CipherGPT: Secure Two-Party GPT Inference

Xiaoyang Hou
Zhejiang University
xiaoyanghou@zju.edu.cn

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

Jingyu Li
Zhejiang University
jingyuli@zju.edu.cn

Wen-jie Lu
Ant Group
juhou.lwj@antgroup.com

Cheng Hong
Ant Group
vince.hc@antgroup.com

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

***Abstract*—ChatGPT is recognized as a significant revolution in the field of artificial intelligence, but it raises serious concerns regarding user privacy, as the data submitted by users may contain sensitive information. Existing solutions for secure inference face significant challenges in supporting GPT-like models due to the enormous number of model parameters and complex activation functions.**

In this paper, we develop CipherGPT, the *first* framework for secure two-party GPT inference, building upon a series of innovative protocols. First, we propose a secure matrix multiplication that is customized for GPT inference, achieving upto 2.5× speedup and 11.2× bandwidth reduction over SOTA. We also propose a novel protocol for securely computing GELU, surpassing SOTA by 4.2× in runtime, 3.4× in communication and 10.9× in precision. Furthermore, we come up with the first protocol for top-k sampling.

We provide a full-fledged implementation and comprehensive benchmark for CipherGPT. In particular, we measure the runtime and communication for each individual operation, along with their corresponding proportions. We believe this can serve as a reference for future research in this area.

1. Introduction

ChatGPT, a large language model (LLM) built upon the groundbreaking *generative pre-trained transformer* (GPT) architecture [28], is regarded as a significant revolution in the field of artificial intelligence. With a vast knowledge base and impressive linguistic capabilities, ChatGPT excels in various tasks, including question answering, article polishing, suggestion offering, and engaging in conversations. It can also serve as a virtual assistant, effectively enabling applications like customer support, information retrieval, and language translation.

OpenAI has made ChatGPT as an online inference service and has even provided a remote API for developers to utilize. Users can conveniently enjoy the services by submitting prompts or messages for GPT inference. However, this service paradigm inevitably put user privacy at risk, as the data submitted by users may contain sensitive information. Such privacy concerns may restrict the deployment of GPT in certain critical scenarios where data confidentiality is crucial.

Secure inference [17], [25], [24], [26], [31], [30], [23], [21] is a two-party cryptographic protocol running the inference stage in way such that the server (S) learns nothing about clients' input and a client (C) learns nothing about the model except the inference results. Roughly, it proceeds by having S and C running the encrypted model over the encrypted input through tailor cryptographic techniques such as homomorphic encryption and secret sharing. A *preprocessing phase* is usually introduced to prepare some expensive and input-independent work so that the *online phase* can be done efficiently.

Unfortunately, existing protocols for secure inference are limited in their ability to support GPT. For instance, Cheetah [23] is specifically tailored for convolutional neural networks such as ResNet50, while Iron [21] operates solely on a single transformer. On the other hand, LLMs such as GPT-2, which consist of 12 transformers, entail a multitude of high-dimensional matrix multiplications and complex mathematical functions like GELU. Therefore, the advent of GPT has indeed introduced new challenges to the field of secure inference.

1.1. Our contributions

In this paper, we develop CipherGPT, the *first* framework for secure GPT inference, building upon a series of novel protocols.

VOLE-based matrix multiplication. GPT takes lengthy sentences as input and autoregressively generates response words. Specifically, after a response word is produced, that word is added to the input sentence, and the new sentence becomes the input to the model to produce the next response word. Each response word generation requires a model inference, which involves a matrix multiplication (MatrixMul for short) at each layer. During the preprocessing phase, at each layer, we combine the MatrixMuls for individual response words into a single *unbalanced* MatrixMul, and process it using sVOLE.

. *Jian Liu is the corresponding author.

Vector oblivious linear evaluation VOLE [4], [6], [38], [5], [13] is used to generate correlations like $\mathbf{w} = \mathbf{u}x + \mathbf{v}$, where a sender with input x learns a vector $\mathbf{w}$ of length n, and a receiver learns $(\mathbf{u}, \mathbf{v})$, both of length n. *Subfield* VOLE (sVOLE) [6] is a generalization of VOLE; ideally, sVOLE accomplishes the same task as k instances of regular VOLE, while maintaining a comparable cost to running a single VOLE instance. sVOLE is more cost-effective when $n \gg k$, making it particularly useful for computing unbalanced MatrixMuls.

Spline-based GELU. GPT uses GELU as its activation function, which can be represented as:

$$\mathsf{GELU}(x) = 0.5x(1 + \mathsf{Tanh}\left[\sqrt{2/\pi}(x + 0.044715x^3)\right]),$$

where $\mathsf{Tanh}(x) = 2\mathsf{Sigmoid}(2x) - 1$ and $\mathsf{Sigmoid}(x) = \frac{1}{1+e^{-x}}$. To securely compute GELU, the SOTA approaches [30], [21] employ a lookup table (LUT) to approximate e^{-x} and another lookup table to approximate the reciprocal. This multi-step process further requires extension or truncation of bitwidths at each step to balance precision and efficiency.

In contrast, we aim to compute GELU as a whole in a single step. To achieve this, we split GELU into several intervals and use a linear function ($y = ax + d$) to approximate the curve within each interval. This *spline*-based approximation was initially proposed by Liu et al. [25], in which garbled circuits were used to find the interval x belongs to and compute the corresponding linear function. We significantly improve its performance by leveraging LUT to find the interval and computing the corresponding linear function in a secret-shared manner.

Compared with the SOTA approach for computing GELU [21], we save one LUT, two truncations and three secret-shared multiplications; and we require a much smaller lookup table. Furthermore, our single-step approach exhibits superior precision, as it avoids the error accumulation inherent in multi-step approaches.

Shuffling-based top-K selection. A straightforward way for selecting the top-K elements from a secret-shared vector of length n is to securely sort the vector, which is typically achieved by securely executing a data-independent sorting algorithm such as Bitonic sorting network [22].

Our first insight is to implement secure sorting based on an idea from [2]: the input elements are securely shuffled first; and then a comparison-based sorting (e.g., quicksort) protocol is applied to arrange the shuffled elements into a sorted order. Notice that the comparison results obtained during the sorting process reveal no information about the original elements as those elements have been shuffled.

Our second insight is that, if quicksort is used, it is unnecessary to sort the entire vector. Instead, we can leverage a modified version of the quicksort algorithm. Typically, quicksort randomly selects an element from the vector as a pivot and compares it with other elements. Based on the comparison results, the vector is partitioned into two parts: elements smaller than the pivot and elements larger than the pivot. The quicksort algorithm then recursively operates on both partitions. In our case, we only need to recursively process the partitions that contain the top-K largest elements, which reduces the number of secure comparisons from $O(n \log n)$ to $O(n)$.

Secure sampling. We also tackle the problem of securely choosing an element from a vector based on secret-shared probabilities. Specifically, given a vector of K elements, each of which is associate a secret-shared probability p_i, the probability for the j-the element to be chosen is p_j. Our protocol only requires $(K-1)$ secure comparisons and K multiplexers. To the best of our knowledge, our study represents the first exploration of secure sampling.

We summarize our contributions as follows:

- A customized secure matrix multiplication for GPT, achieving upto 2.5× speedup and 11.2× bandwidth reduction over SOTA (Section 3);
- A novel protocol for securely computing GELU, surpassing SOTA by 4.2× in runtime, 3.4× in communication and 10.9× in precision. (Section 4);
- An innovative solution for top-K sampling: selecting the top-K probabilities and sampling one element according to the selected probabilities (Section 5 and 6);
- The first framework for secure GPT inference (Section 7), and a comprehensive benchmark that can serve as a reference for endeavors in this research direction (Section 8).

2. Preliminaries

In this section, we present the necessary preliminaries for understanding this paper. Table 1 provides a summary of the frequently used notations in this paper. We use $\langle x \rangle^l$ to denote the secret-sharing of a l-bit value x. For simplicity, we omit the l notation when it is not contextually relevant.

2.1. Secure inference and threat model

Secure inference is a two-party cryptographic protocol that facilitates model inference between a client C and a server S. The protocol ensures that C only obtains knowledge about the model architecture and the inference result, while keeping all other details of S's model hidden. Similarly, S remains unaware of C's input, as well as the output of the inference. In the context of GPT inference, C's input is a prompt and S's model comprises multiple iterations of transformer decoders together with a vec2word layer. More information about the GPT architecture will be provided in Section 7.

Either C or S can be considered a semi-honest adversary, which follows the protocol specifications but attempts to gather as much information as possible during the protocol execution.

Notation	Description
C	client
S	server
n	input vector length (for each layer)
L	# bits left-shifted for initial inputs
l	input bit-length after left-shifting
$\langle x \rangle^l$	$(\langle x \rangle^l_{\mathsf{S}}, \langle x \rangle^l_{\mathsf{C}})$ s.t. $x = \langle x \rangle^l_{\mathsf{S}} + \langle x \rangle^l_{\mathsf{C}} \mod 2^l$
$\mathsf{F}_{\mathsf{Mult}}$	ideal functionality for secret-shared multiplication: multiply a g-bit integer with a h-bit integer and produce an $l = (g + h)$-bit output, with no overflow
$\mathsf{F}_{\mathsf{CMP}}$	ideal functionality for comparison $b \leftarrow \mathsf{CMP}(x, y)$: $b = 1$ if $x \geq y$; $b = 0$ otherwise
$\mathsf{F}_{\mathsf{MUX}}$	ideal functionality for multiplexer $y \leftarrow \mathsf{MUX}(x, b)$: $y = x$ if $b = 1$; $y = 0$ if $b = 0$
$\mathsf{F}_{\mathsf{Trunc}}$	ideal functionality for truncation $y \leftarrow \mathsf{Trunc}(x, s)$: $y = x \gg s$ with $x, y \in \mathbb{Z}_{2^l}$
F_{TR}	ideal functionality for truncate-then-reduce $y \leftarrow \mathsf{TR}(x, s)$: $y = x \gg s$ with $x \in \mathbb{Z}_{2^l}$ and $y \in \mathbb{Z}_{2^{l-s}}$
$\mathsf{F}_{\mathsf{LUT}}$	ideal functionality for lookup table $T[i] \leftarrow \mathsf{LUT}(T, i)$
$\mathsf{F}_{\mathsf{Shuffle}}$	ideal functionality for shuffling
α	the split for $y := \mathsf{GELU}(x)$: $y := 0$ when $x < -\alpha$; $y := \mathsf{GELU}(x)$ when $-\alpha \leq x \leq \alpha$; $y := x$ when $x > \alpha$
s	2^s is the number of intervals within $[-\alpha, \alpha]$
(a_i, d_i)	$y = a_i x + d_i$ is the linear function that approximates $\mathsf{GELU}(x)$ in each interval
g	bit-length of a_i
t	# response words (# input matrices)
N	polynomial modulus degree in FHE
M	# attention heads
T	temperature

TABLE 1: A table of frequent notations.

2.2. Cryptographic Primitives

Multiplication with non-uniform bit-widths The ideal functionality $\mathsf{F}_{\mathsf{Mult}}$ takes $\langle x \rangle^g$ and $\langle y \rangle^h$ as input and returns $\langle z \rangle^l$, where $z = x \cdot y$ and $l = g + h$. A simple way to realize this functionality is to first extend both inputs to l bits and then use a standard protocol for secret-shared multiplication with uniform bit-widths. SIRNN [30] provides a protocol that outperforms this naive solution by $1.5\times$.

Secure comparison. The ideal functionality $\mathsf{F}_{\mathsf{CMP}}$ takes $\langle x \rangle^l$ and $\langle y \rangle^l$ as input and returns $\langle b \rangle^1$, where $b = 1$ if $x \geq y$, otherwise $b = 0$. The SOTA solution for secure comparison is provided by Cheetah [23].

Secure multiplexer. The ideal functionality $\mathsf{F}_{\mathsf{MUX}}$ takes $\langle x \rangle^l$ and $\langle b \rangle^1$ as input and returns $\langle y \rangle^l$, where $y = x$ if $b = 1$, and $y = 0$ if $b = 0$. The SOTA solution for secure multiplexer is provided by SIRNN [30].

Secure truncation. The ideal functionality $\mathsf{F}_{\mathsf{Trunc}}$ takes $\langle x \rangle^l$ and s as input and returns $\langle y \rangle^l$, where $y = x \gg s$. The SOTA solution for secure truncation is provided by SIRNN [30].

Truncate-then-reduce. The ideal functionality F_{TR} takes $\langle x \rangle^l$ and s as input and returns $\langle y \rangle^{l-s}$, where $y = x \gg s$. The SOTA solution for this functionality is also provided by SIRNN [30].

Lookup table. The ideal functionality $\mathsf{F}_{\mathsf{LUT}}$ takes $\langle i \rangle$ as input and returns $\langle T[i] \rangle$ where T is a table with M entries. This functionality can be achieved via a single call to $\binom{M}{1}$-OT [14]. A more efficient solution is to first convert the LUT description into a boolean expression and then evaluate it using a multi-fan-in inner product [9].

Secret-shared shuffle. The ideal functionality $\mathsf{F}_{\mathsf{Shuffle}}$ takes $\langle \mathbf{x} \rangle$ and $\langle \pi \rangle$ as input and returns $\langle \pi(\mathbf{x}) \rangle$, where π is a permutation function. Chase et al. [10] propose an efficient construction for this functionality using lightweight primitives such as OTs and PRGs. Their approach involves using puncturable PRFs to build a permute-and-share protocol, which allows two parties to permute the input vector with the permutation chosen by one party. This permute-and-share protocol is run twice, with each party choosing the permutation once.

Subfield vector oblivious linear evaluation. VOLE is a two-party functionality that takes a scalar $x \in \mathbb{F}_p$ from a sender and generates a VOLE *correlation*:

$$\mathbf{y} = \mathbf{u}x + \mathbf{v}, \tag{1}$$

s.t. the receiver learns $(\mathbf{u}, \mathbf{v}) \in_R \mathbb{F}_p^n \times \mathbb{F}_p^n$ and the sender learns $\mathbf{y} \in_R \mathbb{F}_p^n$. *Subfield* VOLE (sVOLE) is a generalization of VOLE with $\mathbf{u} \in_R \mathbb{F}_p^n$, $x \in \mathbb{F}_q$, $\mathbf{y}, \mathbf{v} \in_R \mathbb{F}_q^n$, and $q = p^m$. Notice that sVOLE achieves the same task as running m instances of normal VOLE, but with less cost.

Homomorphic encryption. *Fully homomorphic encryption* (FHE) is an encryption scheme that allows arbitrary operations to be performed over encrypted data [16]. In practice, it is usually used in a leveled fashion: the operations can only be performed for a limited times, o.w., the ciphertexts cannot be decrypted. In most FHE cryptosystems [8], [7], [15], [11], plaintexts are encoded as polynomials from the quotient ring $\mathbb{Z}_p[x]/(x^N+1)$, where N is a power of 2, and p is the plaintext modulus. The plaintext polynomials are then encrypted into ciphertext polynomials $\mathbb{Z}_q[x]/(x^N+1)$, where q is the ciphertext modulus that determines the security level, as well as how many times the operations can be performed.

3. Secure Matrix Multiplication

The MatrixMul operation takes two input matrices $\mathbf{X} \in \mathbb{Z}_{2^l}^{n \times m}$ and $\mathbf{Y} \in \mathbb{Z}_{2^l}^{m \times k}$ from C and S respectively, and outputs $\langle \mathbf{Z} \rangle$ with $\mathbf{Z} = \mathbf{XY} \in \mathbb{Z}_{2^l}^{n \times k}$. Most of existing solutions use homomorphic multiplications and additions to compute the above formula in a privacy-preserving way. The SIMD technique is typically used to amortize the cost by batching N elements into a single RLWE ciphertext, but it requires expensive homomorphic rotations to sum-up [24]. Cheetah [23] substitutes SIMD with coefficient packing to eliminate the expensive rotations. Nevertheless,

it still requires transferring $\geq \frac{2n\sqrt{mk}}{\sqrt{N}}$ RLWE ciphertexts, and performing $\geq \frac{nmk}{N}$ ciphertext-plaintext homomorphic multiplications.

Recall that GPT needs to autoregressively generate response words. Therefore, a single GPT inference requires running MatrixMul for different $\mathbf{X}$s with the same $\mathbf{Y}$. We aim to reduce the amortized cost of MatrixMul by exploiting this characteristic of GPT.

Let $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_m]$ (with $\mathbf{x}_i \in \mathbb{Z}_{2^l}^n$ being each column of $\mathbf{X}$) and $\mathbf{Y}^T = [\mathbf{y}'_1, \mathbf{y}'_2, \cdots, \mathbf{y}'_m]$ (with $\mathbf{y}'_i \in \mathbb{Z}_{2^l}^k$ being each row of $\mathbf{Y}$), then $\mathbf{Z} = \sum_{i=1}^{m} (\mathbf{x}_i \otimes \mathbf{y}'_i)$. Suppose S and C need to generate t response words, hence there are t input matrices:

$$\begin{aligned} \mathbf{X}_1 &= [\mathbf{x}_{1,1}, \mathbf{x}_{1,2}, \cdots, \mathbf{x}_{1,m}], \\ \mathbf{X}_2 &= [\mathbf{x}_{2,1}, \mathbf{x}_{2,2}, \cdots, \mathbf{x}_{2,m}], \\ &\cdots \quad \cdots \\ \mathbf{X}_t &= [\mathbf{x}_{t,1}, \mathbf{x}_{t,2}, \cdots, \mathbf{x}_{t,m}]. \end{aligned}$$

Let $\mathbf{x}'_i = \mathbf{x}_{1,i} || \mathbf{x}_{2,i} || \cdots || \mathbf{x}_{t,i} \ \forall \ i \in [1, m]$. Then,

$$\mathbf{x}'_i \otimes \mathbf{y}'_i = (\mathbf{x}_{1,i} \otimes \mathbf{y}'_i) || (\mathbf{x}_{2,i} \otimes \mathbf{y}'_i) || \cdots || (\mathbf{x}_{t,i} \otimes \mathbf{y}'_i).$$

Then,

$$\sum_{i=1}^{m} (\mathbf{x}'_i \otimes \mathbf{y}'_i) = \mathbf{Z}_1 || \mathbf{Z}_2 || \cdots || \mathbf{Z}_t.$$

Therefore, we could compute the t times of MatrixMul altogether via m outer products.

Given that $\mathbf{Y}$ is known beforehand, we could introduce a preprocessing phase to have S and C generate m sVOLE correlations:

$$\mathbf{W}_i = \mathbf{u}_i \otimes \mathbf{y}'_i + \mathbf{V}_i, \ \forall \ i \in [1, m].$$

where C holds $\mathbf{u}_i \in \mathbb{Z}_{2^l}^{(t \cdot n)}$ (which is a vector of length $t \cdot n$) and $\mathbf{V}_i \in \mathbb{Z}_{2^l}^{(t \cdot n) \times k}$, and S holds $\mathbf{y}'_i \in \mathbb{Z}_{2^l}^k$ and $\mathbf{W}_i \in \mathbb{Z}_{2^l}^{(t \cdot n) \times k}$.

In the online phase, for an input matrix $\mathbf{X}_j = [\mathbf{x}_{j,1}, \mathbf{x}_{j,2}, \cdots, \mathbf{x}_{j,m}]$, C sends

$$\langle \mathbf{x}_{j,i} \rangle_{\mathsf{S}} := \mathbf{x}_{j,i} - \mathbf{u}_i \left[(j-1)n + 1, \cdots, j \cdot n \right] \ \forall \ i \in [1, m]$$

to S, which then computes:

$$\begin{aligned} \langle \mathbf{x}_{j,i} \rangle_{\mathsf{S}} \otimes \mathbf{y}'_i &= (\mathbf{x}_{j,i} - \mathbf{u}_i \left[(j-1)n + 1, \cdots, j \cdot n \right]) \otimes \mathbf{y}'_i \\ &= \mathbf{x}_{j,i} \otimes \mathbf{y}'_i - \mathbf{u}_i \left[(j-1)n + 1, \cdots, j \cdot n \right] \otimes \mathbf{y}'_i. \end{aligned}$$

Then, we have:

$$\begin{aligned} \mathbf{x}_{j,i} \otimes \mathbf{y}'_i &= \langle \mathbf{x}_{j,i} \rangle_{\mathsf{S}} \otimes \mathbf{y}'_i + \mathbf{u}_i \left[(j-1)n + 1, \cdots, j \cdot n \right] \otimes \mathbf{y}'_i \\ &= \langle \mathbf{x}_{j,i} \rangle_{\mathsf{S}} \otimes \mathbf{y}'_i + \mathbf{W}_i \left[(j-1)kn + 1, \cdots, j \cdot k \cdot n \right] \\ &\quad - \mathbf{V}_i \left[(j-1)kn + 1, \cdots, j \cdot k \cdot n \right]. \end{aligned}$$

Notice that S holds:

$$\langle \mathbf{x}_{j,i} \rangle_{\mathsf{S}} \otimes \mathbf{y}'_i + \mathbf{W}_i \left[(j-1)kn + 1, \cdots, j \cdot k \cdot n \right],$$

and C holds:

$$\mathbf{V}_i \left[(j-1)kn + 1, \cdots, j \cdot k \cdot n \right];$$

that means S and C secret-share $\mathbf{x}_{j,i} \otimes \mathbf{y}'_i$, and consequently they can locally compute the secret-shares of $\mathbf{Z}_j = \sum_{i=1}^{m} (\mathbf{x}_{j,i} \otimes \mathbf{y}'_i)$. They can compute all $\mathbf{Z}$s in this way.

MatrixMul	Overhead
Cheetah [23]	transferring $\geq \frac{2n\sqrt{mk}}{\sqrt{N}}$ RLWE ciphertexts $\geq \frac{nmk}{N}$ ciphertext-plaintext multiplications
Iron [21]	transferring $\geq \frac{2\sqrt{nmk}}{\sqrt{N}}$ RLWE ciphertexts $\geq \frac{nmk}{N}$ ciphertext-plaintext multiplications
Ours	transferring $\frac{2 \cdot e \cdot m \cdot k}{tN}$ RLWE ciphertexts transferring $\frac{e \cdot m \cdot k}{t} + n \cdot m$ masked plaintext $\frac{e \cdot m \cdot k}{tN}$ ciphertext-plaintext multiplications $\frac{c \cdot e \cdot m \cdot \log(n \cdot t/e)}{t}$ OTs and $(c \cdot m \cdot n \cdot k)$ AESs

TABLE 2: Amortized cost for t times of MatrixMul. N is # elements batched in a RLWE ciphertext; e is the dual-LPN noise weight; c is a small constant ($N = 4\ 096$, $e = 144$, $c = 2$ in our benchmarks).

Table 2 compares the MatrixMul overhead among Cheetah [23], Iron [21] and CipherGPT. In terms of computation, we save $\frac{tn}{e}\times$ ciphertext-plaintext multiplications. Suppose $n = 256$, $m = 768$, $k = 64$, $t = 256$ and $e = 144$ (which are the real parameters for GPT-2), we save 3 065 ciphertext-plaintext multiplications, which takes more than 4s[1]. Although we need to do extra $(c \cdot m \cdot n \cdot k)$ AESs to expand the seeds, with the help of AES-NI this can be done in around 100ms. In terms of communication, we transfer at least 94 fewer RLWE ciphertexts, which is around 10MB; whereas the communication overhead introduced by OTs and plaintexts in CipherGPT is only around 1.5MB.

4. Secure GELU

In this section, we begin by providing a high-level overview of our GELU protocol, and then delve into its technical details.

4.1. Intuition

Figure 1 (left) depicts the original curve of $y = \mathsf{GELU}(x)$. It begins at zero for small values of x, and starts deviating from zero when x is around $-\alpha$. As x increases further, $\mathsf{GELU}(x)$ progressively approximates the linear function $y = x$. Based on this observation, we divide the curve into three large intervals:

- $y = 0$ when $x < -\alpha$;
- $y = \mathsf{GELU}(x)$ when $-\alpha \leq x \leq \alpha$;
- $y = x$ when $x > \alpha$.

The computation of the first and third intervals is straightforward. For the second interval, we use *polynomial splines* to approximate the curve. As depicted in Figure 1

1. This includes the time usage for noise flooding.

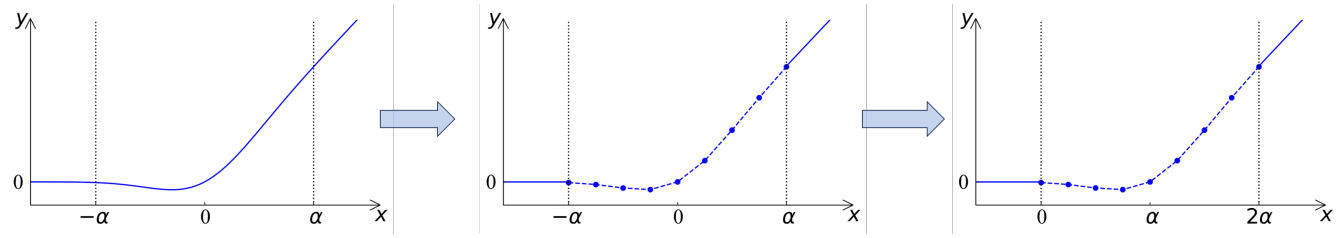


Figure 1: GELU transformation.

(middle), we divide the second interval into several small intervals and use a linear function ($y = ax + d$) to approximate the curve within each small interval. We refer to Section 5.3.2 in [25] for a detailed procedure of finding the linear functions. It is important to note that this approximation does not necessitate any modifications to the training phase of the model.

We could use LUT to find the small interval in which x resides and compute the corresponding linear function in a secret-shared manner. However, for $[-\alpha, \alpha]$, we have to determine the sign of x first, and then lookup in the intervals $[-\alpha, 0]$ and $[0, \alpha]$ separately. To avoid this, we right-shift the entire curve by α as shown in Figure 1 (right), after which the second interval becomes $[0, 2\alpha]$ allowing us to perform a single lookup.

4.2. Details

Algorithm 1 describes in detail how we securely compute $y := \mathsf{GELU}(x)$.

Algorithm 1: Secure GELU: Π_{GELU}

Input: S & C hold $\langle x \rangle^l$
Output: S & C get $\langle y \rangle^l$ for $y = \mathsf{GELU}(x)$

1 Let $\alpha' := 2^L \alpha$
2 S & C (locally) compute $\langle x' \rangle^l := \langle x \rangle^l + \alpha'$
3 Let $\beta := 2\alpha'$
4 Let $h := \log \beta$
5 S & C (locally) extract the lower h bits of $\langle x' \rangle^l$ and get $\langle x' \rangle^h$
6 S & C invoke $\langle i \rangle^s \leftarrow \mathsf{F}_{\mathsf{TR}}(\langle x' \rangle^h, h - s)$
7 S & C invoke $(\langle a_i \rangle^g, \langle d_i \rangle^l) \leftarrow \mathsf{F}_{\mathsf{LUT}}(\langle T \rangle, \langle i \rangle^s)$
8 S & C invoke $\langle ax \rangle^l \leftarrow \mathsf{F}_{\mathsf{Mult}}(\langle a_i \rangle^g, \langle x' \rangle^h)$
9 S & C (locally) compute $\langle z \rangle^l := \langle ax \rangle^l + \langle d_i \rangle^l$
10 S & C invoke $\langle b \rangle^1 \leftarrow \mathsf{F}_{\mathsf{CMP}}(\langle x' \rangle^l, \beta)$ ▷ $b = 1$ if $x' \geq \beta$; $b = 0$ otherwise
11 S & C invoke $\langle b' \rangle^1 \leftarrow \mathsf{F}_{\mathsf{CMP}}(\langle x' \rangle^l, 0)$ ▷ $b' = 1$ if $x' \geq 0$; $b' = 0$ otherwise
12 S & C invoke $\langle u \rangle^l \leftarrow \mathsf{F}_{\mathsf{MUX}}(\langle z \rangle^l, \langle b \rangle^1 \oplus \langle b' \rangle^1)$
13 S & C invoke $\langle v \rangle^l \leftarrow \mathsf{F}_{\mathsf{MUX}}(\langle x \rangle^l, \langle b \rangle^1)$
14 S & C (locally) compute $\langle y \rangle^l := \langle u \rangle^l + \langle v \rangle^l$

Notice that the initial input to the model has undergone a left-shift by L bits, which in turn affects the value of x, resulting in a left-shift of x by L bits as well. To maintain the desired alignment, we scale α up by a factor of 2^L (Line 1). Then, the split value becomes $\alpha' := 2^L \alpha$.

The right-shift of the curve needs to consider the scaling factor as well. Namely, instead of directly right-shifting the curve by α, we should right-shift it by α'. Similarly, to ensure proper alignment, the input to GELU should be adjusted as $x' := x + \alpha'$, which can be achieved by adding α' to any share of x (Line 2).

Handling small intervals. Let $\beta := 2\alpha'$, the second large interval now becomes $[0, \beta]$. We make the initial assumption that x' falls within this large interval; we will address the case where this assumption does not hold later on. As $x' \in [0, \beta]$, we only need to consider the lower $h := \log \beta$ bits of x'. To this end, we have S and C extract the lower h bits of $\langle x' \rangle^l$ and get $\langle x' \rangle^h$ (line 5), which can be done locally without any communication.

Suppose $[0, \beta]$ has been divided into 2^s small intervals. Then, we could find the interval for $\langle x' \rangle^h$ by examining its upper s bits. To this end, we have S and C run the truncate-then-reduce protocol on $\langle x' \rangle^h$ (Line 6), resulting in $\langle i \rangle^s$, where $i \in \mathbb{Z}_{2^s}$ represents the index of the small interval that x' belongs to.

S holds a table T, where each entry stores the coefficients of the linear function corresponding to the respective small interval. After obtaining $i \in \mathbb{Z}_{2^s}$, S and C execute LUT to get the i-th entry of T in a secret-shared from $(\langle a_i \rangle^g, \langle d_i \rangle^l)$ (Line 7). Then, they run "multiplication with non-uniform bitwidths" on $\langle a_i \rangle^g$ and $\langle x' \rangle^h$ (Line 8), resulting in $\langle ax \rangle^l$ with $l = g + h$. After adding $\langle d_i \rangle^l$ to $\langle ax \rangle^l$, they obtain $\langle z \rangle^l$, which is potentially the result of $\mathsf{GELU}(x)$.

Handling large intervals. Notice that the above process for handling small intervals is valid only when $x' \in [0, \beta]$. Indeed, the truncation in Line 5 will result in the loss of information for x' when $x' \notin [0, \beta]$. To this end, we use multiplexer to ensure that $\langle z \rangle^l$ will not be returned when $x' \notin [0, \beta]$.

S and C first securely compare x' with β and get b (Line 10), with $b = 1$ if $x' \geq \beta$ and $b = 0$ otherwise. Then, they securely compare x' with 0 and get b' (Line 11), with $b' = 1$ if $x' \geq 0$ and $b' = 0$ otherwise. Notice that there are only following three possibilities for the combination of b and b' (instead of four):

- $b = 1$ and $b' = 1$,
- $b = 0$ and $b' = 1$,
- $b = 0$ and $b' = 0$.

The second case with $b \oplus b' = 1$ indicates that $x' \in [0, \beta]$, whereas the other two cases with $b \oplus b' = 0$ indicate that $x' \notin [0, \beta]$. Therefore, we could use $b \oplus b'$ as the control signal to implement the multiplexer for z. Specifically, S and C run the multiplexer with input $\langle z \rangle^l$ and $\langle b \rangle^1 \oplus \langle b' \rangle^1$ resulting in $\langle u \rangle^l$ (Line 12), with $u = z$ if $b \oplus b' = 1$, and $u = 0$ otherwise.

Next, S and C run another multiplexer with input $\langle x \rangle^l$ and $\langle b \rangle^1$ resulting in $\langle v \rangle^l$ (Line 13), with $v = x$ if $b = 1$, and $v = 0$ otherwise. This multiplexer determines if $x' > \beta$; if so, returns $v = x'$. The final result of $\mathsf{GELU}(x)$ is $\langle y \rangle^l := \langle u \rangle^l + \langle v \rangle^l$. Notice that there is no need for an additional multiplexer to handle the case of $x' < 0$, because $y = 0$ when $x' < 0$.

Table 3 compares the number of cryptographic operations among SIRNN [30], Iron [21] and our solution for secure GELU. Clearly, our solution is much more lightweight. Furthermore, our solution is also better in precision: the multi-step process in SIRNN and Iron involves approximating exponentiation and reciprocation separately, introducing precision errors at each step; these errors accumulate throughout the process, resulting in a large overall error, which is not the case in our our single-step approach. Our experimental results (cf. Table 4) validate this conjecture.

GELU	Overhead
SIRNN [30]	1LUT (2^{18} entries), 5LUT (2^8 entries), 7Mult, 6Trunc, 5CMP, 2MUX
Iron [21]	1LUT (2^{18} entries), 5LUT (2^8 entries), 6Mult, 5Trunc, 5CMP, 2MUX
Ours	1LUT (2^8 entries), 1Mult, 1TR, 3CMP, 2MUX

TABLE 3: Comparison for GELU.

5. Secure Top-K Selection

In the vec2word layer, the GPT model generates a vector containing probabilities for all possible words. From this vector, the top-K largest probabilities need to be selected and the final response word needs to be sampled based on the selected probabilities. This section focuses on the process of selecting the top-K values from a vector of length n. In the subsequent section, we will discuss how we sample a value from the K selected probabilities.

Algorithm 2 provides a detailed description of our TopK protocol. At a high level, the input elements are securely shuffled first (Line 1); and then a comparison-based selection is employed to identify the top-K elements from the shuffled list (Line 2).

The selection function in Algorithm 2 operates in a recursive manner. Within each recursion, the last element of the vector is selected as the pivot (Line 5); and the vector is partitioned into two parts: elements smaller than the pivot, denoted as S_L, and elements larger than or equal to the pivot, denoted as S_R (Line 6-15). To split the vector,

Algorithm 2: Secure Top-K: Π_{TopK}

Input: S & C hold $\langle \mathbf{x} \rangle$ with $\mathbf{x} \in \mathbb{Z}_{2^l}^n$
Output: S & C get $\langle \mathbf{y} \rangle$ with $\mathbf{y} \in \mathbb{Z}_{2^l}^K$ being the K largest values of $\mathbf{x}$

1 S & C invoke $\langle \mathbf{x}' \rangle \leftarrow \mathsf{F}_{\mathsf{Shuffle}}(\langle \mathbf{x} \rangle)$
2 $\mathbf{y} \leftarrow \texttt{select}(\langle \mathbf{x}' \rangle, K)$
3 **Function** $\texttt{select}(\langle \mathbf{x}' \rangle, K)$**:**
4 $n := |\langle \mathbf{x}' \rangle|$
5 $\langle pivot \rangle := \langle x'_n \rangle$
6 $\langle S_L \rangle := \{\}, \langle S_R \rangle := \{\langle pivot \rangle\}$
7 **for** $i := 1$ **to** $n - 1$ **do**
8 S & C invoke $\langle b \rangle^1 \leftarrow \mathsf{F}_{\mathsf{CMP}}(\langle x'_i \rangle, \langle pivot \rangle)$ ▷ $b = 1$ if $x'_i \geq pivot$; $b = 0$ otherwise
9 S & C reveal $\langle b \rangle^1$ and get b
10 **if** $b = 0$ **then**
11 $\langle S_L \rangle := \langle S_L \rangle \cup \{\langle x'_i \rangle\}$ ▷ $x'_i < pivot$
12 **else**
13 $\langle S_R \rangle \leftarrow \langle S_R \rangle \cup \{\langle x'_i \rangle\}$ ▷ $x'_i \geq pivot$
14 **end**
15 **end**
16 $K' \leftarrow |\langle S_R \rangle|$
17 **switch** $(K' \ ? \ K)$ **do**
18 **case** $(K' = K)$
19 **return** $\langle S_R \rangle$
20 **case** $(K' > K)$
21 **return** $\texttt{select}(\langle S_R \rangle, K)$
22 **case** $(K' < K)$
23 **return** $\texttt{select}(\langle S_L \rangle, K - K') \cup \langle S_R \rangle$
24 **end**
25 **end**
26 **End Function**

all its elements are compared with the pivot (Line 8). The comparison results can be revealed (Line 9) without compromising the privacy of the original elements. This is because the original elements have been shuffled, ensuring that the comparison results are independent of the actual values.

If the size of S_R (denoted by K') is exactly K, it means that all the elements in S_R are the top-K largest elements that we want to select (Line 19). If $K' > K$, the next recursion is executed on S_R to further narrow down the selection (Line 21). On the other hand, if $K' < K$, the next recursion is performed to select the top $(K - K')$ elements from S_L, which are then combined with S_R to obtain the final set of top-K elements (Line 21).

It is worth mentioning that only CMP (line 8) requires interaction between S and C; the remaining steps of the algorithm can be executed locally by each party without the need for interaction. The selection function requires $O(n)$ CMPs.

6. Secure Sampling

In this section, we provide a detailed explanation of our secure sampling protocol. It takes as input K secret-shared probabilities $(p_1, \ldots, p_K)$, where each probability has been scaled to an integer x_i by multiplying it by 2^L and dropping

Algorithm 3: Secure Sampling: Π_{Sample}

Input: S & C hold $\langle \mathbf{x} \rangle$, with $\mathbf{x} \in \mathbb{Z}_{2^l}^K$ being a vector of probabilities scaled by 2^L

Output: S & C get $\langle j \rangle$, with $j \in [1, K]$ and

$$Pr\left(j = i\right) = x_i / \sum_{j=1}^{K} x_j$$

1 C samples $v \xleftarrow{\$} \left[0, 2^L - 1\right]$ with $v \in \mathbb{Z}_{2^l}$
2 S & C init $\langle s_0 \rangle := 0$
3 **for** $i := 1$ **to** $K - 1$ **do**
4 | S & C (locally) compute $\langle s_i \rangle := \langle x_i \rangle + \langle s_{i-1} \rangle$
5 | S & C invoke $\langle b_i \rangle^1 \leftarrow \mathsf{F}_{\mathsf{CMP}}(\langle v \rangle, \langle s_i \rangle)$ ▷ $b = 1$ `if` $v \geq s_i$`;` $b = 0$ `otherwise`
6 **end**
7 S & C init $\langle b_0 \rangle^1 := 1$ and $\langle b_K \rangle^1 := 0$
8 **for** $i := 1$ **to** K **do**
9 | S & C (locally) compute $\langle b'_i \rangle^1 := \langle b_{i-1} \rangle^1 \oplus \langle b_i \rangle^1$ ▷ $b'_i = 1$ `only when` $s_{i-1} \leq v < s_i$
10 **end**
11 S & C compute $\langle j \rangle := \sum_{i=1}^{K} \mathsf{F}_{\mathsf{MUX}}(i, \langle b'_i \rangle^1)$

the fractional part. The output of the protocol is a secret-shared index j:

$$Pr\left(j = i\right) = x_i / \sum_{k=1}^{K} x_k.$$

We will explain how we map this index to a response word in Section 7.5.

Algorithm 3 provides a detailed description of the secure sampling protocol. It is based on the observation is that, for a random $p' \in [0, 1]$, the selected index j satisfies:

$$\sum_{k=1}^{j-1} p_k \leq p' < \sum_{k=1}^{j} p_k.$$

As $(p_1, \cdots, p_K)$ have been scaled by 2^L, p' should should also be scaled accordingly. To this end, we have C sample an integer v from $\left[0, 2^L - 1\right]$ (Line 1). S and C securely compare v with each $\sum_{k=1}^{i} x_k$, $\forall i \in [1, K]$ (Line 2-6), resulting in a secret-shared bit vector $\langle \mathbf{b} \rangle$ that satisfies:

$$b_i = 1 \ \forall \ 1 \leq i < j \text{ and } b_i = 0 \ \forall \ j \leq i \leq K.$$

Our next step is to build another secret-shared bit vector $\langle \mathbf{b}' \rangle$ that satisfies:

$$b'_i = 0 \ \forall \ i \neq j \text{ and } b'_j = 1.$$

This can be achieved by performing an XOR operation on every pair of adjacent bits in $\langle \mathbf{b} \rangle$ (Line 7-10). Then, the desired index is: $\langle j \rangle := \sum_{i=1}^{K} \mathsf{F}_{\mathsf{MUX}}(i, \langle b'_i \rangle^1)$ (Line 11).

We remark that it is acceptable for v to be soley sampled by C, because the final output index j remains unknown to C.

7. The CipherGPT Framework

Figure 2 shows the architecture and workflow of GPT. Roughly, it takes a sequence of words, encodes them into word embeddings, and passes them through multiple iterations[2] of a transformer decoder. Each iteration involves a self-attention layer and a feed-forward neural network. The output from the transformer decoder is fed into a vec2word layer, which generates the predicted response word.

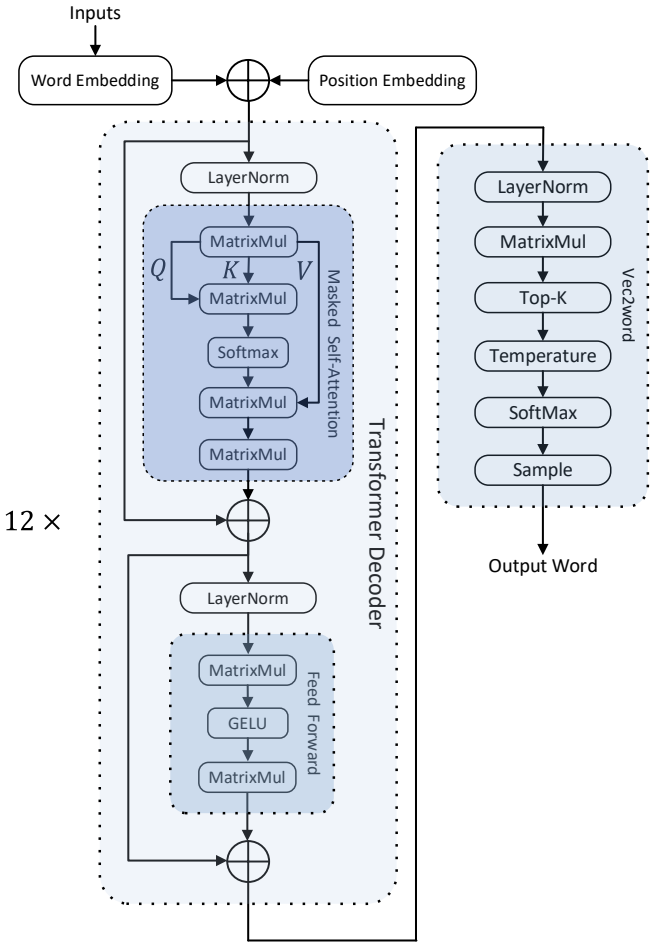


Figure 2: The architecture and workflow of GPT.

Next, we explain in detail how we securely compute this process.

7.1. Embedding

It first maps each input word to a numeric vector of length m, known as a *word embedding*, which is achieved by locating the corresponding row in an *embedding matrix*. Next, each word embedding is augmented by a *position embedding* that is determined by the position of the word within the input sequence. The position embeddings are predefined and added element-wise to the word embeddings.

2. Our benchmarked model involves 12 iterations.

We accomplish word embedding and position embedding altogether using additively homomorphic encryption (AHE):

1) S employs AHE to encrypt each row of the embedding matrix and transmits all the resulting ciphertexts to C. In practice, we encrypt the entire row by representing it as the polynomial coefficients of an RLWE ciphertext. Notice that the word embeddings are floating-point numbers; S scales them up to integers by left-shifting them by L bits and dropping the fractional parts.
2) C locates the corresponding ciphertexts based on its input words, adds a random number to each ciphertext: $E(w_1+r_1), \cdots, E(w_n+r_n)$; and returns them to S.
3) S decrypts the ciphertexts to obtain $w_1 + r_1, \cdots, w_n + r_n$; adds the position embeddings: $w_1 + r_1 + p_1, \cdots, w_n + r_n + p_n$.
4) Now, each embedding is secret-shared, with $\langle x_i \rangle_{\mathsf{C}} = -r_i$ and $\langle x_i \rangle_{\mathsf{S}} = w_i + r_i + p_i$.

We remark that step 1 only needs to be performed once and can be utilized indefinitely, unless there are changes to the embedding matrix.

7.2. Layer normalization

After input encoding, the n input words become a secret-shared matrix $\langle \mathbf{X} \rangle$ with $\mathbf{X} \in \mathbb{Z}_{2^l}^{n \times m}$. Then, layer normalization (LayerNorm) needs to be performed for each of its row $\mathbf{x} \in \mathbb{Z}_{2^l}^{m}$. Specifically, each element x_i in $\mathbf{x}$ is normalized as follows:

$$x_i := \frac{x_i - E[\mathbf{x}]}{\sqrt{Var[\mathbf{x}]+\epsilon}} \cdot \gamma + \beta,$$

where γ and β are learnable parameters and ϵ is a small value used to avoid division by zero.

To securely compute LayerNorm, S and C use LUT to approximate $\frac{1}{\sqrt{Var[\mathbf{x}]+\epsilon}}$. We refer to SIRNN [30] for more details. After the computation of LayerNorm, S and C need to run $\mathsf{F}_{\mathsf{Trunc}}$ to ensure that the scaling remains at L bits. For the sake of simplicity, we omit mentioning truncations in the remaining part of this section.

7.3. Masked Self-Attention

Self-attention is a mechanism that enables the computation of a sequence's representation by relating different positions within the sequence [35]. The first step in calculating self-attention is to create three matrices: a *query matrix* $\mathbf{Q}$, a *key matrix* $\mathbf{K}$ and a *value matrix* $\mathbf{V}$. This is accomplished by multiplying the normalized embeddings $\mathbf{X} \in \mathbb{Z}_{2^l}^{n \times m}$ by three matrices ($\mathbf{W}_Q \in \mathbb{Z}_{2^l}^{m \times m}$, $\mathbf{W}_K \in \mathbb{Z}_{2^l}^{m \times m}$, and $\mathbf{W}_V \in \mathbb{Z}_{2^l}^{m \times m}$) that were trained during the training process:

$$\begin{aligned} \langle \mathbf{Q} \rangle &:= \langle \mathbf{X} \rangle \langle \mathbf{W}_Q \rangle; \\ \langle \mathbf{K} \rangle &:= \langle \mathbf{X} \rangle \langle \mathbf{W}_K \rangle; \\ \langle \mathbf{V} \rangle &:= \langle \mathbf{X} \rangle \langle \mathbf{W}_V \rangle. \end{aligned}$$

As $\mathbf{W}_Q$, $\mathbf{W}_K$ and $\mathbf{W}_V$ are known beforehand, such MatrixMuls can be computed by our sVOLE-based solution described in Section 3.

Multi-headed attention. Each of $\langle \mathbf{Q} \rangle$, $\langle \mathbf{K} \rangle$, $\langle \mathbf{V} \rangle$ is then partitioned into M segments, known as *multi-head attention*, where M represents the number of attention heads[3]. Let $m' = \frac{m}{M}$, we have:

$$\begin{aligned} \langle \mathbf{q}_1 \rangle \,||\, \cdots \,||\, \langle \mathbf{q}_M \rangle &= \langle \mathbf{Q} \rangle, \text{ with each } \mathbf{q}_i \in \mathbb{Z}_{2^l}^{n \times m'}; \\ \langle \mathbf{k}_1 \rangle \,||\, \cdots \,||\, \langle \mathbf{k}_M \rangle &= \langle \mathbf{K} \rangle, \text{ with each } \mathbf{k}_i \in \mathbb{Z}_{2^l}^{n \times m'}; \\ \langle \mathbf{v}_1 \rangle \,||\, \cdots \,||\, \langle \mathbf{v}_M \rangle &= \langle \mathbf{V} \rangle, \text{ with each } \mathbf{v}_i \in \mathbb{Z}_{2^l}^{n \times m'}. \end{aligned}$$

A *score* matrix is calculated by taking the product of a query matrix and a key matrix:

$$\langle \mathbf{s}_i \rangle := \langle \mathbf{q}_i \rangle \left\langle \mathbf{k}_i^T \right\rangle \ \forall\ i \in [M].$$

Each score in $\mathbf{s}_i \in \mathbb{Z}_{2^l}^{n \times n}$ determines how much focus to place on other words when encoding the current word. In this case, where neither $\mathbf{q}_i$ nor $\mathbf{k}_i$ is known beforehand, our sVOLE-based MatrixMul cannot be applied. Instead, we employ the AHE-based MatrixMul proposed in [21].

Self-attention masking. After the multi-headed attention, *self-attention masking* is applied to zero-out the upper-triangle of each $\mathbf{s}_i$. As a result, every word to the left has a much higher attention score than words to the right, so the model in practice only focuses on previous words. This step can be done locally by S and C without any interaction.

Softmax. A softmax operation is applied to each row of each $\langle \mathbf{s}_i \rangle$, ensuring that the scores are normalized within each row, with all values being positive and summing up to 1. To securely compute softmax, we employ a multi-step LUT approach used in [21].

Output. In the final step of self-attention, the softmaxed scores are used to weight the values in the value matrix:

$$\langle \mathbf{z}_i \rangle := \langle \mathbf{s}_i \rangle \langle \mathbf{v}_i \rangle \ \forall\ i \in [M],$$

which is again accomplished by the AHE-based MatrixMul [21]. Then, all $\mathbf{z}$s are reassembled together:

$$\langle \mathbf{Z} \rangle := \langle \mathbf{z}_1 \rangle \,||\, \cdots \,||\, \langle \mathbf{z}_n \rangle.$$

The output of self-attention is:

$$\langle \mathbf{X} \rangle := \langle \mathbf{X} \rangle + \langle \mathbf{Z} \rangle.$$

7.4. Feed forward

The output of self-attention is subjected to a LayerNorm operation. The resulting normalized values are then fed into a feed-forward neural network, which consists of two fully-connected (FC) layers and one activation layer.

The first FC layer is computed as:

$$\langle \mathbf{X}_1 \rangle := \langle \mathbf{X} \rangle \langle \mathbf{W}_1 \rangle,$$

where $\mathbf{X} \in \mathbb{Z}_{2^l}^{n \times m}$, $\mathbf{W}_1 \in \mathbb{Z}_{2^l}^{m \times k}$ and $\mathbf{X}_1 \in \mathbb{Z}_{2^l}^{n \times k}$. Then, Π_{GELU} (cf. Section 4) is applied to each element of $\mathbf{X}_1$, resulting in $\mathbf{X}_1'$. The second FC layer is computed as:

3. In GPT-2, $M = 12$ by default.

$$\langle \mathbf{X}_2 \rangle := \langle \mathbf{X}_1' \rangle \langle \mathbf{W}_2 \rangle,$$

where $\mathbf{X}_1' \in \mathbb{Z}_{2^l}^{n \times k}$, $\mathbf{W}_2 \in \mathbb{Z}_{2^l}^{k \times m}$ and $\mathbf{X}_1 \in \mathbb{Z}_{2^l}^{n \times m}$.

Notice that $\mathbf{W}_1$ and $\mathbf{W}_2$ are known beforehand, hence our sVOLE-based MatrixMul (cf. Section 3) can be applied to the two FC layers.

The output will once again undergo multiple iterations of self-attention and feed-forward, with each iteration employing different weights while preserving the same structure.

7.5. Vec2word

After multiple iterations of self-attention and feed-forward, the resulting output is then passed through a vec2word layer to generate the predicted response word. The initial operation in vec2word involves a MatrixMul to produce a one-hot encoding for all possible words:

$$\langle \mathbf{y}_0 \rangle := \langle \mathbf{x} \rangle \langle \mathbf{W} \rangle,$$

where $\mathbf{W} \in \mathbb{Z}_{2^l}^{m \times k}$, $\mathbf{y}_0 \in \mathbb{Z}_{2^l}^{k}$, and $\mathbf{x} \in \mathbb{Z}_{2^l}^{m}$ is the last row of $\mathbf{X} \in \mathbb{Z}_{2^l}^{n \times m}$ (due to an inference-time optimization employed by GPT). This time, k represents the number of all possible words, which is quite large. Our sVOLE-based MatrixMul is not suitable here, hence we employ the AHE-based MatrixMul [21].

Top-K. To maintain a balance between diversity and high-probability words, the K largest values are selected from $\mathbf{y}_0$:

$$\langle \mathbf{y}_1 \rangle \leftarrow \Pi_{\mathsf{TopK}}(\langle \mathbf{y}_0 \rangle), \text{ with } \mathbf{y}_1 \in \mathbb{Z}_{2^l}^{K}.$$

This is accomplished by our proposed protocol described in Section 5.

Temperature. The *temperature* T determines the creativity and diversity of the text generated by GPT: a higher temperature (e.g., $T = 1.5$) produces more diverse and creative text, whereas a lower temperature (e.g., $T = 0.5$) produces more focused and deterministic text. It is a hyperparameter held by S and to be multiplied with each value in $\mathbf{y}_1$. This can be easily achieved by AHE:

1) C sends S its AHE-encrypted shares $E(\langle y_{1,1} \rangle_{\mathsf{C}}), \cdots, E(\langle y_{1,K} \rangle_{\mathsf{C}})$. In practice, we encrypt them altogether by representing them as the polynomial coefficients of an RLWE ciphertext.
2) S adds its shares to the ciphertexts: $E(\langle y_{1,1} \rangle_{\mathsf{C}} + \langle y_{1,1} \rangle_{\mathsf{S}}), \cdots, E(\langle y_{1,K} \rangle_{\mathsf{C}} + \langle y_{1,K} \rangle_{\mathsf{S}})$.
3) S multiplies all ciphertexts by T: $E(T \cdot y_{1,1}), \cdots, E(T \cdot y_{1,K})$.
4) S adds a random number r_1 to each ciphertext: $E(T \cdot y_{1,1} + r_1), \cdots, E(T \cdot y_{1,K} + r_K)$.
5) S returns the resulting ciphertexts to C.
6) S decrypts the ciphertexts, and now the temperatured values, represented by $\mathbf{y}_2$, are secret-shared:

$$\langle y_{2,i} \rangle_{\mathsf{C}} := T \cdot y_{1,i} + r_i \text{ and } \langle y_{2,i} \rangle_{\mathsf{S}} := -r_i, \quad \forall i \in [K].$$

Random sampling. A softmax operation is applied to $\mathbf{y}_2$ to obtain a probability vector denoted by $\mathbf{y}_3$, and the response word is then randomly sampled based on this probability vector. Such random sampling ensures that the generated output is both diverse and contextually relevant.

We employ the secure sampling protocol described in Section 6 to get an index:

$$\langle j \rangle \leftarrow \Pi_{\mathsf{Sample}}(\mathbf{y}_3).$$

Recall that, in Algorithm 3, the value v is sampled by C (Line 1 in Algorithm 3). Therefore, if C learns j, it could potentially gain some information about the input $\mathbf{x}$. On the other hand, revealing j to S does not disclose any information about $\mathbf{x}$ because v is unknown to S. To this end, we reveal j to S.

As Algorithm 2 does not hide which K elements (in the shuffled vector) were selected, S is able to map j to the corresponding index j' in the shuffled vector. Recall that the shuffling process roughly works as follows:

1) C generates a random permutation π_{C}; S and C jointly apply π_{C} to the input vector, obtaining the corresponding secret-shares.
2) S generates a random permutation π_{S}; S and C jointly apply π_{S} to the output of π_{C}, obtaining the corresponding secret-shares.

To this end, we have S compute $i' := \pi_{\mathsf{S}}^{-1}(j')$ and return i' to C. Notice that i' does not reveal any information since π_{S} is unknown to C. Then, we have C compute $i := \pi_{\mathsf{C}}^{-1}(i')$, which corresponds to the index in the word vector. Given that the word vector is publicly known, C can retrieve the final response word based on the index i.

8. Evaluation

In this section, we provide a full-fledged implementation of CipherGPT and systematically evaluate its performance.

8.1. Implementation

We fully implemented CipherGPT in C++ and set the security parameter as 128. We use the Microsoft SEAL homomorphic encryption library (version 4.0)[4] for AHE and use hexl[5] to accelerate HE operation with AXV-512 instruction. Specifically, we use the Brakerski-Fan-Vercauteren (BFV) [7], [15] scheme, with $N = 4\ 096$ and the default parameters in SEAL for 128-bit security.

- For secure GELU, we implemented LUT, Mult, Trunc, CMP and MUX by leveraging the corresponding open-sourced code in SIRNN[6].
- For sVOLE-based MatrixMul, we used the open-source code of Silver[7] for sVOLE. We re-implemented its reverse-VOLE part with AHE and incorporated the Half-tree [20] optimization to its implementation.

4. https://github.com/Microsoft/SEAL
5. https://github.com/intel/hexl
6. https://github.com/mpc-msri/EzPC/tree/master/SIRNN
7. https://github.com/osu-crypto/libOTe

- For TopK, since the secret-shared shuffle in [10] is not open-sourced, we implemented it by ourselves.
- For LayerNorm, we used the corresponding open-sourced code of SIRNN.
- For Softmax and AHE-based MatrixMul, since Iron [30] is not open-sourced, we reproduced them by leveraging the open-sourced code of SIRNN and Cheetah[8] respectively.

8.2. Experimental Setup

Following SIRNN [30] and Iron [21], we used a LAN network setting, where the bandwidth is 377 MBps and RTT is 0.8ms. All experiments were performed on AWS c5.9xlarge instances with Intel Xeon 8000 series CPUs at 3.6GHz, and they were conducted using a single thread. All results are the average values of 5 runs and the variances are very small.

We benchmark the GPT-2 model proposed by Radford [29], which consists of 117 million parameters, 12 transformer decoders, with an embedding size of 768. Following Cheetah [23] and CrypTFlow2 [31], we left-shift the floating point numbers for $L = 12$ bits and drop the fractional part. During the inference, we use $\mathsf{F}_{\mathsf{Trunc}}$ to make sure the largest value is smaller than $2^l - 1$ with $l = 37$.

8.3. Evaluation results

Evaluation of GELU. When evaluating our GELU protocol (i.e., Algorithm 1), we set $\alpha = 4$ and $s = 8$. Given that $L = 12$, the actual interval to be approximated is $\left[-4 \times 2^{12}, 4 \times 2^{12}\right]$. Specifically, we partition the interval $\left[-4 \times 2^{12}, 4 \times 2^{12}\right]$ into 2^8 small intervals and use a 256-piece spline to approximate the curve within $\left[-4 \times 2^{12}, 4 \times 2^{12}\right]$.

Table 4 shows the comparison between Iron and our solution for GELU. To compute GELU for each 37-bit element in a 3072-length vector, our protocol takes 694ms and 13.1MB of bandwidth. Compared with Iron [21], it achieves a 4.1× speedup in runtime and a 3.3× reduction in communication.

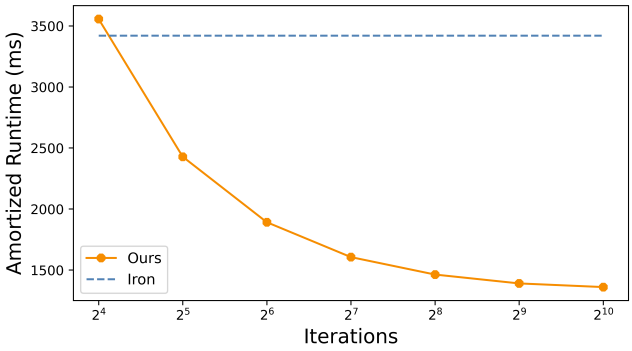
GELU($\mathbb{Z}_{2^{37}}^{3072}$)	Runtime (ms)	Comm. (MB)	Maximal ULP Err.	Average ULP Err.
Iron	694	44.3	9	2.4
Ours	166 4.2× ↓	13.1 3.4× ↓	1 9× ↓	0.22 10.9× ↓

TABLE 4: Evaluation of GELU (we use a 256-piece spline to approximate the curve within $\left[-4 \times 2^{12}, 4 \times 2^{12}\right]$).

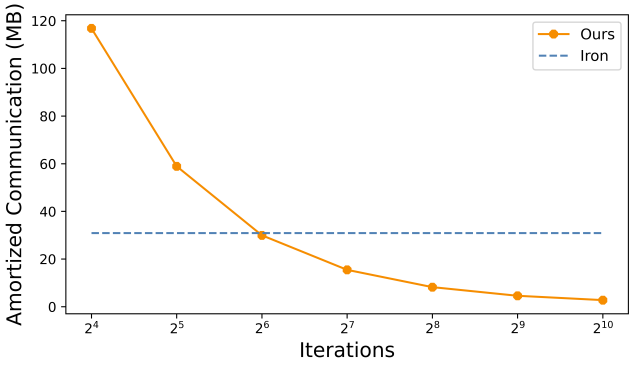
We evaluate the precision of our approximation by testing its *ULP error*, which is defined as the number of representable numbers between the exact real result y and the approximated result $\tilde{y}$ [18]. Since we have scaled the floating-point numbers into integers, the ULP error is exactly $|y - \tilde{y}|$. Following SIRNN [30], we use *exhaustive testing* to evaluate the ULP errors:

1) run the secure GELU protocols on all possible integers within $\left[-16 \times 2^{12}, 16 \times 2^{12}\right]$,
2) compare the ULP error between each output and the corresponding infinite precision real result, and
3) report both the maximal ULP error and the average ULP error.

The results (in Table 4) show that our solution introduces much smaller ULP errors compared with Iron. The multi-step process in Iron (flowing SIRNN) involves approximating exponentiation and reciprocation separately, introducing ULP errors at each step. These errors accumulate throughout the process, resulting in a larger overall error compared to our single-step approach.



(a) Amortized Runtime vs. Iterations.



(b) Amortized Communication vs. Iterations.

Figure 3: Evaluation of MatrixMul (we compute $\mathbb{Z}_{2^{37}}^{256\times768} \times \mathbb{Z}_{2^{37}}^{768\times768}$ for multiple iterations and measure the amortized cost).

Evaluation of MatrixMul. Recall that our sVOLE-based MatrixMul is suitable for the case where the sizes of the two matrices are unbalanced. Therefore, we measure the amortized cost of performing $\mathbb{Z}_{2^{37}}^{256\times768} \times \mathbb{Z}_{2^{37}}^{768\times768}$ for t iterations, where the $\mathbb{Z}_{2^{37}}^{768\times768}$ matrix remains constant across all iterations. While the size of t may not have an impact on other protocols, it is significant for our approach as we

8. https://github.com/Alibaba-Gemini-Lab/OpenCheetah

Layer	Operation	Output←Input	Method	Times	Runtime (ms)	Runtime %	Comm. (MB)	Comm. %
Embedding	Embedding	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{16}}^{256}$, $L=12$	§7.1	1	285	0.02%	8.38	0.01%
LayerNorm	LayerNorm	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}$, $L=12$	[30]	12	12682×12	10.47%	1009.98×12	12.74%
Self-attention	MatrixMul	$(\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}\times\mathbb{Z}_{2^{37}}^{768\times 768})\times 3$, $L=24$	§3	12	4358×12	3.60%	21.79×12	0.27%
	Trunc	$(\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768})\times 3$, $L=12$	[30]	12	4676×12	3.86%	361.76×12	4.56%
	Multi-head	$(\mathbb{Z}_{2^{37}}^{256\times 64}\times 12 \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768})\times 3$, $L=12$	plain	12	$(<1)\times 12$	$\approx 0\%$	0	0%
	MatrixMul	$(\mathbb{Z}_{2^{37}}^{256\times 256} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 64}\times\mathbb{Z}_{2^{37}}^{64\times 256})\times 12$, $L=24$	[21]	12	2854×12	2.36%	58.22×12	0.73%
	Trunc	$(\mathbb{Z}_{2^{37}}^{256\times 256} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 256})\times 12$, $L=12$	[30]	12	2868×12	2.37%	54.28×12	0.68%
	Masking	$(\mathbb{Z}_{2^{37}}^{256\times 256} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 256})\times 12$, $L=12$	plain	12	$(<1)\times 12$	$\approx 0\%$	0	0%
	Softmax	$(\mathbb{Z}_{2^{37}}^{256\times 256} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 256})\times 12$, $L=12$, (by row)	[21]	12	19954×12	16.47%	1175.84×12	14.83%
	MatrixMul	$(\mathbb{Z}_{2^{37}}^{256\times 64} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 256}\times\mathbb{Z}_{2^{37}}^{256\times 64})\times 12$, $L=24$	[21]	12	2817×12	2.33%	54.28×12	0.68%
	Trunc	$(\mathbb{Z}_{2^{37}}^{256\times 64} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 64})\times 12$, $L=12$	[30]	12	1573×12	1.30%	120.58×12	1.52%
	Reassemble	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow (\mathbb{Z}_{2^{37}}^{256\times 64}\times 12)$, $L=12$	plain	12	$(<1)\times 12$	$\approx 0\%$	0	0%
	MatrixMul	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}\times\mathbb{Z}_{2^{37}}^{768\times 768}$, $L=24$	§3	12	1463×12	1.21%	8.20×12	0.10%
	Trunc	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}$, $L=12$	[30]	12	1573×12	1.30%	120.58×12	1.52%
	Matrix Add	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}+\mathbb{Z}_{2^{37}}^{256\times 768}$, $L=12$	plain	12	$(<1)\times 12$	$\approx 0\%$	0	0%
LayerNorm	LayerNorm	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}$, $L=12$	[30]	12	12682×12	10.47%	1009.98×12	12.74%
Feed-forward	MatrixMul	$\mathbb{Z}_{2^{37}}^{256\times 3072} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}\times\mathbb{Z}_{2^{37}}^{768\times 3072}$, $L=24$	§3	12	5997×12	4.95%	28.5×12	0.36%
	Trunc	$\mathbb{Z}_{2^{37}}^{256\times 3072} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 3072}$, $L=12$	[30]	12	6224×12	5.14%	482.34×12	6.08%
	GELU	$\mathbb{Z}_{2^{37}}^{256\times 3072} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 3072}$, $L=12$	§4	12	32314×12	26.68%	3169.97×12	39.98%
	MatrixMul	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 3072}\times\mathbb{Z}_{2^{37}}^{3072\times 768}$, $L=24$	§3	12	5841×12	4.82%	32.42×12	0.41%
	Trunc	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}$, $L=12$	[30]	12	1520×12	1.25%	120.58×12	1.52%
	Matrix Add	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}+\mathbb{Z}_{2^{37}}^{256\times 768}$, $L=12$	plain	12	$(<1)\times 12$	$\approx 0\%$	0	0%
LayerNorm	LayerNorm	$\mathbb{Z}_{2^{37}}^{256\times 768} \leftarrow \mathbb{Z}_{2^{37}}^{256\times 768}$, $L=12$	[30]	1	12682	0.87%	1009.98	1.06%
Vec2Word	MatrixMul	$\mathbb{Z}_{2^{37}}^{50257} \leftarrow \mathbb{Z}_{2^{37}}^{768}\times\mathbb{Z}_{2^{37}}^{768\times 50257}$, $L=24$	[21]	1	1942	0.13%	11.09	0.01%
	Trunc	$\mathbb{Z}_{2^{37}}^{50257} \leftarrow \mathbb{Z}_{2^{37}}^{50257}$, $L=12$	[30]	1	367	0.03%	33.22	0.03%
	Shuffle	$\mathbb{Z}_{2^{37}}^{50257} \leftarrow \mathbb{Z}_{2^{37}}^{50257}$, $L=12$	[10]	1	4004	0.28%	51.3	0.05%
	TopK	$\mathbb{Z}_{2^{37}}^{100} \leftarrow \mathbb{Z}_{2^{37}}^{50257}$, $L=12$	§5	1	1277	0.09%	84.8	0.09%
	Temperature	$\mathbb{Z}_{2^{37}}^{100} \leftarrow \mathbb{Z}_{2^{37}}^{100}$, $L=24$	§7.4	1	18	$<0.01\%$	0.14	$<0.01\%$
	Trunc	$\mathbb{Z}_{2^{37}}^{100} \leftarrow \mathbb{Z}_{2^{37}}^{100}$, $L=12$	[30]	1	18	$<0.01\%$	0.14	$<0.01\%$
	Softmax	$\mathbb{Z}_{2^{37}}^{100} \leftarrow \mathbb{Z}_{2^{37}}^{100}$, $L=12$	[21]	1	257	0.02%	1.22	$<0.01\%$
	Sampling	$\mathbb{Z}_{2^{37}} \leftarrow \mathbb{Z}_{2^{37}}^{100}$, $L=12$	§6	1	7.843	$<0.01\%$	0.11	$<0.01\%$
Total					1 453 610		95 151.98	

TABLE 5: A comprehensive benchmark for CipherGPT in generating a single response word (amortized for the generation of 256 response words).

can preprocess all t iterations together. We acknowledge that this comparison may be considered unfair, but it accurately reflects the setting for GPT inference.

Figure 3 shows the comparison between Iron and our protocol (we did not differentiate between the preprocessing time and online time in this figure). Considering that ChatGPT often generates several hundred words in a single response, $t=256$ would be a reasonable number of iterations. The amortized runtime for our protocol is 1 606ms, $2.1\times$ speedup over Iron; the amortized communication for our protocol is 8.2MB, $3.8\times$ reduction over Iron. In scenarios where the number of response words increases to 1024, which is also quite common, our protocol demonstrates even greater performance advantages. Specifically, our protocol outperforms Iron by $2.5\times$ in runtime and $11.2\times$ in communication.

Evaluation of TopK. We benchmark our TopK protocol (cf. Algorithm 2) for selecting 100 elements from a vector of $\mathbb{Z}_{2^{37}}^{50304}$. It takes 3 281ms and 136.1MB bandwidth. Compared with the commonly used Bitonic sorting network [22], we achieve $8.8\times$ speedup in runtime and $14.8\times$ reduction in communication.

Evaluation of CipherGPT. We run CipherGPT to generate a sentence that consists of 256 response words. Table 5 lists the amortized runtime and communication for each individual operation, along with their corresponding proportions.

9. Related Works

Secure inference can be achieved via generic secure two-party (2PC) computation [39], [19] or fully homomorphic encryption (FHE) [16]. However, such solutions would exhibit high communication and computational cost. Therefore, it is necessary to develop customized protocols

for secure inference. Efforts in this field can be traced back to the early 2010s, with many of the early works primarily focusing on simpler machine learning algorithms such as SVMs and linear regression.

CryptoNets [17] is recognized as the initial endeavor in secure neural network inference. It relies solely on FHE, which limits its applicability to neural networks with a small number of layers. Additionally, it can only support linear operations and low-degree polynomials. MiniONN [25] is the first work that customizes 2PC protocols for secure neural network inference. It proposes a spline-based approximation for non-linear operations, which inspires our solution for secure GELU.

GAZELLE [24] reduces the cost the linear layers by mapping them to SIMD-based matrix-vector multiplication and convolution routines. Cheetah [23] substitutes SIMD with coefficient packing to eliminate the expensive rotations. Iron [21] further reduces the communication complexity of Cheetah. In terms of activations, CrypTFlow2 [31] proposes efficient protocols for secure comparison and division. SIRNN [30] provides crypto-friendly approximations to math functions such as exponential, sigmoid, tanh and reciprocal square root; as well as the corresponding 2PC implementations.

Another research direction for improving the performance of secure inference is to change the model structure to more crypto-friendly ones. For example, DeepSecure [34], XONN [32] and Quotient [1] are specifically designed for binarized neural networks [12]. DeepSecure additionally prunes the model to reduce the number of activations. Delphi [26] provides a planner that leverages neural architecture search to automatically generate neural network architecture configurations that navigate the performance-accuracy trade-offs. However, all such solutions require retraining the model, which is less desirable to machine learning practitioners.

Some solutions [27], [37] leverage GPU parallelism to accelerate the online phase, but they cannot do anything about preprocessing as cryptographic operations dominate the preprocessing phase in such protocols. The most efficient GPU-based solution, i.e. GForce [27], requires 14-15 minutes in total to perform one inference for VGG-16 (trained on CIFAR-10 and CIFAR-100).

The discussion so far focuses on two-party protocols, as we believe secure inference naturally aligns with this setting. However, several other works [33], [36], [3] have instead targeted the three-party setting, where the model is secret-shared between two non-colluding servers and the client interacts with these servers to obtain the prediction. The three-party protocols are generally more efficient than two-party ones, but the assumption of non-colluding servers is often considered to be unrealistic in practice.

10. Conclusion

In response to the privacy concerns raised by ChatGPT, we develop CipherGPT, the first framework for secure GPT inference. It encompasses a series of innovative protocols, including a secure matrix multiplication that is customized for GPT inference, a novel protocol for securely computing GELU, and the first protocol for top-K sampling. We provide a comprehensive benchmark for CipherGPT, which can serve as a reference for future research in this area.

References

[1] 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*, CCS '19, page 1231–1247, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 610–629, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Assi Barak, Daniel Escudero, Anders P. K. Dalskov, and Marcel Keller. Secure evaluation of quantized neural networks. *IACR Cryptol. ePrint Arch.*, page 131, 2019.

[4] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector ole. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 896–912, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 291–308. ACM, 2019.

[6] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 489–518. Springer, 2019.

[7] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. *IACR Cryptol. ePrint Arch.*, page 78, 2012.

[8] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[9] A. Brüggemann, R. Hundt, T. Schneider, A. Suresh, and H. Yalame. Flute: Fast and secure lookup table evaluations. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 515–533, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[10] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 342–372, Cham, 2020. Springer International Publishing.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *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*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3123–3131, 2015.

[13] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part III*, volume 12827 of *Lecture Notes in Computer Science*, pages 502–534. Springer, 2021.

[14] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[15] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[16] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

[17] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 201–210. JMLR.org, 2016.

[18] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mar 1991.

[19] Oded Goldreich, Silvio Micali, and Avi Wigderson. *How to Play Any Mental Game, or a Completeness Theorem for Protocols with Honest Majority*, page 307–328. Association for Computing Machinery, New York, NY, USA, 2019.

[20] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. Half-tree: Halving the cost of tree expansion in cot and dpf. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 330–362, Cham, 2023. Springer Nature Switzerland.

[21] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. In *NeurIPS*, 2022.

[22] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[23] 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, Boston, MA, August 2022. USENIX Association.

[24] 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, Baltimore, MD, August 2018. USENIX Association.

[25] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 619–631. ACM, 2017.

[26] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2505–2522. USENIX Association, 2020.

[27] Lucien K. L. Ng and Sherman S. M. Chow. GForce: GPU-Friendly oblivious and rapid neural network inference. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2147–2164. USENIX Association, August 2021.

[28] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[29] 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.

[30] 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 *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1003–1020. IEEE, 2021.

[31] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 325–342, New York, NY, USA, 2020. Association for Computing Machinery.

[32] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. Xonn: Xnor-based oblivious deep neural network inference. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1501–1518, USA, 2019. USENIX Association.

[33] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 707–721. ACM, 2018.

[34] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[36] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019(3):26–49, 2019.

[37] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU platform for secure computation. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 827–844. USENIX Association, 2022.

[38] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1607–1626. ACM, 2020.

[39] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986.