Secure Transformer Inference Made Non-interactive

Jiawen Zhang*, Xinpeng Yang*, Lipeng He*‡, Kejia Chen*, Wen-jie Lu*,
Yinghao Wang*, Xiaoyang Hou*, Jian Liu*† ✉, Kui Ren*†, Xiaohu Yang*†

*The State Key Laboratory of Blockchain and Data Security, Zhejiang University
†Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute
‡University of Waterloo
{kevinzh, yangxinpeng, chenkejia, asternight, xiaoyanghou, jian.liu, kuiren, yangxh}@zju.edu.cn
lipeng.he@uwaterloo.ca, fionser@gmail.com

***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. The protocol requires the client to engage in just one round of communication with the server during the whole inference process: submitting an encrypted input and receiving an encrypted result. NEXUS introduces several novel primitives, including SIMD ciphertext compression/decompression, SIMD slot folding, and secure Argmax, which enable it to significantly surpass the state-of-the-art in communication while maintaining comparable runtime. Specifically, it reduces bandwidth consumption by 372.5× compared to BOLT (Oakland '24) and 53.6× compared to Bumblebee (NDSS '25). Furthermore, its non-interactive property allows for optimal hardware acceleration, with the GPU version achieving a 42.3× speedup in runtime. This enables NEXUS to run inference on a BERT-based model in just 37.3 seconds, consuming only 164 MB of bandwidth.

I. Introduction

Transformers, such as GPT [48] and BERT [17], 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 the manipulation of sensitive data, leading to growing concerns about user privacy over the years. Recently, OpenAI has developed ChatGPT as an online inference service, along with a public API for developers to easily access the platform by submitting prompts or messages. While this approach is convenient, it poses significant risks to data privacy as the content submitted by the users may sometimes contain private information.

Secure inference is a two-party cryptographic protocol enabling model inference to proceed in a manner that the server $\mathcal{S}$ learns nothing about the input submitted by the clients $\mathcal{C}$s, and $\mathcal{C}$ learns nothing about $\mathcal{S}$'s model, except for the inference results. Many such protocols were developed for convolutional neural networks (CNNs) in the past few years [40], [28], [2], [31], and some recent works also started to support transformer-based models [25], [11], [27], [42], [45].

It is noteworthy that most of these protocols are *interactive*, warranting substantial communication costs and numerous interaction rounds between $\mathcal{C}$ and $\mathcal{S}$. For example, the state-of-the-art solution for secure transformer inference, BOLT [45], documents 59.61GB of bandwidth consumption and 10,509 interaction rounds for a single inference. Such a substantial communication overhead notably contributes to network latency, especially in WAN configurations, and renders conventional hardware acceleration techniques such as GPUs or FPGAs ineffective. In addition, the size of communication payload makes the financial cost of secure inference highly undesirable. According to AWS's pricing standards [1], the cost of each reply token in BOLT [45] is \$5.44, which means practical deployment is expensive and infeasible.

We emphasize the critical importance of establishing a *non-interactive* model for secure inference, where $\mathcal{C}$ only needs to submit one encrypted input in order to receive an encrypted prediction 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 criterion when leveraging hardware acceleration [16], [15], [54], [56], [51], [3], [33]. In non-real-time scenarios such as data warehousing and hospital diagnosis, where $\mathcal{C}$ can tolerate an extended latency in response, the deployment of non-interactive protocols are attainable, whereas interactive ones are not. This is because interactive protocols necessitate $\mathcal{C}$'s computing resources to remain engaged during the waiting period, impeding on $\mathcal{C}$'s ability to execute other tasks.

In the context of non-interactive secure inference, there are two key differences between transformers and CNN models.

1) **Larger scale matrix-matrix multiplications.** Prior works on private neural networks, such as Gazalle [31] and Cheetah [28], proposed optimized protocols for secure matrix-vector multiplications in the fully-connected layers. However, transformers demand large-scale matrix-matrix multiplications. Previous works [31], [28], [25] compute multiplications via inner dot products, and em-

✉ Jian Liu is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
23 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.23868
www.ndss-symposium.org

ploy sparse packing for the resulting ciphertexts in cases of matrix-matrix multiplication. As a result of this technique, the majority of data slots in the ciphertexts are often wasted, which introduces additional communication overhead. Furthermore, the input dimensions of matrices in transformer models are often much higher, leading to increased computational costs as a result of more multiplications being done. Consequently, it is useful to develop a new matrix multiplication protocol that is both time and space-efficient.

2) **Higher dimension inputs to Argmax.** The last layer of both CNN and transformer is Argmax, whose input is a probability vector with each entry being the probability of an output label candidate (m labels in total). The inference output is the label with the highest probability. Current state-of-the-art FHE algorithm for Argmax is presented by Phoenix (CCS'22) [30], which demonstrates a computational complexity of $O(m)$. This metric is often considered acceptable in CNN image classification tasks, since the number of labels are typically not very large, e.g. $m = 1,000$ in ImageNet-1k. However, for transformer-based NLP tasks, m equals to the size of the vocabulary, with m reaching $30,522$ in BERT and $128,256$ in Llama-3-8B. It is evident that the existing algorithm is not suitable for transformers, and a solution with lower complexity is needed.

A. *Our contributions*

In this paper, we propose NEXUS, which, to the best of our knowledge, is the first non-interactive protocol for secure transformer inference. The protocol design of NEXUS begins with the client encrypting its input using RNS-CKKS fully homomorphic encryption (FHE), enabling the server to evaluate the transformer model on FHE-encrypted data. We summarize our contribution as follows:

- **Efficient and communication-optimized matrix multiplications.** Many previous secure inference protocols such as Gazalle [31], Cheetah [28] and Iron [25] have wasted data slots in their output ciphertexts, which introduces unnecessary communication overhead. BumbleBee [42] eliminated wasted slots through ciphertext interleaving, but requires more computations to be done. We adopt the ciphertext compression and decompression strategy and take advantage of the special property of the monomial (cf. Section-III.B) in our our matrix multiplication algorithm to reduce the communication cost. We also propose an amortization-friendly offline-online computing strategy (cf. Section-III.C) to reduce the computation cost.
- **Efficient Argmax and other non-linear functions evaluation.** To defend against membership inference attacks [53], [58], [57], a common approach is to output the logits vector after Argmax, which leaks the least information about the model [53]. For an input of length m ($m = 30,522$ in BERT, $m = 128,256$ in Llama-3-8B), the state-of-the-art protocol [30] requires m times of SGN operations (cf. Section-II.C) and m times of ciphertext rotations. Our method only requires $(\log m+1)$ times of SGN operations and $(\log m + 1)$ times of ciphertext rotations, which brings significant reductions to computation overhead. Additionally, we also implement non-linear functions such as GELU, Softmax, and Layer Normalization using RNS-CKKS.
- **We provide an end-to-end implementation of NEXUS on both CPU and GPU.** Figure-1 illustrates the improvements made by the proposed protocols compared to the baseline. In summary, compared to the state-of-the-art protocol Bumblebee [42], NEXUS(CPU) is 1.79× faster in computation, can save 98.1% of the communication overhead, and able to reduces the financial cost by 2.38×. Leveraging the advantage of the non-interactive property of our protocol, we further provide a GPU-accelerated implementation. NEXUS(GPU) improves inference speed by 42.3× and achieves a financial cost reduction of 17.2×, to only $0.05/token. Our code has been open-sourced at https://github.com/zju-abclab/NEXUS.

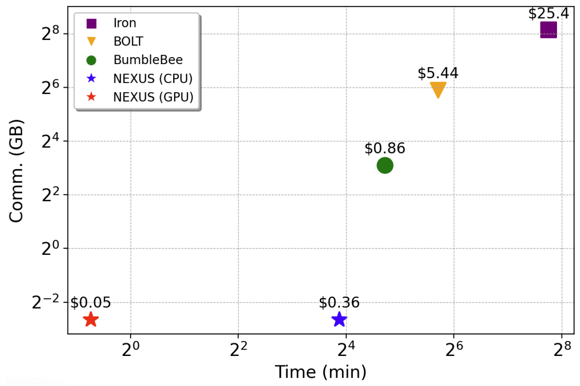


Figure 1: The overall performance improvements of the proposed optimizations on the BERT-base model (128 input tokens, WAN with 100Mbps bandwidth and 80ms latency). The price indicates the financial cost of outputting a word, according to current AWS pricing policy.

II. Preliminaries

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

A. *Secure inference and threat model*

Secure inference is a two-party cryptographic protocol that enables model inference between a client $\mathcal{C}$ and a server $\mathcal{S}$, while preserving the privacy of both parties' inputs. Similar to previous works [25], [45], [42], we assume $\mathcal{C}$ and $\mathcal{S}$ are semi-honest adversaries, adhering to the protocol specifications while endeavoring to gather extra information about the transformer model or the user's data. Additionally, we assume that an adversary is computationally bounded. Formal threat model and security proof are provided in the Appendix-D.

B. *Transformer*

Figure 2 shows the structure and workflow of a transformer. It takes an embedding, represented by 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.

Table I: A table of frequent notations.

Notation	Description
$\mathcal{C}$	client
$\mathcal{S}$	server
$\mathrm{E}()$	encryption
$\pi()$	encoding
$\textsc{Enc}()$	encoding-then encryption
$\tilde{\mathbf{a}}$	FHE ciphertext
$\boxplus$ / $\boxtimes$	homomorphic addition / multiplication
$\textsc{RotL}()$/$\textsc{RotR}()$	left rotation/right rotation
$\textsc{Subs}()$	substitution
$\textsc{Sgn}()$	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

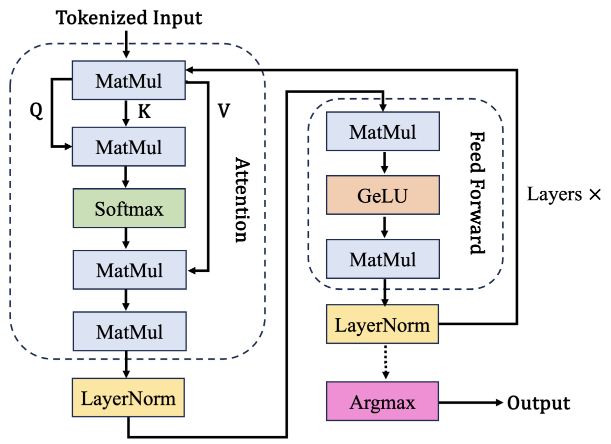


Figure 2: Structure of transformer.

Attention. The first step of the attention layer is to multiply the embedding $\mathbf{A} \in \mathbb{R}^{m\times n}$ with 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 $\mathbf{W}_V$. The 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 is calculated using the formula:

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

In the multi-head attention variant, an H-parallel attention $\textsc{Attention}(\mathbf{Q}_j,\mathbf{K}_j,\mathbf{V}_j)$ for $j \in [H]$ is computed, and the H resulting matrices are concatenated.

Layer normalization. The input to $\textsc{LayerNorm}$ 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:

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

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 placed in between:

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

The GELU function can be evaluated using [26]:

$$\mathrm{GELU}(x) = \frac{1}{2}x \cdot (1 + \mathrm{ERF}(\frac{x}{\sqrt{2}}))$$

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

C. Fully homomorphic encryption

Fully homomorphic encryption (FHE), which allows arbitrary operations to be performed over encrypted data [20], 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) [12], [13].

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 to enable 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 $\mathcal{R}_{q_0} \cong \mathcal{R}_{q_0} \times \mathcal{R}_{q_1} \times \cdots \times \mathcal{R}_{q_L}$ [10]. Suppose the bootstrapping consumes K levels, then a fresh ciphertext can support $L - K$ levels of computation.

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 processing the 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 $\mathrm{E}()$. Throughout this paper, we use $\mathrm{E}()$ to denote the encryption of a polynomial and use $\textsc{Enc}()$ to denote the SIMD encryption of a vector:

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

We summarize the homomorphic operations used in this paper below:

- $p(x) \leftarrow \pi(\mathbf{a})$. The encoding algorithm takes a vector $\mathbf{a} = [a_0, ..., a_{N-1}]$ and outputs a polynomial $p(x) \in \mathcal{R}_Q$.

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

D. Homomorphic sign function

As FHE only supports polynomial operations, it is non-trivial to compare FHE-encrypted values in a non-interactive manner. To enable encrypted comparisons, we leverage the polynomial approximation of the sign function [37], [19], [14]:

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

where $f()$, $g()$ are two polynomials and d_f, d_g are the number of repetitions. 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 $\text{Sgn}()$ function must be normalized beforehand:

$$\Delta = max\{|a_{max}|, |a_{min}|\}$$

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

- $\tilde{\mathbf{b}} \leftarrow \text{Sgn}(\tilde{\mathbf{a}})$: $b_i = f^{d_f}(g^{d_g}(a_i/\Delta)),\ \forall\ i \in [N]$.

In our implementation, both $f()$ and $g()$ are of degree 9; we set $\alpha = 20, d_f = 2, d_g = 2$ and evaluate the polynomials using the Baby-Step-Giant-Step algorithm [24]. In future work, sign function evaluation could benefit from the ongoing efforts in optimizing FHE approximations of non-polynomial functions, such as the latest extension of Lee et al. [35].

III. Efficient Matrix Multiplication

Transformer-based models consist of large matrix multiplications. In this section, we propose an efficient protocol for matrix-matrix multiplications.

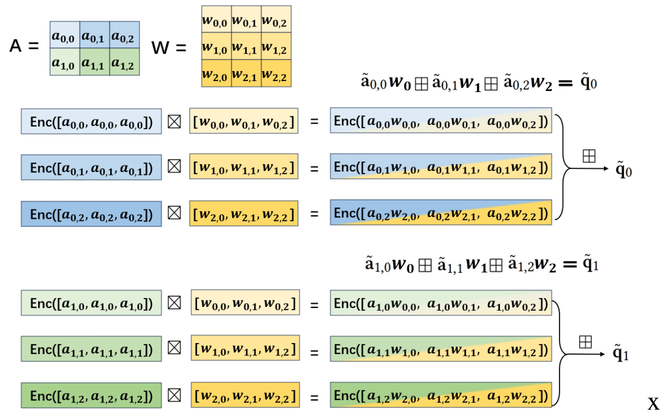
A. Overview

The MatrixMul operation takes input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ form $\mathcal{C}$ and weight matrix $\mathbf{W} \in \mathbb{R}^{n \times k}$ from $\mathcal{S}$, and then outputs $\mathbf{Q} := \mathbf{A} \cdot \mathbf{W} \in \mathbb{R}^{m \times k}$.

Similar to prior works, we use HE to compute them securely, as HE is relatively efficient for linear operations. Some of these works such as Gazalle [31], Cheetah [28] and Iron [25] compute matrix multiplication via inner dot products. In cases of matrix-matrix multiplications, these works employ sparse packing for the resulting ciphertexts. This often leads to the empty slots in the ciphertexts being wasted, introducing additional communication overhead. BumbleBee [42] eliminates these wasted slots through ciphertext interleaving at the cost of requiring a number of Subs operations (same as automorphism). We leverage this technique to develop a secure ciphertext compression algorithm for MatrixMul.

We introduce our optimization based on the observation that different $\mathbf{A} \in \mathbb{R}^{m \times n}$ matrices need to be multiplied with the same $\mathbf{W} \in \mathbb{R}^{n \times k}$ during transformer inference. For example, in GPT, the model autoregressively generates responses based on previous output words (with different $\mathbf{A}$s) [27]; and in BERT, batch inference is typically used to process multiple input samples (with different $\mathbf{A}$s) simultaneously [52]. Our goal is to reduce the amortized cost of MatrixMul by exploiting this fact. In Table-II, we compare the amortized costs of computing t matrix multiplications using NEXUS with the costs of using state-of-the-art MatrixMul protocols.

To present our solution, we start with a toy example with $\mathbf{A} \in \mathbb{R}^{2 \times 3}$ and $\mathbf{W}_Q \in \mathbb{R}^{3 \times 3}$ in Figure-3.



A =
W =
$\tilde{a}_{0,0}w_0 \boxplus \tilde{a}_{0,1}w_1 \boxplus \tilde{a}_{0,2}w_2 = \tilde{q}_0$
$\text{Enc}([a_{0,0}, a_{0,0}, a_{0,0}]) \boxtimes [w_{0,0}, w_{0,1}, w_{0,2}] = \text{Enc}([a_{0,0}w_{0,0}, a_{0,0}w_{0,1}, a_{0,0}w_{0,2}])$
$\text{Enc}([a_{0,1}, a_{0,1}, a_{0,1}]) \boxtimes [w_{1,0}, w_{1,1}, w_{1,2}] = \text{Enc}([a_{0,1}w_{1,0}, a_{0,1}w_{1,1}, a_{0,1}w_{1,2}])$
$\text{Enc}([a_{0,2}, a_{0,2}, a_{0,2}]) \boxtimes [w_{2,0}, w_{2,1}, w_{2,2}] = \text{Enc}([a_{0,2}w_{2,0}, a_{0,2}w_{2,1}, a_{0,2}w_{2,2}])$
$\boxplus$ $\tilde{q}_0$
$\tilde{a}_{1,0}w_0 \boxplus \tilde{a}_{1,1}w_1 \boxplus \tilde{a}_{1,2}w_2 = \tilde{q}_1$
$\text{Enc}([a_{1,0}, a_{1,0}, a_{1,0}]) \boxtimes [w_{0,0}, w_{0,1}, w_{0,2}] = \text{Enc}([a_{1,0}w_{0,0}, a_{1,0}w_{0,1}, a_{1,0}w_{0,2}])$
$\text{Enc}([a_{1,1}, a_{1,1}, a_{1,1}]) \boxtimes [w_{1,0}, w_{1,1}, w_{1,2}] = \text{Enc}([a_{1,1}w_{1,0}, a_{1,1}w_{1,1}, a_{1,1}w_{1,2}])$
$\text{Enc}([a_{1,2}, a_{1,2}, a_{1,2}]) \boxtimes [w_{2,0}, w_{2,1}, w_{2,2}] = \text{Enc}([a_{1,2}w_{2,0}, a_{1,2}w_{2,1}, a_{1,2}w_{2,2}])$
$\boxplus$ $\tilde{q}_1$
X

Figure 3: A toy example of SIMD-based matrix multiplication

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 sum of $(a_{i,j} \cdot \mathbf{w}_j)\ \forall\ j \in [n]$.

$$\mathbf{q}_i = \sum_{j=1}^{n} (a_{i,j} \cdot \mathbf{w}_j)\ \forall\ i \in [m].$$

We could have $\mathcal{C}$ homomorphically encrypt each $a_{i,j}$ and send the corresponding ciphertexts to $\mathcal{S}$, who can then homomorphically evaluate MatrixMul. However, the challenge of this trivial solution is that $\mathcal{C}$ needs to send $m \cdot n$ ciphertexts, each of which is a ciphertext encoded in SIMD. Specifically, we have $\mathcal{C}$ encrypt each $a_{i,j}$ as:

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

To reduce the communication cost, we aim to fully utilize all ciphertext slots. We propose a method enabling $\mathcal{C}$ to compress $m \times n$ ciphertext in the form above into $\frac{m \times n}{N'}$ ciphertexts, while ensuring $\mathcal{S}$ can correctly decompress them and perform the aforementioned computations (cf. Section-III.B). To reduce the

Table II: Amortized cost of t matrix multiplications ($\mathbb{R}^{m\times n}\cdot\mathbb{R}^{n\times k}$). N is the # of elements batched in a ciphertext. # Ciphertexts represents the number of ciphertexts that need to be transmitted. Note that the cost of SUBS is almost equal to the cost of a ciphertext rotation, its main step is key switching, so we use the number of key switchings to represent the computational cost of matrix multiplications. We include an example using real BERT-base and GPT-2 parameters: $m=256, n=768, k=64, N=4096, t=256$

Methods	# Ciphertexts		# Key switching	
Gazalle [31]	$\frac{mn}{k}$	3072	$2k\sqrt{m}$	2048
Cheetah [28]	$\frac{2m\sqrt{nk}}{\sqrt{N}}$	1774	-	0
Iron [25]	$\frac{2\sqrt{mnk}}{\sqrt{N}}$	111	-	0
Bumblebee [42]	$\frac{m(n+k)}{N}$	52	$\frac{mk\log(N)}{2\sqrt{N}}$	1536
BOLT [45]	$\frac{m(n+k)}{N}$	52	$\sqrt{\frac{m^2n^2k}{N^2}}$	384
Ours	$\frac{mk}{N}+\frac{nk}{Nt}$	5	$\frac{2nk-2}{t}$	384

computation cost, we propose an amortization-friendly offline-online computing strategy (cf. Section-III.C)

B. SIMD Ciphertexts Compression and Decompression

Secure Compression. Suppose $\mathcal{C}$ wants to send N' ciphertexts to $\mathcal{S}$ with each ciphertext encrypting N identical values in SIMD format: $\textsc{Enc}(\underbrace{[a_0,...,a_0]}_{N})$, ..., $\textsc{Enc}(\underbrace{[a_{N'-1},...,a_{N'-1}]}_{N})$. We have $\mathcal{C}$ 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 $\tilde{\mathbf{p}}_0 := \mathrm{E}(p(x))$ to $\mathcal{S}$. In this way, we reduce the communication cost from N' ciphertexts to a single ciphertext.

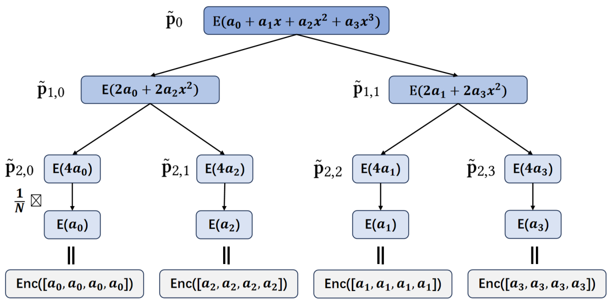
Secure Decompression. We extend SealPIR's ciphertext decompression algorithm [7] to matrix multiplication. In Algorithm-1, $\textsc{Subs}(\tilde{\mathbf{p}}_0, N'+1)$ returns:

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

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

1) $\tilde{\mathbf{p}}_{1,0} \leftarrow \tilde{\mathbf{p}}_0 \boxplus \textsc{Subs}(\tilde{\mathbf{p}}_0, \frac{N'}{2^0}+1)$,
   $\tilde{\mathbf{p}}_{1,1} \leftarrow \tilde{\mathbf{p}}'_0 \boxplus \textsc{Subs}(\tilde{\mathbf{p}}'_0, \frac{N'}{2^0}+1)$
2) $\tilde{\mathbf{p}}_{2,0} \leftarrow \tilde{\mathbf{p}}_{1,0} \boxplus \textsc{Subs}(\tilde{\mathbf{p}}_{1,0}, \frac{N'}{2^1}+1)$,
   $\tilde{\mathbf{p}}_{2,1} \leftarrow \tilde{\mathbf{p}}'_{1,0} \boxplus \textsc{Subs}(\tilde{\mathbf{p}}'_{1,0}, \frac{N'}{2^1}+1)$,
   $\tilde{\mathbf{p}}_{2,2} \leftarrow \tilde{\mathbf{p}}_{1,1} \boxplus \textsc{Subs}(\tilde{\mathbf{p}}_{1,1}, \frac{N'}{2^1}+1)$,
   $\tilde{\mathbf{p}}_{2,3} \leftarrow \tilde{\mathbf{p}}'_{1,1} \boxplus \textsc{Subs}(\tilde{\mathbf{p}}'_{1,1}, \frac{N'}{2^1}+1)$
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 with degree 3. Algorithm-1 describes the full decompression process. Clearly, it only requires $2N'$ substitutions in total. We prove its correctness in Appendix-A.



$\tilde{\mathbf{p}}_0$
$\mathrm{E}(a_0+a_1x+a_2x^2+a_3x^3)$
$\tilde{\mathbf{p}}_{1,0}$
$\mathrm{E}(2a_0+2a_2x^2)$
$\tilde{\mathbf{p}}_{1,1}$
$\mathrm{E}(2a_1+2a_3x^2)$
$\tilde{\mathbf{p}}_{2,0}$
$\mathrm{E}(4a_0)$
$\tilde{\mathbf{p}}_{2,1}$
$\mathrm{E}(4a_2)$
$\tilde{\mathbf{p}}_{2,2}$
$\mathrm{E}(4a_1)$
$\tilde{\mathbf{p}}_{2,3}$
$\mathrm{E}(4a_3)$
$\frac{1}{N}\boxtimes$
$\mathrm{E}(a_0)$
$\mathrm{E}(a_2)$
$\mathrm{E}(a_1)$
$\mathrm{E}(a_3)$
$\mathrm{Enc}([a_0,a_0,a_0,a_0])$
$\mathrm{Enc}([a_2,a_2,a_2,a_2])$
$\mathrm{Enc}([a_1,a_1,a_1,a_1])$
$\mathrm{Enc}([a_3,a_3,a_3,a_3])$

Figure 4: A toy example showcasing the decompression process.

Algorithm 1 Secure Decompression on RNS-CKKS

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

```
1:  function DECOMPRESS($\tilde{\mathbf{p}}_0$)
2:      $\tilde{\mathbf{p}}_{0,0} := \tilde{\mathbf{p}}_0$
3:      for $i = 0$ to $\log N'$ do
4:          for $j = 0$ to $2^i - 1$ do
5:              $\tilde{\mathbf{p}}'_{i,j} \leftarrow \tilde{\mathbf{p}}_{i,j} \boxtimes x^{-2^i}$
6:              $\tilde{\mathbf{p}}_{i+1,2^j-1} \leftarrow \tilde{\mathbf{p}}_{i,j} \boxplus \textsc{Subs}(c, \frac{N'}{2^i}+1)$
7:              $\tilde{\mathbf{p}}_{i+1,2^j} \leftarrow \tilde{\mathbf{p}}'_{i,j} \boxplus \textsc{Subs}(c', \frac{N'}{2^i}+1)$
8:          end for
9:      end for
10:     for $j = 0$ to $N' - 1$ do
11:         $\tilde{\mathbf{a}}_j \leftarrow \tilde{\mathbf{p}}_{\log N',j} \boxtimes \frac{1}{N'}$
12:     end for
13:     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-1 is exactly an SIMD ciphertext encrypting a vector of N a_is.

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

Proof: Given that

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

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

we only need to prove

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

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

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

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:

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

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

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

which is equal to $\mathbf{V}$ multiplied by the left side of the equation. ■

C. *Offline-Online Batch Matrix Multiplication*

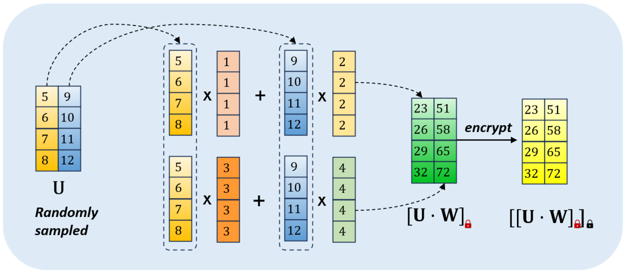
Let $\mathbf{A} = [\mathbf{a}_0, ..., \mathbf{a}_{n-1}]$ with $\mathbf{a}_i \in \mathbb{R}^m$ being each column of $\mathbf{A}$. Suppose $\mathcal{S}$ and $\mathcal{C}$ need to generate t response words, then there are t input matrices:

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

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

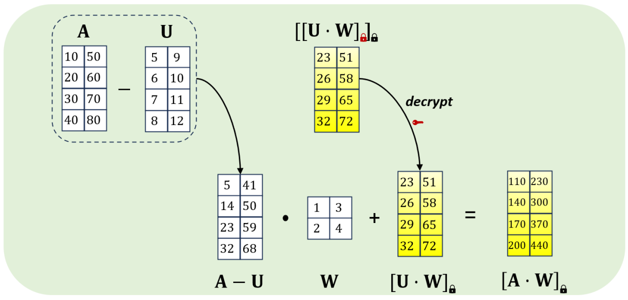
$$\mathbf{Q}' = \mathbf{q}'_0 || \mathbf{q}'_1 || ... || \mathbf{q}'_{k-1} = \begin{bmatrix} \mathbf{A}_0 \mathbf{W} \\ \vdots \\ \mathbf{A}_{t-1} \mathbf{W} \end{bmatrix}$$

To this end, we introduce a preprocessing phase, where $\mathcal{S}$ sends $\mathcal{C}$ the *compressed* ($\mathrm{ENC}_{\mathcal{S}}([\underbrace{w_{i,j}, ..., w_{i,j}}_{t\times m}]) \ \forall\ i \in [n], j \in [k]$),[2] using our compression technique described in Section-III.B. Notice that this transfer occurs only once, unless the model changes. Next, $\mathcal{C}$ performs decompression to obtain $\mathrm{ENC}_{\mathcal{S}}([\underbrace{w_{i,j}, ..., w_{i,j}}_{t\times m}]) \ \forall\ i \in [n], j \in [k]$. If $t \times m > N$,



U
Randomly sampled
x
+
encrypt
$[\mathbf{U}\cdot\mathbf{W}]$
$[[\mathbf{U}\cdot\mathbf{W}]]$

(a) Offline computation at the client



A
U
$[[\mathbf{U}\cdot\mathbf{W}]]$
decrypt
$\mathbf{A}-\mathbf{U}$
W
$[\mathbf{U}\cdot\mathbf{W}]$
$[\mathbf{A}\cdot\mathbf{W}]$

(b) Online computation at the server

Figure 5: Protocol for offline-online matrix multiplication

each $[\underbrace{w_{i,j}, ..., w_{i,j}}_{t\times m}]$ occupies multiple ciphertexts. As $\mathcal{C}$ has no knowledge of the inputs (i.e., $\mathbf{A}$s) in the preprocessing phase, it samples $\mathbf{U} \in_\$ \mathbb{R}^{(tm)\times n}$, and computes:

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

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

In the online phase, after knowing $\mathbf{A}' = \mathbf{a}'_0 || \mathbf{a}'_1 || ... || \mathbf{a}'_{n-1}$, $\mathcal{C}$ sends $(\mathbf{A}' - \mathbf{U})$ to $\mathcal{S}$. $(\mathbf{A}' - \mathbf{U})$ 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:

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

where $\mathbf{q}'_j$ is the j-th column of $\mathbf{Q}'$. Figure-5 give an example of our protocol. The security proof of our matrix multiplication protocol can be found in Appendix-D.

We remark that this optimized MatrixMul protocol does not compromise the non-interactive property of NEXUS: $\mathcal{C}$ only needs to send $(\mathbf{A}' - \mathbf{U})$ to $\mathcal{S}$ and receive the inference result in the online phase.

After the ciphertext-plaintext matrix multiplication, the subsequent matrix multiplication requires row-wise encryption for both $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ in attention layer). It is noteworthy that the multiplication of $\mathbf{Q}$ and $\mathbf{K}^T$ remains feasible even when

[2] We use $\mathrm{ENC}_{\mathcal{S}}$ to denote an encryption under $\mathcal{S}$'s public key. Similarly, we use $\mathrm{ENC}_{\mathcal{C}}$ to denote an encryption under $\mathcal{C}$'s public key.

subjected to column-wise encryption. We show the detailed ciphertext-ciphertext matrix-matrix multiplication process in Appendix-C.

IV. Efficient Non-linear Function Evaluations

A. *Secure Argmax Evaluation*

The number of classes in a text generation task is equal to the number of unique words/tokens. For example, in BERT-base model, the vocabulary size is 30522, hence many of the classes have very small probabilities in the model's prediction vector. Several previous works [53], [58], [57] have designed membership inference attacks based on the class probability distribution of the prediction vector that cause information leakage about the model.

To defend against membership inference attacks, a common approach is to output the probabilities of the most likely k classes [53]. The smaller the k is, the less information the model leaks. In the most extreme case, the model returns only the label of the most likely class and without reporting its probability. ARGMAX is widely used in this scenario.

Suppose the encrypted logits is $\tilde{\mathbf{a}} = \textsc{Enc}([a_0, ..., a_{m-1}])$, the final output of the transformer should be a selection vector $\tilde{\mathbf{b}} = \textsc{Enc}([b_0, ..., b_{m-1}])$, 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 [30]. Phoenix adopts the idea of bubble sorting to compare each element with its adjacent elements by rotating the ciphertext and calculating the difference:

$$\begin{aligned}
\tilde{s_1} &\leftarrow \textsc{Sgn}(\tilde{\mathbf{a}} - \textsc{RotL}(\tilde{\mathbf{a}}, 1)) \\
\tilde{s_2} &\leftarrow \textsc{Sgn}(\tilde{\mathbf{a}} - \textsc{RotL}(\tilde{\mathbf{a}}, 2)) \\
&\dots \\
\tilde{s_m} &\leftarrow \textsc{Sgn}(\tilde{\mathbf{a}} - \textsc{RotL}(\tilde{\mathbf{a}}, m))
\end{aligned}$$

Then a summation $\tilde{\mathrm{s}} \leftarrow \boxplus_{i=1}^{m} \tilde{\mathrm{s}}_i$ is performed over all the comparison results. It follows that the entry of $\tilde{\mathrm{s}}$ that corresponds to the position of the maximum element will have value m, and the values of other entries will be less than m. After that, through simple linear transformations, $\tilde{\mathbf{b}}$ can be obtained from $\tilde{\mathrm{s}}$ (cf. Phoenix [30] for details).

However, this method requires m number of SGN evaluations and m rotations, which makes it very inefficient when m is large (e.g. $m = 30,522$ in BERT, $m = 128,256$ in Llama-3-8B). To solve the problem, we *innovatively* propose to approximate each b_i as:

$$b_i = \textsc{Sgn}(a_i - a_{max}) + 1. \tag{2}$$

We describe the algorithm for computing ARGMAX in Algorithm-2 with the use of QUICKMAX. Before going into details, we first need to introduce a general design called SIMD slots folding. With this technique, we will only require $\log m$ SGNs and $\log m$ rotations to get the maximum value in $\tilde{\mathbf{a}}$.

SIMD slots folding Our goal is to compute function $f()$ over all the SIMD slots of the input $\tilde{\mathbf{a}} = \textsc{Enc}([a_0, ..., a_{N-1}])$, and replace each individual slot with a value in the result $\tilde{\mathrm{s}} = \textsc{Enc}([\underbrace{s, ..., s}_{N}])$. For example, if $f()$ is a max function, then for the input $\textsc{Enc}([2, -1, 3, 1])$, the output is $\textsc{Enc}([3, 3, 3, 3])$.

Algorithm 2 Secure ARGMAX on RNS-CKKS

```
Input: $\tilde{\mathbf{a}} = \textsc{Enc}([a_0, ..., a_{m-1}, 0, ..., 0])$ with $2m < N$
Output: $\textsc{Enc}([b_0, ..., b_{m-1}, 0, ..., 0])$ (cf. Equation 2)
1: function ARGMAX($c$)
2:     $\tilde{\mathbf{a}}_{max} \leftarrow$ QUICKMAX($\tilde{\mathbf{a}}$)
3:     $\tilde{\mathbf{a}} \leftarrow \tilde{\mathbf{a}} \boxminus \tilde{\mathbf{a}}_{max}$
4:     $\tilde{\mathbf{b}} \leftarrow$ SGN($\tilde{\mathbf{a}}$) // $b = 0$ or $-1$
5:     $\tilde{\mathbf{b}} \leftarrow \tilde{\mathbf{b}} \boxplus \mathbf{1}$
6:     return $\tilde{\mathbf{b}}$
7: end function
```

In general, this solution is applicable to all functions that supports *associativity*:

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

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

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

We can aggregate them by employing a binary tree construction:

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

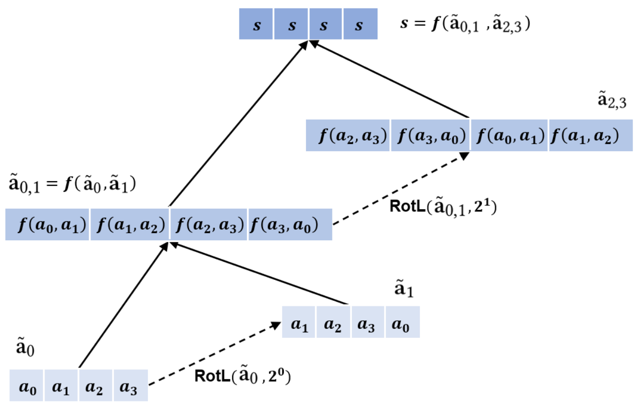
with:

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

This trivial solution requires $N - 1$ rotations.

A 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 a manner akin to a "*binary tree*" (cf. Figure-6). 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.

The solution above is useful only when the length n of the input vector is equal to N. However, in the transformers we evaluate, $n \ll N$. In this case, we transform $\textsc{Enc}([a_0, ..., a_{n-1}, \underbrace{0, ..., 0}_{N-n}])$ into $\textsc{Enc}([a_0, ..., a_{n-1}, a_0, ..., a_{n-1}, \underbrace{0, ..., 0}_{N-2n}])$ before proceeding with the aforementioned process.



s s s s
$s = f(\tilde{a}_{0,1}, \tilde{a}_{2,3})$
$\tilde{a}_{2,3}$
$f(a_2,a_3)$ $f(a_3,a_0)$ $f(a_0,a_1)$ $f(a_1,a_2)$
$\tilde{a}_{0,1} = f(\tilde{a}_0, \tilde{a}_1)$
$f(a_0,a_1)$ $f(a_1,a_2)$ $f(a_2,a_3)$ $f(a_3,a_0)$
RotL$(\tilde{a}_{0,1}, 2^1)$
$\tilde{a}_1$
a_1 a_2 a_3 a_0
$\tilde{a}_0$
a_0 a_1 a_2 a_3
RotL$(\tilde{a}_0, 2^0)$

Figure 6: A toy example for computing $f()$ on SIMD slots ($n = N$).

Algorithm 3 SIMD slots folding on RNS-CKKS

Input: $\tilde{\mathbf{a}} = \text{ENC}([a_0, ..., a_{n-1}, 0, ..., 0])$ with $2n < N$
Output: $\text{ENC}([\underbrace{s, ..., s}_{n}, 0, ..., 0])$, where $s = f(a_0, ..., a_{n-1})$

```
1: function FOLD($\tilde{\mathbf{a}}$)
2:     $\tilde{\mathbf{a}} \leftarrow \tilde{\mathbf{a}} \boxplus \text{ROTR}(\tilde{\mathbf{a}}, n)$
3:     for $i = 0$ to $\log n - 1$ do
4:         $\tilde{\mathbf{t}} \leftarrow \text{ROTL}(\tilde{\mathbf{a}}, 2^i)$ // left-rotate by $2^i$ steps
5:         $\tilde{\mathbf{t}} \leftarrow f(\tilde{\mathbf{t}}, c)$
6:         $\tilde{\mathbf{a}} := \tilde{\mathbf{t}}$
7:     end for
8:     return $\tilde{\mathbf{t}} \boxtimes [\underbrace{1, ..., 1}_{n}, 0, ..., 0]$
9: end function
```

Algorithm-3 gives the detail of SIMD slots folding, as mentioned earlier, $f()$ can be any function that satisfies the associative law, such as sum, max, etc. Based on this, we define QUICKSUM and QUICKSUM.

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-3 by replacing Line 5 with:

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

QuickMax. Given $[a_0, ..., a_{n-1}, 0, ..., 0]$, $\mathcal{S}$ can obtain $\text{ENC}([\underbrace{a_{max}, ..., a_{max}}_{n}, 0, ..., 0])$ through Algorithm-3 by replacing $f()$ with $max()$. We leverage

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

to compute the max function on encrypted values. Then, Line 5 in Algorithm-3 is replaced with:

$$\tilde{\mathbf{t}} \leftarrow 0.5 \boxtimes (\tilde{\mathbf{a}} \boxplus \tilde{\mathbf{t}} \boxplus (\tilde{\mathbf{a}} \boxminus \tilde{\mathbf{t}}) \boxtimes \text{SGN}(\tilde{\mathbf{a}} \boxminus \tilde{\mathbf{t}})).$$

B. *Other Non-linear Functions*

GELU. Referring to BumbleBee [42] and PUMA [18], we adopt the following piecewise polynomial to approximate $\text{GELU}(x)$, which gives an average error within 10^{-4} when $x \in [-8, 8]$[3]:

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

First, we use the SGN operation to obtain 4 encrypted bits: b_0, b_1, b_2, b_3, such that:

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

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

Note that we can accurately estimate $\tilde{\mathbf{a}}$ even when the input is near the dividing points. For example, for an input $|\tilde{\mathbf{a}} - (-1.95)| < 2^{-\alpha}$, we have $\tilde{\mathbf{t}}_0 = 0.5$, $\tilde{\mathbf{t}}_2 = -0.5$, $\tilde{\mathbf{b}}_0 = 0$, $\tilde{\mathbf{b}}_3 = 0$, and $P(x) \approx Q(x) \approx -0.05$, where -0.05 is the ground truth of $\text{GELU}(-1.95)$, hence:

$$\begin{aligned} \tilde{\mathbf{y}} &= (\tilde{\mathbf{b}}_1 \boxtimes P(\tilde{\mathbf{a}})) \boxplus (\tilde{\mathbf{b}}_2 \boxtimes Q(\tilde{\mathbf{a}})) \\ &\approx -0.05 \boxtimes (\tilde{\mathbf{b}}_1 \boxplus \tilde{\mathbf{b}}_2) \\ &= -0.05 \boxtimes (\tilde{\mathbf{t}}_0 \boxminus \tilde{\mathbf{t}}_1 \boxplus \tilde{\mathbf{t}}_1 \boxminus \tilde{\mathbf{t}}_2) \\ &= -0.05 \boxtimes 1 = -0.05 \end{aligned}$$

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

$$y_i = \frac{\text{EXP}(a_i - a_{max})}{\sum_{j=0}^{m-1} \text{EXP}(a_j - a_{max})} \quad (4)$$

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 [34].

Although we can use QUICKMAX to find a_{max}, considering that softmax in BERT-base is executed multiple times and the value of a_{max} does not affect the result of softmax, in order to improve efficiency, we take a_{max} as a constant. Following BumbleBee [42], we approximate the exponentiation using the Taylor series:

$$\text{EXP}(x) \approx (1 + \frac{x}{2^r})^{2^r}, \quad x \leq 0$$

with $r = 8$, which limits the average error to be within 10^{-5}. Then, $\mathcal{S}$ could compute the exponentiation in SIMD format and obtain $\text{ENC}([e_0, ..., e_{m-1}])$, where $e_j = \text{EXP}(a'_j)$. Next, $\mathcal{S}$ applies QUICKSUM to obtain $\text{ENC}([\underbrace{\sum_{j=0}^{m-1} e_j, ..., \sum_{j=0}^{m-1} e_j}_{m}])$. In the end, $\mathcal{S}$ computes the final result in SIMD format using the Goldschmidt division algorithm [22], [46]. Algorithm-4 describes the details of our secure SOFTMAX algorithm.

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

LayerNorm For ease of computation, we perform the following to compute 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

$$y_i = \gamma\sqrt{n} \cdot \frac{z_i}{\sqrt{\sum_{i=0}^{n-1} z_i^2}} + \beta. \tag{5}$$

We apply QUICKSUM 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 [47], which employs Newton's iteration with a proper initial value. Algorithm-4 describes the details of our secure LAYERNORM.

Algorithm 4 Secure Non-linear functions on RNS-CKKS

```
Input: $\tilde{\mathbf{a}} = \text{ENC}([a_0, ..., a_{n-1}, 0, ..., 0])$ with $2n < N$
Output: $\text{ENC}([y_0, ..., y_{n-1}, 0, ..., 0])$ (cf. Equation 3,4,5)
 1: function GELU($\tilde{\mathbf{a}}$)
 2:     Compare $a$ with the breakpoints:
            $\tilde{\mathbf{t}}_0 \leftarrow 0.5 \boxtimes \text{SGN}(\tilde{\mathbf{a}} \boxplus 4)$       // $s_0 = \mathbf{0.5}\{a > -4\}$
            $\tilde{\mathbf{t}}_1 \leftarrow 0.5 \boxtimes \text{SGN}(\tilde{\mathbf{a}} \boxplus 1.95)$ // $s_1 = \mathbf{0.5}\{a > -1.95\}$
            $\tilde{\mathbf{t}}_2 \leftarrow 0.5 \boxtimes \text{SGN}(\tilde{\mathbf{a}} \boxminus 3)$       // $s_2 = \mathbf{0.5}\{a > 3\}$
 3:     Compute segment selection:
            $\tilde{\mathbf{b}}_0 \leftarrow 0.5 \boxminus \tilde{\mathbf{t}}_0$   // $b_0 = \mathbf{1}\{x < -4\}$
            $\tilde{\mathbf{b}}_1 \leftarrow \tilde{\mathbf{t}}_0 \boxminus \tilde{\mathbf{t}}_1$   // $b_1 = \mathbf{1}\{-4 < x < -1.95\}$
            $\tilde{\mathbf{b}}_2 \leftarrow \tilde{\mathbf{t}}_1 \boxminus \tilde{\mathbf{t}}_2$   // $b_2 = \mathbf{1}\{-1.95 < x < 3\}$
            $\tilde{\mathbf{b}}_3 \leftarrow 0.5 \boxplus \tilde{\mathbf{t}}_2$   // $b_3 = \mathbf{1}\{x > 3\}$
 4:     Compute GELU:
            $\tilde{\mathbf{y}} \leftarrow (\tilde{\mathbf{b}}_0 \boxtimes 0) \boxplus (\tilde{\mathbf{b}}_1 \boxtimes P(\tilde{\mathbf{a}})) \boxplus (\tilde{\mathbf{b}}_2 \boxtimes Q(\tilde{\mathbf{a}})) \boxplus (\tilde{\mathbf{b}}_3 \boxtimes \tilde{\mathbf{a}})$
 5:     return $\tilde{\mathbf{y}}$
 6: end function
 7: function SOFTMAX($\tilde{\mathbf{a}}$)
 8:     $\tilde{\mathbf{a}} \leftarrow 1 \boxplus \tilde{\mathbf{a}} \boxtimes \frac{1}{2^r}$
 9:     for $i = 0$ to $r$ do
10:         $\tilde{\mathbf{a}} \leftarrow \text{SQUARE}(\mathbf{a})$
11:     end for
12:     $\tilde{\mathbf{t}} \leftarrow \text{QUICKSUM}(\tilde{\mathbf{e}})$
13:     return $\tilde{\mathbf{e}} \boxtimes \text{INVERSE}(\tilde{\mathbf{t}})$
14: end function
15: function LAYERNORM($\tilde{\mathbf{a}}$)
16:     $\tilde{\mathbf{t}} \leftarrow \text{QUICKSUM}(\tilde{\mathbf{a}})$
17:     $\tilde{\mathbf{z}} \leftarrow (n \boxtimes \tilde{\mathbf{a}}) \boxminus \tilde{\mathbf{t}}$   // $z_i = na_i - \sum_{i=0}^{n-1} a_i$
18:     $\tilde{\mathbf{y}} \leftarrow \text{SQUARE}(\tilde{\mathbf{z}})$
19:     $\tilde{\mathbf{y}} \leftarrow \text{QUICKSUM}(\tilde{\mathbf{y}})$
20:     $\tilde{\mathbf{y}} \leftarrow \text{INVERTSQRT}(\tilde{\mathbf{y}})$
21:     $\tilde{\mathbf{y}} \leftarrow \tilde{\mathbf{z}} \boxtimes \tilde{\mathbf{y}}$   // $y_i = z_i / \sqrt{\sum_{i=1}^{n} z_i^2}$
22:     return $(\tilde{\mathbf{y}} \boxtimes \gamma \boxtimes \sqrt{n}) \boxplus \beta$
23: end function
```

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 ciphertext. For example, in LAYERNORM of BERT-base, $n = 128$, if $N = 32768$, we can batch process 128 inputs using one ciphertext.

V. PLACEMENT OF BOOTSTRAPPING

NEXUS is based on RNS-CKKS, which is a leveled homomorphic encryption scheme that allows at most L multiplications on a ciphertext in 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 and the cost scales linearly relative to the number of ciphertext inputs, its placement is crucial for the overall performance. We observe that the size of GELU input/output matrices is $\mathbb{R}^{128\times3072}$ (packed in 12 ciphertexts), but it is then reduced to $\mathbb{R}^{128\times768}$ (packed in 3 ciphertexts) by the subsequent MATRIXMUL. As a result, bootstrapping will execute much faster if performed after the MATRIXMUL. It is clear that we should circumvent the execution of bootstrapping during operations involving large input/output sizes, such as GELU, by judiciously selecting the multiplicative depth.

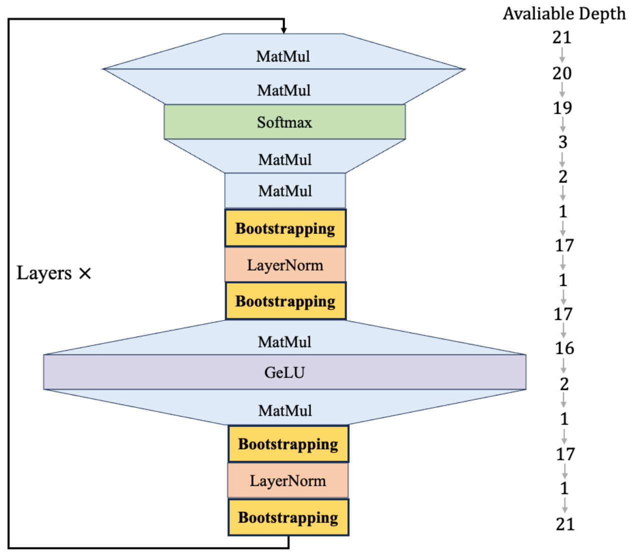


Figure 7: Placement of bootstrapping for a BERT-base transformer.

Figure-7 shows the placement of bootstrapping for a BERT-base transformer employing NEXUS. The width of the building blocks represents the dimension of the input/output. Matrix multiplications will bring about a change in dimensions. We try to perform bootstrapping when the dimension is the smallest (that is, when the number of packed ciphertexts is the least).

VI. EVALUATION

A. Implementation

We implement NEXUS in C++, utilizing the SEAL library[4] for CKKS homomorphic encryption and FHE-MP-CNN[5] for bootstrapping. We use HEXL [9] to accelerate SEAL on Intel

[4]https://github.com/microsoft/SEAL
[5]https://github.com/snu-ccl/FHE-MP-CNN

CPUs and Phantom[6] for GPU implementation. Following the "Homomorphic Encryption Standard" [6], we set the polynomial degree to $N' = 2^{16}$ (hence $N = 2^{15}$) and the ciphertext modulus as 1763-bit to achieve 128-bit security. We set the multiplicative depth to $L = 35$ and the depth for bootstrapping to $K = 14$, which indicates that the multiplicative depth available for normal computations 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 [10] to eliminate the dominant noise components.

B. Experimental setup

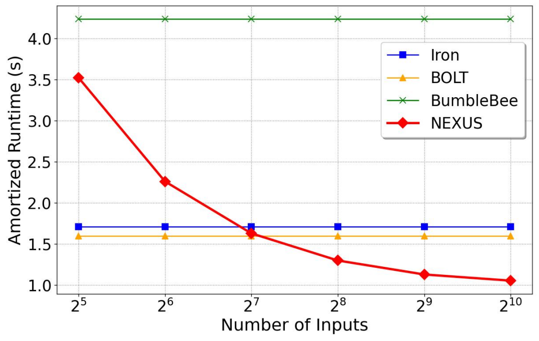
We primarily compare our work with Iron [25], BOLT [45], and Bumblebee [42]. As of current, Bumblebee has been open sourced in the SPU library [43], but Iron and BOLT do not yet have open source implementations. To enable a direct comparison with the results (of both Iron and BOLT) reported in the BOLT paper [45], we conduct our benchmarks under the same experimental settings as BOLT's:

- For the CPU benchmark, we use two instances with 3.70GHz Intel Xeon processors and 128 GB of RAM. We set the the number of threads to 32, same as BOLT. For the GPU benchmark, we use four Tesla A100 GPUs with 40 GB of memory. All results are averaged over 10 runs.
- 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 consists of four settings: {100Mbps, 40ms}, {100Mbps, 80ms}, {200Mbps, 40ms}, and {200Mbps, 80ms}, same as BOLT's.
- We do not apply any machine learning optimizations, such as word elimination or fine-tuning. The model parameters were taken from a pre-trained BERT-base transformer [17].

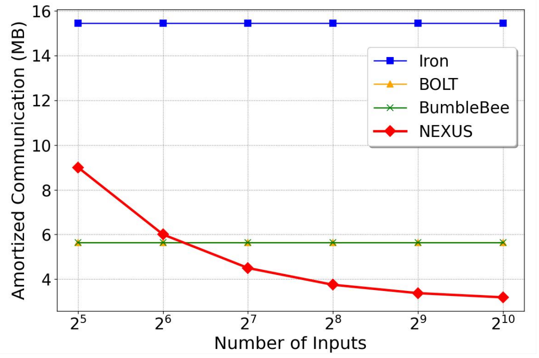
In terms of price, we use the current AWS financial cost structure for running a server with "c6i.16xlarge" specifications [1]. Therefore, the CPU per-hour cost is estimated to be $2.72/2 = $1.36 (since this machine has 64 vCPUs, and we run on 32 threads), the GPU per-hour cost is $1.29, and the download cost is $0.09 per GB.

C. Microbenchmarks

Matrix Multiplication. Figure-8 shows the amortized cost of MATRIXMUL in LAN over multiple inputs. In order to better compare performance metrics, we conduct this microbenchmark using a single thread. For a fair comparison, we did not distinguish the overhead from the offline and online phases, which means that we take into account all the preprocessing and online overhead. Considering that transformer often generates several hundred words in a single response, $t = 256$ would be a reasonable number for inputs. The amortized runtime (for 256 inputs) of NEXUS is 1.31s, $3.3\times$ faster than BumbleBee, $1.3\times$ faster than Iron and $1.2\times$ faster than BOLT. When the number of inputs increases to 1,024, which is also a commonly seen number, NEXUS demonstrates even greater performance advantages. Specifically, it outperforms BOLT by $1.6\times$ in running time and $1.6\times$ in communication costs.



(a) Amortized Runtime vs. #inputs.



(b) Amortized Communication vs. #inputs.

Figure 8: Evaluation of runtime and communication for $\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times768}$ ciphertext-plaintext matrix multiplication with single thread in LAN (amortized cost of multiple inputs).

Table III: Evaluation of non-linear functions. There are $12 \times \mathbb{R}^{128\times3072}$ inputs to GELU, $144 \times \mathbb{R}^{128\times128}$ to SOFTMAX, $24 \times \mathbb{R}^{128\times768}$ to LAYERNORM, and $\mathbb{R}^{30522}$ to ARGMAX. Unit of communication cost is GB, and the WAN setting is 100Mbps bandwidth and 80ms latency.

Setting	Protocol	Comm	LAN(s)	WAN(s)	Error	Price($)
GELU	Iron	93.3	126	4118	5.8e-4	8.453
	BOLT	17.2	14	774	9.8e-4	1.554
	BumbleBee	3.3	24	338	1.1e-3	0.308
	NEXUS	0	44	44	7.7e-4	0.020
	NEXUS*	0	2.1	2.1	7.7e-4	0.003
SOFTMAX	Iron	42.1	60	1900	3.2e-5	3.816
	BOLT	16.9	16	775	1.4e-6	1.528
	BumbleBee	1.7	23	241	7.2e-6	0.170
	NEXUS	0	47	47	3.1e-5	0.019
	NEXUS*	0	1.2	1.2	3.1e-5	0.002
LAYERNORM	Iron	20.4	16	1158	1.7e-3	1.843
	BOLT	14.0	14	914	-	1.266
	NEXUS	0	32	32	4.5e-4	0.013
	NEXUS*	0	2.0	2.0	4.5e-4	0.003
ARGMAX	Phoenix	0	3004	3004	1.9e-2	1.252
	NEXUS	0	54	54	7.6e-4	0.023
	NEXUS*	0	2.5	2.5	7.6e-4	0.004

* GPU accelerated

Non-linear Functions. Table-III shows a comparison of several metrics between NEXUS and other works for

[6]https://github.com/encryptorion-lab/phantom-fhe

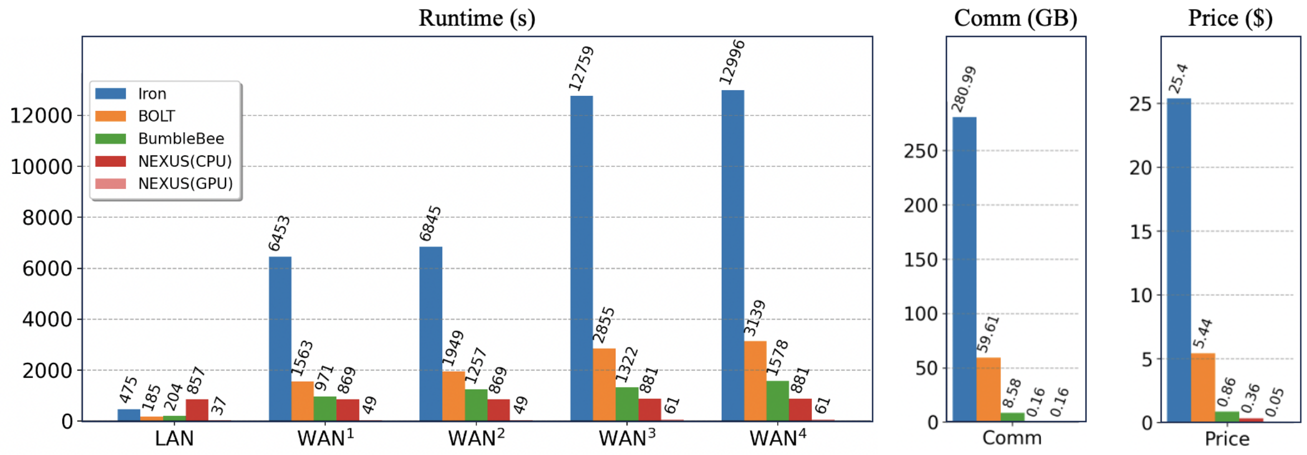


Figure 9: End-to-end comparisons with the existing secure inference frameworks using BERT-base. The input to the model consists of 128 tokens. For the CPU implementation, we use 32 threads for the benchmarks across the board.

evaluating non-linear functions (GELU, LAYERNORM, SOFTMAX, ARGMAX). Iron and BOLT implement these non-linear functions through secure two-party computation, which is expensive in terms of 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 performs:

- for GELU, 93.6× faster and 422.7× cheaper than Iron, 17.6× faster and 77.7× cheaper than BOLT, 7.9× faster and 15.4× cheaper than BumbleBee;
- for SOFTMAX, 40.4× faster and 200.8× cheaper than Iron, 16.5× faster and 80.4× cheaper than BOLT, 5.1× faster and 8.9× cheaper than BumbleBee;
- for LAYERNORM, 36.2× faster and 141.8× cheaper than Iron, 28.6× faster and 97.4× cheaper than BOLT.

The last column of Table-III shows the average error of the three schemes. For Iron and BOLT, we calculate their average errors by multiplying the ULP errors [50] 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 first uniformly sample $[x_1, ..., x_{1000}]$ from the corresponding domain. Then for each x_i, we calculate both the real $y_i = \text{GELU}(x)$ and the approximated $y_i' = \text{GELU}(x)$. 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 works.

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 55.6× in terms of speed and price.

Figure-10 shows the performance of ARGMAX with respect to inputs of different dimensions (vocabulary sizes). In newer models such as Llama-3-8B, the vocabulary size could reach 128,256, in which case NEXUS can achieve up to 136.5× speedup when compared to previous works. The advantage of having logarithmic complexity is highlighted.

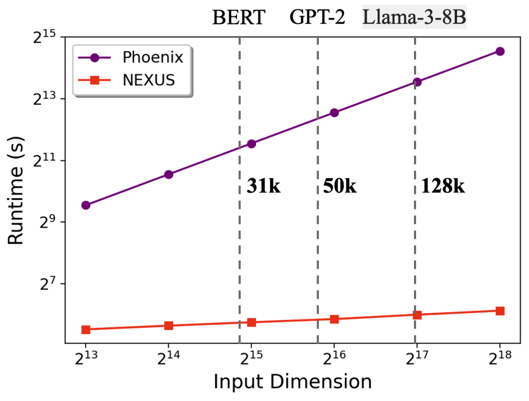


Figure 10: Performance of ARGMAX using RNS-CKKS over inputs of different dimensions. We mark the vocabulary size of BERT, GPT-2, and GPT-3 in the chart.

D. End-to-End Benchmark

End-to-End Performance. The end-to-end performance is roughly the aggregation of the microbenchmarks. Note that for the scaling of floating-point numbers to integers, Iron and BOLT need to perform secure truncations to prevent overflows. In contrast, NEXUS avoids the need for truncations by leveraging RNS-CKKS, which supports floating-point numbers but requires bootstrappings. The end-to-end workflow of NEXUS follows Figure-7.

Figure-9 shows the end-to-end performance (amortized for 128 inputs). Notably, NEXUS only consumes 164MB of bandwidth, which is a 1737.5× reduction over Iron, a 368.6× reduction over BOLT and a 53.7× reduction over BumbleBee. In terms of end-to-end runtime, NEXUS still achieves up to 14.8× speedup over Iron, 3.6× speedup over BOLT and 1.8× speedup over BumbleBee. In terms of price, NEXUS is 70.6× cheaper than Iron, 15.1× cheaper than BOLT and 2.4× cheaper than Bumblebee. As for NEXUS (GPU), it takes only 37 seconds and $0.05 to produce the output, which demonstrates the scalability and practicality of hardware acceleration for

Table IV: Performance breakdown between two transformers generating a one word output using NEXUS. Inputs to the BERT-base model and Llama-3-8B model consist of 128 and 8 tokens respectively. We batched 32 inputs in total and evaluated the benchmark on a machine with a 32-core CPU and four A100 GPUs. Depth represents the change in available multiplication depth before and after each operation. Runtime is the amortized latency of each input.

Operation	Depth	BERT-base (12 layers)			Llama-3-8B (32 layers)		
		Input	CPU(s)	GPU(s)	Input	CPU(s)	GPU(s)
MATRIXMUL	$21 \to 20$	$(\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times768}) \times 3$	65	2.68	$(\mathbb{R}^{8\times4096} \times \mathbb{R}^{4096\times4096}) \times 3$	308	12.71
MATRIXMUL	$20 \to 21$	$(\mathbb{R}^{128\times64} \times \mathbb{R}^{64\times128}) \times 12$	14	0.54	$(\mathbb{R}^{8\times128} \times \mathbb{R}^{128\times8}) \times 32$	1	0.03
SOFTMAX	$19 \to 3$	$(\mathbb{R}^{128\times128}) \times 12$	47	1.15	$(\mathbb{R}^{8\times8}) \times 32$	2	0.04
MATRIXMUL	$3 \to 2$	$(\mathbb{R}^{128\times128} \times \mathbb{R}^{128\times64}) \times 12$	9	0.36	$(\mathbb{R}^{8\times8} \times \mathbb{R}^{8\times128}) \times 32$	1	0.02
MATRIXMUL	$2 \to 1$	$\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times768}$	2	0.06	$\mathbb{R}^{8\times4096} \times \mathbb{R}^{4096\times4096}$	10	0.28
BOOTSTRAPPING	$1 \to 17$	$\mathbb{R}^{128\times768}$	127	5.63	$\mathbb{R}^{8\times4096}$	113	5.00
LAYERNORM	$17 \to 1$	$\mathbb{R}^{128\times768}$	16	1.01	$\mathbb{R}^{8\times4096}$	14	0.91
BOOTSTRAPPING	$1 \to 17$	$\mathbb{R}^{128\times768}$	127	5.63	$\mathbb{R}^{8\times4096}$	113	5.00
MATRIXMUL	$17 \to 16$	$\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times3072}$	48	1.71	$\mathbb{R}^{8\times4096} \times \mathbb{R}^{4096\times14336}$	203	7.33
GELU	$16 \to 2$	$\mathbb{R}^{128\times3072}$	44	3.35	$\mathbb{R}^{8\times14336}$	36	2.72
MATRIXMUL	$2 \to 1$	$\mathbb{R}^{128\times3072} \times \mathbb{R}^{3072\times768}$	8	0.20	$\mathbb{R}^{8\times14336} \times \mathbb{R}^{14336\times4096}$	34	0.88
BOOTSTRAPPING	$1 \to 17$	$\mathbb{R}^{128\times768}$	127	5.63	$\mathbb{R}^{8\times4096}$	113	5.00
LAYERNORM	$17 \to 1$	$\mathbb{R}^{128\times768}$	16	1.01	$\mathbb{R}^{8\times4096}$	14	0.91
BOOTSTRAPPING	$1 \to 21$	$\mathbb{R}^{128\times768}$	153	5.90	$\mathbb{R}^{8\times4096}$	136	5.92
ARGMAX	*	$\mathbb{R}^{30522}$	54	2.48	$\mathbb{R}^{128256}$	110	5.09
Total	-	-	**857**	**37.34**	-	**1088**	**51.84**

secure transformer inference.

Table-IV lists the runtime for each individual operation in NEXUS. Bootstrapping is the most time-consuming part of the entire process, requiring 534s and occupying 62.3% of the total runtime in our CPU implementation.

Accuracy We evaluate the inference accuracy of NEXUS with 3 datasets (RTE, SST-2, and QNLI) from the GLUE benchmark [55], a widely adopted evaluation benchmark for transformers. As shown in Table-V, NEXUS attains comparable levels of accuracy relative to plaintext inference.

Table V: Inference accuracy of BERT-base and Llama-3-8B on the GLUE benchmarks.

Model	Dataset	Plaintext	NEXUS
BERT-base	RTE	70.04%	69.88%
	SST-2	92.36%	92.11%
	QNLI	90.30%	89.90%
Llama-3-8B	RTE	82.75%	81.24%
	SST-2	94.94%	94.46%
	QNLI	90.70%	90.20%

E. *Error Analysis and Trade-offs*

Our analysis aims to answer three research questions:

- **RQ1:** Why is NEXUS more accurate than Phoenix[30]?
- **RQ2:** Why is NEXUS more accurate than MPC-based methods?
- **RQ3:** What's the trade-off between accuracy and latency?

We present the formulas for calculating the errors of specific homomorphic operations in the following table according to [32], [38]. We take a message scaling factor of $\Delta \approx 2^{50}$, the discrete Gaussian distribution with standard deviation $\sigma = 3.2$, hamming weight $h = 192$.

Table VI: Error of specific homomorphic operations

Operation	Error	Bound
encode-encrypt	$\mathbf{e}_{\text{clean}} \leq \frac{18\sigma\sqrt{N}+32\sqrt{6}\sigma N}{3\Delta}$	6.73×10^{-8}
rescaling	$\mathbf{e}_{\text{rs}} \leq \frac{3\sqrt{3N}+8\sqrt{2}N}{3\Delta}$	7.01×10^{-9}
key-switching	$\mathbf{e}_{\text{ks}} \leq \frac{3\sqrt{3N}+8\sqrt{3}\cdot\sigma N+8\sqrt{2}N}{3\Delta}$	3.46×10^{-8}
bootstrapping	parameter II in [10]	4.32×10^{-5}

According to the Lemma 7 in [13], if polynomial $f(x) = \sum_{j=0}^{d} a_j x^j$ and input p has relative error β_0, one can compute $f(p)$ with a relative error bounded by $\beta_d < 2 \cdot d \cdot \beta_0$.

We use 4 polynomials in our SGN implementation and each polynomial has degree 9, hence we can consider SGN as a 36-degree polynomial. And the error introduced by SGN is

$$\mathbf{e}_{\text{sgn}} \approx 2^{-\alpha} + 72 \cdot \mathbf{e}_{\text{clean}} \approx 3.89 \times 10^{-7}$$

To answer **RQ1**, for argmax evaluation, the error introduced by Phoenix [30] is

$$\mathbf{e}_{\text{phoenix}}^{\text{argmax}} = m \cdot (\mathbf{e}_{\text{sgn}} + \mathbf{e}_{\text{ks}}) \approx 1.18 \times 10^{-2}$$

the error introduced by NEXUS is

$$\mathbf{e}_{\text{nexus}}^{\text{argmax}} = \log m \cdot (\mathbf{e}_{\text{sgn}} + \mathbf{e}_{\text{ks}} + \mathbf{e}_{\text{bs}}) \approx 6.91 \times 10^{-4}$$

To answer **RQ2**, we assume that the polynomial fits are identical and only discuss the errors caused by MPC and HE computations.

The error in MPC comes from the float-to-fixed conversion. Given a scale $f \in \mathbb{Z}$, it maps a real number r to a l-bit integer $\lfloor r \cdot 2^f \rfloor \in \mathbb{Z}_{2^l}$. Note that the bits after the f-bit are truncated. So the error introduced by MPC is

$$\mathbf{e}_{\text{mpc}} = \frac{r \cdot 2^f - \lfloor r \cdot 2^f \rfloor}{2^f} \approx 2^{-f}$$

Take GELU evaluation as an example. Both BOLT [45] and BumbleBee [42] set the scale $f = 12$ and bit length $l = 32$, where the minimum error is $2^{-f} \approx 2.4 \times 10^{-4}$. Whereas the error introduced by NEXUS is

$$\mathbf{e}_{\text{nexus}}^{\text{gelu}} = 3 \cdot (\mathbf{e}_{\text{sgn}} + 12 \cdot \mathbf{e}_{\text{clean}}) \approx 1.24 \times 10^{-6}$$

To answer **RQ3**, we should first figure out where the trade-offs come from. The accuracy and latency of NEXUS mainly depends on bootstrapping. It can be observed from Table-IV that bootstrapping takes up 62.3% of the total time, and Table-VII shows that the error of bootstrapping is the largest among all homomorphic operations. The trade-off comes from the polynomial approximation of $sin(x)$ function in the bootstrapping process [10], [38]. Lower-degree polynomials can effectively accelerate this process but will lead to a significant drop in accuracy, and vice versa.

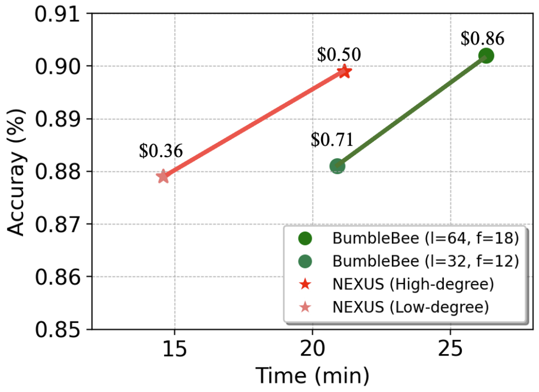


Figure 11: Trade-offs between inference accuracy and amortized latency of BERT-base on QNLI. The network settings are: 100Mbps bandwidth and 80ms latency.

The accuracy-latency trade-off made by MPC-based methods pertains to the bit-length l and scale f in secret sharing, where $\lfloor r \cdot 2^f \rfloor \in \mathbb{Z}_{2^l}$. Secret sharing with larger bit lengths and scales can be computed with higher accuracy but will incur more communication overhead, which significantly increases financial costs and the latency in WAN settings.

Figure-11 shows the trade-off between accuracy and latency. For NEXUS, we use high/low-degree polynomials to approximate $sin(x)$ function in bootstrapping. And for Bumblebee, we use different bit lengths and scales for secret sharing. Both HE and MPC protocols can achieve higher accuracy, but it will be at a cost of increased inference time and higher financial spending.

VII. Related Work

Interactive secure inference for transformers. With the proliferation of ChatGPT, secure transformer inference has become a key area of research. There are works such as Privformer [4], Puma [18] and Sigma [23] that present three-party protocols that require additional trust assumptions. There are also several other works based on 2PCs: [11], [25], [45], [27], [42], [39]. Iron [25] is an optimization of a secure CNN protocol named Cheetah [28] and uses a more efficient packing strategy to reduce the cost of matrix multiplication. Bumblebee [42] further optimized this 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 matrix entries, resulting in a much lower number of ciphertexts needed to be transferred. THE-X [11] and MPCFormer [39] simply replace GELU, Softmax with a combination of ReLU and polynomials, hence both of them require model retraining. BOLT [45] is the state-of-the-art solution for secure transformer inference. Our experimental results show that NEXUS achieves a speedup of 3.6× 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 [49], [41], [36], [8], [21], [29] are designed for convolutional neural networks (CNNs). AutoFHE [8] can automatically optimize the placement of bootstrapping operations in the CNN workflow. CryptoNAS [21] and DeepReduce [29] proposed evaluating non-linear functions like ReLU using FHE, but they cannot support the non-linear functions required by transformers, such as GELU, Softmax and Layer Normalization. NEXUS is arguably the first protocol for non-interactive secure transformer inference.

FHE acceleration. Recent research on optimizing compilers [16], [15], [54], GPU acceleration [56], [5], and specialized hardware accelerators [51], [3], [33] has demonstrated significant speedups for RNS-CKKS. These results can be used directly to accelerate NEXUS.

VIII. Conclusion

We propose NEXUS, the first non-interactive secure transformer inference protocol. We design a series of new protocols based on RNS-CKKS that allow the server to efficiently and accurately compute each layer of the transformer model on encrypted data. Since the scalability of non-interactive protocols is not limited by network bandwidth, we posit that combining NEXUS with carefully designed and deeply integrated hardware acceleration implementations will make secure transformer inference ready for practical deployment.

Acknowledgment

This work is funded by National Key Research and Development Project (Grant No: 2022YFB2703100) and National Key Research and Development Program of China (2023YFB2704000).

References

[1] https://aws.amazon.com/cn/ec2/pricing/on-demand/, 2024. Accessed: 2024-04-18.

[2] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J Kusner, and Adrià Gascón. Quotient: two-party secure neural network training and prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1231–1247, 2019.

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

[4] Yoshimasa Akimoto, Kazuto Fukuchi, Youhei Akimoto, and Jun Sakuma. Privformer: Privacy-preserving transformer with mpc. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroSP)*, pages 392–410, 2023.

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

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

[7] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*, pages 962–979. IEEE, 2018.

[8] Wei Ao and Vishnu Naresh Boddeti. Autofhe: Automated adaption of cnns for efficient evaluation over fhe. 33st USENIX Security Symposium (USENIX Security 24), 2024.

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

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

[11] Tianyu Chen, Hangbo Bao, Shaohan Huang, Li Dong, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. THE-X: Privacy-preserving transformer inference with homomorphic encryption. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Findings of the Association for Computational Linguistics: ACL 2022*, pages 3510–3520, Dublin, Ireland, May 2022. Association for Computational Linguistics.

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

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 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*, pages 409–437. Springer, 2017.

[14] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. 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*, pages 221–256. Springer, 2020.

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

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

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

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

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

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

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

[22] Robert E Goldschmidt. *Applications of division by convergence*. PhD thesis, Massachusetts Institute of Technology, 1964.

[23] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. Sigma: secure gpt inference with function secret sharing. *Cryptology ePrint Archive*, 2023.

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

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

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

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

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

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

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

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

[32] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. Approximate homomorphic encryption with reduced approximation error. In *Cryptographers' Track at the RSA Conference*, pages 120–144. Springer, 2022.

[33] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *Proceedings*

of the 50th Annual International Symposium on Computer Architecture, pages 1–15, 2023.

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

[35] Eunsang Lee, Joon-Woo Lee, Young-Sik Kim, and Jong-Seon No. Optimization of homomorphic comparison algorithm on RNS-CKKS scheme. Cryptology ePrint Archive, Paper 2021/1215, 2021. https://eprint.iacr.org/2021/1215.

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

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

[38] Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 551–580. Springer, 2022.

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

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

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

[42] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Kui Ren, Cheng Hong, Tao Wei, and WenGuang Chen. Bumblebee: Secure two-party inference framework for large transformers. *Cryptology ePrint Archive*, 2023.

[43] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. SecretFlow-SPU: A performant and User-Friendly framework for Privacy-Preserving machine learning. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 17–33, Boston, MA, July 2023. USENIX Association.

[44] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2505–2522. USENIX Association, August 2020.

[45] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. *IEEE Symposium on Security and Privacy (SP)*, 2024.

[46] Hongyuan Qu and Guangwu Xu. Improvements of homomorphic evaluation of inverse square root. *Available at SSRN 4258571.*

[47] Hongyuan Qu and Guangwu Xu. Improvements of homomorphic secure evaluation of inverse square root. In *International Conference on Information and Communications Security*, pages 110–127. Springer, 2023.

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

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

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

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

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

[53] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2017.

[54] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. {HECO}: Fully homomorphic encryption compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4715–4732, 2023.

[55] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[56] Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng. He-booster: An efficient polynomial arithmetic acceleration on gpus for fully homomorphic encryption. *IEEE Transactions on Parallel and Distributed Systems*, 34(4):1067–1081, 2023.

[57] Hongyang Yan, Shuhao Li, Yajie Wang, Yaoyuan Zhang, Kashif Sharif, Haibo Hu, and Yuanzhang Li. Membership inference attacks against deep learning models via logits distribution. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[58] Jiayuan Ye, Aadyaa Maddi, Sasi Kumar Murakonda, Vincent Bindschaedler, and Reza Shokri. Enhanced membership inference attacks against machine learning models. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3093–3106, 2022.

APPENDIX

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 $\mathrm{E}(p(x))$ be the encryption of $p(x)$. Then, the N' output ciphertexts $o_0, ..., o_{N-1}$ of Decompress$(\mathrm{E}(p(x)))$ satisfy:*

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

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

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

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

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

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:

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

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 1). ■

B. Commutable Encryption

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

1) Parse $\textsc{Enc}_\mathcal{S}(m)$ as $(A, As_\mathcal{S} + m + e)$
2) Output $(A, As_\mathcal{S} + As_\mathcal{C} + m + e + e')$

Decrypting it with the server's secret key yields:

$$(A, As_\mathcal{S} + As_\mathcal{C} + m + e + e') - (0, As_\mathcal{S}) = (A, As_\mathcal{C} + m + e + e').$$

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

C. Ciphertext-Ciphertext MatrixMul

Suppose the matrix $\mathbf{Q} \in \mathbb{R}^{m \times n}$ and $\mathbf{K} \in \mathbb{R}^{m \times n}$ packed in column and $\mathbf{q}_j$, $\mathbf{k}_j$ are column vectors for $\forall\ j \in [n]$. We leverage the SIMD element-wise multiplication to find that $(\mathbf{q}_0 \boxtimes \mathbf{k}_0) \boxplus (\mathbf{q}_1 \boxtimes \mathbf{k}_1) \cdots \boxplus (\mathbf{q}_{n-1} \boxtimes \mathbf{k}_{n-1})$ is the diagonal-pack of matrix $\mathbf{Q} \times \mathbf{K}^T$. And we can continue computing the other diagonal of matrix $\mathbf{Q} \times \mathbf{K}^T$ by just rotating the vector $\mathbf{k}_0, \mathbf{k}_1, \cdots, \mathbf{k}_{n-1}$ and get the results $\mathbf{r}_0, \mathbf{r}_1, \cdots, \mathbf{r}_{n-1}$ as shown in Algorithm 5.

D. Security Analysis

1) Security Proof of Matrix Multiplication: We follow the definition of privacy in the simulation paradigm. Namely, the algorithm should be secure against a static semi-honest probabilistic polynomial time adversary corrupting either $\mathcal{C}$ or $\mathcal{S}$.

- **Corrupted client.** We require that a corrupted, semi-honest client to not be able to 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).

Algorithm 5 Ciphertext-Ciphertext MatrixMul

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

```
1:  function MatrixMul($\mathbf{Q}, \mathbf{K}^T$)
2:      for $i = 0$ to $m - 1$ do
3:          $\mathbf{r} \leftarrow \mathbf{0}$
4:          for $j = 0$ to $n - 1$ do
5:              $\mathbf{r}_i \leftarrow \mathbf{r}_i \boxplus (\mathbf{q}_j \boxtimes \mathbf{k}_j)$
6:          end for
7:          for $j = 0$ to $n - 1$ do
8:              $\mathbf{k}_j \leftarrow \textsc{RotL}(\mathbf{k}_j, 1)$
9:          end for
10:     end for
            return $[\mathbf{r}_0, \mathbf{r}_1, \cdots, \mathbf{r}_{m-1}]$
11: end function
```

- **Corrupted server.** We require that a corrupted, semi-honest server to not be able to learn anything about the private input $\mathbf{A}$ of the client. Formally, we require the existence of an efficient simulator $Sim_\mathcal{S}$ such that $View_\mathcal{S} \approx_c Sim_\mathcal{S}(\mathbf{W}, out)$, where $View_\mathcal{S}$ denotes the view of the server in the execution, and out denotes the output, namely $\textsc{Enc}_c(\mathbf{A} \cdot \mathbf{W})$.

The functionality of offine-online batch matrix multiplication (cf. Section III.C) is denoted by $\mathcal{F}_{Matrixmul}$. We summarize the privacy of this functionality in theorem 3.

Theorem 3. *Assuming $\mathcal{F}_{Enc}$ is the homomorphic encryption functionality. The functionality of offine-online batch matrix multiplication (cf. Section III.C) securely realize $\mathcal{F}_{Matrixmul}$ in the $\mathcal{F}_{Enc}$ model.*

Proof: **Corrupted client.** The client view consists of ciphertexts $\{\textsc{Enc}_\mathcal{S}(w_\gamma)\}$. The simulator Sim_C can be constructed by:

1) Output ciphertexts $Enc_\mathcal{S}(0)$.

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 $\{\textsc{Enc}_\mathcal{C}(\textsc{Enc}_\mathcal{S}(\mathbf{v}_{\alpha,\delta}))\}$, plaintext polynomials $\mathbf{a}_{\alpha,\beta} - \mathbf{u}_{\alpha,\beta}$, and the output $\textsc{Enc}_c(\mathbf{A} \cdot \mathbf{W})$. The simulator $Sim_\mathcal{S}$ can be constructed by:

1) The simulator follows step 2 and 3 in the algorithm with knowledge of $\mathbf{W}$. The only difference is that $\mathbf{u}_{\alpha,\beta}$ is replaced with $\hat{\mathbf{u}}_{\alpha,\beta}$ which is sampled by the simulator instead of $\mathcal{C}$ such that
$$\textsc{Enc}_\mathcal{S}(\hat{\mathbf{v}}_{\alpha,\delta}) \leftarrow \boxplus_{\beta \in [n]} \left(\hat{\mathbf{u}}_{\alpha,\beta} \boxtimes \textsc{Enc}_\mathcal{S}(w'_{(\delta-1)n+\beta})\right)$$
The output ciphertexts $\{\textsc{Enc}_\mathcal{C}(\textsc{Enc}_\mathcal{S}(\hat{\mathbf{v}}_{\alpha,\delta}))\}$ are indistinguishable from $\{\textsc{Enc}_\mathcal{C}(\textsc{Enc}_\mathcal{S}(\mathbf{v}_{\alpha,\delta}))\}$ by 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 receives the output $\textsc{Enc}_c(\mathbf{A} \cdot \mathbf{W})$ and forwards it.

■

2) Threat Model and Security: In this work we assume a static semi-honest probabilistic polynomial time adversary $\mathcal{A}$, who corrupts either the server $\mathcal{S}$ or the client $\mathcal{C}$. 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 $\mathcal{A}$ corrupts the client, it tries to learn the model parameters. We adopt the definition of security from [44].

We summarize the correctness and security of our proposed protocol in definition 1.

Definition 1. *A protocol* Π *between* $\mathcal{S}$ *holding a model parameters* $\mathbf{M} = (\mathbf{M}_1, ..., \mathbf{M}_\ell)$ *and* $\mathcal{C}$ *holding an input* $\mathbf{A}$ *is a secure inference protocol if it satisfies:*

- ***Correctness.*** *The output at the end of execution is the correct inference result* $\mathcal{M}(\mathbf{A})$.
- ***Security.***
  - ***Corrupted client.*** *There exists an efficient simulator* $Sim_{\mathcal{C}}$ *such that* $View_{\mathcal{C}}^{\Pi} \approx_c Sim_{\mathcal{C}}(\mathbf{A}, out)$, *where* $View_{\mathcal{C}}^{\Pi}$ *denotes* $\mathcal{C}$*'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.

Proof: Π can be constructed by replacing the function with its secure implementations. The correctness derives directly from the underlining algorithms. It follows that the privacy of Π is also achieved since only HE ciphertexts are exchanged. ■