD ciphertexts compression and decompression.** This technique enables C to compress $2N$ SIMD ciphertexts into a single one, while allowing $\mathcal{S}$ to decompress them back with $4N$ ciphertext-plaintext multiplications and substitutions. As a result, this technique significantly reduces the number of ciphertexts that must be transferred, without introducing any additional overhead for subsequent computations.
- **SIMD slots folding.** This technique enables the computation of an associative function $f()$, such as sum and max, on all SIMD slots of a ciphertext. The resulting value automatically fills the slots of an SIMD ciphertext, allowing it to be applied to each individual slot of the original ciphertext. The entire process only requires $\log N$ rotations. This technique

✉Jian Liu is the corresponding author.

can substantially improve the performance of inner-product, softmax, layer normalization and argmax.

We believe both techniques are of independent interest. It is worth mentioning that the state-of-the-art argmax protocol [29] requires $(N+1)$ rotations and sign operations. Thanks to our SIMD slots folding technique, we successfully reduce both operations to $\log N$, resulting in a remarkable 13.3× speedup.

We provide a full-fledged implementation and systematic evaluation for NEXUS. Running a single inference for a BERT-base transformer [16] consumes 1,103 seconds and 164MB of bandwidth. This indicates a speedup of 2.8× and a bandwidth reduction of 368.6× compared to the state-of-the-art (i.e., BOLT).

We summarize our contribution as follows:

- The first non-interactive protocol for secure transformer inference, achieving a bandwidth reduction of 368.6× over the state-of-the-art (cf. §3);
- An SIMD ciphertexts compression and decompression technique for ciphertext packing (cf. §4);
- An SIMD slots folding technique that efficiently operates on the slots of an SIMD ciphertext (cf. §5);
- A comprehensive implementation and evaluation (cf. §6).

2 PRELIMINARIES

In this section, we provide the necessary preliminaries for this paper. Table 1 shows the notations that are frequently used.

Table 1: A table of frequent notations.

Notation	Description
C	client
$\mathcal{S}$	server
$\mathtt{E}()$	encryption
$\pi()$	encoding
$\mathtt{Enc}()$	encoding-then encryption
$\tilde{\mathbf{a}}$	FHE ciphertext
$\boxplus$	homomorphic addition
$\boxminus$	homomorphic subtraction
$\boxtimes$	homomorphic multiplication
$\mathtt{RotL}()/\mathtt{RotR}()$	left-rotation/right-rotation
$\mathtt{Subs}()$	substitution
$\mathtt{Sgn}()$	sign operation
L	multiplicative depth
N'	polynomial degree in RNS-CKKS
N	# SIMD slots, $N = N'/2$
$\mathbf{A}$	input matrix
$\mathbf{W}$	weight matrix
t	# input instances

2.1 Secure inference and threat model

Secure inference is a two-party cryptographic protocol that enables model inference between a client C and a server $\mathcal{S}$, while preserving the privacy of both parties' inputs. It is formally defined as follows:

Definition 1. *A protocol Π between $\mathcal{S}$ holding a model $\mathcal{M}$ and C holding an input $\mathbf{A}$ is a secure inference protocol if it satisfies:*

- ***Correctness.*** *The output at the end of the protocol is the correct inference $\mathcal{M}(\mathbf{A})$.*
- ***Security.***
  - ***Corrupted client.*** *There exists an efficient simulator Sim_C such that $View_C^{\Pi} \approx_c Sim_C(\mathbf{A}, out)$, where $View_C^{\Pi}$ denotes C's view during the execution of Π (the view includes the client's input, randomness, and the transcript of the protocol), and out denotes the output of the inference.*
  - ***Corrupted server.*** *There exists an efficient simulator $Sim_{\mathcal{S}}$ such that $View_{\mathcal{S}}^{\Pi} \approx_c Sim_{\mathcal{S}}(\mathcal{M})$, where $View_{\mathcal{S}}^{\Pi}$ denotes $\mathcal{S}$'s view during the execution of Π.*

We assume that either C or $\mathcal{S}$ can act as a semi-honest adversary, adhering to the protocol specifications while endeavoring to gather extra information during its execution. Additionally, we assume that the adversary is computationally bounded. Formal definitions of the threat model are provided in the Appendix E.

2.2 Transformer

Figure 1 shows the structure and workflow of a transformer. It takes an embedding, represented as a matrix, and passes it through an *attention* layer and a *feed forward* network. In the end, it outputs a selection vector according to the highest value in the final logits. *Layer normalization* (`LayerNorm`) is applied around each block.

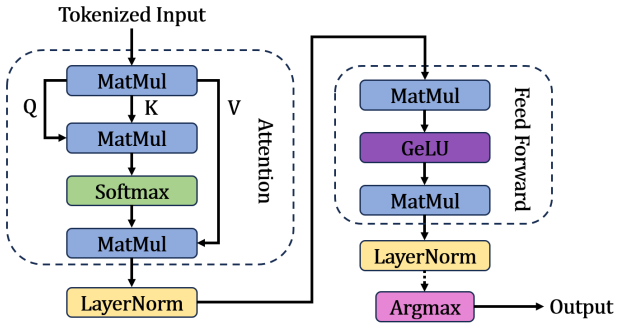


Figure 1: Structure and workflow of a transformer.

Attention. The first step in the attention layer is to multiply the embedding $\mathbf{A} \in \mathbb{R}^{m \times n}$ by three matrices ($\mathbf{W}_Q \in \mathbb{R}^{n \times k}$, $\mathbf{W}_K \in \mathbb{R}^{n \times k}$, and $\mathbf{W}_V \in \mathbb{R}^{n \times k}$) to produce a *query matrix*: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$, a *key matrix*: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$, and a *value matrix*: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$.

For each attention unit, the transformer learns three weight matrices: the query weights $\mathbf{W}_Q$, the key weights $\mathbf{W}_K$, and the value weights $\mathbf{W}_V$. For input token representation $\mathbf{X}$ is multiplied with each of the three weight matrices to produce a query matrix

$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$, a key matrix $\mathbf{K} = \mathbf{X}\mathbf{W}_K$, and a value matrix $\mathbf{V} = \mathbf{X}\mathbf{W}_V$. The attention can be represented as:

$$\texttt{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \texttt{Softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{k}})\mathbf{V}.$$

The multi-head attention variant computes a H-parallel attention $\texttt{Attention}(\mathbf{Q}_j, \mathbf{K}_j, \mathbf{V}_j)$ for $j \in [H]$ and then concatenate these H resulting matrices.

Layer normalization. The input to LayerNorm is $\mathbf{a} \in \mathbb{R}^n$, let $\mu = \frac{1}{n}\sum_{i=0}^{n-1} a_i$ and $\sigma = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(a_i - \mu)^2}$, the output $\mathbf{y} \in \mathbb{R}^n$ is calculated as:

$$y_i = \gamma \cdot \frac{x_i - \mu}{\sigma} + \beta$$

where $\gamma, \beta \in \mathbb{R}$ are two hyper-parameters.

Feed-forward. The fully connected feed-forward network consists of two linear transformations with a GELU activation function in between:

$$\texttt{FeedForward}(\mathbf{X}) = \texttt{GELU}(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2.$$

The GELU function is calculated as [25]:

$$\texttt{GELU}(x) = \frac{1}{2}x \cdot (1 + \texttt{erf}(\frac{x}{\sqrt{2}}))$$

where the Gauss error function is $\texttt{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$. It is used as an activation function due to its favorable curvature and non-monotonicity properties.

2.3 Fully homomorphic encryption

Fully homomorphic encryption (FHE), which allows arbitrary operations to be performed over encrypted data [19], is the the primary tool enabling us to build non-interactive secure transformer inference. The FHE scheme used in this paper is the full *residue number system* (RNS) variant of Cheon-Kim-Kim-Song (CKKS) [11, 12].

RNS-CKKS is a *leveled* FHE, which can support computations up to a multiplicative depth L. Both the plaintexts and ciphertexts of RNS-CKKS are elements in a polynomial ring:

$$\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^{N'} + 1),$$

where $Q = \Pi_{i=0}^{L} q_i$ with distinct primes q_i. Once a ciphertext's level becomes too low, a *bootstrapping* operation is required to refresh it to a higher level, enabling more computations. In a nutshell, bootstrapping homomorphically evaluates the decryption circuit and raises the modulus from q_0 to q_L by leveraging the isomorphism $\mathcal{R}_{q_0} \cong \mathcal{R}_{q_0} \times \mathcal{R}_{q_1} \times \cdots \times \mathcal{R}_{q_L}$ [9]. Suppose the bootstrapping consumes K levels, then a fresh ciphertext can support $L - K$ levels of computations.

RNS-CKKS supports *single instruction multiple data* (SIMD), which enables encrypting a vector $\mathbf{a} \in \mathbb{R}^N$, where $N = N'/2$, into a single ciphertext and process these encrypted elements in a batch without introducing any extra cost. To encrypt $\mathbf{a}$ in SIMD format, it first encodes $\mathbf{a}$ into a polynomial in $\mathcal{R}_Q$ using an encoding algorithm $\pi()$, and then encrypts the polynomial using an encryption algorithm $\texttt{E}()$. Throughout this paper, we use $\texttt{E}()$ to denote the encryption of a polynomial and use $\texttt{Enc}()$ to denote the SIMD encryption of a vector:

$$\texttt{Enc}(\mathbf{a}) = \texttt{E}(\pi(\mathbf{a})).$$

Below, we summarize the homomorphic operations used in this paper:

- $p(x) \leftarrow \pi(\mathbf{a})$. The encoding algorithm takes a vector $\mathbf{a} = [a_0, ..., a_{N-1}]$ and outputs a polynomial $p(x) \in \mathcal{R}_Q$.
- $\tilde{\mathbf{a}} \leftarrow \texttt{Enc}(\mathbf{a})$. The encryption algorithm takes a vector $\mathbf{a} = [a_0, ..., a_{N-1}]$ and outputs an SIMD ciphertext denoted by $\tilde{\mathbf{a}}$.
- $\mathbf{a} \leftarrow \texttt{Dec}(\tilde{\mathbf{a}})$. The decryption algorithm takes an SIMD ciphertext $\tilde{\mathbf{a}}$ and outputs a plaintext vector $\mathbf{a}$.
- $\tilde{\mathbf{c}} \leftarrow \tilde{\mathbf{a}} \boxplus \tilde{\mathbf{b}}$. The addition algorithm takes two SIMD ciphertexts $\tilde{\mathbf{a}}$ and $\tilde{\mathbf{b}}$; outputs $\texttt{Enc}([a_0 + b_0, ..., a_{N-1} + b_{N-1}])$.
- $\tilde{\mathbf{c}} \leftarrow \tilde{\mathbf{a}} \boxtimes \mathbf{b}$. The ciphertext-plaintext multiplication takes $\tilde{\mathbf{a}}$ and a plaintext vector $\mathbf{b}$; outputs $\texttt{Enc}([a_0 b_0, ..., a_{N-1} b_{N-1}])$.
- $\tilde{\mathbf{c}} \leftarrow \tilde{\mathbf{a}} \boxtimes \tilde{\mathbf{b}}$. The ciphertext-ciphertext multiplication takes $\tilde{\mathbf{a}}$ and $\tilde{\mathbf{b}}$; outputs $\texttt{Enc}([a_0 b_0, ..., a_{N-1} b_{N-1}])$.
- $\tilde{\mathbf{a}}' \leftarrow \texttt{RotL}(\tilde{\mathbf{a}}, s)$. The left-rotation algorithm takes $\tilde{\mathbf{a}}$ and an integer $s \in [N]$; left-rotates the vector by s slots.
- $\tilde{\mathbf{a}}' \leftarrow \texttt{RotR}(\tilde{\mathbf{a}}, s)$. The right-rotation algorithm takes $\tilde{\mathbf{a}}$ and an integer $s \in [N]$; right-rotates the vector by s slots.
- $\tilde{\mathbf{a}}' \leftarrow \texttt{Subs}(\tilde{\mathbf{a}}, k)$. The substitution operation takes a ciphertext that encrypts a polynomial $p(x)$ and an odd integer k; outputs a ciphertext that encrypts $p(x^k)$.
- $\tilde{\mathbf{b}} \leftarrow \texttt{Sgn}(\tilde{\mathbf{a}})$. The sign operation, cf.§2.4.

2.4 Homomorphic sign function

As FHE only supports polynomial operations, it is nontrivial to compare FHE-encrypted values in a non-interactive manner. To enable encrypted comparisons, we leverage the polynomial approximation of the sign function [13, 18, 34]:

$$sign(x) = f^{d_f}(g^{d_g}(x)) = \begin{cases} -1 & -1 \le x \le -2^{-\alpha} \\ 0 & x = 0 \\ 1 & 2^{-\alpha} \le x \le 1 \end{cases}$$

where $f()$, $g()$ are two polynomials and d_f, d_g are the number of repetitions for them. Notice that this approximation requires the input to fall within the interval $[-1, 1]$. Therefore, any input $a \in [a_{min}, a_{max}]$ to the $sign()$ function must be normalized beforehand:

$$x := a/max\{|a_{max}|, |a_{min}|\}.$$

We use $\texttt{Sgn}()$ to denote running both the normalization and the sign approximation on an SIMD ciphertext:

- $\tilde{\mathbf{b}} \leftarrow \texttt{Sgn}(\tilde{\mathbf{a}})$: $b_i = f^{d_f}(g^{d_g}(\frac{a_i}{max\{|a_{max}|,|a_{min}|\}}))\ \forall\, i \in [N]$.

In our implementation, both $f()$ and $g()$ are of 9-degree; we set $\alpha = 16, d_f = 2, d_g = 2$ and evaluate the polynomials using the Baby-Step-Giant-Step algorithm [23].

3 BASIC DESIGN

In this section, we provide a basic design for NEXUS, aligning with the workflow depicted in Figure 1. Later in §4 and §5, we will integrate our proposed techniques (i.e., SIMD ciphertext compression/decompression and SIMD slots folding) to enhance this basic design.

3.1 Attention

3.1.1 Matrix multiplication (ciphertext-plaintext). Recall that the first step in the attention layer is to multiply the input matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ by three matrices ($\mathbf{W}_Q \in \mathbb{R}^{n\times k}$, $\mathbf{W}_K \in \mathbb{R}^{n\times k}$, and $\mathbf{W}_V \in \mathbb{R}^{n\times k}$) respectively:

$$\mathbf{Q} := \mathbf{A}\cdot\mathbf{W}_Q;$$
$$\mathbf{K} := \mathbf{A}\cdot\mathbf{W}_K;$$
$$\mathbf{V} := \mathbf{A}\cdot\mathbf{W}_V.$$

We concentrate on explaining the computation of $\mathbf{Q} := \mathbf{A}\cdot\mathbf{W}_Q$, noting that the same process is applicable to $\mathbf{W}_K$ and $\mathbf{W}_V$.

Let $a_{i,j} \in \mathbb{R}$ be the element in the i-th row and j-th column of $\mathbf{A}$, $\mathbf{w}_j \in \mathbb{R}^k$ be the j-th row of $\mathbf{W}_Q$ and $\mathbf{q}_i \in \mathbb{R}^k$ be the i-th row of $\mathbf{Q}$. Then, $\mathbf{q}_i$ is the vector-wise sum of $(a_{i,j}\cdot\mathbf{w}_j)\ \forall\ j \in [n]$.

To this end, we could have C homomorphically encrypt each $a_{i,j}$ and send the corresponding ciphertexts to $\mathcal{S}$, who can then homomorphically evaluate `MatrixMul`. However, this faces two challenges:

(1) C needs to send $m\cdot n$ ciphertexts; and
(2) $\mathcal{S}$ needs to perform $m\cdot n\cdot k$ ciphertext-plaintext multiplications.

We leverage SIMD to address the second challenge. Specifically, we have C encrypt each $a_{i,j}$ as:

$$\tilde{\mathbf{a}}_{i,j} := \mathsf{Enc}(\underbrace{[a_{i,j}, ..., a_{i,j}]}_{k}) \text{ (suppose } k < N).$$

Then, $\mathcal{S}$ is able to compute $\mathsf{Enc}(a_{i,j}\cdot\mathbf{w}_j)$ with a *single* ciphertext-plaintext multiplication:

$$\tilde{\mathbf{a}}_{i,j} \boxtimes \mathbf{w}_j,$$

thereby reducing the total number of ciphertext-plaintext multiplications to $m\cdot n$. Figure 2 shows a toy example with $\mathbf{A} \in \mathbb{R}^{2\times 3}$ and $\mathbf{W}_Q \in \mathbb{R}^{3\times 3}$.

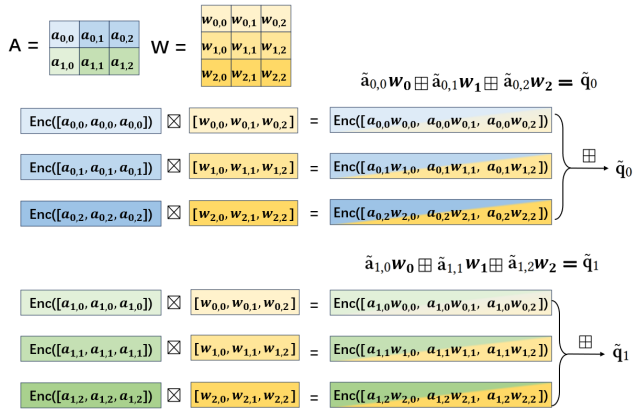


Figure 2: A toy example of SIMD-based matrix multiplication

Nevertheless, C still needs to send $m\times n$ ciphertexts. We propose a method enabling C to compress them into $\frac{m\times n}{N'}$ ciphertexts, while ensuring $\mathcal{S}$ can decompress them and perform the aforementioned computations (cf. §4).

3.1.2 Matrix multiplication (ciphertext-ciphertext). After obtaining the encrypted $(\mathbf{Q}, \mathbf{K}, \mathbf{V})$, $\mathcal{S}$ needs to compute $\mathbf{Q}\cdot\mathbf{K}^T$. Now, each row of $\mathbf{Q} \in \mathbb{R}^{m\times k}$ and each column of $\mathbf{K}^T \in \mathbb{R}^{k\times m}$ are encrypted in SIMD format: $\mathsf{Enc}(\mathbf{q})$ and $\mathsf{Enc}(\mathbf{k}^T)$. If $\mathcal{S}$ can compute the inner-product between $\mathsf{Enc}(\mathbf{q}_i)$ and $\mathsf{Enc}(\mathbf{k}_j^T)\ \forall\ i \in [m], j \in [m]$, it can obtain the encrypted result of $\mathbf{Q}\cdot\mathbf{K}^T$.

Thanks to SIMD, $\mathcal{S}$ can easily obtain $\mathsf{Enc}(\mathbf{u})$, where $\mathbf{u} = [u_0, ..., u_{k-1}]$ is the element-wise product of $\mathbf{q}$ and $\mathbf{k}^T$. Now, to compute the inner-product, $\mathcal{S}$ only needs to compute $s := \sum_{i=0}^{k-1} u_i$ under SIMD. $\mathcal{S}$ could simply rotate $\mathsf{Enc}([u_0, ..., u_{k-1}])$ for $(k-1)$ times and compute the sum, after which $\mathcal{S}$ obtains $\mathsf{Enc}(\underbrace{[s, ..., s]}_{k})$. We propose a `QuickSum` algorithm that allows $\mathcal{S}$ to achieve this goal with only $\log k$ rotations (cf. §5.1).

In the end, $\mathcal{S}$ combines the ciphertexts in each row into a single one: $\boxplus_{i=0}^{m-1}(\mathsf{Enc}([s_i, ..., s_i]) \boxtimes \mathbf{b}_i)$, where only the i-th slot in b_i is 1 and other slots are 0s. We denote the output matrix as $\mathbf{A} \in \mathbb{R}^{m\times m}$, which is the input to `Softmax`. Each row of $\mathbf{A}$ is encrypted in SIMD format.

We remark that the `MatrixMul` after `Softmax` can be computed in this way as well.

3.1.3 Softmax. Recall that the `Softmax` function needs to be applied to each row of $\mathbf{A}$. The `Softmax` function is commonly evaluated as:

$$y_i = \frac{\exp(a_i - a_{max})}{\sum_{j=0}^{m-1}\exp(a_j - a_{max})} \tag{1}$$

where $a_{max} = max(a_0, ..., a_{m-1})$ ensures all inputs to the exponential function (i.e., $a'_j = a_j - a_{max}$) are non-positive, achieving numerical stability [32].

We propose a `QuickMax` algorithm that takes $\mathsf{Enc}([a_0, ..., a_{m-1}])$ as input and outputs $\mathsf{Enc}(\underbrace{[a_{max}, ..., a_{max}]}_{m})$, requiring only $(\log m - 1)$ `Sgn` operations and $\log m$ rotations (cf. §5.2).

With $\mathsf{Enc}([a_0, ..., a_{m-1}])$ and $\mathsf{Enc}([a_{max}, ..., a_{max}])$, $\mathcal{S}$ could easily obtain $\mathsf{Enc}([a'_0, ..., a'_{m-1}])$. Following BumbleBee [38], we approximate the exponentiation using the Taylor series:

$$\exp(x) \approx (1 + \frac{x}{2^r})^{2^r}, \quad x \le 0$$

with $r = 6$, which limits the average error within 10^{-5} (cf. §6.3). Then, $\mathcal{S}$ could compute the exponentiation in SIMD format and obtain $\mathsf{Enc}([e_0, ..., e_{m-1}])$, where $e_j = \exp(a'_j)$. Next, $\mathcal{S}$ applies `QuickSum` (cf. §5.1) to obtain $\mathsf{Enc}(\underbrace{[\sum_{j=0}^{m-1} e_j, ..., \sum_{j=0}^{m-1} e_j]}_{m})$. In the end, $\mathcal{S}$ computes the final result in SIMD format using the Goldschmidt division algorithm [21, 41]. Algorithm 1 describes the details of our secure `Softmax`.

Algorithm 1 Secure `Softmax` on RNS-CKKS

Input: $\tilde{\mathbf{a}} = \mathtt{Enc}([a_0, ..., a_{m-1}, 0, ..., 0])$ with $2m < N$
Output: $\mathtt{Enc}([y_0, ..., y_{m-1}, 0, ..., 0])$ (cf. Equation 1)

function $\mathtt{Softmax}(\tilde{\mathbf{a}})$
  $\tilde{\mathbf{a}}_{max} \leftarrow \mathtt{QuickMax}(\tilde{\mathbf{a}})$ // cf. § 5.2
  $\tilde{\mathbf{e}} \leftarrow \mathtt{Exp}(\tilde{\mathbf{a}} \boxminus \tilde{\mathbf{a}}_{max})$
  $\tilde{\mathbf{s}} \leftarrow \mathtt{QuickSum}(\tilde{\mathbf{e}})$ // cf. § 5.1
  return $\tilde{\mathbf{e}} \boxtimes \mathtt{Inverse}(\tilde{\mathbf{s}})$
end function

3.2 Layer normalization

For the ease of computation, we perform the following transformation to `LayerNorm`:

$$\begin{aligned} y_i &= \gamma \cdot \frac{a_i - \mu}{\sigma} + \beta \\ &= \gamma \cdot \frac{n(a_i - \mu)}{n\sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(a_i - \mu)^2}} + \beta \\ &= \sqrt{n}\gamma \cdot \frac{na_i - n\mu}{\sqrt{\sum_{i=0}^{n-1}(na_i - n\mu)^2}} + \beta. \end{aligned}$$

Let $z_i = na_i - n\mu = na_i - \sum_{i=0}^{n-1} a_i$, then

$$y_i = \gamma\sqrt{n} \cdot \frac{z_i}{\sqrt{\sum_{i=0}^{n-1} z_i^2}} + \beta. \tag{2}$$

We apply `QuickSum` (cf. §5.1) again to compute $\sum_{i=0}^{n-1} a_i$ and $\sum_{i=0}^{n-1} z_i^2$. For the inverse square root, we adopt the method proposed in [42], which employs Newton iteration with a proper initial value. Algorithm 2 describes the details of our secure `LayerNorm`.

Algorithm 2 Secure `LayerNorm` on RNS-CKKS

Input: $\tilde{\mathbf{a}} = \mathtt{Enc}([a_0, ..., a_{n-1}, 0, ..., 0])$ with $2n < N$
Output: $\mathtt{Enc}([y_0, ..., y_{n-1}, 0, ..., 0])$ (cf. Equation 2)

function $\mathtt{LayerNorm}(\tilde{\mathbf{a}})$
  $\tilde{\mathbf{s}} \leftarrow \mathtt{QuickSum}(\tilde{\mathbf{a}})$ // cf. § 5.1
  $\tilde{\mathbf{z}} \leftarrow (n \boxtimes \tilde{\mathbf{a}}) \boxminus \tilde{\mathbf{s}}$ // $z_i = na_i - \sum_{i=0}^{n-1} a_i$
  $\tilde{\mathbf{y}} \leftarrow \mathtt{Square}(\tilde{\mathbf{z}})$
  $\tilde{\mathbf{y}} \leftarrow \mathtt{QuickSum}(\tilde{\mathbf{y}})$ // cf. § 5.1
  $\tilde{\mathbf{y}} \leftarrow \mathtt{InvertSqrt}(\tilde{\mathbf{y}})$
  $\tilde{\mathbf{y}} \leftarrow \tilde{\mathbf{z}} \boxtimes \tilde{\mathbf{y}}$ // $y_i = z_i / \sqrt{\sum_{i=1}^{n} z_i^2}$
  return $(\tilde{\mathbf{y}} \boxtimes \gamma \boxtimes \sqrt{n}) \boxplus \beta$
end function

3.3 Feed forward

The feed forward layer involves two `MatrixMul`s and one `GELU`. For `MatrixMul`, denoted as $\mathbf{A} \cdot \mathbf{W}$, each row of $\mathbf{A}$ is encrypted in SIMD format and $\mathbf{W}$ is known to $\mathcal{S}$. Consequently, $\mathcal{S}$ can employ a similar method as described in §3.1.2:

(1) Compute the element-wise product between each row of $\mathbf{A}$ and each column of $\mathbf{W}$. Notice that this time only requires ciphertext-plaintext multiplication, unlike the ciphertext-ciphertext multiplication used in §3.1.2.
(2) Use `QuickSum` to compute the dot-product.
(3) Combine the ciphertexts in each row into a single one.

We use the following piecewise polynomials to approximate $\mathtt{GELU}(x)$, which limits the average error within 10^{-3} (cf. §6.3) when $x \in [-60, 60]$[1]:

$$\mathtt{GELU}(x) = \begin{cases} 0 & x \le -4 \\ P(x) = \sum_{i=0}^{i=3} c_i x^i & -4 < x \le -1.95 \\ Q(x) = \sum_{i=0}^{i=6} d_i x^i & -1.95 < x \le 3 \\ x & x > 3 \end{cases} \tag{3}$$

We first use the `Sgn` operation to obtain 4 encrypted bits: b_0, b_1, b_2, b_3, s.t.,

$$b_i = 1 \text{ iff } x \text{ belongs to the } i\text{-th segment.}$$

Then, $\mathtt{GELU}(x) = b_0 \cdot 0 + b_1 P(x) + b_2 Q(x) + b_3 x$.

Algorithm 3 shows the details of our secure `GELU`. For simplicity, we omit the evaluation of $P(x)$, $Q(x)$ in Algorithm 3. We remark that our algorithm can implicitly handle the cases where $a = -4, -1.95, -3$:

- $y = 0.5P(a)$ when $a = -4$, which is correct given $P(a) \approx 0$ in this case;
- $y = 0.5P(a) + 0.5Q(a)$ when $a = 1.95$, which is correct given $P(a) \approx Q(a)$ in this case;
- $y = 0.5Q(a) + 0.5$ when $a = -4$, which is correct given $Q(a) \approx 1$ in this case.

Algorithm 3 Secure `GELU` on RNS-CKKS

Input: $\tilde{\mathbf{a}} = \mathtt{Enc}([a_0, ..., a_{N-1}])$
Output: $\mathtt{Enc}([y_0, ..., y_{N-1}])$ (cf. Equation 3)

function $\mathtt{GELU}(\tilde{\mathbf{a}})$
  Compare a with the breakpoints:
    $\tilde{\mathbf{s}}_0 \leftarrow 0.5 \boxtimes \mathtt{Sgn}(\tilde{\mathbf{a}} \boxplus 4)$ // $s_0 = \mathbf{0.5}\{a > -4\}$
    $\tilde{\mathbf{s}}_1 \leftarrow 0.5 \boxtimes \mathtt{Sgn}(\tilde{\mathbf{a}} \boxplus 1.95)$ // $s_1 = \mathbf{0.5}\{a > -1.95\}$
    $\tilde{\mathbf{s}}_2 \leftarrow 0.5 \boxtimes \mathtt{Sgn}(\tilde{\mathbf{a}} \boxminus 3)$ // $s_2 = \mathbf{0.5}\{a > 3\}$
  Compute segment selection:
    $\tilde{\mathbf{b}}_0 \leftarrow 0.5 \boxminus \tilde{\mathbf{s}}_0$ // $b_0 = \mathbf{1}\{x < -4\}$
    $\tilde{\mathbf{b}}_1 \leftarrow \tilde{\mathbf{s}}_0 \boxminus \tilde{\mathbf{s}}_1$ // $b_1 = \mathbf{1}\{-4 < x < -1.95\}$
    $\tilde{\mathbf{b}}_2 \leftarrow \tilde{\mathbf{s}}_1 \boxminus \tilde{\mathbf{s}}_2$ // $b_2 = \mathbf{1}\{-1.95 < x < 3\}$
    $\tilde{\mathbf{b}}_3 \leftarrow 0.5 \boxplus \tilde{\mathbf{s}}_2$ // $b_3 = \mathbf{1}\{x > 3\}$
  Compute `GELU`:
    $\tilde{\mathbf{y}} \leftarrow (\tilde{\mathbf{b}}_0 \boxtimes 0) \boxplus (\tilde{\mathbf{b}}_1 \boxtimes P(\tilde{\mathbf{a}})) \boxplus (\tilde{\mathbf{b}}_2 \boxtimes Q(\tilde{\mathbf{a}})) \boxplus (\tilde{\mathbf{b}}_3 \boxtimes \tilde{\mathbf{a}})$
  return $\tilde{\mathbf{y}}$
end function

3.4 Argmax

Suppose the last layer outputs $\mathtt{Enc}([a_0, ..., a_{m-1}])$, the final output of the transformer should be a selection vector $\mathtt{Enc}([b_0, ..., b_{m-1}])$, where

$$b_i = 1 \text{ iff } a_i = max(a_0, ..., a_{m-1}), \text{ otherwise } b_i = 0.$$

The state-of-the-art non-interactive protocol that can achieve this goal is Phoneix [29], which requires $(m + 1)$ `Sgn`s and $(m + 1)$ rotations.

We *innovatively* propose to approximate each b_i as:

$$b_i = \mathtt{Sgn}(a_i - a_{max}) + 1. \tag{4}$$

[1] Our experimental results show that all inputs are in this range.

Then, the selection vector can be easily computed as described in Algorithm 4. It only requires $\log m$ Sgns and $\log m$ rotations, achieving 13.3× speedup over Phoneix [29].

Algorithm 4 Secure Argmax on RNS-CKKS

Input: $\tilde{\mathbf{a}} = \mathsf{Enc}([a_0, ..., a_{m-1}, 0, ..., 0])$ with $2m < N$
Output: $\mathsf{Enc}([b_0, ..., b_{m-1}, 0, ..., 0])$ (cf. Equation 4)

1: **function** Argmax(c)
2: $\quad\tilde{\mathbf{a}}_{max} \leftarrow \mathsf{QuickMax}(\tilde{\mathbf{a}})$ // cf. § 5.2
3: $\quad\tilde{\mathbf{a}} \leftarrow \tilde{\mathbf{a}} \boxminus \tilde{\mathbf{a}}_{max}$
4: $\quad\tilde{\mathbf{b}} \leftarrow \mathsf{Sgn}(\tilde{\mathbf{a}})$ // $b = 0$ or -1
5: $\quad\tilde{\mathbf{b}} \leftarrow \tilde{\mathbf{b}} \boxplus 1$
6: $\quad$**return** $\tilde{\mathbf{b}}$
7: **end function**

3.5 Placement of bootstrapping

As mentioned in §2.3 , NEXUS is based on RNS-CKKS, which is a leveled homomorphic encryption scheme that allows at most L multiplications on any computation path. Once a ciphertext's level becomes too low, bootstrapping is required to refresh it to a higher level to enable more multiplications. As the bootstrapping operation is expensive, its placement is crucial for the overall performance. For example, the input/output matrix size of GELU is $\mathbb{R}^{128\times3072}$ (packed in 12 ciphertexts), which are then reduced to $\mathbb{R}^{128\times768}$ (packed in 3 ciphertexts) by the subsequent MatrixMul. Therefore, it is crucial to circumvent bootstrapping during operations involving large input/output sizes, such as GELU, by judiciously selecting the multiplicative depth L. Figure 3 shows the placement of bootstrapping for a BERT-base transformer in NEXUS.

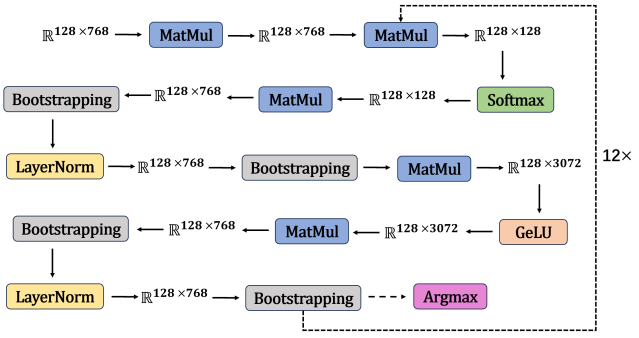


Figure 3: Placement of bootstrapping for a BERT-base transformer.

4 SIMD CIPHERTEXTS COMPRESSION AND DECOMPRESSION

Suppose C wants to send N' ciphertexts to $\mathcal{S}$ with each ciphertext encrypting N identical values in SIMD format: $\mathsf{Enc}(\underbrace{[a_0, ..., a_0]}_{N})$, ..., $\mathsf{Enc}(\underbrace{[a_{N'-1}, ..., a_{N'-1}]}_{N})$. We have C pack $[a_0, a_1, ..., a_{N'-1}]$ into a polynomial $p(x) = a_0 + a_1x + a_2x^2 + ... + a_{N'-1}x^{N'-1}$ and send $\tilde{\mathbf{p}}_0 := \mathsf{E}(p(x))$ to $\mathcal{S}$. In this way, we reduce the communication cost from N' ciphertexts to a single ciphertext. Next, we show how $\mathcal{S}$ performs decompression to recover the N' SIMD ciphertexts.

Notice that $\mathsf{Subs}(\tilde{\mathbf{p}}_0, N'+1)$ (cf. §2.3) returns:

$$\begin{aligned}&\mathsf{E}(a_0 + a_1x^{N'+1} + a_2x^{2(N'+1)} + ... + a_{N'-1}x^{(N'-1)(N'+1)})\\ =&\mathsf{E}(a_0 + a_1x^{N'+1} + a_2(x^{N'+1})^2 + ... + a_{N'-1}(x^{N'+1})^{(N'-1)})\\ =&\mathsf{E}(a_0 + a_1(-x) + a_2(-x)^2 + ... + a_{N'-1}(-x)^{(N'-1)}).^2\end{aligned}$$

It is evident that $\tilde{\mathbf{p}}_0 \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}_0, N'+1)$ eliminates all odd-degree terms of $p(x)$. Then, $\mathcal{S}$ can extract $\mathsf{E}(a_0 + 0x^1 + ... + 0x^{N'-1})$ via $\log N'$ substitutions:

$$\begin{aligned}\tilde{\mathbf{p}}_1 &\leftarrow \tilde{\mathbf{p}}_0, \boxplus\mathsf{Subs}(\tilde{\mathbf{p}}_0, \frac{N'}{2^0}+1)\\ \tilde{\mathbf{p}}_2 &\leftarrow \tilde{\mathbf{p}}_1 \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}_1, \frac{N'}{2^1}+1)\\ \tilde{\mathbf{p}}_3 &\leftarrow \tilde{\mathbf{p}}_2 \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}_2, \frac{N'}{2^2}+1)\\ &\cdots\cdots\\ \tilde{\mathbf{p}}_{\log N'} &\leftarrow \tilde{\mathbf{p}}_{\log N'-1} \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}_{\log N'-1}, \frac{N'}{2^{(\log N'-1)}}+1)\\ \tilde{\mathbf{p}}_{\log N'} &\leftarrow \tilde{\mathbf{p}}_{\log N'} \boxtimes \frac{1}{N'}\end{aligned}$$

given that $\tilde{\mathbf{p}}_i \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}_i, \frac{N'}{2^i}+1)$ eliminates all d-degree terms, where $(d \mod 2^i = 1)$. Later, we will prove that the output of this extraction process is exactly $\mathsf{Enc}(\underbrace{[a_0, a_0, ..., a_0]}_{N})$ (cf. Theorem 1).

Similarly, to extract $\mathsf{E}(a_1 + 0x^1 + ... + 0x^{N'-1})$, $\mathcal{S}$ could left-rotate the encrypted $p(x)$ by one term (i.e., multiply by x^{-1}) and run the above extraction process again. By performing this for N' times, $\mathcal{S}$ can get all individual encryption of $[a_0, a_1, ..., a_{N'-1}]$. However, this involves $(N' \log N')$ substitutions. In contrast, we propose a method that can achieve the same goal with only $2N'$ substitutions. Our method is based on the following observations:

- There are only two terms in the polynomial inside $\tilde{\mathbf{p}}_{\log N'-1}$: $a_0 + a_{N'/2}x^{N'/2}$. If $\mathcal{S}$ left-rotates this polynomial by one term (and obtains $a_{N'/2} + a_0x^{N'/2}$), it can extract $a_{N'/2}$ with only one substitution (instead of $\log N'$).
- There are four terms in the polynomial inside $\tilde{\mathbf{p}}_{\log N'-2}$. If $\mathcal{S}$ left-rotates this polynomial by one term, it can can extract the other two coefficients (other than a_0 and $a_{N'/2}$) with three substitutions in total.
- If $\mathcal{S}$ starts this already from $\tilde{\mathbf{p}}_0$, the extraction process becomes a binary tree and the leaves are exactly $a_0, a_1, ..., a_{N'-1}$.
- To left-rotate the polynomial inside $\tilde{\mathbf{p}}_i$ by one term, $\mathcal{S}$ needs to multiply $\tilde{\mathbf{p}}_i$ by x^{-2^i} instead of x^{-1}, because some terms have been eliminated. Denoted as: $\tilde{\mathbf{p}}'_i \leftarrow \tilde{\mathbf{p}}_i \boxtimes x^{-2^i}$.

Based on these observations, we change the extraction process to:

(1) $\tilde{\mathbf{p}}_{1,0} \leftarrow \tilde{\mathbf{p}}_0 \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}_0, \frac{N'}{2^0}+1)$,
$\quad\tilde{\mathbf{p}}_{1,1} \leftarrow \tilde{\mathbf{p}}'_0 \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}'_0, \frac{N'}{2^0}+1)$

[2] Observe that $x^{N'}+1 \equiv 0 \pmod{x^{N'}+1}$ and hence $x^{N'+1} \equiv -x \pmod{x^{N'}+1}$.

(2) $\tilde{\mathbf{p}}_{2,0} \leftarrow \tilde{\mathbf{p}}_{1,0} \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}_{1,0}, \frac{N'}{2^1}+1)$,
$\tilde{\mathbf{p}}_{2,1} \leftarrow \tilde{\mathbf{p}}'_{1,0} \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}'_{1,0}, \frac{N'}{2^1}+1)$,
$\tilde{\mathbf{p}}_{2,2} \leftarrow \tilde{\mathbf{p}}_{1,1} \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}_{1,1}, \frac{N'}{2^1}+1)$,
$\tilde{\mathbf{p}}_{2,3} \leftarrow \tilde{\mathbf{p}}'_{1,1} \boxplus \mathsf{Subs}(\tilde{\mathbf{p}}'_{1,1}, \frac{N'}{2^1}+1)$
(3) .....

After $\log N'$ steps, $\mathcal{S}$ obtain N' ciphertexts, representing the individual encryption of $[a_0, a_1, ..., a_{N'-1}]$. Figure 4 visualizes this process with a toy polynomial of degree-3. Algorithm 5 describes the full decompression process. Clearly, it only requires $2N'$ substitutions in total. We prove its correctness in Appendix A.

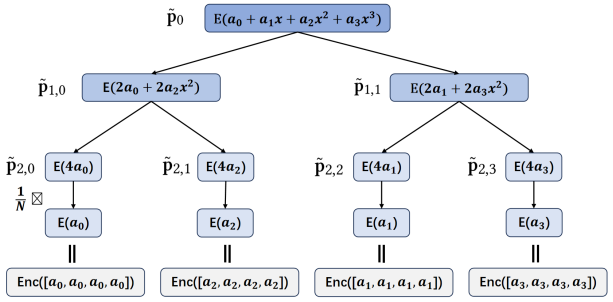


Figure 4: A toy example showcasing the decompression process.

Algorithm 5 Secure `Decompression` on RNS-CKKS

Input: $\tilde{\mathbf{p}}_0 = \mathsf{E}(a_0 + a_1x + a_2x^2 + ... + a_{N'-1}x^{N'-1})$
Output: $[\tilde{\mathbf{a}}_0, ..., \tilde{\mathbf{a}}_{N'-1}]$, where each $\tilde{\mathbf{a}}_i = \mathsf{Enc}(\underbrace{[a_i, ..., a_i]}_{N})$

1: **function** `Decompress`($\tilde{\mathbf{p}}_0$)
2: $\quad\tilde{\mathbf{p}}_{0,0} := \tilde{\mathbf{p}}_0$
3: $\quad$**for** $i = 0$ to $\log N'$ **do**
4: $\quad\quad$**for** $j = 0$ to $2^i - 1$ **do**
5: $\quad\quad\quad\tilde{\mathbf{p}}'_{i,j} \leftarrow \tilde{\mathbf{p}}_{i,j} \boxtimes x^{-2^i}$
6: $\quad\quad\quad\tilde{\mathbf{p}}_{i+1,2^j-1} \leftarrow \tilde{\mathbf{p}}_{i,j} \boxplus \mathsf{Subs}(c, \frac{N'}{2^i}+1)$
7: $\quad\quad\quad\tilde{\mathbf{p}}_{i+1,2^j} \leftarrow \tilde{\mathbf{p}}'_{i,j} \boxplus \mathsf{Subs}(c', \frac{N'}{2^i}+1)$
8: $\quad\quad$**end for**
9: $\quad$**end for**
10: $\quad$**for** $j = 0$ to $N' - 1$ **do**
11: $\quad\quad\tilde{\mathbf{a}}_j \leftarrow \tilde{\mathbf{p}}_{\log N', j} \boxtimes \frac{1}{N'}$
12: $\quad$**end for**
13: $\quad$re-arrange $[\tilde{\mathbf{a}}_0, \tilde{\mathbf{a}}_1, ..., \tilde{\mathbf{a}}_{N'-1}]$ according to the order of $[a_0, a_1, ..., a_{N'-1}]$ and **return** the result
14: **end function**

Next, we prove that each output $\tilde{\mathbf{a}}_i$ of Algorithm 5 is exactly an SIMD ciphertext encrypting a vector of N a_is.

Theorem 1. *The encryption of a polynomial with only constant term:* $\mathsf{E}(a_s + 0x^1 + ... + 0x^{N'-1})$ *is exactly an SIMD encryption of N identical values:* $\mathsf{Enc}(\underbrace{[a_s, a_s, ..., a_s]}_{N})$.

Proof. Given that

$$\mathsf{Enc}(\underbrace{[a_s, a_s, ..., a_s]}_{N}) = \mathsf{E}(\pi(\underbrace{[a_s, a_s, ..., a_s]}_{N})),$$

we only need to prove

$$\mathsf{E}(\pi(\underbrace{[a_s, a_s, ..., a_s]}_{N})) = \mathsf{E}(a_s + 0x^1 + ... + 0x^{N'-1}).$$

The encoding function (i.e., π) is performed as follows:

$$\pi([a_s, ..., a_s]) = \mathbf{V}^{-1} \begin{bmatrix} a_s \\ \vdots \\ a_s \end{bmatrix},$$

where $\mathbf{V}^{-1}$ is the inverse of Vandermonde matrix $\mathbf{V}(\zeta_0, \zeta_1, \cdots, \zeta_{N-1})$. Thereby, we just need to prove:

$$\mathbf{V}^{-1} \begin{bmatrix} a_s \\ a_s \\ \vdots \\ a_s \end{bmatrix} = \begin{bmatrix} a_s \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{5}$$

By multiplying $\mathbf{V}(\zeta_0, \zeta_1, \cdots, \zeta_{N-1})$ to the left-hand side of Equation 5, we can get:

$$\mathbf{V}(\zeta_0, \zeta_1, \cdots, \zeta_{N-1})\mathbf{V}^{-1} \begin{bmatrix} a_s \\ a_s \\ \vdots \\ a_s \end{bmatrix} = \begin{bmatrix} a_s \\ a_s \\ \vdots \\ a_s \end{bmatrix}.$$

By multiplying $\mathbf{V}(\zeta_0, \zeta_1, \cdots, \zeta_{N-1})$ to the right-hand side of Equation 5, we can get:

$$\mathbf{V}(\zeta_0, \zeta_1, \cdots, \zeta_{N-1}) \begin{bmatrix} a_s \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & \zeta_0 & \cdots & \zeta_0^{n-1} \\ 1 & \zeta_1 & \cdots & \zeta_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_{N-1} & \cdots & \zeta_{N-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_s \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} a_s \\ a_s \\ \vdots \\ a_s \end{bmatrix},$$

which concludes the proof. □

4.1 Application to matrix multiplication

This compression-and-decompression technique can be naturally applied to `MatrixMul`, as described in §3.1.1. Additionally, we introduce a further optimization based on the observation that different $\mathbf{A} \in \mathbb{R}^{m \times n}$ matrices need to be multiplied with the same $\mathbf{W} \in \mathbb{R}^{n \times k}$ in transformer inference. For example, in GPT, the model autoregressively generates response words, requiring model inference for multiple words (different $\mathbf{A}$s) [26]; In BERT, batch inference is typically used to process multiple input samples (different $\mathbf{A}$s) simultaneously [47]. Our goal is to reduce the amortized cost of `MatrixMul` by exploiting this fact.

Let $\mathbf{A} = [\mathbf{a}_0, ..., \mathbf{a}_{n-1}]$ with $\mathbf{a}_i \in \mathbb{R}^m$ being each column of $\mathbf{A}$. Suppose $\mathcal{S}$ and C need to generate t response words, then there

are t input matrices:

$$\mathbf{A}_0 = [\mathbf{a}_{0,0}, \mathbf{a}_{0,1}, ..., \mathbf{a}_{0,n-1}]$$
$$\mathbf{A}_1 = [\mathbf{a}_{1,0}, \mathbf{a}_{1,1}, ..., \mathbf{a}_{1,n-1}]$$
$$.....$$
$$\mathbf{A}_{t-1} = [\mathbf{a}_{t-1,0}, ..., \mathbf{a}_{t-1,n-1}]$$

Let $\mathbf{a}'_i = \begin{bmatrix} \mathbf{a}_{0,i} \\ \vdots \\ \mathbf{a}_{t-1,i} \end{bmatrix}$ and $\mathbf{q}'_j := \sum_{i=0}^{n-1} \mathbf{a}'_i w_{i,j}\ \forall\ j \in [k]$, then

$$\mathbf{Q}' = \mathbf{q}'_0||\mathbf{q}'_1||...||\mathbf{q}'_{k-1} = \begin{bmatrix} \mathbf{A}_0\mathbf{W} \\ \vdots \\ \mathbf{A}_{t-1}\mathbf{W} \end{bmatrix}$$

To this end, we introduce a preprocessing phase, where $\mathcal{S}$ sends C the *compressed* $(\mathsf{Enc}_{\mathcal{S}}(\underbrace{[w_{i,j}, ..., w_{i,j}]}_{t\times m})\ \forall\ i \in [n], j \in [k])$,[3] using our compression technique described in §4. Notice that this transfer occurs only once, unless the model changes. Next, C performs decompression to obtain $\mathsf{Enc}_{\mathcal{S}}(\underbrace{[w_{i,j}, ..., w_{i,j}]}_{t\times m})\ \forall\ i \in [n], j \in [k]$. If $t \times m > N$, each $\underbrace{[w_{i,j}, ..., w_{i,j}]}_{t\times m}$ occupies multiple ciphertexts. As C has no knowledge about the inputs (i.e., $\mathbf{A}$s) in the preprocessing phase, it samples $\mathbf{U} \in_{\$} \mathbb{R}^{(tm)\times n}$, and computes:

$$\mathsf{Enc}_{\mathcal{S}}(\mathbf{v}_j) \leftarrow \boxplus_{i=0}^{n-1} \Big(\mathbf{u}_i \boxtimes \mathsf{Enc}_{\mathcal{S}}(\underbrace{[w_{i,j}, ..., w_{i,j}]}_{t\times m})\Big),\ \forall\ j \in [k]$$

where $\mathbf{u}_i$ is the i-th column of $\mathbf{U}$. Next, C encrypts each $\mathsf{Enc}_{\mathcal{S}}(\mathbf{v}_j)$ with its own key and sends each $\mathsf{Enc}_C(\mathsf{Enc}_{\mathcal{S}}(\mathbf{v}_j))$ to $\mathcal{S}$. Notice that $\mathsf{Enc}_C(\mathsf{Enc}_{\mathcal{S}}(\mathbf{v}_j)) \equiv \mathsf{Enc}_{\mathcal{S}}(\mathsf{Enc}_C(\mathbf{v}_j))$ (see Appendix B for more details), hence $\mathcal{S}$ can decrypt it and get $\mathsf{Enc}_C(\mathbf{v}_j)\ \forall\ i \in [k]$.

In the online phase, after knowing $\mathbf{A}' = \mathbf{a}'_0||\mathbf{a}'_1||...||\mathbf{a}'_{n-1}$, C sends $(\mathbf{A}' - \mathbf{U})$ to $\mathcal{S}$, which can be regarded as an one-time-pad encryption of $\mathbf{A}'$, given that $\mathcal{S}$ does not know $\mathbf{U}$. Then, $\mathcal{S}$ computes:

$$\begin{aligned} &(\mathbf{A}' - \mathbf{U})\mathbf{W} \boxplus (\mathsf{Enc}_C(\mathbf{v}_0)||\mathsf{Enc}_C(\mathbf{v}_1)||...||\mathsf{Enc}_C(\mathbf{v}_k)) \\ =\ &(\mathbf{A}'\mathbf{W} - \mathbf{V}) \boxplus (\mathsf{Enc}_C(\mathbf{v}_0)||\mathsf{Enc}_C(\mathbf{v}_1)||...||\mathsf{Enc}_C(\mathbf{v}_k)) \\ =\ &\mathsf{Enc}_C(\mathbf{q}'_0)||\mathsf{Enc}_C(\mathbf{q}'_1)||...||\mathsf{Enc}_C(\mathbf{q}'_{k-1}) \end{aligned}$$

where $\mathbf{q}'_j$ is the j-th column of $\mathbf{Q}'$. Algorithm 6 shows the detailed process of the optimized `MatrixMul`. Its security proof can be found in Appendix C.

Recall that the subsequent `MatrixMul` requires row-wise encryption for both $\mathbf{Q}$ and $\mathbf{K}$ (cf. §3.1.2). Nevertheless, it is noteworthy that the multiplication of $\mathbf{Q}$ and $\mathbf{K}^T$ remains feasible even when subjected to column-wise encryption. Subsequently, the obtained ciphertexts in each row can be combined, leading to a form of row-wise encryption that facilitates the seamless execution of subsequent operations. We provide more details in Appendix D.

[3] We use $\mathsf{Enc}_{\mathcal{S}}$ to denote an encryption under $\mathcal{S}$'s public key. Similarly, we use Enc_C to denote an encryption under C's public key.

Table 2: Amortized communication cost for t times of matrix multiplication ($\mathbb{R}^{m\times n} \cdot \mathbb{R}^{n\times k}$). N' is #elements batched in a ciphertext. We also provide an example with real parameters in BERT-base and GPT-2: $m = 128, n = 768, k = 768, N' = 8192, t = 256, e = 144$ (the e parameter is only for CipherGPT).

Methods	# FHE Ciphertexts	Example
Cheetah [27]	$\geq \frac{2m\sqrt{nk}}{\sqrt{N'}}$	2172
Iron [24]	$\geq \frac{2\sqrt{mnk}}{\sqrt{N'}}$	192
CipherGPT [26]	$\frac{2enk}{tN'}$	81
BOLT [40]	$\frac{m(n+k)}{N'}$	24
Ours	$(\lceil \frac{nk}{N'} \rceil + k\lceil \frac{mt}{N'} \rceil)/t$	13

We remark that this optimized `MatrixMul` does not compromise the non-interactive property of NEXUS: C only needs to send $(\mathbf{A}' - \mathbf{U})$ to $\mathcal{S}$ and receive the inference result in online phase. Table 2 compares the communication cost of NEXUS with the state-of-the-art `MatrixMul` protocols.

5 SIMD SLOTS FOLDING

Recall that the rows of matrices $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ are encrypted in SIMD format. Subsequent operations like inner-product (cf. §3.1.2), `Softmax` (cf. §3.1.3), `LayerNorm` (cf. §3.2) and `Argmax` (cf. §3.4) involve computing a function $f()$ on all SIMD slots and applying the result s to each individual slot. For example, in `Softmax` and `LayerNorm`, given $\mathsf{Enc}([a_0, ..., a_{N-1}])$, $\mathcal{S}$ needs to obtain $\mathsf{Enc}(\underbrace{[s, ..., s]}_{N})$ where $s = \sum_{i=0}^{N-1} a_i$. In this case, $f()$ is a sum function.

In this section, we provide a generic solution applicable to all functions that supports *associativity*:

$$f(f(a_0, a_1), a_2) = f(a_0, f(a_1, a_2))$$

As our solution operates like a *fold* functionality in functional programming, we dub it *SIMD slots folding*.

A trivial solution is to rotate $\mathsf{Enc}([a_0, ..., a_{N-1}])$ for $(N-1)$ times and subsequently apply $f()$ to the resulting ciphertexts. Suppose $N = 4$, the rotated ciphertexts are:

$$\tilde{\mathbf{a}}_0 := \mathsf{Enc}([a_0, a_1, a_2, a_3]),$$
$$\tilde{\mathbf{a}}_1 := \mathsf{Enc}([a_1, a_2, a_3, a_0]),$$
$$\tilde{\mathbf{a}}_2 := \mathsf{Enc}([a_2, a_3, a_0, a_1]),$$
$$\tilde{\mathbf{a}}_3 := \mathsf{Enc}([a_3, a_0, a_1, a_2]).$$

We can aggregate them by employing a binary tree structure:

$$\tilde{\mathbf{a}}_{0,1} := f(\tilde{\mathbf{a}}_0, \tilde{\mathbf{a}}_1) = \mathsf{Enc}([f(a_0, a_1), f(a_1, a_2), f(a_2, a_3), f(a_3, a_0)]),$$
$$\tilde{\mathbf{a}}_{2,3} := f(\tilde{\mathbf{a}}_2, \tilde{\mathbf{a}}_3) = \mathsf{Enc}([f(a_2, a_3), f(a_3, a_0), f(a_0, a_1), f(a_1, a_2)]);$$

and then:

$$f(\tilde{\mathbf{a}}_{0,1}, \tilde{\mathbf{a}}_{2,3}) = \mathsf{Enc}([s, s, s, s]).$$

This trivial solution requires $N - 1$ rotations.

Algorithm 6 Optimized Secure `MatrixMul` on RNS-CKKS

Input: C holds $\mathbf{A}' \in \mathbb{R}^{(tm)\times n}$ and $\mathcal{S}$ holds $\mathbf{W} \in \mathbb{R}^{n\times k}$.

Output: Let $\alpha = \lceil \frac{tm}{N} \rceil$, $\mathcal{S}$ gets $\mathsf{Enc}_c(\mathbf{q}'_{j,0})||...||\mathsf{Enc}_c(\mathbf{q}'_{j,\alpha-1})\ \forall\ j \in [k]$, where $\mathbf{q}'_{j,0}||...||\mathbf{q}'_{j,\alpha-1}$ is the j-th column of $\mathbf{A}' \cdot \mathbf{W}$.

Preprocessing:

1: $\mathcal{S}$ packs the elements in $\mathbf{W}$ into $\lceil \frac{n\cdot k}{N'} \rceil$ polynomials (notice that the order is not important), and sends the encrypted polynomials to C.

2: C runs Algorithm 5 to decompress these encrypted polynomials, and obtains $\mathsf{Enc}(\underbrace{[w_{i,j}, ..., w_{i,j}]}_{N})\ \forall\ i \in [n]\ j \in [k]$.

3: C samples $\mathbf{U} \in_{\$} \mathbb{R}^{(tm)\times n}$, and computes:

$$\mathsf{Enc}_{\mathcal{S}}(\mathbf{v}_{j,0})||...||\mathsf{Enc}_{\mathcal{S}}(\mathbf{v}_{j,\alpha-1}) \leftarrow \boxplus_{i=0}^{n-1}\Big((\mathbf{u}_{i,0} \boxtimes \mathsf{Enc}_{\mathcal{S}}(\underbrace{[w_{i,j}, ..., w_{i,j}]}_{N}))||...||(\mathbf{u}_{i,\alpha-1} \boxtimes \mathsf{Enc}_{\mathcal{S}}(\underbrace{[w_{i,j}, ..., w_{i,j}]}_{N}))\Big)\ \forall\ j \in [k],$$

where $\mathbf{u}_{i,0}||...||\mathbf{u}_{i,\alpha-1}$ is the i-th column of $\mathbf{U}$. In RNS-CKKS, C samples each $\mathbf{u}_{i,j}$ as a random polynomial and multiplies it directly to $\mathsf{Enc}_{\mathcal{S}}(\underbrace{[w_{i,j}, ..., w_{i,j}]}_{N})$.

4: C encrypts each $\mathsf{Enc}_{\mathcal{S}}(\mathbf{v}_{j,i})$ with its own key, and then sends each $\mathsf{Enc}_C(\mathsf{Enc}_{\mathcal{S}}(\mathbf{v}_{j,i}))$ to $\mathcal{S}$, who decrypts and gets $\mathsf{Enc}_C(\mathbf{v}_{j,i})$.

Online:

5: For $\forall\ i\ \in [n]$, C sends $(\mathbf{a}'_{i,0} - \mathbf{u}_{i,0})||...||(\mathbf{a}'_{i,\alpha-1} - \mathbf{u}_{i,\alpha-1})$ to $\mathcal{S}$, where $\mathbf{a}'_{i,0}||...||\mathbf{a}'_{i,\alpha-1}$ is the i-th column of $\mathbf{A}'$. In RNS-CKKS, C encodes each $\mathbf{a}'_{i,j}$ as a polynomial and subtracts the $\mathbf{u}_{i,j}$ polynomial.

6: $\mathcal{S}$ computes $\mathbf{Z} := (\mathbf{A}' - \mathbf{U})\mathbf{W}$. For $\forall\ j\ \in [k]$, $\mathcal{S}$ computes $(\mathbf{z}_{j,0} \boxplus \mathsf{Enc}_C(\mathbf{v}_{j,0}))||...||(\mathbf{z}_{j,\alpha-1} \boxplus \mathsf{Enc}_C(\mathbf{v}_{j,\alpha-1}))$.

Our key observation is that $\tilde{\mathbf{a}}_{2,3}$ can be obtained by left-rotating $\tilde{\mathbf{a}}_{0,1}$ by two slots, hence there is no need to compute $\tilde{\mathbf{a}}_2$ and $\tilde{\mathbf{a}}_3$ at all. More generally, each right-child in the binary tree can be obtained by left-rotating the corresponding right-child by 2^i slots. Given that we know the left-most leaf (i.e., $\tilde{\mathbf{a}}_0$), we can compute the root (i.e., the final result s) in the form akin to a "Merkle tree" (cf. Figure 5). Notice that when the number of rotated slots is a power of two, the rotation overhead is equal to a single rotation. As a result, our solution only requires $(\log N - 1)$ rotations.

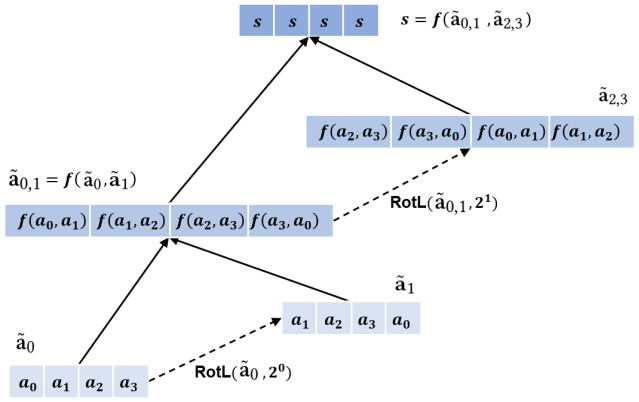


Figure 5: A toy example for computing $f()$ on SIMD slots ($n = N$).

However, this solution is applicable only when the length n of the input vector is equal to N, whereas $n \ll N$ in the transformer we evaluated. In this case, we could transform $\mathsf{Enc}([a_0, ..., a_{n-1}, \underbrace{0, ..., 0}_{N-n}])$ into $\mathsf{Enc}([a_0, ..., a_{n-1}, a_0, ..., a_{n-1}, \underbrace{0, ..., 0}_{N-2n}])$ and then proceed with the aforementioned process. Figure 6 provides a visualization of the entire process and Algorithm 7 outlines the detailed steps. It requires $\log n$ rotations in total.

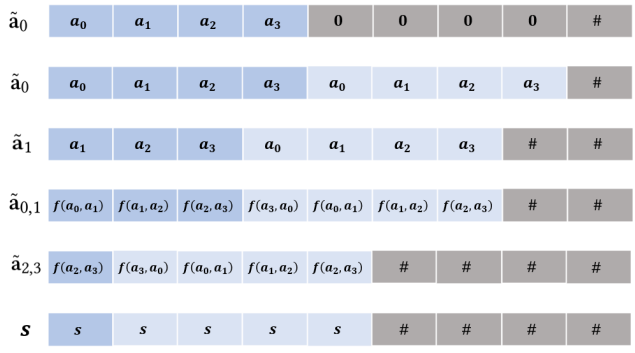


Figure 6: A toy example for computing $f()$ on SIMD slots ($n \ll N$); # means the value is not important.

We remark that the $(N - 2n)$ empty slots can be used to fold other $\tilde{\mathbf{a}}$s, thereby we can process $\frac{N}{2n}$ vectors with a single SIMD ciphertext. For example, in `LayerNorm` of BERT-base and GPT-2, $n = 128$; when $N = 16,384$, we can process 32 vectors with one ciphertext.

Algorithm 7 SIMD slots folding on RNS-CKKS

Input: $\tilde{\mathbf{a}} = \mathsf{Enc}([a_0, ..., a_{n-1}, 0, ..., 0])$ with $2n < N$

Output: $\mathsf{Enc}([\underbrace{s, ..., s}_{n}, 0, ..., 0])$, where $s = f(a_0, ..., a_{n-1})$

1: **function** $\mathsf{Fold}(\tilde{\mathbf{a}})$
2: $\quad\tilde{\mathbf{a}} \leftarrow \tilde{\mathbf{a}} \boxplus \mathsf{RotR}(\tilde{\mathbf{a}}, n)$
3: $\quad$**for** $i = 0$ to $\log n - 1$ **do**
4: $\quad\quad\tilde{\mathbf{s}} \leftarrow \mathsf{RotL}(\tilde{\mathbf{a}}, 2^i)$ // left-rotate by 2^i steps
5: $\quad\quad\tilde{\mathbf{s}} \leftarrow f(\tilde{\mathbf{s}}, c)$
6: $\quad\quad\tilde{\mathbf{a}} := \tilde{\mathbf{s}}$
7: $\quad$**end for**
8: $\quad$**return** $\tilde{\mathbf{s}} \boxtimes [\underbrace{1, ..., 1}_{n}, 0, ..., 0]$
9: **end function**

5.1 QuickSum

Given $[a_0, ..., a_{n-1}, 0, ..., 0]$, $\mathcal{S}$ can obtain $[\underbrace{\sum_{i=0}^{N-1} a_i, ..., \sum_{i=0}^{N-1} a_i}_{n}, 0, ..., 0]$ through Algorithm 7 by replacing Line 5 with:

$$\tilde{\mathbf{s}} \leftarrow \tilde{\mathbf{s}} \boxplus \tilde{\mathbf{a}}.$$

5.2 QuickMax

Given $[a_0, ..., a_{n-1}, 0, ..., 0]$, $\mathcal{S}$ can obtain $\mathsf{Enc}([\underbrace{a_{max}, ..., a_{max}}_{n}, 0, ..., 0])$ where $a_{max} = max(a_0, a_2, ..., a_{n-1})$ through Algorithm 7 by replacing $f()$ with $max()$. We leverage

$$max(a, b) = \frac{a + b + (a - b) \cdot \mathsf{Sgn}(a - b)}{2}$$

to compute the *max* function on encrypted values. Then, Line 5 in Algorithm 7 is replaced with:

$$\tilde{\mathbf{s}} \leftarrow 0.5 \boxtimes (\tilde{\mathbf{a}} \boxplus \tilde{\mathbf{s}} \boxplus (\tilde{\mathbf{a}} \boxminus \tilde{\mathbf{s}}) \boxtimes \mathsf{Sgn}(\tilde{\mathbf{a}} \boxminus \tilde{\mathbf{s}})).$$

6 EVALUATION

6.1 Implementation

We implement NEXUS in C++, utilizing the SEAL library [48] for RNS-CKKS homomorphic encryption and FHE-MP-CNN [1] For bootstrapping. We use HEXL [8] to accelerate SEAL on Intel CPUs. We set the polynomial degree as $N' = 2^{16}$ (hence $N = 2^{15}$) and the ciphertext modulus as 1763-bit to achieve 128-bit security, following the "homomorphic encryption standard" [6]. We set the multiplicative depth as $L = 35$ and the depth for bootstrapping as $K = 14$, which indicates that the available multiplicative depth is $L - K = 21$. We set $q_0 \approx 2^{60}$ and $q_i \approx 2^{50}$ $\forall\, i \geq 1$. We leverage the scale propagation technique [9] to eliminate the dominant noise component.

6.2 Experimental setup

We primarily compare our work with Iron [24] and BOLT [40]. However, neither of their implementations is currently publicly available. To enable a direct comparison with the results (of both BOLT and Iron) reported in the BOLT paper, we conducted our benchmarks under the same experimental setting as BOLT:

- We conducted our benchmarks on two VM instances with 3.20GHz Intel Xeon processors and 128 GB RAM.
- We control the communication bandwidth between them using the Linux Traffic Control (tc) command. We set the bandwidth to 3Gbps and the round-trip latency to 0.8 ms to simulate the communication in LAN.
- Our simulation for WAN consisted of four settings: {100Mbps, 40ms}, {100Mbps, 80ms}, {200Mbps, 40ms}, and {200Mbps, 80ms}.
- The model parameters were taken from a pre-trained BERT-base transformer [16]. The hyperparameters are the same with BOLT.
- We set the the number of threads to 32, same as BOLT.

When measuring NEXUS, we did not distinguish between the pre-processing time and online time,[4] as BOLT did not do this. Additionally, BOLT introduces a *word elimination* technique that can significantly improve its performance but requires model fine-tuning. As NEXUS does not require any fine-tuning, we compare with their results that were obtained w.o. using word elimination. All results were averaged from 10 runs.

6.3 Microbenchmarks

Matrix Multiplication. Figure 7 shows the amortized cost of `MatrixMul` (cf. Algorithm 6) in LAN for multiple inputs. Considering that ChatGPT often generates several hundred words in a single response, $t = 256$ would be a reasonable number of inputs. The amortized runtime (of 256 inputs) of NEXUS is 2.26s, 15.9× faster than Iron and 3.3× faster than BOLT. When the number of inputs increases to 1,024, which is also quite common, NEXUS demonstrates even greater performance advantages. Specifically, it outperforms BOLT by 4.8× in runtime and 2.6× in communication.

Non-linear Functions. Table 3 shows the comparison of non-linear functions (`GELU, Softmax, LayerNorm, Argmax`). Iron and BOLT implement such non-linear functions through secure two-party computation, which is expensive in both bandwidth consumption and communication rounds. In contrast, NEXUS holds a superiority particularly in poor network conditions, owing to its non-interactive feature. For example, when the bandwith is 100Mbps and round-trip latency is 80ms, NEXUS achieves:

- 11.7× speedup over Iron and 3.3× speedup over BOLT for `GELU`;
- 14.3× speedup over Iron and 11.3× speedup over BOLT for `LayerNorm`;
- 7.9× speedup over Iron and 3.2× speedup over BOLT for `Softmax`;

Regarding `Argmax`, while both Phoenix and NEXUS are non-interactive, NEXUS demands notably fewer rotations and `Sgn` operations. As a result, NEXUS outperforms Phoenix by 13.3× in terms of speed.

[4] In NEXUS, C needs to transfer rotate keys and bootstrapping keys, 21.6 GB in total. However, this operation is a one-time requirement and has not been included in our reported results.

Table 3: Evaluation of non-linear functions.

Setting	Protocol	Comm (MB)	Rounds	LAN	WAN[1]	WAN[2]	WAN[3]	WAN[4]	Average Error
GELU $12 \times \mathbb{R}^{128\times3072}$	Iron [24]	7960.00 × 12	256 × 12	126	2049	2075	4114	4118	5.8×10^{-4}
	BOLT [40]	1471.67 × 12	88 × 12	14	389	416	762	774	9.8×10^{-4}
	NEXUS	**0**	**0**	**233**	**233**	**233**	**233**	**233**	7.7×10^{-4}
LayerNorm $24 \times \mathbb{R}^{128\times768}$	Iron [24]	871.46 × 24	218 × 24	16	558	759	996	1158	1.7×10^{-3}
	BOLT [40]	599.40 × 24	220 × 24	14	430	653	723	914	-
	NEXUS	**0**	**0**	**81**	**81**	**81**	**81**	**81**	4.5×10^{-4}
Softmax $12 \times \mathbb{R}^{128\times128}$	Iron [24]	3596.32 × 12	252× 12	60	930	964	1877	1900	3.2×10^{-5}
	BOLT [40]	1447.65 × 12	232 × 12	16	382	434	754	775	1.4×10^{-6}
	NEXUS	**0**	**0**	**242**	**242**	**242**	**242**	**242**	3.1×10^{-6}
Argmax $\mathbb{R}^{128}$	Phoenix [29]	0	0	366	366	366	366	366	4.0×10^{-4}
	NEXUS	**0**	**0**	**28**	**28**	**28**	**28**	**28**	2.5×10^{-6}

[1] 200Mbps, 40ms [2] 200Mbps, 80ms [3] 100Mbps, 40ms [4] 100Mbps, 80ms

The last column of Table 3 shows the average errors of these three schemes. For Iron and BOLT, we calculated their average errors via multiplying the ULP errors [45] reported in their papers by their respective scales (BOLT did not report their ULP errors for LayerNorm). For NEXUS, taking GELU as an example, we uniformly sampled $[x_1, ..., x_{1000}]$ from the corresponding domain. For each x_i, we calculated both the real $y_i = \texttt{GELU}(x)$ and the approximated $y_i' = \texttt{GELU}(x)$. The average error is then computed as $\frac{\sum_{i=1}^{1000} |y_i - y_i'|}{1000}$. The results indicate that the average errors introduced by NEXUS are comparable to those of prior work.

6.4 Macrobenchmarks

End-to-End Performance. The end-to-end performance is roughly the aggregation of the microbenchmarks. Additionally, Iron and BOLT need to perform secure truncations to prevent overflows, given their scaling of floating-point numbers to integers. In contrast, NEXUS avoids the need for truncations by leveraging RNS-CKKS, which supports floating-point numbers, but it incurs bootstrappings. Specifically, the end-to-end workflow of NEXUS follows Figure 3.

Figure 8 shows the end-to-end performance (amortized for 128 inputs). Notably, NEXUS only consumes 164MB bandwidth, 1737.5× reduction over Iron and 368.6× reduction over BOLT. In terms of end-to-end runtime, NEXUS still achieves upto 11.6× speedup over Iron and 2.8× speedup over BOLT.

Table 4 lists the runtime for each individual operation in NEXUS, along with their corresponding proportions. Bootstrapping is the most time-consuming part of the entire process, requiring 315s and occupying 37.72% of the total runtime. Following bootstrapping, Softmax and GELU are the next most time-consuming parts, occupying 21.72% and 20.92% of the total runtime, respectively.

Accuracy. We evaluate accuracy with 4 datasets from the GLUE benchmark [50], a widely adopted evaluation measure for BERT and GPT-based transformers. Three of these datasets pertain to BERT-base: RTE, SST-2, and QNLI, all involving classification tasks.

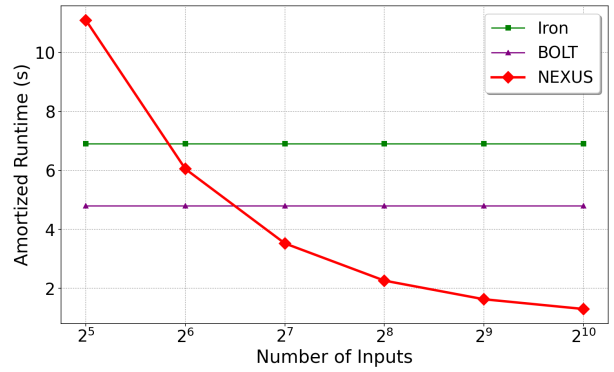
Table 4: Breaking down the end-to-end runtime of NEXUS.

Operation	Input	Time (s)	%
MatrixMul	$\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times768}$	9	0.81%
Softmax	$\mathbb{R}^{128\times128}$	242	21.72%
MatrixMul	$\mathbb{R}^{128\times128} \times \mathbb{R}^{128\times768}$	32	2.87%
Bootstrapping	$\mathbb{R}^{128\times768}$	105	9.43%
LayerNorm	$\mathbb{R}^{128\times768}$	40	3.59%
Bootstrapping	$\mathbb{R}^{128\times768}$	105	9.43%
MatrixMul	$\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times3072}$	36	3.23%
GELU	$\mathbb{R}^{128\times3072}$	233	20.92%
MatrixMul	$\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times3072}$	23	2.06%
Bootstrapping	$\mathbb{R}^{128\times768}$	105	9.43%
LayerNorm	$\mathbb{R}^{128\times768}$	40	3.59%
Bootstrapping	$\mathbb{R}^{128\times768}$	105	9.43%
Argmax	$\mathbb{R}^{128}$	28	2.51%
Total	-	1,103	-

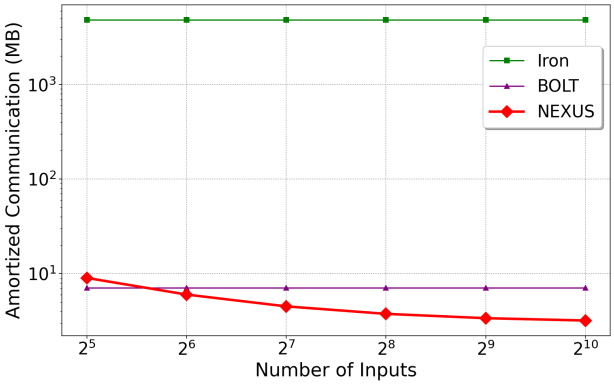
The remaining dataset, Children's Book Test (CBT), pertains to GPT, evaluating accuracy by having GPT predict the correct word choice from 10 options in a cloze test. As shown in Table 5, NEXUS attains comparable levels of accuracy when compared to the plaintext inference.

7 RELATED WORK

Interactive secure inference for transformers. With the proliferation of ChatGPT, secure transformer inference has become a key area of research. Privformer [4], Puma [17] and Sigma [22] are



(a) Amortized Runtime vs. #inputs.



(b) Amortized Communication vs. #inputs.

Figure 7: Evaluation of ciphertext-plaintext matrix multiplication for $\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times768}$ in LAN (amortized cost of multiple inputs).

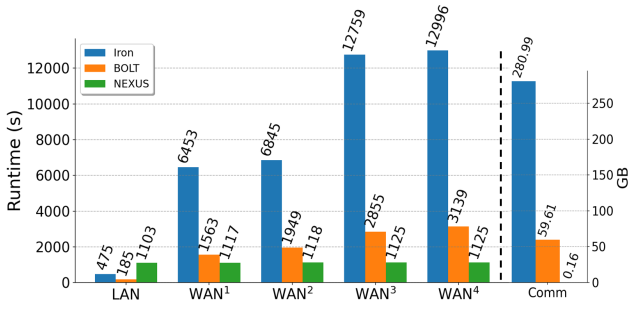


Figure 8: End-to-end inference performance (amortized cost for 128 inputs).

three-party protocols that requires additional trust assumptions. There are several works on secure transformer inference based on 2PC [10, 24, 26, 35, 38, 40]. Iron [24] is an optimization of a secure CNN protocol named Cheetah [27] and uses a more efficient packing strategy to reduce the cost of matrix multiplication.

Table 5: Inference accuracy on the GLUE benchmarks using BERT-base and GPT-2.

Model	Dataset	Plaintext	NEXUS
BERT-base	RTE	70.04%	69.88%
	SST-2	92.36%	92.11%
	QNLI	90.30%	89.92%
GPT-2	CBT-CN	85.70%	85.31%

Bumblebee [38] further optimizes the packing strategy. Similar to NEXUS, all these three protocols use polynomial coefficients to pack matrices, but they did not make full use of the coefficients (i.e., a large number of coefficients are wasted). In contrast, NEXUS can use all coefficients to pack matrices, resulting in a much less number of ciphertexts needed to be transferred. THE-X [10] and MPCFormer [35] simply replace GELU, Softmax with a combination of ReLU and polynomials, hence both of them require model retraining. BOLT [40] is the state-of-the-art solution for secure transformer inference. Our experimental results show that NEXUS achieves a speedup of 2.8× and a remarkable bandwidth reduction of 368.6×, compared to BOLT.

Non-interactive secure inference. To the best of our knowledge, all existing non-interactive secure inference protocols [7, 20, 28, 33, 37, 44] are designed for convolutional neural networks (CNNs). CryptoNAS [20] and DeepReduce [28] are proposed to run the non-linear functions like ReLU under FHE, but they cannot support the non-linear functions required by transformers, such as GELU, softmax and layer normalization. AutoFHE [7] can automatically optimize the placement of bootstrapping operations in a CNN workflow. NEXUS is arguably the first protocol for non-interactive secure transformer inference.

FHE acceleration. Recent research on optimizing compilers [14, 15, 49], GPU acceleration [5, 51], and specialized hardware accelerators [3, 31, 46] has demonstrated significant speedups for RNS-CKKS. These results can be used directly to accelerate NEXUS. We did not utilize them in our benchmarks because we aim to provide a fair comparison with prior work.

8 CONCLUSION

We propose NEXUS, the first secure inference protocol for transformers without the need of further interactions between the client and the server. We propose a series of new protocols for RNS-CKKS so that the server can efficiently and accurately compute each layer of the transformer on the encrypted data. Given that the non-interactive protocol is not limited by network bandwidth, we posit that combining NEXUS with hardware acceleration makes secure transformer inference ready for practical deployment.

REFERENCES

[1] [n. d.]. FHE-MP-CNN. https://github.com/snu-ccl/FHE-MP-CNN.
[2] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J Kusner, and Adrià Gascón. 2019. QUOTIENT: two-party secure neural network training and prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1231–1247.

[3] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 882–895.

[4] Yoshimasa Akimoto, Kazuto Fukuchi, Youhei Akimoto, and Jun Sakuma. 2023. Privformer: Privacy-preserving Transformer with MPC. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroSP)*. 392–410. https://doi.org/10.1109/EuroSP57164.2023.00031

[5] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. 2020. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (2020), 379–391.

[6] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, et al. 2021. Homomorphic encryption standard. *Protecting privacy through homomorphic encryption* (2021), 31–62.

[7] Wei Ao and Vishnu Naresh Boddeti. 2024. AutoFHE: Automated Adaption of CNNs for Efficient Evaluation over FHE. 33st USENIX Security Symposium (USENIX Security 24).

[8] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. 2021. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 57–62.

[9] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 587–617.

[10] Tianyu Chen, Hangbo Bao, Shaohan Huang, Li Dong, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. 2022. THE-X: Privacy-Preserving Transformer Inference with Homomorphic Encryption. In *Findings of the Association for Computational Linguistics: ACL 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 3510–3520. https://doi.org/10.18653/v1/2022.findings-acl.277

[11] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. 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*. Springer, 347–368.

[12] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 409–437.

[13] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. 2020. Efficient homomorphic comparison methods with optimal complexity. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part II 26*. Springer, 221–256.

[14] Sangeeta Chowdhary, Wei Dai, Kim Laine, and Olli Saarikivi. 2021. Eva improved: Compiler and extension library for ckks. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 43–55.

[15] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*. 142–156.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[17] Ye Dong, Wen-jie Lu, Yancheng Zheng, Haoqi Wu, Derun Zhao, Jin Tan, Zhicong Huang, Cheng Hong, Tao Wei, and Wenguang Cheng. 2023. Puma: Secure inference of llama-7b in five minutes. *arXiv preprint arXiv:2307.12533* (2023).

[18] Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. 2024. BLEACH: cleaning errors in discrete computations over CKKS. *Journal of Cryptology* 37, 1 (2024), 3.

[19] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Stanford university.

[20] Zahra Ghodsi, Akshaj Kumar Veldanda, Brandon Reagen, and Siddharth Garg. 2020. Cryptonas: Private inference on a relu budget. *Advances in Neural Information Processing Systems* 33 (2020), 16961–16971.

[21] Robert E Goldschmidt. 1964. *Applications of division by convergence*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[22] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. 2023. SIGMA: secure GPT inference with function secret sharing. *Cryptology ePrint Archive* (2023).

[23] Kyoohyung Han and Dohyeong Ki. 2020. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*. Springer, 364–390.

[24] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. 2022. Iron: Private inference on transformers. *Advances in Neural Information Processing Systems* 35 (2022), 15718–15731.

[25] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* (2016).

[26] Xiaoyang Hou, Jian Liu, Jingyu Li, Yuhan Li, Wen-jie Lu, Cheng Hong, and Kui Ren. 2023. Ciphergpt: Secure two-party gpt inference. *Cryptology ePrint Archive* (2023).

[27] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and fast secure {two-party} deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)*. 809–826.

[28] Nandan Kumar Jha, Zahra Ghodsi, Siddharth Garg, and Brandon Reagen. 2021. Deepreduce: Relu reduction for fast private inference. In *International Conference on Machine Learning*. PMLR, 4839–4849.

[29] Nikola Jovanovic, Marc Fischer, Samuel Steffen, and Martin Vechev. 2022. Private and reliable neural network inference. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1663–1677.

[30] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*. 1651–1669.

[31] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[33] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *International Conference on Machine Learning*. PMLR, 12403–12422.

[34] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. 2021. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2021), 3711–3727.

[35] Dacheng Li, Rulin Shao, Hongyi Wang, Han Guo, Eric P Xing, and Hao Zhang. 2023. MPCFormer: fast, performant and private Transformer inference with MPC. *International Conference on Learning Representations (ICLR)* (2023).

[36] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. 2017. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 619–631.

[37] Qian Lou and Lei Jiang. 2021. Hemet: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. In *International conference on machine learning*. PMLR, 7102–7110.

[38] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Kui Ren, Cheng Hong, Tao Wei, and WenGuang Chen. 2023. BumbleBee: Secure Two-party Inference Framework for Large Transformers. *Cryptology ePrint Archive* (2023).

[39] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2505–2522. https://www.usenix.org/conference/usenixsecurity20/presentation/mishra

[40] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. 2024. BOLT: Privacy-Preserving, Accurate and Efficient Inference for Transformers. *IEEE Symposium on Security and Privacy (SP)* (2024).

[41] Hongyuan Qu and Guangwu Xu. [n. d.]. Improvements of Homomorphic Evaluation of Inverse Square Root. *Available at SSRN 4258571* ([n. d.]).

[42] Hongyuan Qu and Guangwu Xu. 2023. Improvements of Homomorphic Secure Evaluation of Inverse Square Root. In *International Conference on Information and Communications Security*. Springer, 110–127.

[43] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[44] Ran Ran, Xinwei Luo, Wei Wang, Tao Liu, Gang Quan, Xiaolin Xu, Caiwen Ding, and Wujie Wen. 2023. SpENCNN: orchestrating encoding and sparsity for fast homomorphically encrypted neural network inference. In *International Conference on Machine Learning*. PMLR, 28718–28728.

[45] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. Sirnn: A math library for secure rnn inference. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1003–1020.

[46] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 238–252.

[47] SEAL 2023. Microsoft batch-inference. https://github.com/microsoft/batch-inference. Microsoft Research, Redmond, WA..

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

[49] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2023. {HECO}: Fully Homomorphic Encryption Compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4715–4732.
[50] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).
[51] Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng. 2023. HE-Booster: An Efficient Polynomial Arithmetic Acceleration on GPUs for Fully Homomorphic Encryption. *IEEE Transactions on Parallel and Distributed Systems* 34, 4 (2023), 1067–1081.

A CORRECTNESS OF CIPHERTEXT DECOMPRESSION

Theorem 2. *Let N' be a power of 2, $p(x) = a_0 + a_1x^1 + ... + a_{N'-1}x^{N'-1}$ be the polynomial encoding of* $\mathbf{A}$*, and* $\mathsf{E}(p(x))$ *be the encryption of $p(x)$. Then, the N' output ciphertexts $o_0, ..., o_{N-1}$ of Decompress*$(\mathsf{E}(p(x)))$ *satisfy:*

$$o_s = \mathsf{Enc}(a_s + 0x^1 + 0x^2 + ... + 0x^{N-1}) \; \forall s \in [N']$$

Proof. It suffices to prove the case $N' = 2^\ell$. For $j = \{0, 1, ..., \ell - 1\}$, we claim that after j^{th} iteration of the outer loop, we have ciphertexts $= [c_0, \ldots, c_{2^{j+1}-1}]$ such that

$$c_s = \mathsf{E}\left(2^{j+1} \sum_{i=0}^{N'-1} \left[a_i x^{i-s}\right]_{i \equiv s \bmod 2^{j+1}}\right)$$

We prove the claim by induction on j. The base case $j = 0$ is explained before. Suppose the claim is true for some $j \geq 0$. Then in the next iteration, there is an integer r such that $i - s = 2^{j+1} \cdot r$, then we compute an array

$$\begin{aligned}
c_s^{'} &= c_s + \mathsf{Subs}(c_s, N/2^{j+1} + 1) \\
&= c_s + \mathsf{E}\left(2^{j+1} \sum_{i=0}^{N-1} \left[a_i x^{(N/2^{j+1}+1)(2^{j+1}r)}\right]_{i \equiv s \bmod 2^{j+1}}\right) \\
&= c_s + \mathsf{E}\left(2^{j+1} \sum_{i=0}^{N-1} \left[a_i (-1)^r x^{i-s}\right]_{i \equiv s \bmod 2^{j+1}}\right) \\
&= \mathsf{E}\left(\left[1 + (-1)^r\right] \cdot 2^{j+1} \sum_{i=0}^{N-1} \left[a_i x^{i-s}\right]_{i \equiv s \bmod 2^{j+1}}\right) \\
&= \mathsf{E}\left(2^{j+2} \sum_{i=0}^{N-1} \left[a_i x^{i-s}\right]_{i \equiv s \bmod 2^{j+2}}\right)
\end{aligned}$$

It is necessary to explain that when r is odd, it is clear that the corresponding term will be eliminated. When r is even, let's denote it as $r = 2r'$ (where r' is an integer). In this case, only the terms satisfying $i - s = 2^{j+1} \cdot 2r'$ will be left, and this condition can also be expressed as $i \equiv s \bmod 2^{j+2}$.

Finally, with the above claim we show that after the outer loop, where $j = \ell - 1$, we have an array of N ciphertexts such that:

$$\begin{aligned}
o_s &= \mathsf{E}\left(2^{j+1} \cdot \sum_{i=0}^{N-1} \left[a_i x^{i-s}\right]_{i \equiv s \bmod 2^{j+1}}\right) \cdot \frac{1}{N} \\
&= \mathsf{E}\left(N \cdot \sum_{i=0}^{N-1} \left[a_i x^{i-s}\right]_{i \equiv s \bmod N}\right) \cdot \frac{1}{N} \\
&= \mathsf{E}(a_s + 0x^1 + 0x^2 + ... + 0x^{N-1})
\end{aligned}$$

Note that $i < N = 2^\ell$, so $i \equiv s \bmod N$ implies $i = s$. Hence o_s is an encryption of monomial $Na_s + 0x^1 + ... + 0x^{N-1}$. To obtain an encryption of a_s, we multiply o_s by $\frac{1}{N}$ in the last step (Line 12-15 in Algorithm 5). □

B COMMUTABLE ENCRYPTION

A RLWE ciphertext consists of a pair of polynomials $(A, As + m + e)$. Then, $\mathsf{Enc}_C(\mathsf{Enc}_S(m))$ can be obtained by letting the client run the following procedure:

(1) Parse $\mathsf{Enc}_S(m)$ as $(A, As_S + m + e)$
(2) Output $(A, As_S + As_C + m + e + e')$

Decrypting it with the server's secret key yields:

$$(A, As_S + As_C + m + e + e') - (0, As_S) = (A, As_C + m + e + e').$$

Which is a valid ciphertext under client's secret key.

C SECURITY PROOF FOR MATRIX MULTIPLICATION

We here provide a security proof for Algorithm 6.

Correctness. The algorithm calculates $\mathbf{U} \cdot \mathbf{W}$ in the offline phase and $\mathbf{A} \cdot \mathbf{W} = \mathbf{A} - \mathbf{U} \cdot \mathbf{W} + \mathbf{U} \cdot \mathbf{W}$ in the online phase. The correctness of the rest of the algorithm derives directly from the correctness of Algorithm 5 and the homomorphic encryption scheme.

Privacy. We define privacy by the simulation paradigm. Namely, the algorithm should be secure against a static semi-honest probabilistic polynomial time adversary corrupt either C or $\mathcal{S}$.

- **Corrupted client.** We require that a corrupted, semi-honest client does not learn anything about the server's input $\mathbf{W}$. Formally, we require the existence of an efficient simulator Sim_C such that $View_C \approx_c Sim_C(\mathbf{A})$, where $View_C$ denotes the view of the client in the execution (the view includes the client's input, randomness, and the transcript of the protocol).
- **Corrupted server.** We require that a corrupted, semi-honest server does not learn anything about the private input $\mathbf{A}$ of the client. Formally, we require the existence of an efficient simulator Sim_S such that $View_S \approx_c Sim_S(\mathbf{W}, out)$, where $View_S$ denotes the view of the server in the execution, and out denotes the output, namely $\mathsf{Enc}_c(\mathbf{A} \cdot \mathbf{W})$.

The functionality of Algorithm 6 is denoted by $\mathcal{F}_{Matrixmul}$. We summarize the privacy of Algorithm 6 by theorem 3.

Theorem 3. *Assuming $\mathcal{F}_{Enc}$ is the homomorphic encryption functionality. Algorithm 6 is a protocol that securely realize $\mathcal{F}_{Matrixmul}$ in the $\mathcal{F}_{Enc}$ model.*

Proof. **Corrupted client.** The client view consists of ciphertexts $\{\mathsf{Enc}_S(w_Y)\}$. The simulator Sim_C can be constructed by:

(1) Output ciphertexts $Enc_S(0)$.

The security against a corrupted client is directly reduced to the semantic security of the underlying RLWE encryption.

Corrupted server. The server view consists of ciphertexts $\{\mathsf{Enc}_C(\mathsf{Enc}_S(\mathbf{v}_{\alpha,\delta}))\}$, plaintext polynomials $\mathbf{a}_{\alpha,\beta} - \mathbf{u}_{\alpha,\beta}$, and the output $\mathsf{Enc}_c(\mathbf{A} \cdot \mathbf{W})$. The simulator Sim_S can be constructed by:

(1) The simulator follows step 2 and 3 in the algorithm with the knowledge of $\mathbf{W}$. The only difference is that $\mathbf{u}_{\alpha,\beta}$ is replaced with $\hat{\mathbf{u}}_{\alpha,\beta}$ which is sample by the simulator instead of C such that

$$\mathsf{Enc}_{\mathcal{S}}(\hat{\mathbf{v}}_{\alpha,\delta}) \leftarrow \boxplus_{\beta\in[n]} \left(\hat{\mathbf{u}}_{\alpha,\beta} \boxtimes \mathsf{Enc}_{\mathcal{S}}(w'_{(\delta-1)n+\beta})\right)$$

The output ciphertexts $\{\mathsf{Enc}_C(\mathsf{Enc}_{\mathcal{S}}(\hat{\mathbf{v}}_{\alpha,\delta}))\}$ are indistinguishable from $\{\mathsf{Enc}_C(\mathsf{Enc}_{\mathcal{S}}(\mathbf{v}_{\alpha,\delta}))\}$ from the semantic security of the underlying RLWE encryption.

(2) The simulator samples and outputs random plaintext polynomials $\{p_{\alpha,\beta}\}$. The random plaintext polynomials are indistinguishable from $\{\mathbf{a}_{\alpha,\beta}-\mathbf{u}_{\alpha,\beta}\}$ as $\{\mathbf{u}_{\alpha,\beta}\}$ are uniformly random one-time-pads in the plaintext ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N+1)$. Therefore, $\{\mathbf{a}_{\alpha,\beta} - \mathbf{u}_{\alpha,\beta}\}$ are also uniformly random in $\mathcal{R}_Q$.

(3) The simulator receive the output $\mathsf{Enc}_c(\mathbf{A}\cdot\mathbf{W})$ and forward it.

□

D COLUMN-PACKED MATRIX MULTIPLICATION

Suppose the matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{m\times n}$ packed in column and $\mathbf{a}_j$, $\mathbf{b}_j$ are column vectors for $\forall\ j \in [n]$. We leverage the element wise multiplication of SIMD and find that $(\mathbf{a}_0 \boxtimes \mathbf{b}_0) \boxplus (\mathbf{a}_1 \boxtimes \mathbf{b}_1) \cdots \boxplus (\mathbf{a}_{n-1} \boxtimes \mathbf{b}_{n-1})$ is the diagonal-packed of matrix $\mathbf{A}\times\mathbf{B}^T$. And we can continue computing the other diagonal of matrix $\mathbf{A}\times\mathbf{B}^T$ by just rotating the vector $b_0, b_1, \cdots, b_{n-1}$ and get the result $r_0, r_1, \cdots, r_{n-1}$ in Algorithm 8.

Algorithm 8 Column-packed `MatrixMul`

Input: Column-packed matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{m\times n}$
Output: $\mathbf{A}\times\mathbf{B}^T \in \mathbb{R}^{m\times m}$

function `MatrixMul`(A, B)
  for $i = 0$ to $m-1$ **do**
    $\mathbf{r} \leftarrow \mathbf{0}$
    for $j = 0$ to $n-1$ **do**
      $\mathbf{r}_i \leftarrow \mathbf{r}_i \boxplus (\mathbf{a}_j \boxtimes \mathbf{b}_j)$
    end for
    for $j = 0$ to $n-1$ **do**
      $\mathbf{b}_j \leftarrow \mathsf{RotL}(\mathbf{b}_j, 1)$
    end for
  end for
  return $[\mathbf{r}_0, \mathbf{r}_1, \cdots, \mathbf{r}_{m-1}]$
end function

E SECURE INFERENCE

In this work we assume a static semi-honest probabilistic polynomial time adversary $\mathcal{A}$, who corrupt either the server $\mathcal{S}$ or the client C. The adversary $\mathcal{A}$ follows the protocol honestly. When $\mathcal{A}$ corrupts the server, it may try to learn the input of the client. When $\mathcal{A}$ corrupt the client, it tries to learn the model parameters. We adopt the definition of security from [39].

We summarize the correctness and security of our proposed protocol by theorem 4.

Theorem 4. *Suppose a server having as input model parameters $\mathbf{M} = (\mathbf{M}_1, ..., \mathbf{M}_\ell)$ and a client having as input a feature vector $\mathbf{x}$. The model M can be computed with functions $Gelu$, $LayerNorm$, $Softmax$, and Matrix multiplication. A secure inference protocol Π can be constructed utilizing Algorithm 3, 2, 1, 6, and homomorphic operations described in Section 2.3 in the $\mathcal{F}_{Gelu}, \mathcal{F}_{LayerNorm}, \mathcal{F}_{Softmax}, \mathcal{F}_{Matrixmul}$-hybrid model.*

Proof. Π can be constructed by replacing the function with their secure implementations. The correctness derives directly from the underlining algorithms. The privacy of Π simply follows in the hybrid model since only HE ciphertexts are exchanged. □