SFPDML: Securer and Faster Privacy-Preserving Distributed Machine Learning Based on MKTFHE

Hongxiao Wang[1], Zoe L. Jiang[2,3], Yanmin Zhao[1], Siu-Ming Yiu[1]*, Peng Yang[2], Man Chen[4], Zejiu Tan[2], and Bohan Jin[2]

[1] University of Hong Kong, Hong Kong, China
{hxwang, ymzhao, smyiu}@cs.hku.hk
[2] Harbin Institute of Technology, Shenzhen, Shenzhen, China
zoeljiang@hit.edu.cn
{stuyangpeng, 23s151118, 23s051024}@stu.hit.edu.cn
[3] Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, Shenzhen, China
[4] Shandong University, Jinan, China
chenman19961121@gamil.com

Abstract

In recent years, distributed machine learning has garnered significant attention. However, privacy continues to be an unresolved issue within this field. Multi-key homomorphic encryption over torus (MKTFHE) is one of the promising candidates for addressing this concern. Nevertheless, there may be security risks in the decryption of MKTFHE. Moreover, to our best known, the latest works about MKTFHE only support Boolean operation and linear operation which cannot directly compute the non-linear function like Sigmoid. Therefore, it is still hard to perform common machine learning such as logistic regression and neural networks in high performance. In this paper, we first discover a possible attack on the existing distributed decryption protocol for MKTFHE and subsequently introduce secret sharing to propose a securer one. Next, we design a new MKTFHE-friendly activation function via *homogenizer* and *compare quads*. Finally, we utilize them to implement logistic regression and neural network training in MKTFHE. Comparing the efficiency and accuracy between using Taylor polynomials of Sigmoid and our proposed function as an activation function, the experiments show that the efficiency of our function is 10 times higher than using 7-order Taylor polynomials straightly and the accuracy of the training model is similar to using a high-order polynomial as an activation function scheme.

Keywords: Privacy-Preserving Machine Learning, Multi-Key Fully Homomorphic Encryption, Multi-Key Decryption, Distributed Machine Learning

1 Introduction

In the big data era, it is necessary to transform centralized systems into distributed ones in machine learning tasks. However, these distributed systems lead to new challenges, and one of the most pressing is privacy [1, 2].

Privacy computing is a technique that enables data computation without any risk of information leakage. To outsource private computations, fully homomorphic encryption (FHE), a cryptographic tool, is employed. FHE is a unique form of encryption that allows users to perform computations on encrypted data without the need to first decrypt it. FHE can be divided

The 7th International Conference on Mobile Internet Security (MobiSec'23), Dec. 19-21, 2023, Okinawa, Japan, Article No. 34
*Corresponding author: University of Hong Kong, Hong Kong, China, Email: smyiu@cs.hku.hk

into two categories: *single-key* fully homomorphic encryption and *multi-key* fully homomorphic encryption.

Single-key FHE only allows a server to perform addition and multiplication on data encrypted by the same key. In contrast, *multi-key* FHE (MKFHE) proposed in [3] enables users to encrypt their own data under their own keys, but during the decryption of MKFHE, all secret keys of all participants are used. It prevents conspiracy between a user and a server to steal the data of other users.

In recent years, multi-key fully homomorphic encryption over the torus (MKTFHE) has attracted significant attention from researchers, particularly in the areas of evaluation and decryption algorithms. Chen et al. [4] developed a library for implementing MKTFHE, focusing on an evaluation algorithm that takes a NAND gate as input. Subsequently, Jiang et al. [5] expanded the evaluation algorithm to include arithmetic operators such as adders, subtracters, multipliers, and dividers, enabling linear multi-key homomorphic arithmetic evaluation in MKTFHE. However, the inability to evaluate non-linear operations, like the Sigmoid function, restricts the direct application of more complex machine learning schemes such as logistic regression and neural networks.

To overcome this limitation, we borrow the SecureML [6] idea of replacing the non-linear activation function with a piecewise function (MKTFHE-friendly activation function). During the evaluation of the activation function, operands are transformed into Booleans and evaluated using Boolean operations. However, implementing piecewise functions in the FHE scheme is challenging due to their discontinuous nature and the need for input ciphertext comparisons with each bound. Instead of the online interactive comparison utilized in SecureML, we propose a simpler method: We develop *compare quads* to *select* ciphertext ct_c and ct_d by comparing messages between ciphertext ct_a and ct_b. This is achieved by subtracting ct_a and ct_b, extracting the most significant bit (MSB) of the result, and using the MSB to *select* the appropriate value through homomorphic evaluation of the equation $\mathsf{MSB} \wedge ct_c + \neg\mathsf{MSB} \wedge ct_d$. Our *compare quads* supports more complex branching programs and enables the implementation of the SecureML-like piecewise function with only two *compare quads*.

Concerning the decryption algorithm, Chen et al. [4] provided a naive decryption algorithm for the original MKTFHE, which requires all users' secret keys as input. In practical scenarios, access to others' secret keys during decryption should be restricted. To address this issue, Lee et al. [7] proposed a distributed decryption algorithm that separates the decryption process into two sub-algorithms: partial decryption and final decryption. Each user employs their secret key for partial decryption, ensuring that no user has access to others' secret keys. However, we discover some *possible attacks* on the distributed decryption algorithm, described in Appendix A.

Why the previous decryption is not secure. In brief, the existing MKTFHE scheme leaks information about user u_i's secret key s_i when provided with ciphertext and partial decryption. Suppose an MKTLWE ciphertext $(\mathbf{a}_1, \ldots, \mathbf{a}_k, b)$, with $b = \frac{1}{4}m - \sum_{i=1}^{k}\langle \mathbf{a}_i, \mathbf{s}_i\rangle + e$, k as the number of users, m as a one-bit message, and e as an error to randomize b. With the partial decryption $p_i = b + \langle \mathbf{a}_i, \mathbf{s}_i\rangle$ and ciphertext b, one can obtain $\langle \mathbf{a}_i, \mathbf{s}_i\rangle$ via their subtraction which may leak information about $\mathbf{s}_i$. Besides, an external adversary obtaining all partial decryption results can finish the final multi-key encryption results alone by computing $\sum_{i=1}^{k} p_i - (k-1)\, b$.

To address these security concerns, we introduce additive secret sharing to protect partial decryption and facilitate final decryption. Each participant performs partial decryption using their secret key and shares the result with decryption parties. Secret sharing is then employed to complete the final decryption, preventing internal adversaries from accessing others' secret keys and external adversaries from obtaining partial decryption results. Since only additional

operations are involved, decryption parties do not need to interact, making our decryption protocol secure against both internal and external adversaries while maintaining a similar cost to the original algorithm.

Finally, we combine these advancements to propose our Secure and Faster Privacy-Preserving Distributed Machine Learning (SFPDML) scheme, applying it to train and predict logistic regression models and neural networks using the Iris dataset.

Our contributions can be summarized as follows:

1. We develop a secure distributed decryption protocol for MKTFHE by introducing a secret sharing scheme, addressing the information leakage problem. We define our security goal for MKTFHE against possible static adversaries, then prove the correctness and analyze the security of our protocol. Experimental results demonstrate acceptable performance for the overall scheme and achieve similar performance to the original algorithm when the number of participants increases.

2. We apply our SFPDML scheme to train and evaluate logistic regression and neural network models. We design a *homogenizer* to modify the bit length of operands, allowing the use of fewer-bit operators to reduce operation time. To implement a new MKTFHE-friendly activation function, we develop *compare quads* for comparing and selecting input ciphertext. Experimental results show that our function's efficiency is 10 times higher than using 7-order Taylor polynomials directly, and the accuracy of the trained model is similar to that of high-order polynomial activation function schemes.

2 Related Work

There are numerous works on privacy-preserving machine learning prediction [8–12] and training [13–15]. These solutions are based on single-key FHE which cannot support the data participants using different secret keys to encrypt their own data. The prediction can support more complex models such as logistic regression and even neural networks in the second level, but the training only focuses on simpler models such as logistic regression in the hour level or higher. Besides, we also note that there are other approaches based on MPC, e.g. [6, 16] and compared with the above works based on FHE, the performances of the solutions based on MPC are very impressive. But they need interactivity between the data participants and the computation parties which may lead to many problems such as network latency or high bandwidth usage. Considering the above downsides, we focus on FHE, especially multi-key FHE.

The concept of multi-key fully homomorphic encryption was first proposed by Lopez et al. [3], which is intended to apply to on-the-fly multiparty computation based on NTRU. Then first LWE-based MKFHE was constructed by Clear et al. [17], and later, was improved by Mukherjee and Wichs [18]. These schemes are single-hop MKFHE schemes, which means all the participants must be known in advance. The multi-hop MKFHE schemes were proposed by Peikert et al. [19] and Brakerski et al. [20], but their schemes are impractical and without implementation.

The first implementation of the MKFHE scheme was achieved by Chen et al. [4], named MKTFHE which is the variant of TFHE [21–23]. Their scheme only provided a bootstrapped NAND gate to evaluate. Then, Lee and Park [7] first formalized the distributed decryption for MKFHE and improved the decryption part of MKTFHE, but a passive adversary still can recover the decryption result through the partial results. Then Jiang et al. [5] designed other bootstrapped gates, utilized them to build arithmetic operators including adder, subtractor,

multiplier, and divider, and then implemented a privacy-preserving linear regression in the GD method.

However, only using arithmetic operators cannot directly compute the non-linear activation function like the Sigmoid function. So, there is still a gap between MKTFHE and the implementation of more complex privacy-preserving machine learning such as logistic regression and neural networks.

3 Preliminaries

Notation: In the rest of this paper, $\mathbb{R}$ denotes the real numbers, $\mathbb{Z}$ denotes the integers, and $\mathbb{T}$ indicates $\mathbb{R}/\mathbb{Z}$, the torus of real numbers modulo 1. We use TLWE to denote the (scalar) binary Learning With Error problem over Tours, and TRLWE for the ring mode. We define *params* as the parameter set in TFHE, *mkparams* in MKTFHE, and our scheme. Besides, k is used to represent the number of participants in MKTFHE, l is used to represent the bit length of a message or ciphertext, and $ct[l]$ is denoted as l-bit ciphertext in MSB order. Then we use bold letters, e.g. $\boldsymbol{a}$, to denote vector and use $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ to represent the inner product between vector $\boldsymbol{a}$ and vector $\boldsymbol{b}$.

3.1 Multi-key Fully Homomorphic Encryption over Torus

MKTFHE scheme is the multi-key version of TFHE scheme [21–23]. In the MKTFHE scheme, the ciphertext length increases linearly with the number of users, and a homomorphic NAND gate with bootstrapping is given. The MKTFHE scheme is comprised of the following algorithms:

- $mkparams \leftarrow \mathsf{MKTFHE.SETUP}\left(1^{\lambda}\right)$: Take a security parameter λ as input and output the public parameter set $mkparams$.
- $\{sk_i, pk_i\} \leftarrow \mathsf{MKTFHE.KEYGEN}\left(mkparams\right)$: Take the $mkparams$ as input, and output secret key sk_i and public key pk_i for a single participant i.
- $ct \leftarrow \mathsf{MKTFHE.ENC}\left(\mu\right)$: Encrypt an input bit $\mu \in \{0,1\}$ and output a TLWE ciphertext with the scaling factor $\frac{1}{4}$. The output ciphertext $ct = (\boldsymbol{a}, b) \in \mathbb{T}^{n+1}$, satisfing $b + \langle \boldsymbol{a}, \boldsymbol{s} \rangle \approx \frac{1}{4}\mu$.
- $\mu \leftarrow \mathsf{MKTFHE.DEC}\left(ct, \{sk_i\}_{i\in[k]}\right)$: Input a TLWE ciphertext $ct = (\boldsymbol{a_i}, \cdots, \boldsymbol{a_k}, b) \in \mathbb{T}^{kn+1}$ and a set of secret keys $\{sk_i\}_{i\in[k]}$, and output the message $\mu \in \{0,1\}$ which satisfies $b + \sum_{i=1}^{k} \langle \boldsymbol{a_i}, sk_i \rangle \approx \frac{1}{4}\mu \left(\mod 1\right)$.
- $ct \leftarrow \mathsf{MKTFHE.NAND}\left(ct_1, ct_2\right)$: Input two TLWE ciphertext $ct_1 = \mathsf{MKTFHE.\ ENC}\left(\mu_1\right) \in \mathbb{T}^{n+1}$, $ct_2 = \mathsf{MKTFHE.ENC}\left(\mu_2\right) \in \mathbb{T}^{n+1}$, where ct_1, ct_2 can be constructed by different participants respectively, and output the multi-key ciphertext result $ct = \mathsf{MKTFHE.ENC}\left(\mu_1 \oplus \mu_2\right) \in \mathbb{T}^{kn+1}$:
  - Extend ct_1 and ct_2 to ct_1' and ct_2' to make them encrypted under the multi-key $\{sk_i\}_{i\in[k]}$ by putting zero in the empty extending slots.
  - Evaluate $\mathsf{GB}\left(\left(0, \cdots, 0, \frac{5}{8}\right) - ct_1' - ct_2'\right)$ and return the result.

For the part GB, we will not discuss it in this paper and refer to the original paper. And we call the evaluated ciphertext as multi-key ciphertext whose dimension is the number of participants in the MKTFHE scheme.

3.2 Distributed Decryption

The decryption algorithm of the existing MKTFHE is a single decryptor case, that is, the decryptor holds a set of secret keys $\{sk_i\}_{i\in[k]}$ of all participants. However, in practical use, for security reasons, the decryptor should not hold any secret key sk_i of participants. Therefore, distributed decryption which involves all participants jointly decrypting a multi-key ciphertext is more practical. The most common distributed decryption for MKFHE [24] has been defined below:

- $p_i \leftarrow \mathsf{PartDec}\left(ct, sk_i\right)$: Input a multi-key ciphertext ct under a set of secret keys $\{sk_i\}_{i\in[k]}$ of all participants, and the i-th secret key sk_i, output a partial decryption result p_i;
- $m \leftarrow \mathsf{FinDec}\left(p_1, \cdots, p_k\right)$: Input a set of partial decryption results $\{p_i\}_{i\in[k]}$ of all participants and output the plaintext of the multi-key ciphertext.

3.3 Homomorphic Gates and Operators Based on MKTFHE

Based on the TFHE scheme and its multi-key variant, Jiang et al. [5]. designed other binary gates with the same efficiency as NAND gates in MKTFHE including AND, OR, NOT, etc., and used their designed binary gates to implement the fixed k-bit complement arithmetic operators, so that addition, subtraction, multiplication, and division of both positive and negative numbers can be evaluated in MKTFHE. The definition of operators in MKTFHE is as follows:

- $ct\left[l\right] \leftarrow \mathsf{MKADD}\left(ct_1\left[l\right], ct_2\left[l\right]\right)$: Input two l-bit TLWE ciphertexts $ct_1\left[l\right]$ and $ct_2\left[l\right]$ and output a l-bit MKTLWE ciphertext $ct\left[l\right]$.
- $ct\left[l\right] \leftarrow \mathsf{MKSUB}\left(ct_1\left[l\right], ct_2\left[l\right]\right)$: Input two l-bit TLWE ciphertexts $ct_1\left[l\right]$ and $ct_2\left[l\right]$ and output a l-bit MKTLWE ciphertext $ct\left[l\right]$.
- $ct\left[2l\right] \leftarrow \mathsf{MKMUL}\left(ct_1\left[l\right], ct_2\left[l\right]\right)$: Input two l-bit TLWE ciphertexts $ct_1\left[l\right]$ and $ct_2\left[l\right]$ and output a $2l$-bit MKTLWE ciphertext $ct\left[2l\right]$.
- $ct\left[l\right] \leftarrow \mathsf{MKDIV}\left(ct_1\left[2l\right], ct_2\left[l\right]\right)$: Input a $2l$-bit TLWE ciphertext $ct_1\left[2l\right]$ and an l-bit TLWE ciphertext $ct_2\left[l\right]$ and output an l-bit length TLWE ciphertext $ct\left[l\right]$.

Note that the input of the gate circuit is a single-bit ciphertext, while the input of the operator is a multi-bit ciphertext. Therefore, before inputting a multi-bit integer, firstly encode it as complement, and then encrypt it by bit. In addition, the bit-number of the multiplication and division input data and output data are different, which is prone to data overflow or the bits of the input data do not match the operator.

3.4 Secret Sharing Based on Arithmetic Circuit

The secret sharing protocol on the arithmetic circuit is carried out on a finite field. In the secure 2-party computation, the l-bit value x is shared by the participants into two elements on $\mathbb{Z}_{2^l}$ ring and sned the two elements to two computing parties P_0 and P_1 respectively. Make $[x]_i^A$ represents the sub secret owned by the computing party P_i, and the superscript A represents the secret share on the arithmetic circuit. Secret sharing on arithmetic circuits is in $\mathbb{Z}_{2^l}$ which satisfies $[x]_0^A + [x]_1^A = x \mod 2^l$ where $[x]_0^A$, $[x]_1^A \in \mathbb{Z}_{2^l}$. So, the participants can share and reconstruct the secret x by the following algorithms:

- $\{[x]_0^A, [x]_1^A\} \leftarrow Share^A(x)$: Input an l-bit secret x, randomly choose $r \in \mathbb{Z}_{2^l}$, set $[x]_0^A = x - r$, $[x]_1^A = r$ and then output $[x]_0^A, [x]_1^A$.
- $x \leftarrow Rec^A\left([x]_0^A, [x]_1^A\right)$: Input $[x]_0^A$,$[x]_1^A$, compute $[x]_0^A + [x]_1^A$, then output the result.

In this protocol, if one party does not abide by the rules and sends the wrong value during the final reconstruction, the honest party cannot reconstruct the secret, while the fraudulent party can reconstruct the real secret. Therefore, this agreement is semi-honest and the participants need to abide by the rules of the agreement.

Besides, the addition operation of this protocol is free and both computing parties can directly perform the calculation locally which follows below:

- $\{[z]_0^A, [z]_1^A\} \leftarrow Add^A\left([x]_0^A, [x]_1^A, [y]_0^A, [y]_1^A\right)$: Input the secret shares of x, y, compute $[z]_0^A = [x]_0^A + [y]_0^A$, $[z]_1^A = [x]_1^A + [y]_1^A$, and output $[z]_0^A$,$[z]_1^A$.

3.5 Machine Learning

3.5.1 Logistic Regression

Logistic regression is a generalized linear regression model. It is a classical method to solve the binary classification problem by using the activation function. In the traditional logistic regression, the activation function is defined as a Sigmoid function, function $f(x) = \frac{1}{1+e^{-x}}$. The Batch Gradient Descent (BGD) method for logistic regression updates the coefficients in each iteration as follows:

$$z = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n = \theta^T x$$

$$h_\theta(x) = f\left(\theta^T x\right) = \frac{1}{1 + e^{-\theta^T x}}$$

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1} n\left(h_\theta(x_i) - y_i\right) x_i^j.$$

The phase to calculate the predicted output $h_\theta(x_i)$ is called *forward propagation*, and the phase to calculate the gradient $\alpha \frac{1}{m} \sum_{i=1} n\left(h_\theta(x_i) - y_i\right) x_i^j$ is called *backward propagation*.

3.5.2 Neural Networks

Neural networks are a more generalized regression model compared to logistic regression to learn more complex relationships between high dimensional input data and multiple output labels. Traditional activation functions are similar to the Sigmoid function $f(x)$ or the RELU function.

Standard error Back Propagation (BP) neural networks can be trained by the Gradient Descent with Momentum (GDM) method so that the coefficient convergence will be faster than BGD and more stable than Stochastic Gradient Descent (SGD). By the chain rule, the coefficients are updated as follows:

$$o_i = f(\sum_{i=1}^{m} w_{ji} x_i^k), \hat{y}_j^k = \sum_{i=1}^{n} o_i v_{ji}$$

$$\Delta v_{ji}^{(n+1)} = \beta_1 \Delta_1 v_{ji}^{(n)} + (1 - \beta_1)(\hat{y}_j^k - y_j^k) o_i$$

$$\Delta w_{ij}^{(n+1)} = \beta_2 \Delta w_{ij}^{(n)} + (1-\beta_2)\, o_i\,(1-o_i)\, x_j^k \sum_{j=1}^{p} \left(\hat{y}_j^k - y_j^k\right) v_{ji}$$

$$v_{ji}^{(n+1)} = v_{ji}^{(n)} - \alpha_1 \Delta v_{ji}^{(n+1)} w_{ij}^{(n+1)} = w_{ij}^{(n)} - \alpha_2 \Delta w_{ij}^{(n+1)}$$

4 Distributed Decryption Protocol

4.1 Our Security Goal

In the MKFHE scheme, participants generally generate their own secret keys independently, encrypt their own data by their own secret key, and decrypt the multi-key ciphertext by all secret keys jointly. Therefore, our security goal of MKFHE decryption is to protect individual single-key encrypted messages and common multi-key encrypted messages.

As mentioned in Section 1, we can observe that there are at least two kinds of static (passive) adversaries: an internal adversary and an external adversary. The internal adversary is one of the participants in the scheme, but the external adversary is not. Both adversaries want to know any information about each participant's message and the external adversary also hopes to obtain the computation result.

We define the security using the framework of Universal Composition (UC) [25]. To simplify, we take both of them together as *semi-honest* adversaries $\mathcal{A}$. And we assume the adversary $\mathcal{A}$ can corrupt any subset of the participants and servers (at least two participants and the server is uncorrupted) which only the data of the corrupted participants but nothing else about the remaining honest participants' data can be learned by the adversary $\mathcal{A}$.

4.2 Our Distributed Decryption Protocol for MKTFHE

We already know that MKFHE is IND-CPA secure [26], so the multi-key ciphertext is guaranteed to be secure for both adversaries and we force the protection of single-key ciphertext. Thanks to the technique of two-party secret sharing, no one can learn the whole single-key ciphertext, even the participants can only obtain their own part.

Denote the multi-key ciphertext by $\hat{ct} = (\boldsymbol{a_1}, \ldots, \boldsymbol{a_k}, b) \in \mathbb{T}^{kn+1}$, which satisfies $b = \frac{1}{4}m - \sum_{j=1}^{k} \langle \boldsymbol{a_j}, \boldsymbol{s_j} \rangle + e(\mod 1)$ [4], where $m \in \{0,1\}$ is the plaintext after decryption of the ciphertext, k is the number of participants, s_j is the private key of the j-th participant. The secure distributed decryption algorithm based on MKTFHE is defined as follows:

- Partial decryption algorithm:

  $p_i \leftarrow \mathsf{Part\_Dec}(\hat{ct}, s_i)$: The input is the multi-key ciphertext $\hat{ct}$ and the secret key s_i of the i-th participant. The output p_i is the partial decryption result of the i-th participant, and the computation is $p_i = b + \langle a_i, s_i \rangle$

- Final decryption algorithm:

  $\frac{1}{4}m + \bar{e} \leftarrow \mathsf{Fin\_Dec}(\{p_i\}_{i\in k})$: The input $\{p_i\}_{i\in k}$ is the partial decryption result of all participants, and the output is plaintext with noise before rounding, and the computation is $\frac{1}{4}m + \bar{e} = \sum_{i=1}^{k} p_i - (k-1)b$

The protocol can be divided into four steps, as shown in Fig 1

Step 1 Partial decryption: each participant $P_1, ..., P_k$ uses their own secret key s_i to compute partial decryption $\mathsf{Part\_Dec}(\hat{ct}, s_i)$ to obtain their personal partial decryption results p_i;

Step 2 Secret sharing: each participant runs the secret share algorithm for its own partial decryption result p_i. and get secret shares$[p_i]_0^A$ and $[p_i]_1^A$, and send them to the cloud server and the decryption party respectively;

Step 3 Offline computing: the cloud server and the decryption party receive secret shares received from participants, and then offline compute final decryption $\mathsf{Fin\_Dec}(\{[p_i]_0^A\}_{i\in k})$ and $\mathsf{Fin\_Dec}(\{[p_i]_1^A\}_{i\in k})$ respectively, so as to obtain the secret share of final decryption result $\left[\frac{1}{4}m+\bar{e}\right]_0^A$ and $\left[\frac{1}{4}m+\bar{e}\right]_1^A$;

Step 4 Secret reconstruction: the decryption party DP_1, DP_2 send the secret share of the final decryption result $\left[\frac{1}{4}m+\bar{e}\right]_0^A$ and $\left[\frac{1}{4}m+\bar{e}\right]_1^A$ to participants and the participants use the secret recovery protocol to reconstruct the final decryption result and obtain the final decryption result $\frac{1}{4}m+\bar{e}$.

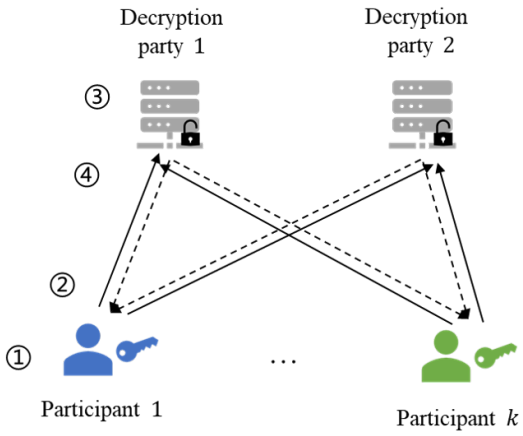

Figure 1: Our proposed distributed decryption protocol

4.2.1 Correctness Proof

The correctness of the protocol follows:

$$\begin{aligned}
&\left(\sum_{i=1}^{k}[p_i]_0^A-(k-1)[b]_0^A\right)+\left(\sum_{i=1}^{k}[p_i]_1^A-(k-1)[b]_1^A\right)\\
&=\left(\sum_{i=1}^{k}[p_i]_0^A+\sum_{i=1}^{k}[p_i]_1^A\right)-\left((k-1)[b]_0^A+(k-1)[b]_1^A\right)\\
&=\sum_{i=1}^{k}p_i-(k-1)b=\sum_{i=1}^{k}(b+\langle a,s_i\rangle)-(k-1)b\\
&=b+\sum_{i=1}^{k}\langle a,s_i\rangle=\frac{1}{4}m+e
\end{aligned}$$

If the error e is less than $\frac{1}{8}$, the decryption will work correctly.

4.2.2 Security Proof

In the UC framework, security is defined by comparing the *real* world and *ideal* world. The *real* world is involved in the protocol, adversary $\mathcal{A}$, and honest participants. And the *ideal* world includes the trusted party to represent the protocol, the simulator $\mathcal{S}$ to simulate the *ideal* world, and honest participants. If the view of *real* world and *ideal* world is undistinguished, the protocol is secure.

We consider security in the semi-honest model in which all participants and servers follow the protocol exactly. We assume that the two servers are non-colluding. We choose the adversary $\mathcal{A}$ who corrupts a server DP_1 and all but two of the participants $\{P_1, ..., P_{k-2}\}$ as an example that can cover all scenarios in our security goal. The simulator $\mathcal{S}$ is to simulate the above in *ideal* world which submits the partial decryption of the participants and receives the final decryption from the trusted party.

During simulation, on behalf of honest participants $\mathcal{S}$ sends a randomized partial decryption share $[p_i]_0^A$, $[p_i]_1^A$ in $\mathbb{Z}_{2^l}$ to DP_1 and DP_2. This is the only phase where participants are involved. Then each server evaluates independently until the final decryption is recovered.

We can briefly argue that the view of the *real* world and the *ideal* world is indistinguishable due to the security of the arithmetic secret sharing. The share of partial decryption is generated by participants at random. In particular, all messages sent, received and reconstructed in our protocol are generated using uniformly random shares in both the *real* world our protocol involved and *ideal* world simulator simulated, so the view of both identically distributed concludes our argument.

5 Privacy-Preserving Distributed Machine Learning

5.1 Pre-Work

5.1.1 Extract Sign Bit

Thanks to the encryption and evaluation of MKTFHE bit by bit, we can easily extract any bit in a multi-bit ciphertext. In complement coding, the highest bit can represent the sign of the operand, which is that the highest bit is 0 and the operand is positive, and the highest bit is 1 and the operand is negative. Therefore, we can use this property to extract the sign bit of any ciphertext. The operation of extracting the sign bit is defined as follows:

- $ct_{sign} \leftarrow \mathsf{Extract\_Sign}\,(ct\,[l])$: Input a l-bit length TLWE ciphertext $ct\,[l]$, and output the sign bit ct_{sign} of $ct\,[l]$.

5.1.2 Cut off and Expand

Considering the bit-length of the existing arithmetic operators in MKTFHE is predefined, and the larger the bit-length, the more time consumed. Therefore, flexibly adjusting the bit number of ciphertext and selecting the less bit-length arithmetic operators can improve the efficiency and accuracy of the overall scheme of machine learning based on MKTFHE.

We continue to use the encoding method in MKTFHE arithmetic operators [5] which is to use complement to encode both positive and negative integers. When the large-bit ciphertext operands are put into the little-bit arithmetic operators, the operands will automatically cut off the remaining bits of the input ciphertext operands; in other words, only the part of the data with the same bit number as the arithmetic operators will be calculated. For example, we can get a 16-bit ciphertext product from an 8-bit multiplier with a couple of 8-bit ciphertext

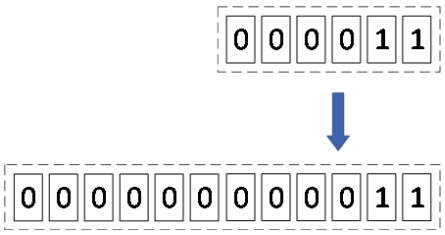

Figure 2: Expand the ciphertext

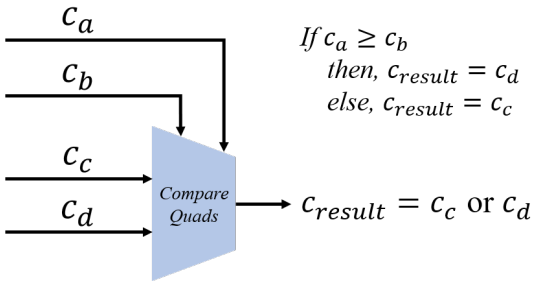

Figure 3: Select a ciphertext

operands input. And if this 16-bit ciphertext product doesn't overflow 8-bit size, it can be put into the next 8-bit arithmetic operator directly with a little time of computation. But if the 16-bit ciphertext product overflows the size of 8-bit, it must be put into the 16-bit arithmetic operator in the next calculation with more time consumed, and its corresponding 8-bit operand must be expanded from 8-bit to 16-bit. Until the operand recovers the 8-bit size by subsequent operations, such as division, we can continue to use the less-bit number operator.

To flexibly expand the bit number of the operand and keep its sign, we design and implement a device named *homogenizer*, as shown in Fig. 2, and the specific design is as follows:

MKTFHE does not allow the ciphertext to be copied directly (it is considered unsafe), so we use the trivial $\mathsf{TLWE}(0)$ and the sign bit ct_{sign} of the original small-bit ciphertext operand to calculate $\mathsf{MKAND}\left(\mathsf{TLWE}\left(0\right), ct_{sign}\right)$, and fill the ciphertext results into high bits. The operation of expanding is defined as follows:

- $ct\left[l'\right] \leftarrow \mathsf{Homogenizer}\left(ct\left[l\right], l'\right)$: Input the l bit length TLWE ciphertext $ct\left[l\right]$ and the bit length l', output the l' bit length ciphertext $ct\left[l'\right]$, the plaintext of which is the same as $ct\left[l\right]$.

5.1.3 Compare

In the practical machine learning scheme, the comparison operation is usually required, but the existing MKTFHE scheme cannot support the comparison operation without decryption. We believe that the comparison can be divided into two categories. One needs to know the comparison results, such as the millionaire problem, and the other is to determine the next calculation through the comparison result, which is similar to branch selection. At present, the comparison in the machine learning scheme is mainly the second category. Therefore, we utilize the Boolean operation and arithmetic operations in MKTFHE to design and implement the basic elements of the comparison operation, named *compare quads*, which is used to pick one from two ciphertext operands based on the results of comparison between the other two ciphertext operands, as shown in Fig. 3. And the details of the *compare quads* are described in Algorithm 1. The operation of comparison is defined as follows:

- $c_{result}\left[l\right] \leftarrow \mathsf{Compare\_Quads}\left(c_a\left[l\right], c_b\left[l\right], c_c\left[l\right], c_d\left[l\right]\right)$: Input four l bit length ciphertext $c_a\left[l\right]$, $c_b\left[l\right]$, $c_c\left[l\right]$, $c_d\left[l\right]$, if the plaintext of $c_a\left[l\right]$ is greater than $c_b\left[l\right]$, then output $c_d\left[l\right]$, otherwise output $c_c\left[l\right]$.

Algorithm 1: *Compare quads*

Input: MKTFHE parameter set $mkparams$, four l-bit ciphertext $c_a[l] = \mathsf{MKENC}(a, l)$, $c_b[l] = \mathsf{MKENC}(b, l)$, $c_c[l] = \mathsf{MKENC}(c, l)$, $c_d[l] = \mathsf{MKENC}(d, l)$, and the public keys of all participants $\{pk_i\}_k$

Output: l-bit ciphertext $c_{result}[l] = \mathsf{MKENC}((a \geq b) : d : c, l)$

1 Compute $\mathsf{MKSUB}(c_a[l], c_b[l])$ to obtain $c_a[l] - c_b[l]$
2 Compute $\mathsf{Extract\_Sign}(c_a[l] - c_b[l])$ to extract the sign bit c_{sign}
3 Compute $\mathsf{MKNOT}(c_{sign})$ to reverse the sign bit in order to obtain $\neg c_{sign}$
4 Compute $\mathsf{MKAND}(c_{sign}, c_c[t])$, $\mathsf{MKAND}(\neg c_{sign}, c_d[t])$ to obtain $c_{sign} \wedge c_c[n]$, $\neg c_{sign} \wedge c_d[n]$ which t is from 1 to n and can be computed in parallel.
5 Compute $\mathsf{MKADD}(c_{sign} \wedge c_c[l], \neg c_{sign} \wedge c_d[l])$ to obtain $c_{sign} \wedge c_c[n] + \neg c_{sign} \wedge c_d[n]$
6 Return $c_{sign} \wedge c_c[n] + \neg c_{sign} \wedge c_d[n]$

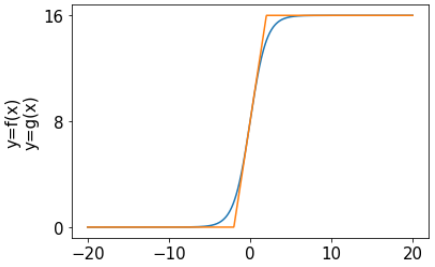

Figure 4: Our function $g(x)$

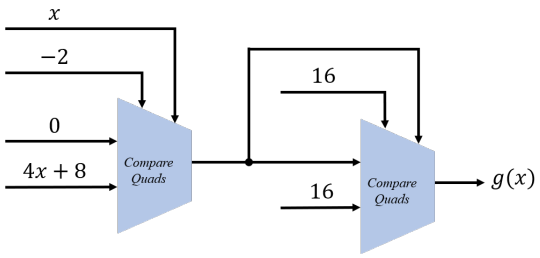

Figure 5: Constructure of our function

5.2 MKTFHE Friendly Activation Function

At present, the existing MKTFHE only supports integer linear operations and Boolean operations, so how to compute the Sigmoid function in logical regression and neural networks has become a main additional challenge. Prior work shows that polynomials can be used to fit Sigmoid function [27], and high-degree polynomials can achieve very high accuracy [28]. Hence, it is obvious that we can use the above method to implement the Sigmoid function, but high-degree polynomials will seriously reduce the efficiency, and using low-degree polynomials will lose a lot of accuracy.

We borrow the idea in SecureML [6] which discusses that the piecewise function can also achieve high accuracy. Hence, we design a new MKTFHE-friendly activation function $g(x)$. In addition, in order to improve the accuracy, we fit the sigmoid function $f(x)$ in the form of the tangent at the origin, as shown in Fig. 4. Considering that MKTFHE only supports integers, we have zoomed in on the new activation function 16 times. The description of the function is as follows:

$$g(x) = \begin{cases} 16, & x > 2 \\ 4x + 8, & -2 \leq x \leq 2 \\ 0, & x < -2 \end{cases}$$

Note that the comparison in our proposed activation function belongs to the second category of comparison, which is the comparison results are used for the next calculation instead of

Algorithm 2: New activation function

Input: MKTFHE parameter set $mkparams$, ciphertext $ct\,[l] = \mathsf{MKENC}\,(x, l)$ and the public keys of all participants $\{pk_i\}_k$

Output: Ciphertext $c_{result}\,[l] = \mathsf{MKENC}\,(g\,(x)\,, l)$

1 Prepare the ciphertext $c\,(-2)\,, c\,(16)\,, c\,(0)$ and $c(4x+8)$
2 Compute $\mathsf{Compare\_Quads}\,(ct\,[n]\,, c\,(-2)\,, c\,(0)\,, c\,(4x+8))$ to obtain the middle result $c_{mid}\,[l]$
3 Compute $\mathsf{Compare\_Quads}\,(c_{mid}\,[l]\,, c\,(16)\,, c_{mid}\,[l]\,, c\,(16))$ to obtain the final result $c_{result}\,[l]$
4 Return $c_{result}\,[l]$

knowing the comparison results. Therefore, we can use two *compare quads* in Section 5.1.3 to implement the activation function following Fig. 5. And details are shown in Algorithm 2.

5.3 Privacy-Preserving Machine Learning based on MKTFHE

After we propose the new MKTFHE-friendly activation function, we will utilize it to compute the *back propagation* in the logistic regression and neural networks. Considering the accuracy, we continue to use the Sigmoid function to calculate the partial derivative and maintain the structure of the iterative equation. And prior research also shows that if we change to compute the partial derivative of the linear activation function, the cross-entropy function is no longer convex, and the accuracy of training will incur more losses [6].

In addition, since MKTFHE is for integers and Boolean, it is necessary to zoom the learning rate and other parameters into integers and modify the relevant calculation equation, to ensure that the model coefficients after training are also enlarged in proportion.

We set the expansion factor q, use the integer learning rate α' and the new iterative computation for logistic regression is below:

$$h_\theta\,(x) = g(x)$$

$$\theta_j = \theta_j q - \alpha\prime \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\,(x_i) - y_i\right) x_i^j$$

There are m neurons in the input layer, n neurons in the hidden layer, and p neurons in the output layer. Like the above logistic regression, we set the expansion factor q, use the integer learning rate α'_1, α'_2, β'_1, β'_2, and the rest definition is the same as Section 3.5, the iterative computation for neural networks is below:

$$o_i = g(\sum_{i=1}^{m} w_{ji} x_i^k q), \hat{y}_j^k = \sum_{i=1}^{n} o_i v_{ji} q$$

$$\Delta v_{ji}^{(n+1)} = \beta'_2 \Delta v_{ji}^{(n)} q - \alpha'_1 \Delta v_{ji}^{(n+1)}$$

$$\Delta w_{ij}^{(n+1)} = \beta'_2 \Delta w_{ij}^{(n)} + (q - \beta'_2)\, o_i\, (q - o_i)\, x_j^k \sum_{j=1}^{p} \left(\hat{y}_j^k - y_j^k q\right) v_{ji}$$

$$v_{ji}^{(n+1)} = v_{ji}^{(n)} q - \alpha'_1 \Delta v_{ji}^{(n+1)}, w_{ij}^{(n+1)} = w_{ij}^{(n)} q - \alpha'_2 \Delta w_{ij}^{(n+1)}$$

5.4 Our Framework

After implementing privacy-preserving logical regression and neural network training, we replace the original decryption with our proposed distributed decryption protocol and finally propose a distributed privacy-preserving machine learning framework based on MKTFHE, including four types of entities: participants, a cloud server, a CRS server, and a decryption party. The participants want to outsource computation and each of them holds their own part of data for model training which should not be learned by a cloud server and other participants; The cloud servers are usually composed of one or more high-performance servers, which do not have their own data and only provide computing power. The CRS server is only responsible for generating the public parameters (that is, common reference string) of the framework which can be included in the cloud server. The decryption party only joins the distributed decryption and does not need too much computing power, which can be acted on by a participant or a single server.

We take two participants as examples. The steps of the whole scheme are shown in Fig. 6:

Step 1 Set up parameters: The CRS server calls $\mathsf{MKTFHE.SETUP}(1^\lambda)$ to generate $mkparams$ and communicates with the participants on the expansion factor q, then sends the set of parameters and factors $\{mkparams, q\}$ to each participant and the cloud server.

Step 2 Preprocess and encrypt data: Each participant uses the expansion factor q to zoom or round the origin data and uses the $mkparams$ to call $\mathsf{MKTFHE.KEYGEN}\,(mkparams)$ to generate their own secret key and public key, then utilize their own secret key to encrypt the preprocessed data bit by bit by calling $\mathsf{MKTFHE.ENC}\,(m)$ and finally sends the public key and the ciphertext to the cloud server.

Step 3 Train ML models: The cloud server first expands the single key ciphertext from each participant to multi-key ciphertext, then uses the arithmetic operators like MKADD, MKSUB, MKMUL, MKDIV, and our newly designed *homogenizer*, *compare quads* to train the model. In the end, the cloud server sends the ciphertext model to the decryption party and all participants.

Step 4 Decrypt data: All the participants, the decryption party and the cloud server run the distributed decryption protocol together and the participants will get the plaintext in the end.

6 Implementation and Experiment

6.1 Implementation of Distributed Decryption Protocol

6.1.1 The Implementation and Experimental Environment

Our code for the distributed decryption protocol is written in C + +, mainly using the arithmetic secret sharing of ABY [29] which is a very efficient two-party secure computing scheme. Considering that the final decryption only involves addition and subtraction without multiplication, this experiment neither needs to use oblivious transfer (OT) and other operations that require online interaction nor needs a semi-honest third party (STP) to assist in generating multiplication triples.

Note that the ABY library only uses unsigned numbers and negative numbers cannot be directly computed. Therefore, we naturally regard the unsigned numbers as the number encoded by the complement code and convert them into signed original code data after the final

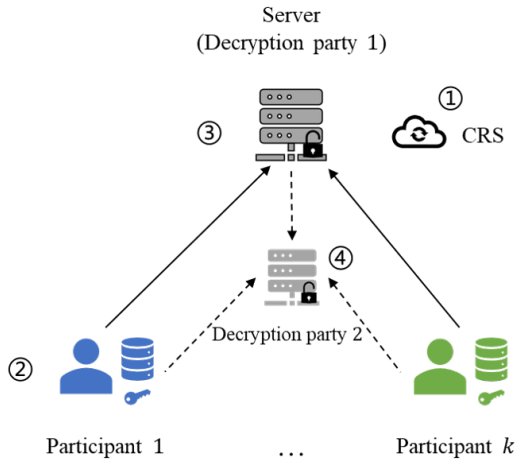

Figure 6: Framework of privacy-preserving machine learning

operation. In addition, since the finite field used in the MKTFHE scheme is a 32-bit torus, we straightly use the 32-bit arithmetic secret sharing in the ABY library for computation.

The experiment on this subject runs on the Linux environment based on the following configuration.

(1) Cloud server S_0 and S_1, is configured as Intel Xeon gold 5220 @2.2GHz processor, 256GB memory, and the operating system is Ubuntu 18.04 LTS;

(2) The client is Windows PC, Intel Core i7-8750H@2.20GHz processor, 16GB memory, and the operating system is Windows 10.

In the LAN experiment, we use two servers located in the same area. The network bandwidth is 512MB/s and the network delay is 0.35ms.

6.1.2 Accuracy and Efficiency Analysis

In this experiment, we compare with the original decryption scheme in MKTFHE and set the number of participants k to 2, 4, and 8 respectively. We tested in 10 groups for both decryption schemes and each group included 1000 bits of ciphertext. Then we record the average time of decryption. Note that in the specific implementation, in the distributed decryption protocol, we use the SIMD technique in the ABY library for parallel optimization to improve efficiency. The experimental results are shown below:

Table 1: Implementation of distributed decryption

Participants k	MKTFHE/s	Our protocol/s	Accuracy
2	0.024	0.261	100%
4	0.050	0.268	100%
8	0.112	0.263	100%

The result in Table 1 shows that compared to the original MKTFHE, the efficiency of our scheme is relatively lower, but it is still acceptable. We think the reason is mainly in the establishment of the secret sharing scheme and ciphertext transmission because the original scheme does not involve additional schemes and transmissions. By using the SIMD technique in the ABY library, the decryption time basically remains unchanged with the increase of participants, while the decryption time of the original MKTFHE scheme increases linearly with the increase of participants. Therefore, in the case of multi-party participation, our scheme has more advantages in both security and efficiency.

6.2 Implementation of Privacy Preserving Distributed Machine Learning

6.2.1 Data preprocessing

Considering that MKTFHE only supports integers and Boolean, we need to preprocess the input data. We have two methods to preprocess; one is rounding and the other is zooming. The input data of logistic regression is in a rather large range while the input data of neural networks is relatively small, so we apply the rounding method on logistic regression and the zooming method on neural networks to keep the data precise. We store the zooming factor for the following computation to guarantee accuracy in an acceptable range.

6.2.2 Implementation of Privacy-Preserving Logistic Regression

The input data are generated by ourselves which are several sets of linear data with small random noise, and we mainly use 16-bit and 32-bit operators in this implementation.

We first use the 7-order Taylor polynomial (high enough order) formed Sigmoid function and our proposed activation function as the activation function in logistic regression to train the models with *plaintext* of integer and float data. The result shows in Table 2 that in both integer and floating numbers, the accuracy of using a 7-order Taylor polynomial as an activation function is the highest, and using our proposed activation function can be close to that of a 7-order Taylor polynomial.

Then, we utilize the operators and other tools in MKTFHE to train the logistic regression models with the above different types of activation functions in *ciphertext*. In addition to recording the accuracy and time in training in Table 3, we also compare the computation time of different activation functions under MKTFHE in Table 4. The result of the experiments shows that our scheme has no accuracy loss which means that the model trained in *ciphertext* is the same as that in *plaintext*, and the loss only occurs in the integer transfer stage. Using our proposed activation function can shorten the computing activation function time in *ciphertext* by 10 times and significantly shorten the training time compared with the 7-order Taylor polynomial and the accuracy is close to it. Note that we also compare the 3-order Taylor polynomial with our proposed function in both *plaintext* and *ciphertext* and the result shows that comparing with the 3-order Taylor polynomial we can also shorten the computing activation time by 5 times with much better accuracy.

6.2.3 Implementation of Privacy Preserving Neural Networks

We use the Iris data set in sklearn [30] as input data for neural networks, half of them for training and the rest for prediction. Like the above logistic regression, we also implement neural networks in both plaintext and ciphertext.

Table 2: Logistic regression accuracy in plaintext

Data type	7-order Taylor polynomial	3-order Taylor polynomial	Our function
Floating data	98%	85%	95%
Integer data	95%	80%	92%

Table 3: Logistic regression accuracy in ciphertext

Activation function	Accuracy	Training time/iter/piece/s
7-order Taylor polynomial	95%	4049
3-order Taylor polynomial	80%	2549
Our function	92%	611

In *plaintext*, we use the above different kinds of functions as activation functions to train the model with both integer data and floating data, and the same in *ciphertext*. Note that we also compute every neuron in the same layer in parallel to optimize the code. The result of the experiments is shown in Table 5.

As the result shown in Table 6, using a 7-order Taylor polynomial as an activation function is more accurate and costs more time, but using our proposed activation function can greatly reduce the training time with close accuracy to it. Note that we also compare our function with a 3-order Taylor polynomial and the result shows that we shorten the training time with much better accuracy as well.

6.2.4 Security analysis

Our scheme is semantically secure under the (R)LWE assumption. We choose the same parameters in MKTFHE [4,5]. The achievement estimated security level of our scheme is 110-bit while the dimension of the TLWE problem is $k = 1$.

6.2.5 Distributed server cluster

The training data, intermediate result e.g. descent gradient, and model parameters are all encrypted by single-key or multi-key and the evaluations among them are all fully homomorphic. So our scheme is "plug and play" for server clusters via most distributed machine learning strategies and typologies in [1] and does not leak any privacy in the distributed structure.

7 Conclusion and Discussion

In this paper, we implement privacy-preserving logistic regression and neural networks with a distributed decryption protocol based on MKTFHE. Firstly, we introduce secret sharing to

Table 4: Computing activation function in ciphertext

	7-order Taylor polynomial	3-order Taylor polynomial	Our function
Time/s	1440	549	130

Table 5: Neural networks accuracy in plaintext

Data type	7-order Taylor polynomial	3-order Taylor polynomial	Our function
Floating data	96.23%	72.17%	95.46%
Integer data	94.67%	68.12%	94.15%

Table 6: Neural networks accuracy in ciphertext

Activation function	Accuracy	Training time/iter/piece/s
7-order Taylor polynomial	94.67%	7301
3-order Taylor polynomial	68.12%	6736
Our function	94.15%	4654

protect partial decryption and final decryption. Secondly, we design *homogenizer* and *compare quads* to implement our proposed MKTFHE-friendly activation function. Then, we utilize them to train privacy-preserving logistic regression and privacy-preserving neural networks. Finally, we formalize our distributed privacy-preserving machine learning framework. The experimental results show that the efficiency of our distributed decryption protocol is acceptable. Compared with using the Sigmoid function, the efficiency is greatly improved with our activation function and the accuracy is basically unchanged.

Acknowledgments

This work is supported by National Natural Science Foundation of China (No. 62272131), National Science and Technology Major Project Carried on by Shenzhen, (CJGJZD20200617103000 001), Shenzhen Basic Research Project of China (JCYJ20200109113405927), and Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (2022B1212010005).

References

[1] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. A survey on distributed machine learning. *ACM Computing Surveys (CSUR)*, 53(2):1–33, 2020.

[2] Qi Jia, Linke Guo, Zhanpeng Jin, and Yuguang Fang. Preserving model privacy for machine learning in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(8):1808–1822, 2018.

[3] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012*, pages 1219–1234, 2012.

[4] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Multi-key homomorphic encryption from TFHE. In *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security*, volume 11922 of *Lecture Notes in Computer Science*, pages 446–472, 2019.

[5] Zoe L Jiang, Jiajing Gu, Hongxiao Wang, Yulin Wu, Junbin Fang, Siu-Ming Yiu, Wenjian Luo, and Xuan Wang. Privacy-preserving distributed machine learning made faster. In *Proceedings of the 2023 Secure and Trustworthy Deep Learning Systems Workshop*, pages 1–14, 2023.

[6] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[7] Hyang-Sook Lee and Jeongeun Park. On the security of multikey homomorphic encryption. In *Cryptography and Coding - 17th IMA International Conference, IMACC 2019*, volume 11929 of *Lecture Notes in Computer Science*, pages 236–251, 2019.

[8] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 201–210, 2016.

[9] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Advances in Cryptology - CRYPTO 2018*, volume 10993 of *Lecture Notes in Computer Science*, pages 483–512, 2018.

[10] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019*, pages 3–13, 2019.

[11] Yifan Tian, Laurent Njilla, Jiawei Yuan, and Shucheng Yu. Low-latency privacy-preserving outsourcing of deep neural network inference. *IEEE Internet Things J.*, 8(5):3300–3309, 2021.

[12] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, pages 142–156, 2019.

[13] Hao Chen, Ran Gilad-Bachrach, Kyoohyung Han, Zhicong Huang, Amir Jalali, Kim Laine, and Kristin E. Lauter. Logistic regression over encrypted data from fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2018:462, 2018.

[14] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, and Xiaoqian Jiang. Secure logistic regression based on homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2018:74, 2018.

[15] Jung Hee Cheon, Duhyeong Kim, Yongdai Kim, and Yongsoo Song. Ensemble method for privacy-preserving logistic regression based on homomorphic encryption. *IEEE Access*, 6:46938–46948, 2018.

[16] Eleftheria Makri, Dragos Rotaru, Nigel P Smart, and Frederik Vercauteren. Pics: Private image classification with svm. *IACR Cryptol. ePrint Arch.*, 2017:1190, 2017.

[17] Michael Clear and Ciaran McGoldrick. Multi-identity and multi-key leveled FHE from learning with errors. In *Advances in Cryptology - CRYPTO 2015*, volume 9216 of *Lecture Notes in Computer Science*, pages 630–656, 2015.

[18] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In *Advances in Cryptology - EUROCRYPT 2016*, volume 9666 of *Lecture Notes in Computer Science*, pages 735–763, 2016.

[19] Chris Peikert and Sina Shiehian. Multi-key FHE from lwe, revisited. In *Theory of Cryptography - 14th International Conference, TCC 2016-B*, volume 9986 of *Lecture Notes in Computer Science*, pages 217–238, 2016.

[20] Zvika Brakerski and Renen Perlman. Lattice-based fully dynamic multi-key FHE with short ciphertexts. In *Advances in Cryptology - CRYPTO 2016*, volume 9814 of *Lecture Notes in Computer Science*, pages 190–213, 2016.

[21] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology - ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, 2016.

[22] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *Advances in Cryptology - ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408. Springer, 2017.

[23] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, 2020.

[24] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key fhe. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 735–763, 2016.

[25] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[26] Eunkyung Kim, Hyang-Sook Lee, and Jeongeun Park. Towards round-optimal secure multiparty computations: Multikey fhe without a crs. Cryptology ePrint Archive, Report 2018/1156, 2018. https://ia.cr/2018/1156.

[27] Yoshinori Aono, Takuya Hayashi, Le Trieu Phong, and Lihua Wang. Scalable and secure logistic regression via homomorphic encryption. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 142–144, 2016.

[28] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. *Advances in neural information processing systems*, 27, 2014.

[29] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[30] Gaël Varoquaux, Lars Buitinck, Gilles Louppe, Olivier Grisel, Fabian Pedregosa, and Andreas Mueller. Scikit-learn: Machine learning without learning the machinery. *GetMobile: Mobile Computing and Communications*, 19(1):29–33, 2015.

A Attack on Existing Distributed Decryption Protocol

In this section, we propose a possible attack on [7] for the external passive adversary to obtain the final decryption result m. We show that the adversary only needs to collect the partial decryption broadcasted by each user to recover the message in the multi-key ciphertext.

As we mentioned above, the MKTLWE ciphertext is $(\mathbf{a}_1, \ldots, \mathbf{a}_k, b)$, which $b = \frac{1}{4}m - \sum_{i=1}^{k}\langle \mathbf{a}_i, \mathbf{s}_i\rangle + e$, k is the number of total users, m is a one-bit message, and e is an error of b. In the partial decryption protocol of [7], there are two kinds of partial decryption $p_{i,i}$ and $p_{i,j}$ both generated by the user u_i:

- $p_{i,i} = b + \langle \mathbf{a}_i, \mathbf{s}_i\rangle$ directly computed by its own secret key. Note that $p_{i,i}$ can also be viewed as a TLWE ciphertext that contains the message mess encrypted by the j-th user u_j that satisfies $p_{i,i} = \mathsf{mess} - \langle \mathbf{a}_j, \mathbf{s}_j\rangle + e$ and $\mathsf{mess} = \frac{1}{4}m - \sum_{t\neq i,j}^{k}\langle \mathbf{a}_t, \mathbf{s}_t\rangle$.
- $p_{i,j} = (\mathbf{a}_{i,j}, b_{i,j})$ for each $j \in [k]\backslash\{i\}$ is the external product between the TLWE sample $(\mathbf{a}_j, p_{i,i})$ and a trivial $\mathsf{TGSW}(1)$.

Then, user u_i keeps the $p_{i,i}$ secret and broadcasts all the $p_{i,j}$ for each $j \in [k]\backslash\{i\}$.

Note that the trivial $\mathsf{TGSW}(1) = \mathbf{Z}_t + \mathbf{H}$, each row of $\mathbf{Z}_t$ is $(\mathbf{0}, e)$ which is a trivial TLWE sample of zero with some noise. Thus, after the external product, $\mathbf{a}_{i,j} = \mathbf{a}_j$ and $b_{i,j}$ is still encrypted by user u_j which still contains the message mess. The only change is that the noise e becomes $\tilde{e_j} = e + e_{add_j}$. So the $b_{i,j}$ satisfies the following equation:

$$b_{i,j} = \langle \mathbf{a}_j, \mathbf{s}_j\rangle + \mathsf{mess} + \tilde{e_j} = \frac{1}{4}m - \sum_{t\neq i}^{k}\langle \mathbf{a}_t, \mathbf{s}_t\rangle + \tilde{e_j}$$

So, the attack for the external adversary is that: after he collects the $p_{i,j}$ for some $j \neq i$ of all the users u_i for $i \in [k]$, he can finish the final decryption to obtain m on his own by

computing $\sum_{i=1}^{k} p_{i,*} - (k-1)b$ and $*$ is denoted as some j we don't care. For example, when $k = 3$, one of the sets of $p_{i,j}$ supporting the attack is $p_{1,2}$, $p_{2,3}$, $p_{3,1}$.

Correctness of attack. For $b = \frac{1}{4}m - \sum_{i=1}^{k}\langle \mathbf{a}_i, \mathbf{s}_i\rangle + e$, we have

$$\begin{aligned}\sum_{i=1}^{k} p_{i,*} &= k \cdot \frac{1}{4}m - k \cdot \sum_{t=1}^{k}\langle \mathbf{a}_t, \mathbf{s}_t\rangle + \sum_{i=1}^{k}\langle \mathbf{a}_i, \mathbf{s}_i\rangle + k \cdot \tilde{e_*} \\ &= k \cdot \frac{1}{4}m - (k-1) \cdot \sum_{t}^{k}\langle \mathbf{a}_t, \mathbf{s}_t\rangle + k \cdot \tilde{e_*}\end{aligned}$$

$$\sum_{i=1}^{k} p_{i,*} - (k-1)b = \frac{1}{4}m + k\tilde{e_*} - (k-1)e = \frac{1}{4}m + \tilde{e}$$

If the magnitude of the error term $\tilde{e}$ is less than $\frac{1}{8}$, the final decryption (attack) made by the adversary works correctly.

Error growth estimation. Notice e is the error in b and $p_{i,i}$. After the external product, the noise becomes $\tilde{e_*} = e + e_{add_*}$. We can estimate the magnitude of the growth e_{add_*} from the external product noise propagation formula. So that we have

$$\tilde{e} = k\tilde{e_*} - (k-1)e = ke + ke_{add_*} - (k-1)e = e + ke_{add_*}$$

$$\|\tilde{e}\|_\infty \leq \|e\|_\infty + k \max_{*}\{\|e_{add_*}\|_\infty\} = \|e\|_\infty + 2kl\beta \max_{*}\{\|\mathsf{Err}(\mathbf{A}_*)\|_\infty\} + 2\epsilon$$

To simplify, we omit some preliminaries only used in this section found in [7], and some notions and symbols are the same as [7]. Therefore, the noise growth after the attack is quite small because $\mathsf{Err}(\mathbf{A}_*)$ is the error of the fresh TGSW ciphertext $\mathbf{A}_*$.