Efficient Privacy-Preserving LSTM Inference on Encrypted Sequential Data

Qiang He[a,b,c], Jingwei Chen[a,b,c], Wenyuan Wu[a,b,c], Yong Feng[a,b,c]

[a] *Chongqing Institute of Green and Intelligent Technology, CAS, 400714, Chongqing, China*
[b] *Chongqing School, University of Chinese Academy of Sciences, 400714, Chongqing, China*
[c] *Chongqing Key Laboratory of Secure Computing for Biology, 400714, Chongqing, China*

Abstract

Recent advances in fully homomorphic encryption (FHE) have enabled privacy-preserving machine learning directly over encrypted data. As a representative recurrent architecture, the long short-term memory (LSTM) network is widely used for modeling sequential dependencies, yet existing FHE-based LSTM inference schemes still suffer from high latency and limited scalability. In this paper, we present an efficient privacy-preserving LSTM inference protocol on encrypted sequential data based on FHE. We insert a lightweight normalization module before nonlinear activations to bound the hidden states, thereby enabling accurate low-degree polynomial approximations of the Sigmoid, Tanh, and inverse-square-root functions via a hybrid Remez and least-squares strategy. We implement the proposed protocol using the Lattigo library, incorporating ciphertext packing optimization, rotation minimization, and SIMD-based parallelization. Experiments show that on standard text classification benchmarks, our encrypted LSTM achieves competitive accuracy compared with plaintext models and consistently outperforms the state-of-the-art FHE-based method, achieving up to a 4.7x speedup.

1. Introduction

With the advances in artificial intelligence (AI) and cloud computing, Machine Learning as a Service (MLaaS) has become a widely adopted paradigm. It enables users to utilize powerful computational resources and sophisticated pre-trained models hosted on remote servers, thereby reducing the demand for hardware and expertise. Despite these advantages, outsourcing data and computations to third-party servers inevitably raises serious privacy concerns [22]. Sensitive user data—such as medical records, financial information, and personal images—must be transmitted to external servers for model inference, posing potential risks of data leakage or unauthorized access. Meanwhile, model providers may also be reluctant to expose their proprietary models due to intellectual property and security considerations.

To address these challenges, privacy-preserving machine learning (PPML) has become an active research area, aiming to enable collaborative computation between users and servers while preserving privacy. Various cryptographic techniques have been explored in PPML, including secure multi-party computation (MPC) [51] and homomorphic encryption (HE) [18]. Among them, lattice-based HE stands out as a promising approach because it allows arithmetic operations to be performed directly on encrypted data, thereby ensuring end-to-end confidentiality. Recent progress in approximate homomorphic encryption schemes, such as the Cheon-Kim-Kim-Song (CKKS) scheme [11], has significantly improved the efficiency and practicality of encrypted machine learning inference, paving the way toward secure and privacy-preserving intelligent systems.

Recurrent neural networks (RNNs), particularly Long Short-Term Memory (LSTM) networks [23], play a crucial role in processing sequential data, such as natural language, speech, and financial time series, which often contain sensitive information. Recently, several studies have explored the use of fully homomorphic encryption (FHE) with recurrent architectures to protect data privacy during model inference. Most of these works [7, 40, 48, 49], however, focus on simple RNNs or Gated Recurrent Units (GRUs) [13], whose relatively lightweight structures are easier to adapt to the computational constraints of HE. In contrast, LSTM networks introduce multiple gates: input, forget, output, and cell state update, which involve more nonlinear operations and deeper computational dependencies. These characteristics make FHE-based LSTM inference significantly more challenging in both ciphertext-depth management and computational efficiency.

Only a few works [45, 57] have attempted to implement encrypted LSTM inference entirely under FHE. Trama *et al.* [45] developed TFHE-based building blocks for implementing LSTM, in particular realizing look-up tables (LUTs) for the Sigmoid and Tanh activation functions via functional bootstrapping. However, they did not provide

*Citation: Qiang He, Jingwei Chen, Wenyuan Wu, and Yong Feng. Efficient privacy-preserving LSTM inference on encrypted sequential data. Neurocomputing, 701:134508, 2026. Doi: 10.1016/j.neucom.2026.134508

Email addresses: heqiang@cigit.ac.cn (Qiang He), chenjingwei@cigit.ac.cn (Jingwei Chen), wenyuanwu@cigit.ac.cn (Wenyuan Wu), yongfeng@cigit.ac.cn (Yong Feng)

an end-to-end implementation of LSTM inference over encrypted data. PrivLSTM [57] is an important step forward, demonstrating the feasibility of CKKS-based LSTM inference. Nevertheless, it has several limitations in practice. First, the supported model size is limited to relatively modest configurations (e.g., the reported hidden dimension is at most 50), which falls short of typical text-classification deployments. Second, the integration-based packing strategy couples the four gates into a single ciphertext, reducing the effective batch size and requiring additional rotations and plain-ciphertext multiplications to extract gate-wise components before applying different nonlinear activations. Third, all nonlinear functions are approximated using a single family of minimax polynomials, which either increases the multiplicative depth or degrades end-to-end accuracy. Finally, bootstrapping is performed conservatively, leading to long inference latency even for moderate sequence lengths. These limitations indicate that there is still room for further improvement in both scalability and efficiency.

1.1. Contribution

In this work, we propose an efficient LSTM inference protocol, named PPLSTM, based on the FHE scheme CKKS. PPLSTM preserves the expressive power of standard LSTM models while integrating several algorithmic and implementation-level optimizations that jointly improve accuracy, scalability, and runtime performance. Our main contributions are summarized as follows:

- We design and implement an FHE-based LSTM inference protocol for classification under a standard client-server setting. Our protocol supports LSTM models of realistic scale (e.g., embedding dimension 100 and hidden dimension 128). It is evaluated on four widely used text-classification datasets, allowing a direct comparison with existing privacy-preserving approaches, including PrivLSTM [57]. Our implementation is available on GitHub. [1]

- We introduce an RMSNorm-based LSTM variant tailored for encrypted inference. The incorporation of RMSNorm stabilizes the input ranges of the Sigmoid and Tanh activations, making the overall protocol more HE-friendly. We further adopt a hybrid approximation strategy, using Remez polynomials for Sigmoid/Tanh and least-squares polynomials for the inverse square root, which keeps the multiplicative depth low while preserving the end-to-end accuracy of the plaintext model.

- We develop a throughput-aware linear-algebra backend for encrypted LSTM. This includes a batching and packing scheme aligned with the LSTM gate structure, diagonal encoding, and a Baby-Step Giant-Step (BSGS) implementation of plaintext–ciphertext matrix multiplication (PCMM) that fully exploits single instruction multiple data (SIMD) parallelism supported by CKKS. Compared with the integration strategy of PrivLSTM, our design increases the effective batch size and reduces the amortized number of rotations per input, thereby significantly improving per-sample latency.

- We carefully analyze the multiplicative depth of the encrypted LSTM and optimize the placement and frequency of bootstrapping operations to obtain an efficient CKKS parameter configuration. We provide a complete implementation of PPLSTM and report its practical end-to-end performance. On real-world datasets, our protocol consistently achieves higher classification accuracy and up to a 4.7x speedup over PrivLSTM under comparable security and model settings.

1.2. Related Work

FHE has enabled practical progress in PPML, with many representative systems targeting convolutional and attention-based models. Prior work has demonstrated efficient encrypted CNN inference through optimized packing and polynomial activation approximations [5, 29, 27, 32]. Similar techniques have recently been extended to FHE-based Transformer inference architectures via attention-specific optimizations [53, 56, 36, 54, 50, 17]; we refer the readers to [9, 1] and references therein for more details on this topic. These works commonly combine encrypted matrix multiplication, ciphertext packing, and polynomial approximations to support Transformer layers under homomorphic encryption. Nevertheless, the LSTM setting studied in this paper has a different computational structure. As a result, the depth accumulation, packing layout, rotation-reuse opportunities, and bootstrapping schedule differ substantially from those in CNN and Transformer protocols. Although recent FHE-based CNN and Transformer protocols have achieved impressive progress, LSTMs remain attractive for lightweight sequential tasks, short-text classification, and resource-constrained inference. Our work, therefore, focuses on a complementary setting, namely, architecture-specific optimization for encrypted LSTM inference, where PrivLSTM [57] remains the closest baseline and leaves substantial room for improving both latency and accuracy.

In fact, research on encrypted recurrent networks remains comparatively limited. The recurrent dependency chain and repeated nonlinearities introduce depth and noise-growth challenges that are less pronounced in feed-forward architectures. As a result, existing studies on privacy-preserving RNNs are relatively limited. We now briefly summarize recent progress closely related to this work in Table 1 that evaluates RNN inference under FHE,

[1] `https://github.com/sana2333/PPLSTM`.

including schemes based on simple RNNs, GRUs, and LSTMs.

Table 1: Comparison of HE-based RNN inference protocols.

Work	Primitive	RNN type	NI	BTS	Norm	LSS
[7]	CKKS	Simple RNN	●	○	○	○
[40]	CKKS	Simple RNN	●	○	○	○
[16]	CKKS+GC	GRU	○	○	○	●
[25]	CKKS	GRU	●	●	○	●
[48]	CKKS	GRU	●	○	●	○
[45]	TFHE	LSTM	●	●	○	○
[49]	CKKS	GRU	●	○	●	○
[57]	CKKS	LSTM	●	●	○	●
Ours	CKKS	LSTM	●	●	●	●

NI: Non-interactive. BTS: Bootstrapping. Norm: Normalization. LSS: Long-sequence supported. ● indicates the feature is adopted, while ○ indicates it is not used.

CryptoRNN[7] explored approximate activation function substitutions and ciphertext refresh strategies to stabilize inference under HE constraints. Podschwadt and Takabi proposed [40] a non-interactive encrypted prediction framework for sequence modeling. Their approach reduced communication costs but suffered from increased ciphertext noise due to repeated multiplications in recurrent operations. These studies established the feasibility of HE-based recurrent computation.

GRU is a simplified variant of the LSTM that merges the forget and input gates, thereby reducing computational depth and making it more suitable for encrypted inference. CryptoGRU [16] introduced a hybrid cryptographic framework (FHE and Garble Circuit (GC)) specifically tailored for GRU, utilizing GC-friendly ReLU activations with substantially lower circuit evaluation cost than the commonly used Tanh function and small-bitwidth quantization to achieve a substantial speedup in secure inference compared to previous privacy-preserving neural network implementations. This design significantly reduces the multiplicative depth of homomorphic computation, thereby avoiding bootstrapping. However, this comes at the cost of additional communication overhead and reliance on interactive protocols. Building upon this line of research, Jang et al.[25] later proposed MatHEGRU, a privacy-preserving deep sequential model built upon an extended CKKS-based matrix homomorphic encryption scheme (MatHEAAN), which enables efficient matrix representations, homomorphic sequence operations, and improved noise control for encrypted GRU inference. Additionally, Wang and Ikeda [48] propose a parallelized GRU architecture that increases throughput by exploiting ciphertext-level parallelism. Their approach limits circuit depth through structural simplifications and parallel evaluation, enabling efficient encrypted inference without bootstrapping. Nevertheless, the achievable sequence length is constrained by the remaining noise budget, and memory consumption increases due to parallel ciphertext processing.Building upon these ideas, Wang and Ikeda [49] further introduce a bootstrapping-free GRU model for text classification by jointly optimizing quantization, parameter scaling, and circuit depth. While this design achieves significantly reduced latency compared to bootstrapping-based approaches, it typically involves trade-offs in model accuracy and restricts the supported sequence length to relatively short inputs.

Compared with GRU, LSTM has a more complex gating structure (input, forget, output, and cell states), which significantly increases multiplicative depth. As a result, research on encrypted LSTM inference remains limited. Some works rely on hybrid cryptographic frameworks. For example, Liu et al. [8] utilized HE and MPC-enhanced LSTM for biomedical signal analysis, ensuring data confidentiality while maintaining inference accuracy. However, the hybrid scheme introduces communication overhead and partially compromises the non-interactive advantage of pure FHE systems. In the domain of pure FHE, while Trama et al. [45] explored using TFHE [12] and LUTs to evaluate activations, most homomorphic learning tasks prioritize the CKKS scheme for its vector-packing efficiency. Unlike before, PrivLSTM [57] is a work implemented solely using FHE, which implements all LSTM operations under the CKKS scheme without relying on MPC or inter-party communication, using polynomial approximations for nonlinearities and ciphertext packing to control multiplicative depth and noise growth. Although our overall architecture is similar to PrivLSTM [57], PPLSTM integrates the proposed optimizations in normalization, polynomial approximation, packing, and bootstrapping. As summarized in Table 2, when evaluating an LSTM with a hidden size of 64 using 8 threads, the CKKS-based schemes, ours and PrivLSTM [57], substantially outperform the TFHE-based scheme [45]. This performance gap primarily stems from CKKS's ability to pack multiple values into a single ciphertext and to perform vectorized operations at low amortized cost, thereby enabling much faster batched computation. Moreover, our work not only achieves higher accuracy than PrivLSTM [57] but also provides up to a 4.7x speedup.

2. Preliminaries

In this section, we first review the basic pipeline of LSTM inference, then summarize the core functionalities supported by the CKKS fully homomorphic encryption scheme, and finally briefly describe the public datasets used in this work.

2.1. The LSTM Unit

Recurrent neural networks (RNNs) are a standard modeling tool for sequential data because they can capture temporal dependencies across time steps. However, conventional RNNs suffer from vanishing and exploding gradients, which limit their ability to learn long-term dependencies. To address this issue, Hochreiter and Schmidhuber proposed the Long Short-Term Memory (LSTM)

Table 2: Comparison with Non-interactive HE-based LSTM inference schemes.

Work	FHE	Normalization	Activation	Packing	Bootstrapping	Accuracy	Time (s)
[45]	TFHE	–	LUTs	–	Per-gate	–	6.00
[57]	CKKS	–	Remez	Integrated	Baseline	85.06%	0.31
Ours	CKKS	RMSNorm (Section 3)	Remez+LS (Section 4)	Optimized (Section 5)	Placement-optimized (Section 7)	87.11%	0.09

The protocol in [45] is not end-to-end. Its running time was estimated based on the number of LUTs and the reported per-LUT time. – indicates it is not used.

[23], which introduces a memory cell and gating mechanisms—namely, the input, forget, and output gates—to control the information flow. In what follows, we briefly recall the standard LSTM formulation used throughout the paper. As shown in Fig. 1, at each time step t, given an input vector $\boldsymbol{x}_t$, the previous hidden state $\boldsymbol{h}_{t-1}$, and the previous cell state $\boldsymbol{c}_{t-1}$, the standard LSTM computes

$$\begin{aligned}
\boldsymbol{i}_t &= \mathrm{Sigmoid}(\boldsymbol{W}_{ii}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i),\\
\boldsymbol{f}_t &= \mathrm{Sigmoid}(\boldsymbol{W}_{if}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f),\\
\boldsymbol{g}_t &= \mathrm{Tanh}(\boldsymbol{W}_{ig}\boldsymbol{x}_t + \boldsymbol{W}_{hg}\boldsymbol{h}_{t-1} + \boldsymbol{b}_g),\\
\boldsymbol{o}_t &= \mathrm{Sigmoid}(\boldsymbol{W}_{io}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o),\\
\boldsymbol{c}_t &= \boldsymbol{f}_t \otimes \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \otimes \boldsymbol{g}_t,\\
\boldsymbol{h}_t &= \boldsymbol{o}_t \otimes \mathrm{Tanh}(\boldsymbol{c}_t),
\end{aligned} \tag{1}$$

where $\boldsymbol{i}_t, \boldsymbol{f}_t, \boldsymbol{o}_t$ represent the input, forget, and output gates, respectively, and $\boldsymbol{g}_t$ denotes the candidate cell state. The function Sigmoid$(\cdot)$ denotes the sigmoid activation, Tanh$(\cdot)$ is the hyperbolic tangent, and $\otimes$ denotes element-wise multiplication. Through its gating mechanism, the LSTM can selectively retain or discard information across time steps, thereby capturing both short-term and long-term dependencies in the sequence.

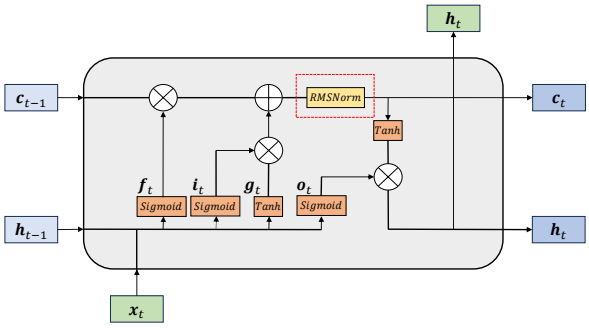

Figure 1: Operations in an LSTM unit.

Note that the standard LSTM does not include the Root Mean Square Normalization (RMSNorm) module (see Eq. (2) for its definition), which is highlighted by the red dashed box in Fig. 1. When this module is included, we obtain the variant referred to as *LSTM with RMSNorm*. A detailed analysis of how RMSNorm affects the LSTM is provided in Section 3.

2.2. The CKKS Scheme

Among existing fully homomorphic encryption schemes, we adopt the Cheon–Kim–Kim–Song (CKKS) scheme [11], which supports approximate arithmetic over packed complex vectors and is therefore well suited for encrypted machine learning. We assume the standard key generation, encryption, and decryption procedures as in [11], and only briefly recall the homomorphic operations used in this work.

- $\mathsf{Add}(\mathsf{ct}_1, \mathsf{ct}_2)$: Given ciphertexts ct_1 and ct_2 encrypting two vectors $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$, respectively, this operation outputs a ciphertext encrypting $\boldsymbol{m}_1 + \boldsymbol{m}_2$.
- $\mathsf{CMul}(\mathsf{ct}, \boldsymbol{p})$: Given a ciphertext ct encrypting a vector $\boldsymbol{m}$ and a vector $\boldsymbol{p}$, this operation outputs a ciphertext encrypting $\boldsymbol{p} \otimes \boldsymbol{m}$.
- $\mathsf{Mul}(\mathsf{ct}_1, \mathsf{ct}_2)$: Given ciphertexts ct_1 and ct_2 encrypting $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$, respectively, this operation outputs a ciphertext encrypting $m_1 \otimes m_2$.
- $\mathsf{Rot}_k(\mathsf{ct}.\boldsymbol{x})$: Rotates the packed slots of a ciphertext ct cyclically by k positions towards left. If ct encrypts $(x_0, x_1, \ldots, x_{n-1})$, then $\mathsf{Rot}_k(\mathsf{ct}.\boldsymbol{x})$ encrypts $(x_k, x_{k+1}, \ldots, x_{n-1}, x_0, \ldots, x_{k-1})$.
- $\mathsf{Bts}(\mathsf{ct})$: This is the well-known bootstrapping, which refreshes a ciphertext whose noise has grown too large by homomorphically evaluating the decryption circuit, and enables further homomorphic multiplications, effectively turning leveled CKKS into an FHE scheme [10].

Among these functionalities, Bts is by far the most time-consuming operation, while ciphertext-ciphertext multiplication (Mul) and ciphertext rotation (Rot) also incur significant overhead. CKKS is semantically secure under the RLWE assumption [34].

2.3. Datasets

We evaluate the standard LSTM, the LSTM with RMSNorm introduced in Section 3, and our optimized privacy-preserving LSTM inference protocol in Section 7 on four widely used text classification benchmarks in Table 3.

Table 3: Summary of benchmark datasets.

Dataset	Task	Total Samples	Classes
IMDB[35]	Sentiment	50K	2
YELP[55]	Review	650K	5
AGNEWS[55]	Topic	127.6K	4
DBPedia[55]	Document	630K	14

3. LSTM with RMSNorm

In this section, we introduce a variant of the LSTM architecture, referred to as *LSTM with RMSNorm*, to improve the stability of LSTM inference over encrypted data.

3.1. Motivation

In HE-based computation, non-polynomial functions (e.g., Sigmoid and Tanh) cannot be directly evaluated on ciphertexts due to the lack of native support for exponentiation, division, and other non-polynomial operations. Consequently, existing homomorphic-encryption-based PPML systems typically replace such nonlinearities with polynomial approximations. However, a wider input range requires a higher-degree polynomial to attain the same approximation accuracy, which in turn increases the multiplicative depth of the homomorphic circuit and, consequently, the number of required bootstrapping operations.

For example, in the case of $\mathrm{Tanh}(x)$, achieving a maximum absolute approximation error below 10^{-1} via the Remez algorithm [42] requires increasing the polynomial degree from 7 to 32 as the target domain expands from $[-5, 5]$ to $[-25, 25]$. When evaluating polynomials, a degree-7 polynomial can be computed within a multiplicative depth of 3, whereas a degree-32 polynomial requires depth 6. Such higher-degree polynomials lead to deeper multiplicative circuits and substantially higher computational costs.

In Table 4, we report the empirical input ranges of the activation functions in a standard LSTM with an embedding dimension of 100 and a hidden dimension of 128, evaluated on the IMDB, YELP, AGNEWS, and DBPedia datasets introduced in Section 2.3. The "Sigmoid" row corresponds to the affine transformations that produce the gate activations $\boldsymbol{i}_t$, $\boldsymbol{f}_t$, and $\boldsymbol{o}_t$. The "Tanh_g" row corresponds to the input of the Tanh_g function used to compute the candidate cell state $\boldsymbol{g}_t$, and the "Tanh_c" row corresponds to the Tanh_c operation applied when generating the hidden state $\boldsymbol{h}_t$; see the last equation in Eq. (1).

Table 4: Domain of the activation functions in the standard LSTM.

	IMDB	YELP	AGNEWS	DBPedia
Sigmoid	$-10.3 \sim 11.5$	$-10.5 \sim 12.6$	$-14.0 \sim 16.6$	$-10.2 \sim 11.8$
Tanh_g	$-5.3 \sim 5.1$	$-5.1 \sim 5.2$	$-8.9 \sim 9.3$	$-4.7 \sim 5.5$
Tanh_c	$-220.4 \sim 231.9$	$-88.1 \sim 227.9$	$-75.7 \sim 43.2$	$-109.2 \sim 49.8$

We observe that the input to the Tanh_c activation in the standard LSTM exhibits a considerable dynamic range, reaching approximately $[-220.40, 231.89]$ on IMDB and similarly wide intervals on the other datasets. Such expansive intervals make polynomial approximation highly challenging and numerically unstable. In contrast, the inputs to the sigmoid and Tanh_g functions remain within much smaller intervals—around $[-14.01, 16.61]$ for sigmoid and $[-8.93, 9.34]$ for Tanh_g at worst—which are far more suitable for accurate and efficient polynomial approximation. These observations motivate the introduction of a normalization step to constrain the input range before the Tanh_c activation.

3.2. Root Mean Square Normalization

Normalization is a standard technique for stabilizing neural network activations and improving convergence. Common approaches include Batch Normalization (BatchNorm) [24], Layer Normalization (LayerNorm) [4], and Root Mean Square Normalization (RMSNorm) [52].

The application of BatchNorm to LSTMs is constrained by its dependence on the batch size and the number of time steps. Crucially, BatchNorm requires reusing the statistics collected for each time step during training for inference. This constraint prevents the model from naturally adapting to sequences that exceed the training horizon, thereby limiting its practical deployment. In fact, Cooijmans et al. [14] apply BatchNorm to all gate computations and before the Tanh_c activation. However, in the context of HE, applying BatchNorm throughout the computation will significantly increase the multiplicative depth and computational overhead.

LayerNorm normalizes each input vector across its feature dimensions: $\mathrm{LayerNorm}(\boldsymbol{x}) = \frac{\boldsymbol{x}-\mu}{\sqrt{\sigma^2+\epsilon}} \otimes \boldsymbol{\gamma} + \boldsymbol{\beta}$, $\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$, $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$, where m is the feature dimension, ϵ is a small constant for numerical stability, and $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable parameters. However, computing both the mean μ and the variance σ^2 under CKKS encryption requires multiple ciphertext additions, subtractions, and multiplications, which significantly increase noise growth and multiplicative depth.

To mitigate these costs, we adopt RMSNorm, which removes the mean subtraction and replaces the variance with a simpler root mean square term:

$$\mathrm{RMSNorm}(\boldsymbol{x}) = \frac{\boldsymbol{x} \otimes \boldsymbol{\gamma}}{\sqrt{\frac{1}{m}\sum_{i=1}^{m} x_i^2 + \epsilon}}, \tag{2}$$

where $\epsilon > 0$ is a small constant for numerical stability and $\boldsymbol{\gamma} \in \mathbb{R}^m$ is a learnable scaling parameter.

The following proposition implies that even if the pre-normalization activations have a large numerical range, their post-RMSNorm values are confined to a worst-case interval depending only on the learned scaling parameter $\|\boldsymbol{\gamma}\|_\infty$ and the hidden dimension m.

Proposition 1 (Infinity-norm range control of RMSNorm)**.** *Let* $\boldsymbol{y} = \mathrm{RMSNorm}(\boldsymbol{x})$ *for some* $\boldsymbol{x} \in \mathbb{R}^m$ *with* $\epsilon > 0$*. Then* $\|\boldsymbol{y}\|_\infty \le \sqrt{m}\|\boldsymbol{\gamma}\|_\infty$.

Proof. By the definition of RMSNorm,

$$|y_i| = \frac{|x_i\gamma_i|}{\sqrt{\frac{1}{m}\|\boldsymbol{x}\|_2^2 + \epsilon}} \leq \frac{|x_i| \cdot \|\boldsymbol{\gamma}\|_\infty}{\sqrt{\frac{1}{m}\|\boldsymbol{x}\|_2^2 + \epsilon}}.$$

Taking the maximum over i gives

$$\|\boldsymbol{y}\|_\infty \leq \frac{\|\boldsymbol{x}\|_\infty \cdot \|\boldsymbol{\gamma}\|_\infty}{\sqrt{\frac{1}{m}\|\boldsymbol{x}\|_2^2 + \epsilon}}. \tag{3}$$

Since $\|\boldsymbol{x}\|_2 \geq \|\boldsymbol{x}\|_\infty$, we further obtain $\|\boldsymbol{y}\|_\infty \leq \sqrt{m}\|\boldsymbol{\gamma}\|_\infty$. This completes the proof. □

The bound given in Proposition 1 may be conservative in practice. Taking the bound in Eq. (3) directly, if the nonzero entries of $\boldsymbol{x}$ are concentrated on a few coordinates, the ratio can approach $\sqrt{m}\|\boldsymbol{\gamma}\|_\infty$; whereas for dense hidden states with more evenly distributed energy, the ratio is typically much smaller, and can be close to $\|\boldsymbol{\gamma}\|_\infty$ when the coordinate magnitudes are comparable.

We integrate RMSNorm into the LSTM as illustrated in Fig. 1. The standard LSTM formulation in Eq. (1) remains unchanged except for the penultimate equation $\boldsymbol{c}_t = \boldsymbol{f}_t \otimes \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \otimes \boldsymbol{g}_t$, which is replaced by

$$\boldsymbol{c}_t = \mathrm{RMSNorm}(\boldsymbol{f}_t \otimes \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \otimes \boldsymbol{g}_t).$$

We refer to this variant as *LSTM with RMSNorm*. Note that LSTM with RMSNorm only introduces RMSNorm before Tanh_c, unlike BatchNorm throughout the computation in [14]. The introduction of RMSNorm does add a new nonlinear operation, the inverse square root $\mathrm{InvSqrt}(x) = 1/\sqrt{x}$. In our experiments, the input to this operation lies in a small and well-bounded interval, approximately $[\epsilon, 2.25]$ with $\epsilon = 10^{-5}$.

3.3. Performance of Plain LSTM with RMSNorm

Due to the recurrent and multilayer structure of LSTM, the stabilization effect also propagates to subsequent layers. Fortunately, this RMSNorm strategy proves more friendly in our encrypted LSTM protocol than the standard one: It significantly reduces the domain of Tanh_c while keeping other domains still narrow.

Table 5: Activation function domains in LSTM with RMSNorm.

	IMDB	YELP	AGNEWS	DBPedia
Sigmoid	$-7.1 \sim 7.3$	$-11.6 \sim 9.4$	$-8.9 \sim 8.5$	$-10.7 \sim 14.6$
Tanh_g	$-2.6 \sim 3.0$	$-5.6 \sim 5.2$	$-7.1 \sim 5.7$	$-6.5 \sim 10.7$
Tanh_c	$-8.0 \sim 10.5$	$-11.5 \sim 7.4$	$-7.7 \sim 8.1$	$-14.6 \sim 13.7$
InvSqrt	9.6e-5 $\sim$ 1.6	1.9e-3 $\sim$ 1.9	2.5e-3 $\sim$ 1.8	2.6e-3 $\sim$ 2.2

Table 5 summarizes the input ranges of the nonlinear functions in LSTM with RMSNorm. Compared with Table 4, the input range of Tanh_c becomes dramatically narrower, and the ranges of the sigmoid, Tanh_g, and inverse square root functions also remain within moderate intervals. These tighter ranges translate into more favorable domains for polynomial approximation and improved numerical stability during homomorphic evaluation.

Table 6 compares the classification accuracy of LSTM models with and without RMSNorm across the four datasets. Compared with LayerNorm, RMSNorm avoids explicitly computing the mean and variance, resulting in simpler computation while still effectively controlling the magnitude of activations. Moreover, relative to the standard LSTM, it consistently yields higher classification accuracy across all four datasets considered. Although RMSNorm introduces an additional non-polynomial function, the input to this operation remains confined to a small interval. As we will show in Section 7, LSTM with RMSNorm also achieves both higher accuracy and better efficiency in encrypted LSTM inference than the standard LSTM [57]. In the next section, we describe how to evaluate these non-polynomial functions on homomorphically encrypted data.

Table 6: Accuracy (%) of LSTM with and without RMSNorm.

	IMDB	YELP	AGNEWS	DBPedia
Standard LSTM	86.31	94.95	92.03	98.53
LSTM with RMSNorm	87.35	95.86	92.78	98.78
Improvement	+1.04	+0.91	+0.75	+0.25

4. Encrypted Non-Polynomial Function Evaluation

Since FHE schemes support only additions and multiplications, non-polynomial activation functions in neural networks must be approximated by polynomials to enable encrypted inference. In our setting, we need to approximate three nonlinear functions, namely Sigmoid, Tanh, and InvSqrt, in a way that is compatible with encrypted arithmetic. After applying RMSNorm, the inputs to these functions become well-bounded and numerically stable, typically lying within small intervals (see Table 5). This property allows us to employ low-degree polynomials while still preserving sufficient numerical accuracy. However, numerical approximation error alone does not fully capture the practical effectiveness of a polynomial approximation; in addition to pointwise error, we must also account for its impact on the end-to-end performance of the encrypted LSTM model.

4.1. A Hybrid Strategy

For evaluating a polynomial of degree d, the required multiplicative depth is at least $\lceil \log d \rceil$, so it is common to choose degrees of the form $2^n - 1$ to obtain balanced multiplicative circuits.

Sigmoid and Tanh. In this work, we use minimax polynomials of degree 7 derived from the Remez algorithm [42] for Sigmoid and Tanh. The Remez algorithm minimizes the maximum absolute error over a target interval. It thus

provides a nearly uniform approximation quality across the entire range, in contrast to the least-squares approach, which minimizes the mean squared error and focuses on average performance. Since Tanh saturates more slowly and spans a wider effective interval than Sigmoid, it is generally more challenging to approximate accurately with low-degree polynomials. As illustrated in Fig. 2, on the interval $[-15, 15]$ the Remez-based polynomial of degree 7 yields an almost uniform error profile for Tanh, with a maximum absolute error of approximately 0.32. Although this worst-case error may appear relatively large from a purely numerical perspective, our experiments show that the resulting approximation is sufficient to maintain high classification accuracy in the downstream LSTM model. Similarly, Fig. 3 shows that the Remez approximation of Sigmoid attains a maximum absolute error of about 0.07 on the same interval. These observations are consistent with prior work [57], where Remez-based polynomials also demonstrated strong approximation performance for Sigmoid and Tanh.

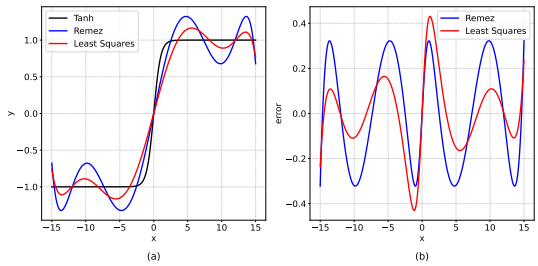

Figure 2: Tanh: (a) the function and its polynomial approximation of degree 7; (b) the error.

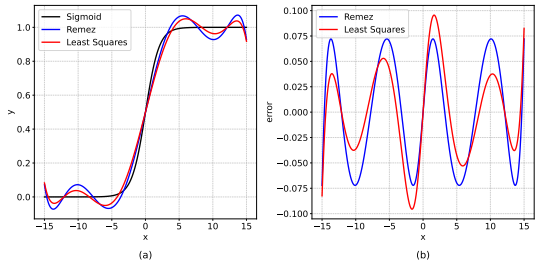

Figure 3: Sigmoid: (a) the function and its polynomial approximation of degree 7; (b) the error.

Inverse of Square Root. In our setting, the original approximation domain of InvSqrt is $[10^{-5}, 2.25]$. Direct polynomial approximation over such a wide interval is challenging due to the rapid variation of the function near zero, which typically requires a higher-degree polynomial to maintain sufficient accuracy. To address this issue, we introduce a constant shift ϵ_1 to the InvSqrt input. So the function InvSqrt is defined as $\text{InvSqrt}(x) = \frac{1}{\sqrt{x+\epsilon_1}}$, where ϵ_1 is a small constant introduced for numerical stability. At inference time, we evaluate the trained model using ϵ_1 in the InvSqrt function. Note that ϵ_1 is introduced only for inference to improve the polynomial approximation of the inverse square root function, and the model is neither retrained nor fine-tuned with this shift. As illustrated in Table 7, we evaluate different choices of ϵ_1, with $\epsilon_1 = 0.1$ yielding at most a 0.03% accuracy loss across all four datasets. With this constant added, the effective approximation domain becomes $[0.1, 2.35]$. Over this significantly narrower interval, the InvSqrt function becomes much smoother, allowing us to attempt to approximate it effectively using a polynomial of degree 3. Prior work, including [31] and [41], employs Newton-based iterative schemes to achieve high-precision InvSqrt evaluation under homomorphic encryption. However, these methods typically require multiple iterative refinements, increasing multiplicative depth and computational overhead. Compared with such iterative approaches, our method aims to reduce the approximation difficulty at the function level, thereby lowering the homomorphic evaluation depth and making it more suitable for efficient CKKS-based inference under limited bootstrapping budgets.

Table 7: Impact of the InvSqrt ϵ_1 on Accuracy (%).

ϵ_1	IMDB	YELP	AGNEWS	DBPedia
1e-5	87.35	95.86	92.78	98.78
1e-4	87.34	95.87	92.78	98.77
1e-3	87.35	95.87	92.78	98.76
1e-2	87.33	95.86	92.74	98.76
1e-1	87.33	95.87	92.75	98.75
1	78.59	84.28	75.79	85.25

At first glance, it is natural to apply the same Remez-based minimax approximation strategy to InvSqrt as for Sigmoid and Tanh. However, our experimental results indicate that this uniform choice is suboptimal. As summarized in Table 9, replacing all nonlinear functions with Remez-based polynomials leads to worse classification performance than a hybrid strategy that uses Remez polynomials for Sigmoid and Tanh but employs a least-squares polynomial for InvSqrt. The key reason lies in the distinct optimization objectives of the two criteria. The Remez algorithm optimizes the global worst-case error over the entire interval. But the InvSqrt function exhibits rapid variation and steep gradients near one end of the interval, while remaining relatively smooth elsewhere. Although the Remez algorithm guarantees a tight uniform bound, it often yields a higher mean-squared error. In contrast, least-squares fitting minimizes the mean squared error over the target domain $[0.1, 2.35]$. In particular, we first sample 100 points uniformly within this interval and then fit a low-degree (e.g., degree-3) polynomial to the sampled values, producing an approximation that better captures the function's average behavior. As illustrated in Fig. 4 and Table 9, although Remez provides a better worst-case guarantee, the least-squares approximation offers better overall fidelity. And the least-squares approximation achieves a significantly lower mean squared error of 0.06, compared to 0.12 for the Remez method.

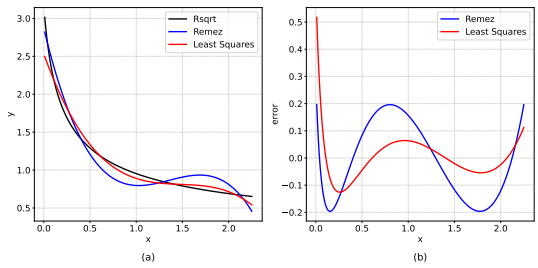

Figure 4: InvSqrt: (a) The function and its polynomial approximation of degree 3; (b) the error.

Our Hybrid Strategy. We use polynomial approximations of degree 7 produced by the Remez algorithm [42] to replace functions Sigmoid and Tanh, and replace the function InvSqrt with a degree-3 least square-based polynomial approximation for encrypted inference of LSTM with RMSNorm.

Table 8: Hybrid strategy for activation function approximation.

	Sigmoid	Tanh_g	Tanh_c	InvSqrt
Method	Remez	Remez	Remez	Least Square
Degree	7	7	7	3

4.2. End-to-End Accuracy in Plaintext

Effect of polynomial approximation strategy (degree-7). We first study the impact of different polynomial approximation strategies, fixing the approximation degree to 7 for the Sigmoid and Tanh functions and to 3 for the InvSqrt function. Table 9 reports the classification accuracy of LSTM with RMSNorm under four settings: no approximation (*Original*), applying least-squares polynomials to all nonlinearities (*Least-squares (all)*), applying Remez-based minimax polynomials to all nonlinearities (*Remez (all)*), and our hybrid strategy that uses Remez for Sigmoid and Tanh and least-squares fitting for InvSqrt (*Hybrid (deg-7)*).

Across all four datasets, the hybrid strategy incurs an absolute accuracy loss of at most 1.05% relative to the original model. It consistently outperforms both "Least-squares (all)" and "Remez (all)", which shows that *a single approximation family is not uniformly optimal for all nonlinearities*, and that combining Remez for Sigmoid/Tanh with least-squares fitting for InvSqrt provides the best accuracy-efficiency trade-off under a fixed polynomial degree.

Table 9: Effect of approximation strategy on accuracy (%).

	IMDB	YELP	AGNEWS	DBPedia
Original	87.35	95.86	92.78	98.78
Least-squares (all)	86.54	93.87	91.96	95.33
Remez (all)	86.92	94.76	91.49	95.11
Hybrid (deg-7, ours)	87.32	95.53	92.33	97.73

Effect of polynomial degree for Tanh_c *under the hybrid strategy.* Having identified the hybrid strategy as the most effective approximation family, we next investigate the impact of the polynomial degree. In particular, we compare our default setting, which uses a polynomial of degree 7 for Tanh_c under the hybrid strategy, with a more aggressive choice that replaces Tanh_c with a degree-3 polynomial while keeping the other nonlinearities unchanged. The results are summarized in Table 10.

Table 10: Effect of the polynomial degree for Tanh_c on accuracy (%).

	IMDB	YELP	AGNEWS	DBPedia
Original	87.35	95.86	92.78	98.78
Hybrid (deg-7, ours)	87.32	95.53	92.33	97.73
Hybrid (deg-3 for Tanh_c)	85.70	91.53	89.42	95.39

As shown in Table 10, reducing the degree of the polynomial used for Tanh_c from seven to three leads to an accuracy drop of up to 4.33% (for the YELP dataset) across the four datasets, despite the reduced multiplicative depth. This indicates that overly aggressive degree reduction introduces approximation noise that the LSTM architecture cannot fully compensate for, and that a degree-7 polynomial is necessary to maintain high end-to-end accuracy in our encrypted LSTM setting.

In summary, we show that a hybrid approximation strategy (using Remez-based minimax polynomials for Sigmoid and Tanh and least-squares polynomials for InvSqrt) is more effective than relying on any single approximation family for all nonlinearities. In general, the proposed approximation strategy can be summarized as follows.

- Given a target task, the first key step is to estimate the input ranges of the nonlinear functions (e.g., Sigmoid, Tanh, and InvSqrt) from the distribution of intermediate activations observed on a validation set.
- If the activation range is excessively large, normalization techniques such as RMSNorm can be incorporated to reduce the dynamic range, thereby enabling lower-degree polynomial approximations and improving noise efficiency under CKKS. Furthermore, excluding a marginal proportion of extreme outliers allows for a more compact approximation interval without compromising accuracy, as these rare values have negligible impact on the inference accuracy.
- Based on these estimated intervals, polynomial approximations are then constructed by choosing suitable approximation methods (e.g., Remez or least-squares) and polynomial degrees, with the goal of minimizing the end-to-end inference accuracy degradation rather than the pointwise approximation error. In practice, we adopt an accuracy-driven search strategy: starting from low-degree polynomials, we gradually increase the degree until the accuracy loss is within a predefined tolerance (e.g., within 2%).

In the next section, we turn to the design of encrypted linear operations in LSTM, which, together with the nonlinear approximations described above, form the core of our homomorphic inference protocol.

5. Encrypted Linear Operations in LSTM

In the standard LSTM formulation shown in Eq. (1), each gate (input, forget, cell, and output) performs a linear transformation of the current input $\boldsymbol{x}_t \in \mathbb{R}^m$ and the previous hidden state $\boldsymbol{h}_{t-1} \in \mathbb{R}^m$, followed by a nonlinear activation. These gate computations can be compactly represented as $\boldsymbol{Z}_{gate} = \boldsymbol{W}_{i,gate}\boldsymbol{x}_t + \boldsymbol{W}_{h,gate}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{gate}$, where $\boldsymbol{W}_{i,gate}$ and $\boldsymbol{W}_{h,gate} \in \mathbb{R}^{m\times m}$, and $\boldsymbol{b}_{gate} \in \mathbb{R}^m$ denote the weight matrices and bias vectors corresponding to the input, forget, cell, and output gates, respectively. To fully exploit SIMD parallelism and increase throughput, a batch of size B of inputs $\boldsymbol{x}_t$ and hidden states $\boldsymbol{h}_{t-1}$ is processed in a single computation, which transforms the equation into

$$\boldsymbol{Z}_{gate} = \boldsymbol{W}_{i,gate}\boldsymbol{X}_t + \boldsymbol{W}_{h,gate}\boldsymbol{H}_{t-1} + \boldsymbol{B}_{gate}, \qquad (4)$$

where $\boldsymbol{X}_t$ and $\boldsymbol{H}_{t-1}$ are in $\mathbb{R}^{m\times B}$ and $\boldsymbol{B}_{gate}$ is obtained by repeating $\boldsymbol{b}_{gate}$ B times. This equation provides a compact notation for expressing the independent linear transformations of all gates within the LSTM cell. When applied to the encrypted inference protocol illustrated in Fig. 6, these linear operations correspond to plaintext-ciphertext matrix multiplications (PCMM), which are the most computationally expensive operations besides Bts (bootstrapping) in our experiments.

5.1. The Method Adopted in Zhang et al. [57]

First, we note that there has been substantial recent progress on ciphertext-ciphertext matrix multiplication (CCMM); see, e.g., [26]. However, methods designed for CCMM are not, in general, optimal for PCMM (see Table 11). In the literature, PCMM has been extensively studied, e.g., [20, 33, 6]. The algorithm in [6] is mainly intended for PCMM in high-dimensional settings (e.g., dimension 2^{16}) [43]. By contrast, in our PPLSTM experiments, only two PCMM configurations are used: $128 \times 128 \times 256$ and $64 \times 64 \times 512$. Although in [6] the authors do not report such small cases, the closest result is a 256-dimensional square PCMM taking about 0.3s. Our single-threaded implementation takes 0.78 s and 0.40 s, respectively. However, [5] uses $N = 2^{13}$ (degree of the CKKS ring) and AVX-512 acceleration, whereas we use $N = 2^{15}$ in Lattigo without AVX-512. Moreover, it relies on coefficient encoding and therefore requires coefficient-to-slot and slot-to-coefficient conversions when adapted to our setting, which introduces additional overhead. The approach in [33] imposes restrictions on the admissible matrix dimensions. Recent advances in encrypted Transformer inference also rely heavily on PCMM kernels. However, these kernels are typically designed and optimized for attention computation, especially $(\boldsymbol{Q}\boldsymbol{K}^\top)\boldsymbol{V}$, in which the matrix dimensions, data reuse patterns, and packing layouts differ substantially from those used in LSTM inference. Therefore, they are not necessarily optimal for the LSTM setting considered in this work. Our goal is not to develop a universally optimal PCMM algorithm, but to adopt and optimize a PCMM strategy that is well-suited to encrypted LSTM inference. To this end, we employ diagonal encoding [20], which is well aligned with the gate-wise matrix–vector products in LSTM (see Eq. (1)). This encoding strategy has also been used in recent encrypted Transformer inference protocols [37, 36, 54, 50] and has been validated for LSTM inference by Zhang *et al.* in PrivLSTM [57]. Building on this observation, we further optimize the diagonal-encoding-based design to improve both scalability and efficiency in encrypted LSTM inference.

Linear Transformation on Encrypted Vectors. For a plaintext square matrix $\boldsymbol{W} = (w_{i,j})$ of dimension m, the i-th *diagonal vector* of $\boldsymbol{W}$ is defined as $\boldsymbol{d}_i(\boldsymbol{W}) = (w_{0,i}, w_{1,i+1}, \ldots, w_{m-i-1,m-1}, w_{m-i,0}, \ldots, w_{m-1,i-1})$, denoted by $\boldsymbol{d}_i$ for simplicity. Now, for an encrypted m-dimensional vector $\boldsymbol{x}$, Halevi and Shoup in [20] utilize the formula $\boldsymbol{W}\boldsymbol{x} = \sum_{0\le i<m} \boldsymbol{d}_i \otimes \mathsf{Rot}_i(\boldsymbol{x})$. Computing $\boldsymbol{W}\boldsymbol{x}$ with this formula requires m Rots. In [21], they employ the baby-step-giant-step (BSGS) strategy, reducing m Rots to $O(\sqrt{m})$. The idea is based on the following formula (assuming $m = \ell \cdot k$ with $k \approx \ell \approx \sqrt{m}$):

$$\boldsymbol{W}\boldsymbol{x} = \sum_{0\le i<\ell} \mathsf{Rot}_{ik}\Big(\sum_{0\le j<k} \mathsf{Rot}_{-ik}(\boldsymbol{d}_{ik+j}) \otimes \mathsf{Rot}_j(\boldsymbol{x}) \Big) \qquad (5)$$

The reason BSGS can significantly reduce the number of required Rots is that the matrix $\boldsymbol{W}$ is in plaintext, allowing us to precompute $\mathsf{Rot}_{-ik}(\boldsymbol{d}_{ik+j})$ directly in plaintext.

Integration Method. When evaluating Eq. (4), Zhang *et al.* [57] proposed a method called *integration*, which computes all four gates (input, forget, cell, and output) simultaneously in a single batched operation. Using the notation introduced in Eq. (1) and (4), their computation can be rewritten as

$$\begin{bmatrix} \boldsymbol{Z}_i \\ \boldsymbol{Z}_f \\ \boldsymbol{Z}_g \\ \boldsymbol{Z}_o \end{bmatrix} = \boldsymbol{W}_i \begin{bmatrix} \boldsymbol{X}_t \\ \boldsymbol{X}_t \\ \boldsymbol{X}_t \\ \boldsymbol{X}_t \end{bmatrix} + \boldsymbol{W}_h \begin{bmatrix} \boldsymbol{H}_{t-1} \\ \boldsymbol{H}_{t-1} \\ \boldsymbol{H}_{t-1} \\ \boldsymbol{H}_{t-1} \end{bmatrix} + \begin{bmatrix} \boldsymbol{B}_i \\ \boldsymbol{B}_f \\ \boldsymbol{B}_g \\ \boldsymbol{B}_o \end{bmatrix}, \qquad (6)$$

where $\boldsymbol{W}_i = \mathrm{diag}(\boldsymbol{W}_{i,i}, \boldsymbol{W}_{i,f}, \boldsymbol{W}_{i,g}, \boldsymbol{W}_{i,o})$ and $\boldsymbol{W}_h = \mathrm{diag}(\boldsymbol{W}_{h,i}, \boldsymbol{W}_{h,f}, \boldsymbol{W}_{h,g}, \boldsymbol{W}_{h,o})$. Here $\mathrm{diag}(\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}, \boldsymbol{Y})$ denotes the block-diagonal matrix with matrices $\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}$, and $\boldsymbol{Y}$. Both $\boldsymbol{W}_i$ and $\boldsymbol{W}_h$ are of dimension $(4m) \times (4m)$. The ciphertext matrices are both of dimension $(4m) \times B$. Subsequently, the computations in Eq. (4) for all gates are reduced to two PCMMs in Eq. (6). For each PCMM, the ciphertext matrix is treated column-wise, yielding B linear transformations on encrypted $4m$-dimensional vectors.

According to the analysis in [57], each such linear transformation requires $2\sqrt{4m} = 4\sqrt{m}$ rotation operations. However, we note that this estimate overstates the actual cost. In fact, since both $\boldsymbol{W}_i$ and $\boldsymbol{W}_h$ are block-diagonal matrices, each of them has at most $2m-1$ nonzero diagonals. Therefore, by the BSGS method, a single PCMM in Eq. (5) can be carried out using at most $2\sqrt{2m-1} \le 3\sqrt{m}$ Rot operations.

Nevertheless, the integration method has two main drawbacks. First, for a fixed number of ciphertext slots, it effectively reduces the usable batch size, because each sample must be replicated across the four gates. Second, in subsequent steps, additional operations are required to extract the corresponding gates from the integrated ciphertext so that different nonlinear functions can be applied to different gates. To address these issues, we adopt a divide-and-conquer strategy, i.e., computing each gate separately.

5.2. Batched Linear Transformation on Encrypted Data

We now turn to a gate-by-gate computation of the PCMMs in Eq. (4). For simplicity, we consider $\boldsymbol{WX}$ with plaintext matrix $\boldsymbol{W} \in \mathbb{R}^{m\times m}$ and encrypted matrix $\boldsymbol{X} \in \mathbb{R}^{m\times B}$. Without loss of generality, we assume that the number of ciphertext slots is exactly $m \cdot B$ so that $\boldsymbol{X}$ can be encrypted as a single ciphertext. In particular, we pack the entries of $\boldsymbol{X} \in \mathbb{R}^{m\times B}$ in to a single vector of dimension mB in the following way:

$$\begin{aligned}\boldsymbol{x}' = [\ & x_{0,0}, x_{1,0}, \ldots, x_{B-1,0}, \\ & x_{0,1}, x_{1,1}, \ldots, x_{B-1,1}, \ldots, \\ & x_{0,m-1}, x_{1,m-1}, \ldots, x_{B-1,m-1}],\end{aligned} \tag{7}$$

as shown in Fig. 5. The resulting vector $\boldsymbol{x}'$ is then encrypted as one ciphertext. For the plaintext matrix $\boldsymbol{W}$, it has m diagonal vectors $(\boldsymbol{d}_i)_{i<m}$ of dimension m. For each i, we encode $\boldsymbol{d}_i = (d_{i,j})_{j<m}$ as

$$\boldsymbol{d}'_i = (\underbrace{d_{i,0}, \ldots, d_{i,0}}_{B \text{ times}}, \underbrace{d_{i,1}, \ldots, d_{i,1}}_{B \text{ times}}, \ \ldots, \ \underbrace{d_{i,m-1}, \ldots, d_{i,m-1}}_{B \text{ times}}),$$

that is, each entry $d_{i,j}$ is repeated B times to form an (mB)-dimensional vector. Under this setting, the batched version of Eq. (5) can be rewritten as

$$\begin{aligned}\boldsymbol{z}' &= \sum_{0\le i<m} \boldsymbol{d}'_i \otimes \mathsf{Rot}_{iB}(\boldsymbol{x}') \\ &= \sum_{0\le i<\ell} \mathsf{Rot}_{ikB}\Big(\sum_{0\le j<k} \mathsf{Rot}_{-ikB}\big(\boldsymbol{d}'_{ik+j}\big) \otimes \mathsf{Rot}_{jB}(\boldsymbol{x}')\Big).\end{aligned} \tag{8}$$

The output vector $\boldsymbol{z}'$ corresponds exactly to the result of $\boldsymbol{Z} = \boldsymbol{WX}$ whose encoding matches that of the input $\boldsymbol{X}$ (see Eq. (7)), which facilitates the subsequent computations. Clearly, such a computation requires only m ciphertext–plaintext multiplications (CMul) and $2\sqrt{m}$ rotations (Rot). Recall that we need to apply four different weight matrices $\boldsymbol{W}$ to the same input $\boldsymbol{X}$. In Eq. (8), the inner summation terms $\mathsf{Rot}_{jB}(\boldsymbol{x}')$ depend only on $\boldsymbol{X}$ and can therefore be *reused* across all four matrices. As a result, to evaluate the first summand in Eq. (4) for four gates with the same $\boldsymbol{X}$, the total number of rotations required is only $5\sqrt{m}$, rather than $8\sqrt{m}$.

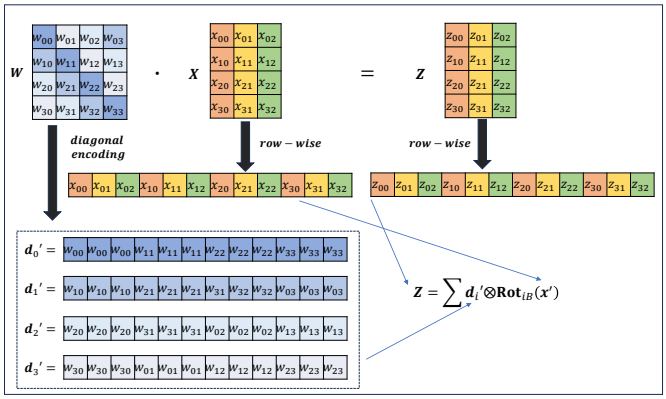

Figure 5: PCMM with the diagonal encoding strategy. ($B = 3$)

Comparison. To quantitatively compare our method with the integration approach, we focus on the cost of computing the *four* PCMMs $\boldsymbol{W}_{i,i}\boldsymbol{X}_t$, $\boldsymbol{W}_{i,f}\boldsymbol{X}_t$, $\boldsymbol{W}_{i,g}\boldsymbol{X}_t$, and $\boldsymbol{W}_{i,o}\boldsymbol{X}_t$. As summarized in Table 11, both methods require the same number of ciphertext–plaintext multiplications, namely $4m$ CMul operations. The main difference lies in how they trade batch size for rotations.

Table 11: Cost comparison with the methods in [26] and [57] for the four PCMMs. For the method in [26], the reported upper bound on the number of operations is derived by partitioning the $m \times B$ input matrix into B/m blocks of size $m \times m$ and applying the corresponding $m \times m$ PCMM algorithm to each block.

Method	Batch Size	CMul		Rot	
		Total	Amortized	Total	Amortized
JKLS [26]	B	$8B$	8	$4B + \frac{8B}{\sqrt{m}}$	$4 + \frac{8}{\sqrt{m}}$
Integration [57]	B	$2m-1$	$\frac{2m-1}{B}$	$3\sqrt{m}$	$\frac{3\sqrt{m}}{B}$
Ours	$4B$	$4m$	$\frac{m}{B}$	$5\sqrt{m}$	$\frac{5\sqrt{m}}{4B}$

Under the integration method [57], a batch of size B is processed with $3\sqrt{m}$ rotations ($4\sqrt{m}$ in the analysis of [57]), yielding an amortized rotation cost of $3\sqrt{m}/B$ per input. In contrast, our method processes a batch of size $4B$ under the same parameter setup as the integration method, but requires only $5\sqrt{m}$ rotations, yielding an amortized cost of $5\sqrt{m}/(4B)$. Hence, the per-input rotation cost of our method is

$$\frac{5\sqrt{m}/(4B)}{3\sqrt{m}/B} = \frac{5}{12} \approx 0.417$$

of that of the integration method, corresponding to a 58.3% reduction in rotations per input (or equivalently a 2.4× improvement).

In summary, compared with the integration method, for the same number of CMul operations, our scheme achieves

a larger effective batch size and hence a lower amortized rotation and CMul cost. Since rotations are computationally expensive, this reduction directly improves the efficiency of encrypted LSTM inference.

For completeness, Table 11 also reports the cost of applying the method in [26] to the same computational task, where the $m \times B$ input matrix is partitioned into B/m square blocks of size $m \times m$. Since, in the LSTM setting considered here, B is divisible by m (and hence $B \geq m$), our method achieves a lower amortized cost.

6. Put Everything Together

Given the above preparations, we present a complete algorithm for LSTM inference over encrypted batched data in this section (Algorithm 1). Before doing so, we first introduce the following building blocks required by the algorithm:

- $\mathsf{PCMM}(\boldsymbol{W}, \mathsf{ct}.\boldsymbol{x})$ implements the batched plaintext-ciphertext matrix multiplication algorithm described in Section 5.2.
- $\mathsf{PolyEval}(\mathsf{ct}.\boldsymbol{x}, \boldsymbol{p})$ evaluates a polynomial with coefficient vector $\boldsymbol{p}$ on the encrypted input $\mathsf{ct}.\boldsymbol{x}$. For example, it can be instantiated using the Paterson-Stockmeyer algorithm [38].
- $\mathsf{SumRows}(\mathsf{ct}.\boldsymbol{x})$ takes as input a ciphertext $\mathsf{ct}.\boldsymbol{x}$ encrypting a matrix $\boldsymbol{X} \in \mathbb{R}^{m \times B}$ and returns a ciphertext encrypting the vector obtained by summing each row of $\boldsymbol{X}$.

6.1. The Main Algorithm

Algorithm 1 presents the homomorphic inference procedure for LSTM with RMSNorm. Given a batched input sequence $\{\boldsymbol{X}_t \in \mathbb{R}^{m \times B}\}_{t=1}^{T}$ of length T, we first pack each $\boldsymbol{X}_t$ according to Eq. (7) and encrypt it to obtain a ciphertext sequence $\{\mathsf{ct}.\boldsymbol{x}_t\}_{t=1}^{T}$. The (encrypted) computation in Algorithm 1 then follows the forward pass of LSTM with RMSNorm.

In Algorithm 1, the for loop in Step 3 computes the linear transformations for each gate before applying the nonlinear activations. For every $gate \in \{i, f, g, o\}$, after this loop we obtain $\boldsymbol{Z}_{gate}$ as Eq. (4). The core computation in this step is carried out by the PCMM algorithm described in Section 5.2. Steps 7–10 then apply the nonlinear activations for each gate, where the corresponding activation functions are evaluated homomorphically via the polynomial approximations specified in Table 8 (i.e., the proposed hybrid strategy). Step 11 corresponds to the computation of $\boldsymbol{c}_t$ in Eq. (1), while Steps 12–15 implement the RMSNorm operation. The latter involves the SumRows primitive and a polynomial approximation to the InvSqrt function. Steps 16–17 correspond to the last equation in Eq. (1), namely $\boldsymbol{H}_t = \boldsymbol{O}_t \otimes \mathrm{Tanh}(\boldsymbol{C}_t)$. After repeating the above steps for T times, the resulting hidden states are passed through a final fully connected layer, yielding a ciphertext $\mathsf{ct}.\boldsymbol{r}$ that encrypts the predictions for the B input sequences in the batch.

Algorithm 1 Batched LSTM Network Inference on Encrypted Data

Input: A ciphertext sequence $\{\mathsf{ct}.\boldsymbol{x}_t\}_{t=1}^{T}$ with each $\boldsymbol{x}_t \in \mathbb{R}^{mB}$ obtained by packing $\boldsymbol{X}_t \in \mathbb{R}^{m \times B}$ as Eq. (7); plaintext weights $\boldsymbol{W}_{i,gate}$ and $\boldsymbol{W}_{h,gate}$, and plaintext bias $\boldsymbol{b}_{gate}$ for $gate \in \{i, f, g, o\}$; plaintext polynomial coefficients $\boldsymbol{p}_{\mathrm{Sigmoid}}$, $\boldsymbol{p}_{\mathrm{Tanh}_g}$, $\boldsymbol{p}_{\mathrm{Tanh}_c}$, and $\boldsymbol{p}_{\mathrm{InvSqrt}}$ for approximation of non-polynomial functions in Table 8; and the parameter $\boldsymbol{\gamma}$ used in Eq. (2) for RMSNorm; the plaintext weights and bias for the last fully connected layer $\boldsymbol{W}_{\mathrm{fc}}$ and $\boldsymbol{b}_{\mathrm{fc}}$.

Output: Ciphertext output $\mathsf{ct}.\boldsymbol{r}$.

1: Initialize $\mathsf{ct}.\boldsymbol{h}_0$ and $\mathsf{ct}.\boldsymbol{c}_0$ as an encryption of zero-vector. ▷ Initialize $\boldsymbol{H}_0$ and $\boldsymbol{C}_0$ as zero matrix.
2: **for** $t = 1$ to T **do**
3: **for** $gate \in \{i, f, g, o\}$ **do**
4: $\mathsf{ct}.\boldsymbol{a}_1 \leftarrow \mathsf{PCMM}(\boldsymbol{W}_{i,gate}, \mathsf{ct}.\boldsymbol{x}_t)$ ▷ $\boldsymbol{A}_1 = \boldsymbol{W}_{i,gate} \cdot \boldsymbol{X}_t$
5: $\mathsf{ct}.\boldsymbol{a}_2 \leftarrow \mathsf{PCMM}(\boldsymbol{W}_{h,gate}, \mathsf{ct}.\boldsymbol{h}_{t-1})$ ▷ $\boldsymbol{A}_2 = \boldsymbol{W}_{h,gate} \cdot \boldsymbol{H}_{t-1}$
6: $\mathsf{ct}.\boldsymbol{z}_{gate} \leftarrow \mathsf{Add}(\mathsf{ct}.\boldsymbol{a}_1, \mathsf{ct}.\boldsymbol{a}_2, \boldsymbol{b}_{gate})$
7: $\mathsf{ct}.\boldsymbol{i}_t \leftarrow \mathsf{PolyEval}(\mathsf{ct}.\boldsymbol{z}_i, \boldsymbol{p}_{\mathrm{Sigmoid}})$
8: $\mathsf{ct}.\boldsymbol{f}_t \leftarrow \mathsf{PolyEval}(\mathsf{ct}.\boldsymbol{z}_f, \boldsymbol{p}_{\mathrm{Sigmoid}})$
9: $\mathsf{ct}.\boldsymbol{o}_t \leftarrow \mathsf{PolyEval}(\mathsf{ct}.\boldsymbol{z}_o, \boldsymbol{p}_{\mathrm{Sigmoid}})$
10: $\mathsf{ct}.\boldsymbol{g}_t \leftarrow \mathsf{PolyEval}(\mathsf{ct}.\boldsymbol{z}_g, \boldsymbol{p}_{\mathrm{Tanh}_g})$
11: $\mathsf{ct}.\hat{\boldsymbol{c}} \leftarrow \mathsf{Add}(\mathsf{Mul}(\mathsf{ct}.\boldsymbol{f}_t, \mathsf{ct}.\boldsymbol{c}_{t-1}), \mathsf{Mul}(\mathsf{ct}.\boldsymbol{i}_t, \mathsf{ct}.\boldsymbol{g}_t))$ ▷ $\hat{\boldsymbol{C}} = \boldsymbol{F}_t \otimes \boldsymbol{C}_{t-1} + \boldsymbol{I}_t \otimes \boldsymbol{G}_t$
12: $\mathsf{ct}.\boldsymbol{v} = \mathsf{CMul}(\mathsf{SumRows}(\mathsf{Mul}(\mathsf{ct}.\hat{\boldsymbol{c}}, \mathsf{ct}.\hat{\boldsymbol{c}})), \frac{1}{m})$ ▷ Beginning of RMSNorm
13: $\mathsf{ct}.\boldsymbol{t} \leftarrow \mathsf{PolyEval}(\mathsf{ct}.\boldsymbol{v}, \boldsymbol{p}_{\mathrm{InvSqrt}})$
14: $\mathsf{ct}.\hat{\boldsymbol{c}}_t \leftarrow \mathsf{Mul}(\mathsf{ct}.\hat{\boldsymbol{c}}, \mathsf{ct}.\boldsymbol{t})$
15: $\mathsf{ct}.\boldsymbol{c}_t \leftarrow \mathsf{CMul}(\mathsf{ct}.\hat{\boldsymbol{c}}_t, \boldsymbol{\gamma})$ ▷ $\boldsymbol{C}_t = \mathrm{RMSNorm}(\hat{\boldsymbol{C}})$
16: $\mathsf{ct}.\boldsymbol{t} \leftarrow \mathsf{PolyEval}(\mathsf{ct}.\boldsymbol{c}_t, \boldsymbol{p}_{\mathrm{Tanh}_c})$
17: $\mathsf{ct}.\boldsymbol{h}_t \leftarrow \mathsf{Mul}(\mathsf{ct}.\boldsymbol{o}_t, \mathsf{ct}.\boldsymbol{t})$ ▷ $\boldsymbol{H}_t = \boldsymbol{O}_t \otimes \mathrm{Tanh}(\boldsymbol{C}_t)$
18: $\mathsf{ct}.\boldsymbol{r} \leftarrow \mathsf{Add}(\mathsf{PCMM}(\boldsymbol{W}_{\mathrm{fc}}, \mathsf{ct}.\boldsymbol{h}_T), \boldsymbol{b}_{\mathrm{fc}})$ ▷ Fully connected

6.2. Our Protocol

As shown in Fig. 6, our privacy-preserving LSTM inference protocol, called PPLSTM, follows the most common architecture used in homomorphic-encryption-based privacy-preserving machine learning (PPML): a two-party client-server setting under the semi-honest (honest-but-curious) adversarial model.

PPLSTM works as follows:

- The client holds a batch of input data $\boldsymbol{X} \in \mathbb{R}^{m \times B}$, where m is the feature dimension and B is the batch size per inference. After generating a secret/public key pair $(\mathsf{sk}, \mathsf{pk})$ for an homomorphic encryption scheme (say, CKKS), the client encrypts $\boldsymbol{X}$ under pk and sends the resulting ciphertexts to the

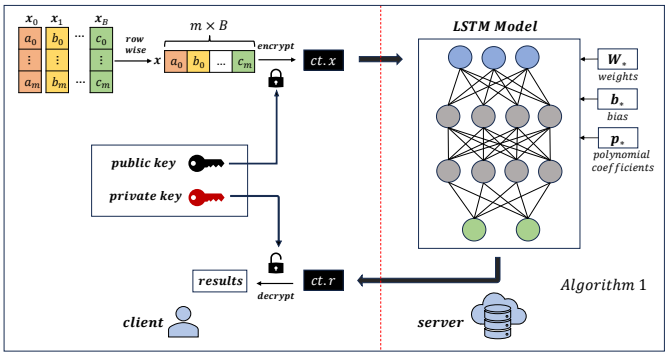

Figure 6: Architecture of Our PPLSTM Protocol.

server, together with the public key and the evaluation keys required for homomorphic computation (e.g., relinearization keys, rotation keys, and bootstrapping keys).

- The server holds the model parameters, including weights, biases, and normalization parameters. Upon receiving the encrypted data and keys from the client, the server runs our algorithm for batched LSTM inference on encrypted data (Algorithm 1) and obtains a ciphertext $\mathsf{ct}.\boldsymbol{r}$ encrypting the prediction results. This ciphertext is then returned to the client.

- Finally, the client uses the secret key sk to decrypt $\mathsf{ct}.\boldsymbol{r}$ and recovers the plaintext prediction vector $\boldsymbol{r}$.

6.2.1. Security

We now analyze the security of PPLSTM under the *semi-honest* (a.k.a. *honest-but-curious*) model. There are three main reasons for adopting this model. First, the parties must trust each other to some extent, as the server is legitimately providing the inference service. Second, protocols in the semi-honest model offer much better performance and are therefore more practical in real-world deployments. Third, although this setting cannot protect against fully malicious adversaries, it still provides strong confidentiality guarantees for the client's data and the server's model parameters.

Threat Model. In the semi-honest model, all parties execute the protocol as specified but may attempt to infer additional information from the messages they observe; security requires that they learn nothing beyond their inputs and outputs. In particular, we use the real-world vs ideal-world paradigm, which is commonly used in the study of privacy-preserving machine learning with homomorphic encryption. In this paradigm, the security of the protocol is assessed by comparing what an adversarial server (or client) can learn in the real execution of the protocol with what can be simulated in an ideal execution where only legitimate outputs are revealed.

Data Privacy for Client. In the real world, the client encrypts the plaintext inputs $\{\boldsymbol{X}_t\}_{1\le t\le T}$ using the CKKS public key and sends the ciphertexts $\mathsf{ct}.\boldsymbol{x}_t$ to the server, as illustrated in Fig. 6. The server performs the LSTM inference entirely over encrypted data following Algorithm 1, producing the ciphertext result $\mathsf{ct}.\boldsymbol{r}$, which is returned to the client. The server therefore observes only: $\mathcal{V}_{real}^{server} = \{\mathsf{ct}.\boldsymbol{x}_t, \boldsymbol{W}_*, \boldsymbol{b}_*, \gamma, \mathsf{ct}.\boldsymbol{r}\}$. In the ideal world, a trusted third party (TTP) takes the client's plaintext input $\boldsymbol{x}_t$ and the server's plaintext model parameters $(\boldsymbol{W}_*, \boldsymbol{b}_*)$, performs the LSTM inference in the clear, and returns only the final plaintext output r to the client. The view of the server in this ideal execution is: $\mathcal{V}_{ideal}^{server} = \{\boldsymbol{W}_*, \boldsymbol{b}_*, \boldsymbol{\gamma}\}$, and the view of the client is: $\mathcal{V}_{ideal}^{client} = \{\boldsymbol{x}_t, \boldsymbol{r}\}$. By the semantic security of CKKS, the ciphertexts in the real world are computationally indistinguishable from random values. Hence, the adversarial server's view $\mathcal{V}_{real}^{server}$ is indistinguishable from $\mathcal{V}_{ideal}^{server}$. Therefore, no additional information about the client's data can be inferred beyond what is revealed in the ideal world.

To prove security, we construct simulators that generate views computationally indistinguishable from the real views, relying solely on the information available in the ideal world. We construct a simulator that takes $\mathcal{V}_{ideal}^{server}$ as input. Since the simulator lacks access to the real inputs $\boldsymbol{x}_t$, it generates dummy ciphertexts by encrypting zero vectors of the corresponding dimension: $\mathsf{ct}.\tilde{\boldsymbol{x}}_t \leftarrow \mathsf{Enc}_{pk}(\boldsymbol{0})$, so $\mathcal{V}_{sim}^{server} = \{\mathsf{ct}.\tilde{\boldsymbol{x}}_t, \boldsymbol{W}_*, \boldsymbol{b}_*\}$. The simulator then performs the homomorphic operations in PPLSTM to produce a simulated output ciphertext: $\mathsf{ct}.\tilde{\boldsymbol{r}} \leftarrow \mathsf{PPLSTM}(\mathsf{ct}.\tilde{\boldsymbol{x}}_t, \boldsymbol{W}_*, \boldsymbol{b}_*)$. Due to the IND-CPA security of the CKKS scheme, the encryption of the real input is computationally indistinguishable from the encryption of zero, i.e., $\mathsf{Enc}_{pk}(\boldsymbol{x}_t) \approx_c \mathsf{Enc}_{pk}(\boldsymbol{0})$. Consequently, the distribution of the simulated view is computationally indistinguishable from the real view ($\mathcal{V}_{real}^{server} \approx_c \mathcal{V}_{siml}^{server}$). This guarantees that the server learns no information about the client's inputs, since it cannot distinguish between processing real data and processing zero vectors.

Model Privacy for Server. Similarly, the client receives only the final ciphertext $\mathsf{ct}.\boldsymbol{r}$ and decrypts it locally using the CKKS private key. Thus the client's real-world view is $\mathcal{V}_{real}^{client} = \{\{\boldsymbol{X}_t\}_t, \mathsf{ct}.\boldsymbol{r}, \mathsf{Dec}(\mathsf{ct}.\boldsymbol{r})\}$, from which the client learns nothing about the server-side model parameters $(\boldsymbol{W}_*, \boldsymbol{b}_*, \boldsymbol{\gamma})$ beyond what is inherently implied by the final plaintext output.

We remark that as in other black-box inference services, preventing model extraction [46] under repeated adaptive queries requires additional mechanisms, e.g., [28, 44], which are orthogonal to our cryptographic inference protocol and outside the scope of this paper.

6.2.2. Computational Cost

We now analyze the computational cost of the PPLSTM protocol. For the client, the computational cost is limited to key generation, encryption, and decryption. In what

follows, we focus on the server-side cost, i.e., the complexity of Algorithm 1.

Recall that a single $\mathsf{PCMM}(\boldsymbol{W} \in \mathbb{R}^{m\times m}, \mathsf{ct}.\boldsymbol{x})$ in Section 5.2 requires at most $2\sqrt{m}$ rotations (Rot) and m plaintext-ciphertext multiplications (CMul), and that some intermediate results in the BSGS procedure (in particular, the rotated versions of $\mathsf{ct}.\boldsymbol{x}_t$ and $\mathsf{ct}.\boldsymbol{h}_{t-1}$) can be reused. For the four different weight matrices $\boldsymbol{W}_{i,gate}$ applied to the same input $\mathsf{ct}.\boldsymbol{x}_t$ and the four different weight matrices $\boldsymbol{W}_{h,gate}$ applied to the same $\mathsf{ct}.\boldsymbol{h}_{t-1}$, the entire for loop in Step 3 therefore consumes a total of $10\sqrt{m}$ rotations and $8m$ CMul operations. This part costs a multiplicative depth of 1, corresponding to a single layer of CMul operations.

If the Paterson-Stockmeyer algorithm is used to implement PolyEval then for a polynomial of degree d, the number of required ciphertext-ciphertext multiplications (Mul) is $\sqrt{2d} + \log d + O(1)$, and the number of required CMul is $d - \sqrt{d/2}$. However, the degrees of the approximation polynomials listed in Table 8 are relatively small, so we still prioritize minimizing the multiplicative depth. Specifically, for a cubic polynomial $P(X) = a_0 + a_1X + a_2X^2 + a_3X^3$, we can rewrite it as $P(X) = a_0 + a_1X + X^2(a_2 + a_3X)$, and evaluate it using only $\lceil \log d \rceil = 2$ multiplicative-depth layers, with 2 Muls and 2 CMuls. For the case of $d = 7$, we follow the same principle of keeping the multiplicative depth as small as possible. Let $P(X) = a_0 + a_1X + a_2X^2 + a_3X^3 + a_4X^4 + a_5X^5 + a_6X^6 + a_7X^7$. We first precompute a small set of ciphertext powers of X and then reuse them throughout the evaluation. In particular, we compute $X^2 = X \cdot X$ and $X^4 = X^2 \cdot X^2$. Using these precomputed powers, the polynomial can be evaluated in a factorized form, for example, $P(X) = (a_0 + a_1X + a_2X^2 + a_3X^3) + X^4(a_4 + a_5X + a_6X^2 + a_7X^3)$. As a result, the entire degree-7 polynomial can be evaluated within three multiplicative-depth layers, using only 5 Mul and 4 CMul operations.

Step 11 corresponds to the computation of $\boldsymbol{c}_t$ in Eq. (1). It requires two Mul operations and costs one level of multiplicative depth. Steps 12-15 implement the RMSNorm operation, where SumRows needs no multiplications but $\log m$ Rots. In total, Steps 12-17 require three additional Muls and two CMuls, together with the polynomial evaluations for the InvSqrt and Tanh_c functions (see Table 8). Based on the above analysis of polynomial evaluation, the total cost of Steps 12-17 amounts to 10 levels of multiplicative depth, 8 CMul operations, 10 Mul operations, and $\log m$ rotations (Rot).

The overall cost of Algorithm 1 is summarized in Table 12. As can be seen, each outer-loop iteration (for each time step t) requires a multiplicative depth of 15. When the sequence length T is large, bootstrapping must be invoked to preserve the non-interactive property of the protocol. The next section describes our strategy for scheduling these bootstrapping operations and presents the corresponding empirical performance.

Table 12: Cost of Algorithm 1.

Step	CMul	Mul	Rot	Mult Depth
Steps 3-6	$8m$	0	$10\sqrt{m}$	1
Steps 7-10	16	20	0	3
Step 11	0	2	0	1
Steps 12-17	8	10	$\log m$	10
Steps 3-17	$8m + 24$	32	$10\sqrt{m} + \log m$	15
Step 18	m	0	$2\sqrt{m}$	1
Total	$(8m + 24)T + m$	$32T$	$(10T + 2)\sqrt{m} + T \log m$	$15T + 1$

7. Experiments

In this section, we present the concrete implementation of the PPLSTM protocol and evaluate its practical performance on the datasets introduced in Section 2.3. We further compare our implementation with state-of-the-art schemes under the same experimental settings.

7.1. Implementation

We implement PPLSTM based on the Lattigo v6 library [15], which provides a high-performance protocol for the CKKS scheme [11]. The ring dimension is set to $N = 2^{15}$, and a conjugate-invariant variant [30] of CKKS is employed, which allows us to utilize all N plaintext slots for encoding real-valued data. The coefficient modulus Q is composed of a series of prime moduli with bit sizes of 55 and ten 40-bit primes, the special modulus P consists of three 61-bit primes, and the default scaling factor is set to $\Delta = 2^{40}$. For bootstrapping parameters, the ring dimension is increased to 2^{16} to meet the requirements of Lattigo v6, and the special modulus P is expanded to four 61-bit primes. Under this setup, our implementation can achieve a security of at least 128-bits, according to an HE standard [2] and the latest Lattice Estimator [3].

7.2. Evaluation

Setup. Experiments were conducted on a workstation equipped with an Intel(R) Xeon(R) Gold 6530 CPU (2.1GHz) and 128 GB of RAM with the Ubuntu 24.04 system. The experimental evaluation was conducted on four benchmark text classification datasets introduced in Section 2.3, and all reported results are averaged over five independent runs. For the IMDB and YELP datasets, the input sequence lengths were set to $T = 100$, 150, and 200, while for the AGNEWS and DBpedia datasets, the sequence lengths were set to $T = 30$, 50, and 70, respectively. The model architecture consists of a single-layer LSTM network with an embedding dimension of 100 and a hidden dimension of $m = 128$, yielding a batch size of $B = N/m = 256$. The input tokens are represented using 100-dimensional pretrained GloVe embeddings [39], which are kept fixed during training to provide stable semantic representations. The fully connected (output) layer contains 1, 1, 4, and 14 units for IMDB, YELP, AGNEWS, and DBPedia, respectively. In our implementation, the sequence length only affects the number of recurrent iterations, the peak memory and the time per iteration remains essentially unchanged across different sequence lengths.

Bootstrapping Management. As shown in Table 12, each inference round requires approximately 15 levels of multiplicative depth. Since both the cell state $\mathsf{ct}.\boldsymbol{c}_t$ (at the end of Step 15) and the hidden state $\mathsf{ct}.\boldsymbol{h}_t$ (at the end of Step 17) are propagated to the next iteration, they both must undergo bootstrapping to refresh the ciphertext noise budget. To determine the optimal position of the bootstrapping operation, we consider two alternative strategies. (1) End-of-round bootstrapping: This strategy postpones the bootstrapping of $\mathsf{ct}.\boldsymbol{c}_t$ and schedules it together with the bootstrapping of $\mathsf{ct}.\boldsymbol{h}_t$ to exploit multithreading. However, this is only a scheduling optimization; logically, the computation still requires two bootstrapping operations. It requires a modulus-chain depth of 15. (2) Mid-round bootstrapping: perform bootstrapping on $\mathsf{ct}.\boldsymbol{c}_t$ before its multiplication with the RMSNorm weights, followed by bootstrapping on $\mathsf{ct}.\boldsymbol{h}_t$. The second strategy requires two bootstrapping operations but allows a shorter modulus chain of 10. As shown in Table 13, an increased modulus chain not only prolongs the bootstrapping process itself but also amplifies the latency of basic operations such as addition, multiplication, and rotation. Experimental results show that the first method requires approximately 62.85 seconds per iteration and uses 30.37 GB of memory. In contrast, the second method reduces the time to 53.41 seconds and the memory to 24.64 GB due to a shorter modulus chain. Therefore, in this work, we adopt the *Mid-round* strategy, which achieves better computational efficiency.

Table 13: Breakdown runtime of different Bootstrapping strategy.

Step	End-of-round		Mid-round	
	Time	Depth	Time	Depth
Steps 3-6	26.67s	$15 \to 14$	17.16s	$10 \to 9$
Steps 7-10	1.52s	$14 \to 11$	829.24ms	$9 \to 6$
Step 11	55.24ms	$11 \to 10$	30.78ms	$6 \to 5$
Step 12	580.79ms	$10 \to 8$	246.20ms	$5 \to 3$
Step 13	82.09ms	$8 \to 6$	32.28ms	$3 \to 1$
Step 14	29.85ms	$6 \to 5$	8.56ms	$1 \to 0$
Bootstrapping	-	-	17.43s	$0 \to 10 \to 5$
Step 15	13.69ms	$5 \to 4$	11.41ms	$5 \to 4$
Step 16	88.07ms	$4 \to 1$	92.03ms	$4 \to 1$
Step 17	56.76ms	$1 \to 0$	31.23ms	$1 \to 0$
Bootstrapping	33.75s	$0 \to 15$	17.54s	$0 \to 10$
Total	62.85s	-	53.41s	-
Peak Memory	30.37GB	-	24.64GB	-

Although Table 13 suggests that a smaller initial depth (e.g., 8 or 9) might be sufficient based on the remaining levels after each step, such configurations are in fact not feasible when considering the dependency between ciphertexts in subsequent operations. In particular, if the initial depth is set to 8, the ciphertext $\mathsf{ct}.\hat{\boldsymbol{c}}$ would already drop to a low level (level 3) after Step 11. Although $\mathsf{ct}.\boldsymbol{t}$ can be refreshed by bootstrapping, the multiplication in Step 14 requires both operands to have sufficiently aligned and adequate levels. Due to the low level of $\mathsf{ct}.\hat{\boldsymbol{c}}$, the resulting ciphertext $\mathsf{ct}.\hat{\boldsymbol{c}}_t$ would not retain enough levels to support the remaining operations, including RMSNorm scaling and the subsequent Tanh evaluation. Therefore, a minimum initial depth of 10 is required to ensure that all intermediate ciphertexts maintain sufficient levels throughout the computation.

Required Keys and Peak Memory. We denote the hidden dimension by m and the batch size by B (where $B = N/m$ and N is the number of ciphertext slots). The required rotation keys consist of two parts: BSGS rotations $\{j \cdot B \mid 1 \le j < k\} \cup \{i \cdot n_1 \cdot B \mid 1 \le i < \ell\}$ and SumRows rotations $\{\pm B \cdot 2^i \mid 0 \le i \le \lfloor \log_2(\lfloor m/2 \rfloor) \rfloor\}$. For BSGS, we set $k = \lceil \sqrt{m} \rceil$ for baby-step and $\ell = \lceil m/k \rceil$ for giant-step. The analysis of the distinct key-switching-key usage as shown in Table 14. Peak memory additionally includes ciphertexts and temporary buffers during evaluation.

Table 14: Key-switching key and peak memory usage.

m	B	Relin key	Rot key	Bts key	Peak Memory
64	512	28 MB	$20 \cdot 28$ MB	10.66 GB	24.05 GB
128	256	28 MB	$31 \cdot 28$ MB	10.66 GB	24.64 GB

Relin:Relinearization; Rot:Rotation; Bts:Bootstrapping

Table 15: Performance of PPLSTM on various datasets.

Dataset	Length	PT(%)	PT*(%)	CT(%)	ERR(%)	ET(s)
IMDB	100	88.28	89.06	89.06	0.98	21.26
	150	88.67	87.50	87.11	−1.56	31.73
	200	88.67	88.28	88.28	−0.39	41.73
YELP	100	97.27	96.48	96.48	-0.79	21.18
	150	96.88	96.48	96.48	−0.40	31.57
	200	96.88	95.31	94.53	−2.35	42.07
AGNEWS	30	92.97	92.97	92.97	0.00	6.38
	50	94.53	94.53	94.53	0.00	10.69
	70	92.97	93.36	93.36	0.39	14.64
DBPedia	30	99.61	99.61	99.61	0.00	6.35
	50	98.44	98.83	98.44	0.00	10.70
	70	99.22	98.44	98.05	−1.17	14.40

PT, PT* and CT denote the classification accuracy of the LSTM with RMSNorm under plaintext, plaintext with polynomial approximation and encrypted inference settings, respectively, both evaluated with a single batch of size 256. ERR is the rate difference of PT and CT, and ET is the average time spent on a sequence.

Performance. As shown in Table 15, the accuracy difference between the plaintext LSTM and the encrypted LSTM based on the CKKS scheme is within 2.35% across all datasets and sequence lengths. In several cases, the encrypted model slightly outperforms the original plaintext model (the column PT in Table 15). This behavior is mainly attributed to the polynomial approximation used to replace nonlinear activation functions, since the column PT* is no less than the column CT. For a sequence length of 100, the inference time per sample is approximately 22 seconds, demonstrating the feasibility of performing practical inference over encrypted data. Moreover, Table 15 shows that the running time of PPLSTM grows linearly

with the sequence length T, which is consistent with the analysis in Section 6.2.

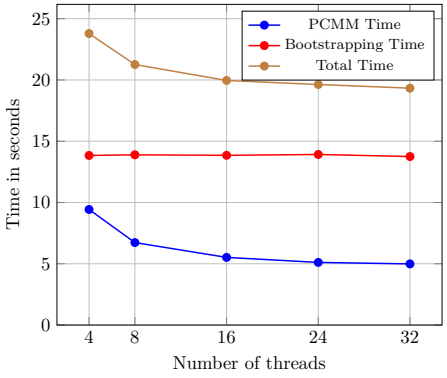

Figure 7: Running time for a sequence length of 100 vs #threads.

To further evaluate protocol efficiency, we benchmarked encrypted inference using 4, 8, 16, 24, and 32 threads on sequences of length 100. As shown in Fig. 7, the corresponding runtimes were approximately 23.79, 21.26, 19.96, 19.63, and 19.33 seconds per sample, respectively. These results indicate that increasing the number of threads provides speedup, primarily because parallelism accelerates the PCMM algorithm in the gate computations. However, the overall acceleration remains limited since the bootstrapping procedure—being inherently sequential and computationally dominant—cannot benefit from multithreading. This observation highlights a fundamental bottleneck in current FHE-based recurrent models. Future work may explore optimized bootstrapping algorithms, lazy or selective bootstrapping strategies, and hardware-level acceleration to reduce latency further and enable larger-scale encrypted sequence models.

Compared with Trama et al. [45]. The TFHE approach [45] demonstrates the feasibility of fully homomorphic LSTM inference by using discretization and functional bootstrapping with large lookup tables for the Sigmoid and Tanh activation functions. However, its computational cost remains high. Based on the reported numbers, evaluating an encrypted LSTM with a hidden dimension of 64 takes approximately 6 seconds per time step on 8 threads. In contrast, our work is implemented using the approximate-number CKKS scheme, which supports SIMD packing and vectorized arithmetic at low amortized cost. PPLSTM achieves an amortized time of 0.09 seconds with 8 threads for an LSTM with a hidden dimension of 64 in a single time step.

Compared with Zhang et al. [57]. To further evaluate the performance of our proposed encrypted LSTM protocol, we compare it with PrivLSTM proposed in [57]. In PrivLSTM, the model configuration uses an embedding dimension of 40 and a hidden dimension of $m = 50$. For a fair comparison, we adopt a similar configuration with an embedding dimension of 40, increasing the hidden dimension to $m = 64$ and the batch size to $B = 2^{15}/64 = 512$ to fully utilize the available ciphertext slots in the CKKS encoding. Since PrivLSTM adopts the integration method described in Section 5.1, the BSGS procedure requires a total of 21 rotation keys under this parameter setting. In comparison, our method requires 20 rotation keys in total (see Table 14), including the five additional rotation keys needed for the RMSNorm computation. Detailed comparisons in terms of accuracy and runtime are reported in Table 16.

The PrivLSTM results reported in Table 16 are borrowed from [57]. Their implementation is also based on the Lattigo library (v4), and their experiments were conducted on a PC equipped with an Intel i9-12900K CPU (3.2 GHz) and 128 GB RAM. To ensure a fair comparison, we reran all relevant experiments on a PC with the same CPU model, namely an Intel i9-12900K (3.2 GHz), but with a smaller memory capacity of 32 GB RAM. As illustrated in Table 16, our method achieves slightly higher precision compared to PrivLSTM, while demonstrating a 79% improvement (or equivalently a 4.7x speedup)[2] with 8 threads in the inference speed. The performance gains over PrivLSTM arise from improvements in both the plaintext model design and the encrypted execution strategy.

- From the plaintext perspective, our method introduces an RMSNorm module into the standard LSTM architecture, which consistently improves classification accuracy across all four datasets, as shown in Table 6. In contrast, PrivLSTM is built upon a standard LSTM without normalization. Therefore, part of the accuracy improvement can be attributed to the enhanced model architecture itself. To make the modified model compatible with homomorphic evaluation, Section 4 further introduces a hybrid Remez and least-squares polynomial approximation strategy to efficiently approximate the newly introduced non-polynomial functions while maintaining high inference accuracy with low polynomial degree.

- From the encrypted-system perspective, our method further improves inference efficiency through two optimizations. First, compared with the Integration packing strategy adopted in PrivLSTM, we employ a no-Integration ciphertext packing scheme, which reduces the costs of evenly distributed ciphertext multiplications and rotations, as demonstrated in Table 11. Sec-

[2]The implementation of PrivLSTM in [57] is reportedly based on Lattigo v4, whereas our implementation uses Lattigo v6.1. To assess the potential impact of this version difference, we benchmarked CKKS bootstrapping under the Lattigo parameter set `N16QP1546H192H32`. The bootstrapping time is 18.60 s in Lattigo v4.1 and 17.54 s in Lattigo v6.1, corresponding to only a modest improvement of about 1.06x. If this factor were taken into account, the speedup would be 4.65x. This suggests that the performance gains reported in Table 16 mainly come from our algorithmic optimizations.

Table 16: Overall performance comparison between ours and PrivLSTM. PrivLSTM: Data are taken from [57] (Intel i9-12900K CPU (3.2 GHz) with 128 GB RAM); Ours-1: Intel Xeon Gold 6530 CPU (2.1 GHz) with 128 GB RAM; Ours-2: Intel i9-12900K CPU (3.2 GHz) with 32 GB RAM.

(a) IMDB and YELP datasets (T = 100, 150, 200)

Dataset	Scheme	Accuracy (%)			Computation Time (s)		
		100	150	200	100	150	200
IMDB	PrivLSTM [57]	85.06	85.74	86.52	30.66	45.93	61.22
	Ours-1	87.11	88.09	85.74	8.99	13.56	17.96
	Improvment-1	+2.05%	+2.35%	-0.78%	-70.68%	-70.48%	-70.66%
	Ours-2	87.11	88.09	85.74	5.92	9.26	11.61
	Improvment-2	+2.05%	+2.35%	-0.78%	-80.69%	-79.84%	-81.04%
YELP	PrivLSTM [57]	92.87	94.14	93.16	30.73	45.91	65.47
	Ours-1	95.31	95.31	94.14	9.08	13.48	18.05
	Improvment-1	+2.44%	+1.17%	+0.98%	-70.45%	-70.64%	-72.43%
	Ours-2	95.31	95.31	94.14	5.88	9.30	12.40
	Improvment-2	+2.44%	+1.17%	+0.98%	-81.35%	-79.75%	-79.75%

(b) AGNEWS and DBPedia datasets (T = 30, 50, 70)

Dataset	Scheme	Accuracy (%)			Computation Time (s)		
		30	50	70	30	50	70
AGNEWS	PrivLSTM [57]	89.16	89.94	90.14	9.20	16.46	21.51
	Ours-1	91.60	94.92	91.41	2.76	4.58	6.28
	Improvement-1	+1.44%	+4.98%	+1.27%	-70.00%	-72.17%	-70.80%
	Ours-2	91.60	94.92	91.41	1.71	3.07	4.36
	Improvement-2	+1.44%	+4.98%	+1.27%	-81.04%	-81.35%	-79.73%
DBPedia	PrivLSTM [57]	87.60	91.70	93.26	9.25	16.40	21.52
	Ours-1	99.41	98.05	97.27	2.74	4.75	6.23
	Improvement-1	+11.81%	+6.35%	+4.01%	-70.38%	-71.04%	-71.05%
	Ours-2	99.41	98.05	97.27	1.86	3.05	4.23
	Improvement2	+11.81%	+6.35%	+4.01%	-79.78%	-81.47%	-80.33%

ond, while PrivLSTM implicitly performs one bootstrapping operation at the end of each sequence step, as can be inferred from Table 3 of the original paper where the number of bootstrapping operations equals the sequence length, we adopt a mid-round bootstrapping strategy. As shown in Table 13, this strategy further reduces the overall computational overhead.

Overall, the experimental results across multiple datasets demonstrate that our PPLSTM protocol achieves a favorable trade-off between accuracy and efficiency, maintaining near-plaintext performance while significantly accelerating encrypted inference compared to existing approaches.

8. Conclusion and Discussion

In this paper, we present and implement an accurate, efficient, non-interactive LSTM inference protocol, PPLSTM, built upon the CKKS homomorphic encryption scheme. PPLSTM incorporates three major optimizations. First, for nonlinear activations, it introduces RMSNorm to improve approximation accuracy and computational efficiency simultaneously. Second, it employs a carefully designed batching and packing strategy, together with the SIMD capabilities of CKKS and a BSGS-based implementation, to optimize plaintext-ciphertext matrix multiplication. Third, it strategically places bootstrapping operations to minimize overall latency while preserving the correctness of encrypted LSTM inference. As a result, the proposed protocol not only narrows the performance gap between encrypted and plaintext inference but also demonstrates that LSTM architectures can be efficiently reimplemented using HE schemes.

Finally, we note that the time complexity of LSTM grows linearly with the sequence length T, which constitutes a fundamental performance bottleneck in both plaintext and encrypted inference. This limitation has, in part, motivated the development of more parallelizable architectures such as Transformers [47]. Nevertheless, PPLSTM remains relevant in several important respects. Many real-world systems, especially in healthcare, finance, and time-series forecasting, still rely heavily on LSTM models and are difficult to migrate to Transformer-based architectures. In addition, LSTM may still offer advantages over Transformers in scenarios involving limited training data, real-time prediction, or resource-constrained deployment. From the perspective of fully homomorphic encryption (FHE), LSTM also provides a convenient and rep-

resentative testbed for analyzing the depth and computational characteristics of encrypted recurrent structures and serves as a natural step toward supporting more advanced architectures such as Mamba [19].

Acknowledgments. This work was partially supported by the National Key Research and Development Program of China (2025YFA1017201) and the National Natural Science Foundation of China (12571553). The authors would like to thank the anonymous reviewers for their valuable comments and suggestions, which have helped improve the clarity and quality of this paper significantly.

Statement. During the preparation of this work, the authors used ChatGPT and Gemini for language polishing and auxiliary coding. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the article.

References

[1] A. Al Badawi, A. Alexandru, Y. Polyakov, and V. Vaikuntanathan. SoK: Private LLM inference using approximate homomorphic encryption, 2026. https://eprint.iacr.org/2026/935.

[2] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. *Homomorphic Encryption Standard*, pages 31–62. Springer, Cham, 2021. https://doi.org/10.1007/978-3-030-77287-1_2.

[3] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015. https://doi.org/10.1515/jmc-2015-0016. Lattice Estimator: https://github.com/malb/lattice-estimator.

[4] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization, 2016. https://arxiv.org/abs/1607.06450.

[5] R. Gilad-Bachrach, N. Dowlin, K. Laine, et al. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML '16*, pages 201–210. PMLR, New York, 2016. http://proceedings.mlr.press/v48/gilad-bachrach16.html.

[6] Y. Bae, J. H. Cheon, G. Hanrot, J. H. Park, and D. Stehlé. Plaintext-ciphertext matrix multiplication and FHE bootstrapping: Fast and fused. In *CRYPTO '24*, volume 14922 of *LNCS*, pages 387–421. Springer, Cham, 2024. https://doi.org/10.1007/978-3-031-68382-4_12.

[7] M. Bakshi and M. Last. CryptoRNN - privacy-preserving recurrent neural networks using homomorphic encryption. In *CSCML '20*, volume 12161 of *LNCS*, pages 245–253. Springer, 2020. https://doi.org/10.1007/978-3-030-49785-9_16.

[8] M. Bian, G. He, G. Feng, X. Zhang, and Y. Ren. Verifiable privacy-preserving heart rate estimation based on lstm. *IEEE Internet of Things Journal*, PP:1–1, 01 2023. https://doi.org/10.1109/JIOT.2023.3290651.

[9] Y. Chen, T. Lu, Z. Tang, B. Zhang, Z. Shi, Y. Luan, and Z. Wang. SoK: Private transformer-based model inference, 2026. https://eprint.iacr.org/2026/491.

[10] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. Bootstrapping for approximate homomorphic encryption. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT '18*, volume 10820 of *LNCS*, pages 360–384. Springer, Cham, 2018. https://doi.org/10.1007/978-3-319-78381-9_14.

[11] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT 2017*, volume 10624 of *LNCS*, pages 409–437. Springer, Heidelberg, 2017. https://doi.org/10.1007/978-3-319-70694-8_15.

[12] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020. https://doi.org/10.1007/s00145-019-09319-x.

[13] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *EMNLP '14*, pages 1724–1734. ACL, 10 2014.

[14] T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville. Recurrent batch normalization. In *ICLR '17*, 2017. https://openreview.net/forum?id=r1VdcHcxx.

[15] EPFL-LDS and Tune-Insight. Lattigo v6. Online: https://github.com/tuneinsight/lattigo, 2024.

[16] B. Feng, Q. Lou, L. Jiang, and G. Fox. CRYPTOGRU: Low latency privacy-preserving text analysis with GRU. In *EMNLP '21*, pages 2052–2057. ACL, 11 2021. https://doi.org/10.18653/v1/2021.emnlp-main.156.

[17] X. Gao, S. Fu, L. Liu, Z. Liu, Y. Luo, and Y. Wang. Euston: Efficient and user-friendly secure transformer inference with non-interactivity. In *IEEE S&P*, pages

882–901. IEEE, 2026. https://doi.org/10.1109/SP63933.2026.00048.

[18] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *STOC '09*, pages 169–178. ACM, New York, 2009. https://doi.org/10.1145/1536414.1536440.

[19] A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024. https://arxiv.org/abs/2312.00752.

[20] S. Halevi and V. Shoup. Algorithms in HElib. In *CRYPTO '14*, volume 8616 of *LNCS*, pages 554–571. Springer, Heidelberg, 2014. https://doi.org/10.1007/978-3-662-44371-2_31.

[21] S. Halevi and V. Shoup. Bootstrapping for HElib. In *EUROCRYPT '15*, volume 9056 of *LNCS*, pages 641–670. Springer, Heidelberg, 2015. https://doi.org/10.1007/978-3-662-46800-5_25.

[22] X. He, G. Xu, X. Han, Q. Wang, L. Zhao, C. Shen, C. Lin, Z. Zhao, Q. Li, L. Yang, S. Ji, S. Li, H. Zhu, Z. Wang, R. Zheng, T. Zhu, Q. Li, C. He, Q. Wang, H. Hu, S. Wang, S.-F. Sun, H. Yao, Z. Qin, K. Chen, Y. Zhao, H. Li, X. Huang, and D. Feng. Artificial intelligence security and privacy: a survey. *Science China Information Sciences*, 68(8):181101, 2025. https://doi.org/10.1007/s11432-025-4388-5.

[23] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. https://doi.org/10.1162/neco.1997.9.8.1735.

[24] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML'15*, pages 448–456. PMLR, New York, 2015. http://proceedings.mlr.press/v37/ioffe15.pdf.

[25] J. Jang, Y. Lee, A. Kim, B. Na, D. Yhee, B. Lee, J. H. Cheon, and S. Yoon. Privacy-preserving deep sequential model with matrix homomorphic encryption. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 377–391, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3488932.3523253.

[26] X. Jiang, M. Kim, K. Lauter, and Y. Song. Secure outsourced matrix computation and application to neural networks. In *CCS '18*, pages 1209–1222. ACM, New York, 2018. https://doi.org/10.1145/3243734.3243837.

[27] J. H. Ju, J. Park, J. Kim, M. Kang, D. Kim, J. H. Cheon, and J. H. Ahn. NeuJeans: Private neural network inference with joint optimization of convolution and FHE bootstrapping. In *CCS '24*, pages 4361–4375. ACM, New York, 2024. https://doi.org/10.1145/3658644.3690375.

[28] S. Kariyappa and M. K. Qureshi. Defending against model stealing attacks with adaptive misinformation. In *CVPR*, pages 767–775. 2020. https://doi.org/10.1109/CVPR42600.2020.00085.

[29] D. Kim and C. Guyot. Optimized privacy-preserving CNN inference with fully homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 18:2175–2187, 2023. https://doi.org/10.1109/TIFS.2023.3263631.

[30] D. Kim and Y. Song. Approximate homomorphic encryption over the conjugate-invariant ring. In *ICISC 2018*, volume 11396 of *LNCS*, pages 85–102. Springer, Cham, 2019. https://doi.org/10.1007/978-3-030-12146-4_6.

[31] Y. Lee, J. Seo, Y. Nam, J. Chae, and J. H. Cheon. Heaan-stat: A privacy-preserving statistical analysis toolkit for large-scale numerical, ordinal, and categorical data. *IEEE Transactions on Dependable and Secure Computing*, 21(3):1224–1241, 2024. https://doi.org/10.1109/TDSC.2023.3275649.

[32] C. Li, X. Ma, X. Wang, F. Kong, Y. Tao, and C. Ge. Optimized homomorphic linear computation in privacy-preserving CNN inference. *Expert Systems with Applications*, 298:129767, 2026. https://doi.org/10.1016/j.eswa.2025.129767.

[33] Y. Liu, L. Yang, J. Chen, W. Wu, and Y. Feng. Matrix computation over homomorphic plaintext-ciphertext and its application. *Journal on Communications*, 45(2):150–161, 2024. in Chinese, https://doi.org/10.11959/j.issn.1000-436x.2024024.

[34] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *Journal of ACM*, 60(6):43:1–35, 2013. https://doi.org/10.1145/2535925.

[35] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. Learning word vectors for sentiment analysis. In D. Lin, Y. Matsumoto, and R. Mihalcea, editors, *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[36] J. Moon, D. Yoo, X. Jiang, and M. Kim. THOR: Secure transformer inference with homomorphic encryption. In *CCS '25*, pages 3765 – 3779. ACM, New York, 2025. https://doi.org/10.1145/3719027.3765150.

[37] D. Park, E. Lee, and J.-W. Lee. Powerformer: Efficient privacy-preserving transformer with batch rectifier-power max function and optimized homomorphic attention. In *ACL '25*. ACL, 2025. https://ia.cr/2024/1429.

[38] M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973. https://doi.org/10.1137/0202007.

[39] J. Pennington, R. Socher, and C. D. Manning. GloVe: Global vectors for word representation. In *EMNLP '14*, pages 1532–1543. ACL, 2014. https://doi.org/10.3115/v1/D14-1162.

[40] R. Podschwadt and D. Takabi. Non-interactive privacy preserving recurrent neural network prediction with homomorphic encryption. In *CLOUD '21*, pages 65–70, 2021. https://doi.org/10.1109/CLOUD53861.2021.00019.

[41] H. Qu and G. Xu. Improvements of homomorphic secure evaluation of inverse square root. In *ICICS '23*, volume 14252 of *LNCS*, pages 110–127. Springer, Singapore, 2023. https://doi.org/10.1007/978-981-99-7356-9_7.

[42] E. Y. Remez. Sur le calcul effectif des polynomes d'approximation de Tschebyschef. *Compt. Rend. Acad. Sci.*, 199:337–340, 1934.

[43] D. Sthel'e. Personel communication, 2025.

[44] M. Tang, A. Dai, L. DiValentin, A. Ding, A. Hass, N. Z. Gong, Y. Chen, and H. H. Li. ModelGuard: Information-theoretic defense against model extraction attacks. In *USENIX Security '24*, pages 5305–5322. USENIX Association, Philadelphia, 2024. https://www.usenix.org/system/files/usenixsecurity24-tang.pdf.

[45] D. Trama, P.-E. Clet, A. Boudguiga, and R. Sirdey. Building blocks for LSTM homomorphic evaluation with TFHE. In S. Dolev, E. Gudes, and P. Paillier, editors, *CSCML '23*, pages 117–134, Cham, 2023. Springer. https://doi.org/10.1007/978-3-031-34671-2_9.

[46] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction APIs. In *Proceedings of the 25th USENIX Conference on Security Symposium (August 10–12, 2016, Austin, TX, USA)*, pages 601–618. USENIX Association, USA, 2016. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_tramer.pdf.

[47] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS '17*. Curran Associates, Inc., New York, 2017. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[48] Z. Wang and M. Ikeda. High-throughput privacy-preserving GRU network with homomorphic encryption. In *IJCNN '23*, pages 1–9, 2023. https://doi.org/10.1109/IJCNN54540.2023.10191194.

[49] Z. Wang and M. Ikeda. Toward bootstrapping-free homomorphic encryption-based GRU network for text classification. *IEEE Access*, 12:94008–94017, 2024. https://doi.org/10.1109/ACCESS.2024.3422455.

[50] L. Yang, J. Chen, W. Dai, S. Wang, W. Wu, and Y. Feng. ARION: Attention-optimized transformer inference on encrypted data, 2025. https://eprint.iacr.org/2025/2271.

[51] A. C. Yao. Protocols for secure computations. In *SFCS '82*, page 160–164. IEEE, 1982.

[52] B. Zhang and R. Sennrich. Root mean square layer normalization. In *NIPS '19*. Curran Associates Inc., Red Hook, 2019. https://doi.org/10.5555/3454287.3455397.

[53] J. Zhang, X. Yang, L. He, K. Chen, W.-J. Lu, Y. Wang, X. Hou, J. Liu, K. Ren, and X. Yang. Secure transformer inference made non-interactive. In *NDSS '25*. The Internet Society, Fredericksburg, 2025. https://dx.doi.org/10.14722/ndss.2025.230868.

[54] L. Zhang, X. Wang, J. J. Sim, Z. Huang, J. Zhong, H. WANG, P. Duan, and K.-Y. Lam. MOAI: Module-optimizing architecture for non-interactive secure transformer inference. In *ICLR*. OpenReview, 2026. https://openreview.net/forum?id=qJn4HtTzhH.

[55] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 649–657, Cambridge, MA, USA, 2015. MIT Press.

[56] Y. Zhang, J. Xue, M. Zheng, M. Xie, M. Zhang, L. Jiang, and Q. Lou. CipherPrune: Efficient and scalable private transformer inference. In *ICLR'25*. OpenReview.net, 2025. https://openreview.net/forum?id=mUMvr33FTu.

[57] Z. Zhang, K. Shao, R. Deng, X. Wang, Y. Zhang, and M. Wang. PrivLSTM: A privacy-preserving LSTM inference framework by fusing encryption and network structure for multi-sourced data. *Information Fusion*, 127:103711, 2025. https://doi.org/10.1016/j.inffus.2025.103711.