Privacy-Preserving Distributed Machine Learning Made Faster

Zoe L. Jiang
Harbin Institute of Technology, Shenzhen & Peng Cheng Laboratory
Shenzhen, Guangdong, China
zoeljiang@hit.edu.cn

Jiajing Gu
Harbin Institute of Technology, Shenzhen
Shenzhen, Guangdong, China
3256337209@qq.com

Hongxiao Wang
University of Hong Kong
Hong Kong, China
hxwang@cs.hku.hk

Jinjun Gu
Harbin Institute of Technology, Shenzhen
Shenzhen, Guangdong, China
3256337209@qq.com

Yulin Wu
Harbin Institute of Technology, Shenzhen & Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies
Shenzhen, Guangdong, China
wuyulin@hit.edu.cn

Hongxiao Wang
University of Hong Kong
Hong Kong, China
hxwang@cs.hku.hk

Siow-Ming Yiu
University of Hong Kong
Hong Kong, China
smyiu@cs.hku.hk

Wenjian Luo
Harbin Institute of Technology, Shenzhen & Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies
Shenzhen, Guangdong, China
luowenjian@hit.edu.cn

Xuan Wang
Harbin Institute of Technology, Shenzhen & Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies
Shenzhen, Guangdong, China
wangxuan@cs.hitsz.edu.cn

ABSTRACT

With the development of machine learning, it is difficult for a single server to process all the data. So machine learning tasks need to be spread across multiple servers, turning the centralized machine learning into a distributed one. Multi-key homomorphic encryption is one of the suitable candidates to solve the problem. However, the most recent result of the Multi-key homomorphic encryption scheme (MKTFHE) only supports the NAND gate. Although it is Turing complete, it requires efficient encapsulation of the NAND gate to further support mathematical calculation. This paper designs and implements a series of operations on positive and negative integers accurately. First, we design basic bootstrapped gates, the efficiency of which is times that of using NAND to build. Second, we construct practical k-bit complement mathematical operators based on our basic binary bootstrapped gates. The constructed created can perform addition, subtraction, multiplication, and division on both positive and negative integers accurately. Finally, we demonstrated the generality of the designed operators by achieving a distributed privacy-preserving machine learning algorithm, i.e. linear regression with two different solutions. Experiments show that the consumption time of the operators built with our gates is about 50−70% shorter than built directly with NAND gate and the iteration time of linear regression with our gates is 67.5% shorter than with NAND gate directly.

KEYWORDS

Privacy-preserving machine learning, distributed machine learning, multi-key fully homomorphic encryption

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:
Zoe L. Jiang, Jiajing Gu, Hongxiao Wang, Yulin Wu, Junbin Fang, Siow-Ming Yiu, and Xuan Wang. 2023. Privacy-Preserving Distributed Machine Learning Made Faster. In Proceedings of the 2023 Secure and Trustworthy Deep Learning Systems Workshop (SecTL ’23), July 10–14, 2023, Melbourne, VIC, Australia. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3591197.3591306

1 INTRODUCTION

In the big data era, it is necessary to spread the machine learning tasks across multiple servers and transform centralized systems into distributed ones [38]. These distributed systems present new challenges and especially privacy remains an unsolved challenge in it [27].

Privacy computing is a kind of techniques which perform data computation without specified information leakage. Homomorphic encryption is a special form of encryption that permits users to perform computations on encrypted data without first decrypting it, which has great practical implications in the outsourcing of private computations. It is called fully homomorphic encryption (FHE) if both addition and multiplication are supported. The security of FHE is usually guaranteed by the hard problems on lattices, which is also considered as one of the post-quantum cryptographic schemes [31]. However, computing on encrypted data using such FHE schemes will increase the noise in the ciphertext. If the noise
in the ciphertext grows beyond some threshold, it will result in wrong decryption [19]. This problem can be solved by a bootstrapping technique, first proposed by Gentry [22]. Bootstrapping can reduce the noise in ciphertext to refresh it so that the homomorphic evaluation can be performed constantly.

Obviously, single-key fully homomorphic encryption only allows server to perform addition and multiplication on data encrypted by same key. Multi-key fully homomorphic encryption (MKFHE) was proposed to circumvent this shortcoming [33]. It enables users to encrypt their own data under their own keys, while homomorphic evaluations can be performed on encrypted data directly at server side without decryption. It avoids the possibility that user and server conspire to steal the data of other users. Therefore, MKFHE indeed realizes secure multi-party computation with untrusted party. While Chen et al. [8] developed the library Multi-key fully homomorphic encryption over torus (MKTFHE) for NAND gate.

However, MKTFHE library only provides multi-key homomorphic NAND gates. Perform multi-key homomorphic computation using only homomorphic NAND gates is a complex and time-consuming work. As a result, a simple multi-key homomorphic NAND gate is not practical to be used directly. In order to better apply multi-key homomorphic encryption on privacy computing, we design and implement a series of practical multi-key homomorphic mathematical operators.

In this study, we make the following contributions:

1. We designed a series of fundamental multi-key bootstrapped gates based on MKTFHE, including multi-key bootstrapped AND gate, multi-key bootstrapped OR gate, multi-key bootstrapped NOR gate, multi-key bootstrapped XOR gate, multi-key bootstrapped XNOR gate, and multi-key NOT gate (without gate bootstrapping). Experiment results show that the proposed fundamental multi-key bootstrapped gates are more efficient than directly joining multi-key bootstrapped NAND gates to build basic binary gates.

2. We designed a series of multi-key homomorphic operators based on the basic multi-key bootstrapped gates we constructed. The operators we designed include k-bit complement array integer adder, k-bit complement array integer subtractor, k-bit complement array integer multiplier and k-bit complement array integer divider, which can perform addition, subtraction, multiplication and division on arbitrary bits integers in both positive and negative. We construct the complement array adder linearly and used a structure similar to the 4-bit array integer multiplier in [26] to construct our k-bit complement multiplier. In the similar way we construct the k-bit complement array integer divider. The subtractor is constructed with a adder and a simple multiplier. The time of the adder and the subtractor grows linearly with the bits of input numbers, while the time of the multiplier and divider grows quadratically. The time of the divider grows almost linearly with the number of layers in divider.

3. We train linear regression model by utilizing our proposed multi-key homomorphic operators. Taking linear regression as an example, we implement a whole multi-key fully homomorphic machine learning scheme. Experimental results show that the practicability and generality of our homomorphic operator and the training time grows linearly with the number of participants.

The rest of this paper is organized as follows. Section 2 discusses the related research work. In Section 3, it clarifies the notation and review some constructions related. Section 4 describes the k-bit complement array operators with implementation based on MKTFHE and the distributed machine learning model we evaluate, followed by the performance analysis and experimental results of our implementation in Section 5. Section 6 concludes the paper with future work.

2 RELATED WORK

Previous research efforts have made a number of contributions to the development of privacy-preserving machine learning. Some work has focused on securely outsourcing the training of ML models to the cloud, typically by using homomorphic encryption techniques [21]. In 2016, Aono et al. [1] proposed a secure system for protecting the training data in logistic regression via homomorphic encryption. In 2018, Crawford et al. [18] built a system that uses fully-homomorphic encryption to approximate the coefficients of a logistic-regression model built from genomic data, while Kim et al. [28, 30] presented a method to train a logistic regression model without information leakage. However, these scheme are all based on single-key fully homomorphic encryption, which are not suitable for distributed machine learning models.

In 2012, Graepel et al. [25] first proposed that it is possible to perform machine learning algorithm on encrypted data by homomorphic encryption scheme. Then, several single-key homomorphic encryption schemes are used for machine learning prediction [2, 5, 20, 24, 37] or machine learning training [10, 13, 29]. A few machine learning algorithms has adapted multi-key homomorphic encryption and demonstrated the premising future [9].

Multi-key fully homomorphic encryption was first proposed based on the NTRU assumption by Lopez et al. [33] at 2012, which is intended to apply to on-the-fly multiparty computation. In 2015, Clear et al. [17] constructed the first LWE-based MKFHE scheme, and improved by Mukherjee and Wichs [34] in 2016. These schemes are single-hop MKFHE schemes, which means all the participants must be known in advance. This problem was solved by Peikert et al. [35] and Brakerski et al. [7] in 2016 by constructing multi-hop MKFHE schemes. However, these schemes are impractical and without implementation.

The first implementation of the MKFHE scheme was achieved by Chen et al. [8] in 2019, named MKTFHE. In their scheme, a multi-key bootstrapped NAND gate is described. This scheme was improved by Lee and Park [32] in 2019, realizing the distributed decryption in MKTFHE. However, using multi-key bootstrapped NAND gates to construct homomorphic encryption operation directly has disadvantages such as low practicability, high complexity and error-prone construction process. At the same time, the complex structure lead to expensive optimization.
3 PRELIMINARIES

3.1 Notation
The rest of the paper uses the following notations. T denotes the real Torus $\mathbb{R}/\mathbb{Z}$, the set of real numbers modulo 1. $\mathbb{T}_N[X]$ denotes $\mathbb{R}[X]/(X^N + 1)$ mod 1. k represents the number of parties. TFHE is used to denote the (scalar) binary learning with error problem over Torus, while for the ring mode, we use the notation of TRLWE. $\text{params}$ represents the parameter sets used in TFHE (fully homomorphic encryption over torus) scheme, while $\text{mkmparams}$ represents the parameter sets used in MKTFHE scheme and our scheme.

3.2 MKTFHE
MKTFHE scheme is the multi-key version of TFHE scheme. TFHE, constructed by Chillotti et al. [14–16], is a fast fully homomorphic encryption (FHE) scheme over the torus, which generalizes and improves the FHE based on GSW [23] and its ring variants. In the TFHE scheme, bootstrapped binary gates are designed to represent the functions developers need.

The main idea of TFHE is to bootstrap after every binary gate evaluation to refresh the ciphertext in order to make it usable for the following operations, resulting in that arbitrarily deep circuits can be homomorphically evaluated. That is to say, a single parameter set allows the sever to evaluate any function. The entire homomorphic evaluation of the circuits will take time proportional to the number of the binary gates used or, if parallelism is involved, to the number of the circuit layers.

The message space of TFHE bootstrapping gates is $\mathbb{T}$. A TLWE ciphertext $(a, b) \in T^{n+1}$ encrypted a message $\mu \in \mathbb{T}$ with noise parameter $\alpha$.

In the TFHE scheme, the homomorphic evaluation of a binary gate is achieved with operations between TLWE samples and a gate-bootstrapping just after that (except the bootstrapping NOT gate, which does not need bootstrapping). By using this approach, all the basic gates can be evaluated with a single bootstrapping process (GB):

- TFHE.BootsNAND($c_1, c_2$) = $\text{GB}((0, \frac{1}{2}) - c_1 - c_2)$
- TFHE.BootsAND($c_1, c_2$) = $\text{GB}((0, \frac{1}{4}) + c_1 + c_2)$
- TFHE.BootsOR($c_1, c_2$) = $\text{GB}((0, 1) - c_1 - c_2)$
- TFHE.BootsXOR($c_1, c_2$) = $\text{GB}(2 \cdot (c_1 - c_2))$
- TFHE.NOT($c$) = $(0, \frac{1}{4}) - c$

The TFHE scheme has the advantages of fast bootstrapping, efficient homomorphic logic circuit evaluation, and so on. Its multi-key version, named MKTFHE [8], was constructed by Chen et al. in 2019. MKTFHE is the first attempt in the literature to implement an MKFHE scheme in codes.

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 [8]:

- MKTFHE.Setup(1/4): The cloud equipment take a security parameter $\lambda$ as input, and output the public parameter set $\text{mkmparams}$.
- MKTFHE.KeyGen($\text{mkmparams}$): Each party generates its keys independently. First sample the TLWE secret key $sk_i$.

Then the algorithm set public key $pk_i$, bootstrapping key $BK_i$ and key-switching key $KS_i$.

- MKTFHE.SymEnc($\mu$): This algorithm encrypts an input bit $\mu \in \{0, 1\}$, and returns a TLWE ciphertext with the scaling factor $\frac{1}{4}$. The output ciphertext $c = (b, a) \in T^{n+1}$ satisfies $b+c < a, b_i > -\frac{1}{4} m$.
- MKTFHE.SymDec($\mu$, $sk_i$): Input a TLWE ciphertext $c = (b, a)$ and a set of secret keys $\{sk_i\}$, and return the message $\mu$ which minimizes $|b + \sum_{i=1}^k a_i, b_i| > -\frac{1}{4} m$.
- MKTFHE.MKKeySwitch($\mu$, $(KS_i)_c$): Input an expand ciphertext $c$ corresponding to $t = (t_1,...,t_k)$ and a sequence of key-switching keys from $t_1$ to $t_k$ and returns an encryption of the same message under $s = (t_1,...,t_k)$.

3.3 Complement operators
Assuming that there are two k-bit inputs $a_{in}$ and $b_{in}$, a k-bit complement array integer multiplier can be constructed in the following steps [36].

1. Construct a semi-adder with two XOR gates: Input two addends $a$ and $b$ and the carry $c$, return $out = (c \quad (a \oplus b))$.
2. Construct a carrier with three AND gates and two OR gates: Input two addends $a$ and $b$ and the carry $c$, return $c = (a \land b) \lor (a \land c) \lor (b \land c)$.
3. Construct a 1-bit adder with a semi-adder and a carrier: Input two addends $a$ and $b$ and the carry $c$, return $out = (c \quad (a \oplus b))$ and $c = (a \land b) \lor (a \land c) \lor (b \land c)$.
4. Construct a k-bit complement array integer adder with k 1-bit adder: Input two addends $a_{in}$ and $b_{in}$, set the carry $c[0] = 0$, compute $out[i] = (c[i] \quad (a_{in}[i] \quad b_{in}[i]))$, $c[i+1] = (a_{in}[i] \land b_{in}[i]) \lor (a_{in}[i] \lor c[i]) \lor (b_{in}[i] \lor c[i])$ with $i = \{0,1,...,k-1\}$. Return the result $out$.

Assuming that there are two k-bit inputs $a_{in}$ and $b_{in}$, a k-bit complement array integer multiplier can be constructed in the following steps [36].

1. Prepare three kinds of 1-bit adder for complement multiplier:
   (a) 0-adder: The same as 1-bit adder above.
   (b) 1-adder: Input two addends $a$ and $b$ and the carry $c$, return $out = (c \quad (a \quad b))$ and $out = (a \land b) \lor (a \land c) \lor (b \land c)$.
   (c) 2-adder: Input two addends $a$ and $b$ and the carry $c$, return $out = (c \quad (a \quad b))$ and $out = (a \land b) \lor (a \land c) \lor (b \land c)$.

2. Construct a k-bit complement array integer multiplier according to the following rules, as shown in Figure 1. (a) Arrange the adders in $k$ rows and $k - 1$ columns. The input $a$ and $b$ of the adder in $i$ row and $j$ column are $a_{in}[j] \land b_{in}[i - 1]$ and $a_{in}[j - 1] \land b_{in}[i]$. The inputs $c$ of...
Figure 1: The construction of \( k \)-bit complement array integer multiplier. The index of the adder stands for the type of it (0-adder, 1-adder, or 2-adder) [36].

the adders are 0 in the first line while the \( cout \) of adders above in other lines.

(b) The highest bit of each addend (that is to say, \( a_{in}[k-1] \) and \( b_{in}[k-1] \)) has the weight \(-1\).

(c) If two input addends have no weight then use 0-adder.

(d) If one of the input addends has the weight \(-1\) after AND gate or 1-adder then use 1-adder.

(e) If one of the addends is an output of a 1-adder and the other addend has the weight \(-1\) after AND gate, use 2-adder.

(f) If one of the addends is an output of a 2-adder, use 2-adder.

Obviously, a subtractor can be constructed by an adder and a multiplier. Assume that there are two \( k \)-bit inputs \( a_{in} \) and \( b_{in} \), a \( k \)-bit complement array integer subtractor can be calculated by:

\[
out = a_{in} \times (-1) + b_{in}.
\]

Assume that there are two \( k \)-bit inputs \( a_{in} \) and \( b_{in} \), a \( k \)-bit complement array integer divider can be constructed in the following steps [39].

Figure 2: The construction of CAS [39]

(1) Prepare the Controlled Adder/subtractor (CAS) for the divider. We use three bootstrapped XOR gates, two bootstrapped OR gates, and two bootstrapped AND gates to construct the CAS. The CAS takes \( a_{in}[i] \), \( b_{in}[i] \), \( c_{in}[i] \) and \( p \) as input, and output \( out[i] \) and the carrier \( c[i+1] \). The construction of a CAS is shown in Figure 2. The CAS will perform addition if \( p = 0 \) and perform subtraction if \( p = 1 \).

(2) Then design an absolute value array divider with the CAS above. It takes the 2\( k \)-bit dividend \( a_{in} \) and the \( k \)-bit divisor \( b_{in} \) as input and outputs the quotient \( q \) and the remainder \( r \).

(3) XOR gates are used to decide the sign bit of the quotient to realize the division of both positive and negative integers.

(4) In order to facilitate the operation, the input and output of the divider is unified to complement format. As a result, a compensation device is needed while processing the input and the output. To realize the complement, we first perform XOR gates on the sign bit and each bit, and then add the sign bit to the result, and finally get the complement code.

(5) Construct the \( k \)-bit complement array divider with a absolute value array divider, a XOR gate and two compensation device. The \( k \)-bit complement array divider takes the 2\( k \)-bit complement dividend \( a_{in} \) and the \( k \)-bit complement divisor \( b_{in} \) as input, and output the complement quotient \( q \) and the complement remainder \( r \). The structure of the \( k \)-bit complement array divider is shown in Figure 3. For simplicity, we set \( k = 3n \) in this figure.

Figure 3: The construction of \( k \)-bit complement array divider [39]

3.4 Linear regression

In unary linear regression, there is a calculation formula to obtain the global optimal solution during the training of the model. According to the definition of linear regression, we can evaluate the
accuracy of the model. Suppose we observe $n$ data pairs and call them $\{(x_i, y_i), i = 1, \ldots, n\}$. We implement unary linear regression and utilize its calculation formula to train the model [40]:

$$(\omega^*, b^*) = \arg\min_{(\omega, b)} \sum_{i=1}^{m} (f(x_i) - y_i)^2$$

(1)

By deriving the above formula and making the derivative zero, we can get the calculation formula.

$$\omega = \frac{\sum_{i=1}^{m} y_i(x_i - \bar{x})}{\sum_{i=1}^{m} x_i^2 - \frac{1}{m} (\sum_{i=1}^{m} x_i)^2}, \bar{x} = \frac{1}{m} \sum_{i=1}^{m} x_i$$

(2)

$$b = \frac{1}{m} \sum_{i=1}^{m} (y_i - \omega x_i)$$

(3)

Where $\omega$ is the slope of the line and $b$ is the y-intercept. $\bar{x}$ stands for the average of $x_1, x_2, \ldots, x_m$.

In most linear regression, there is often no calculation formula to help us obtain the global optimal solution, but the common practice is to utilize the Gradient descent method to obtain the local optimal through multiple iterations. The iterative formula of the classical gradient descent method is as follows [40].

$$\theta_j^{t+1} = \theta_j^t - \alpha \frac{\partial f(\theta_j)}{\partial \theta_j}$$

(4)

$$f(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$$

(5)

Where $\alpha$ is the learning rate, $t$ is the number of iterations, $\theta$ is the parameter of model, $\theta_j$ is the gradient of $\theta$, $j$ is the index of parameters to be solved, $m$ is the number of training data and $h_\theta$ stands for the prediction model.

4 THE PROPOSED PRIVACY-PRESERVING DISTRIBUTED MACHINE LEARNING

Although MKTFHE scheme provides bootstrapped NAND gates, it is time-consuming to use NAND gates directly for privacy computing on Industrial Internet. We expand NAND gates to other fundamental gates to better support mathematical operators. At the same time, we construct an adder, a subtractor, a multiplier and a divider, so that users can perform mathematical operators infinitely.

4.1 Homomorphic fundamental gates

In previous works, the other fundamental gates (except the NAND gate) are constructed by joining the multi-key bootstrapped NAND gates together. Although only the multi-key bootstrapped NAND gate is described in [26], any arbitrary binary bootstrapped gate can be evaluated in the same way. Our scheme uses a similar approach in MKTFHE to directly evaluate the following basic gates in a multi-key version.

- **MKBootsAND (c1, c2, {pk_i} in {k_i})**: Given ciphertexts $c_1$ and $c_2$, this algorithm first extends two input ciphertexts into $c'_1$ and $c'_2$ under the same key. Then it evaluates the AND gate by computing:

$$c' = (0, -\frac{1}{8}) + c_1 + c_2$$

(6)

homomorphically on encrypted bits. Then this algorithm evaluates the decryption circuit of the ciphertext $c'$ to bootstrap it. Finally, this algorithm runs the MKTFHE.MKKeySwitch(c, KS\{i\} $\in$ {k_i}) algorithm and outputs a TLWE ciphertext encrypting $m = m_1 \land m_2$.

- **MKBootsOR (c1, c2, {pk_i} in {k_i})**: Given ciphertexts $c_1$ and $c_2$, this algorithm first extends two input ciphertexts into $c'_1$ and $c'_2$ and evaluates the OR gate by computing:

$$c' = (0, \frac{1}{4}) + c_1 + c_2$$

(7)

homomorphically on encrypted bits. Then this algorithm bootstraps it, runs the MKTFHE.MKKeySwitch(c, KS\{i\} $\in$ {k_i}) algorithm and outputs a TLWE ciphertext encrypting $m = m_1 \lor m_2$.

- **MKNOT (c, mparams)**: Take ciphertext $c$ as input, this algorithm evaluates the NOT gate by computing:

$$c' = (0, \frac{1}{4}) - c$$

(8)

donot increase the noise in ciphertext or change the key so that no bootstrapping and key-switching process is needed.

- **MKBootsXOR (c1, c2, {pk_i} in {k_i})**: Given ciphertexts $c_1$ and $c_2$, this algorithm first extends two input ciphertexts into $c'_1$ and $c'_2$ and evaluates the OR gate by computing:

$$c' = (0, \frac{1}{8}) - c_1 - c_2$$

(9)

homomorphically on encrypted bits. Then this algorithm bootstraps it, runs the MKTFHE.MKKeySwitch(c, KS\{i\} $\in$ {k_i}) algorithm and outputs a TLWE ciphertext encrypting $m = m_1 \oplus m_2$.

- **MKBootsXNOR (c1, c2, {pk_i} in {k_i})**: Given ciphertexts $c_1$ and $c_2$, this algorithm first extends two input ciphertexts into $c'_1$ and $c'_2$ and evaluates the XOR gate by computing:

$$c' = \frac{1}{4} - 2 \cdot (c_1 - c_2)$$

(10)

homomorphically. Then run the bootstrapping and key-switching algorithm, and output a TLWE ciphertext encrypting $m = m_1 \odot m_2$.

First of all, a mapping function ModToT is required to transform a message bit $m$ to an element on torus, like $(0, \frac{1}{2})$ in Equation (6), such that all of the following computations are executed on torus. The goal of Equation (6) is to compute $m_1 \land m_2$. Let us look into
Algorithm 1: Multi-key TLWE ciphertexts addition (LweAdd)

**Input:** MKTFHE parameter set, number of the participants $p$ and two MKTLWE ciphertexts $c_1 = (a_1, b_1)$ and $c_2 = (a_2, b_2)$

**Output:** A MKTLWE ciphertext $c = (a, b) = c_1 + c_2$

1. Extract the dimension of the lattice $n$ in parameter set;
2. for $i = 1$ to $p$ do
   3. for $j = 1$ to $n$ do
      4. Compute $\{a\}_i = \{a_1\}_i + \{a_2\}_i$
   5. end
3. end
4. Compute $b = b_1 + b_2$;
5. Refresh the current variance;

Algorithm 2: Multi-key TLWE ciphertexts multiplication (LweMul)

**Input:** MKTFHE parameter set, number of the participants $p$ and two MKTLWE ciphertexts ciphertexts $c_1 = (a_1, b_1)$ and $c_2 = (a_2, b_2)$

**Output:** A MKTLWE ciphertext $c = (a, b) = c_1 \cdot c_2$

1. Extract the dimension of the lattice $n$ in parameter set;
2. for $i = 1$ to $p$ do
   3. for $j = 1$ to $n$ do
      4. Compute $\{a\}_i = \{a_1\}_i + \{a_2\}_i$;
   5. end
3. end
4. Compute $b = b_1 + b_2$;
5. Refresh the current variance;

Algorithm 3: k-bit complement array integer adder

**Input:** MKTFHE parameter set, two ciphertexts $c_1 = $ MKTFHE.SymEnc$(\mu_1)$ and $c_2 = $ MKTFHE.SymEnc$(\mu_2)$ and the public keys of all participants

**Output:** Ciphertext $c = $ MKTFHE.SymEnc$(\mu_1 + \mu_2)$

1. Set the first carry $c_0[0] = 0$;
2. Encrypt addends and carry with public keys;
3. for $i = 1$ to $k - 1$ do
   4. Compute the out$[i]$ and $c_e[i]$;
4. end

Table 1: Operations used in each basic gate in our scheme

| Gate | BS | Add | Sub | Mul | ModToT |
|------|----|-----|-----|-----|--------|
| AND  | √  | √  |     | √  |        |
| OR   | √  | √  |     |     |        |
| NOT  | √  |     | √   |     |        |
| NAND | √  | √  |     | √   |        |
| NOR  | √  | √  |     |     |        |
| XOR  | √  |     | √   | √   |        |
| XNOR | √  |     |     | √   |        |

4.2 Two important components used in bootstrapped gates

This subsection illustrates two components used in bootstrapped gates, LweAdd and LweMul, which are missed in [8]. The idea of LweAdd is to execute addition on the $n + 1$-dimension vectors $c_1$ and $c_2$ element by element. Remember to refresh the current variance by adding the variance of ciphertext $c_1$ and the variance of ciphertext $c_2$ to get the variance of ciphertext $c$, as shown in Algorithm 1. As shown in Equation (4), LweMul is only required in XOR. Such multiplication is dot multiplication, which can be expressed by addend $\{a\}_i \cdot j$ plus $k$-time $\{a\}_i \cdot j$. Remember to refresh the current variance by adding the variance of ciphertext $c_1$ and the variance of ciphertext $c_2$ times $k^2$ to get the variance of ciphertext $c$, as shown in Algorithm 2.

However, two components used in bootstrapped gates above, LweAdd and LweMul, which are missed in [8]. The idea of LweAdd is to execute addition on the $n + 1$-dimension vectors $c_1$ and $c_2$ element by element. Remember to refresh the current variance by adding the variance of ciphertext $c_1$ and the variance of ciphertext $c_2$ to get the variance of ciphertext $c$, as shown in Algorithm 1. As shown in Equation (4), LweMul is only required in XOR. Such multiplication is dot multiplication, which can be expressed by addend $\{a\}_i \cdot j$ plus $k$-time $\{a\}_i \cdot j$. Remember to refresh the current variance by adding the variance of ciphertext $c_1$ and the variance of ciphertext $c_2$ times $k^2$ to get the variance of ciphertext $c$, as shown in Algorithm 2.

We summarize the operations between two LWE ciphertexts used in each basic binary gate in Table 1.

4.3 k-bit homomorphic adder

After achieving various basic gate operations in subsection 4.1, it is still a gap between such gates and complicated machine learning functions or private computing directly. In this subsection, we construct a $k$-bit complement array integer adder based on the MKTFHE scheme. We choose complement representation instead of true form or inverse code, for the complement can compute both positive and negative numbers in the same way. A $k$-bit complement array integer adder can be constructed in the following four steps. The main idea was described in subsection 3.3. The only difference is that bootstrapping BS is necessary after each computation to decrease noise. The $k$-bit complement array integer adder is shown in Algorithm 3.

1. Construct a homomorphic semi-adder with two bootstrapped XOR gates: Input two ciphertexts $c_a$ and $c_b$ and the
We also construct a homomorphic 1-bit adder by merging a homomorphic semi-adder and a homomorphic carrier: Input two ciphertexts $c_a$ and $c_b$ and the encrypted carry $c_c$, return $\text{out} = \text{BS}(c_c \oplus (\text{BS}(c_a \oplus c_b)))$, where $\text{BS}$ stands for the bootstrapping process.

(2) Construct a homomorphic carrier with three bootstrapped AND gates and two bootstrapped OR gates: Input two ciphertexts $c_a$ and $c_b$ and the encrypted carry $c_c$, return $\text{cout} = \text{BS}(\text{BS}(c_a \land c_b) \lor \text{BS}(c_a \land c_c) \lor \text{BS}(c_b \land c_c))$.

(3) Construct a homomorphic 1-bit adder by merging a homomorphic semi-adder and a homomorphic carrier: Input two ciphertexts $c_a$ and $c_b$ and the encrypted carry $c_c$, return $\text{out} = \text{BS}(c_c \oplus (\text{BS}(c_a \oplus c_b)))$ and $\text{cout} = \text{BS}(\text{BS}(c_a \land c_c) \lor \text{BS}(c_b \land c_c))$.

(4) Construct a homomorphic k-bit complement array integer adder with k homomorphic 1-bit adder: Input two ciphertexts $c_a$ and $c_b$, set the encrypted carry $c_c[0] = 0$, compute $out[i] = \text{BS}(c_a[i] \oplus (\text{BS}(c_a[i] \oplus c_b[i])))$ and $c_c[i + 1] = \text{BS}(\text{BS}(c_a[i] \land c_b[i]) \lor \text{BS}(c_a[i] \land c_c[i]) \lor \text{BS}(c_b[i] \land c_c[i]))$, where $i = [0, 1, \ldots, k-1]$. Return the result $\text{out}$.

In k-bit complement array integer adder, the output carry of the last 1-bit adder is abandoned because of the capture of the complement.

![Figure 4: The construction of k-bit homomorphic complement array integer adder. GB stands for gate-bootstrapping.](image)

4.4 k-bit homomorphic multiplier

We also construct a k-bit complement array integer multiplier based on the MKTFHE scheme. A k-bit complement array integer multiplier can be constructed in the following steps. The main idea was described in subsection 3.3. The only difference is that bootstrapping $\text{BS}$ is necessary after each computation to decrease noise. A k-bit complement array integer multiplier is shown in Algorithm 4.

1. Prepare three kinds of homomorphic 1-bit adder:
   (a) 0-homadder: The same as the homomorphic 1-bit adder above.
   (b) 1-homadder: Input two ciphertexts $c_a$ and $c_b$ and the encrypted carry $c_c$, return $out = \text{BS}(c_c \oplus (\text{BS}(c_a \oplus c_b)))$ and $c_{out} = \text{BS}(\text{BS}(\text{BS}(\text{BS}(c_a \land c_b) \lor \text{BS}(\text{BS}(c_a \land c_c) \lor \text{BS}(c_b \land c_c))))$.
   (c) 2-homadder: Input two ciphertexts $c_a$ and $c_b$ and the encrypted carry $c_c$, return $out = BS(c_c \oplus (BS(c_a \oplus c_b)))$ and $c_{out} = BS(BS(BS(c_a \land c_b) \lor BS(c_a \land c_c)) \lor BS(c_b \land c_c))$.

2. Construct a k-bit homomorphic complement array integer multiplier according to the rules in Section 3.4 but use homomorphic adder instead of the previous adder.

We use Algorithm 4 to evaluate k-bit complement array integer multiplier homomorphically, where $k$ is the number of bits of the addends. In k-bit complement array integer multiplier, the output carry of the last 1-bit adder is considered as the highest bit of the result.

4.5 k-bit homomorphic subtractor and divider

We construct a k-bit complement array integer subtractor by using a k-bit complement array integer adder and a k-bit complement array integer multiplier. A k-bit complement array integer subtractor is shown in Algorithm 5.

We also construct a k-bit complement array integer divider based on the MKTFHE scheme. The main idea was described in subsection 3.3. The only difference is that bootstrapping $\text{BS}$ is necessary after each computation to decrease noise. A k-bit complement array integer divider can be constructed in the following steps:

1. Prepare the homomorphic CAS for the divider. We use three homomorphic bootstrapped XOR gates, two homomorphic bootstrapped OR gates and two homomorphic bootstrapped AND gates to construct the homomorphic CAS. The homomorphic CAS takes ciphertexts $c_a[i]$, $c_b[i]$, $c_c[i]$ and $c_p$
Algorithm 5: $k$-bit complement array integer subtractor

**Input:** MKTFHE parameter set, two ciphertexts
$c_1 = \text{MKTFHE}.\text{SymEnc}(\mu_1)$ and $c_2 = \text{MKTFHE}.\text{SymEnc}(\mu_2)$ and the public keys of all participants.

**Output:** A ciphertext $c = \text{MKTFHE}.\text{SymEnc}(\mu_1 - \mu_2)$

1. Change every bit of $c_2$ and add 1 to calculate $-c_2$;
2. Set the first carry $c_c[0] = 0$;
3. Encrypt addends and carry with public keys;
4. for $i = 1$ to $k - 1$
   5. Compute the $\text{out}[i]$ and $c_e[i]$;
end

Algorithm 6: $k$-bit complement array integer divider

**Input:** MKTFHE parameter set, two ciphertexts
$c_1 = \text{MKTFHE}.\text{SymEnc}(\mu_1)$ and $c_2 = \text{MKTFHE}.\text{SymEnc}(\mu_2)$ and the public keys of all participants.

**Output:** Two ciphertexts $q = \text{MKTFHE}.\text{SymEnc}(\mu_1/\mu_2)$ and $c = \text{MKTFHE}.\text{SymEnc}(\mu_1 \mod \mu_2)$

1. Encrypt the $p_0$ with public keys;
2. Change the inputs into complement format with compensation device;
3. Calculate the first later of divider;
4. for $i = 1$ to $k - 1$
   5. for $j = 0$ to $k - 1$
   6. Compute the $\text{CAS}$ unit;
end

as input, and output $c_{\text{out}}[i]$ and the carrier $c_c[i + 1]$ satisfying $c_{\text{out}}[i] = \text{BS}(c_1[i] \oplus \text{BS}(c_2[i] \oplus c_p) \oplus c_c[i])$ and $c_c[i + 1] = \text{BS}(\text{BS}(c_1[i] \lor c_c[i]) \land \text{BS}(c_2[i] \lor c_p) \lor \text{BS}(c_1[i] \land c_c[i]))$. The homomorphic CAS will perform addition if the message of $c_p$ is 0 and perform subtraction if the message of $c_p$ is 1.

(2) Then design the homomorphic absolute value array divider with the homomorphic CAS above. It takes the $2k$-bit dividend $c_a$ and the $k$-bit divisor $c_b$, as input, and output the quotient $q$ and the remainder $c_r$.

(3) Homomorphic XOR gates are used to decide the sign bit of the ciphertext of quotient to realize the homomorphic division of both positive and negative integers.

(4) A homomorphic compensation device is also designed to realize the complement division.

(5) Construct the $k$-bit homomorphic complement array divider with a homomorphic absolute value array divider, a homomorphic XOR gate and two homomorphic compensation devices. The $k$-bit homomorphic complement array divider takes the ciphertexts of $2k$-bit complement dividend $a_{2k}$ and the $k$-bit complement divisor $b_{2k}$ as input, and output the ciphertexts of complement quotient $q$ and the complement remainder $r$.

We use Algorithm 6 to evaluate $k$-bit complement array integer divider homomorphically, where $k$ is the number of bits of the divisor.

In summary, this section first expands NAND gates to other six basic gates, AND, OR, NOT, NOR, XOR and XNOR. In order to compute these gates, two important components, $\text{lweAdd}$ and $\text{lweMult}$ are then proposed. At last, we construct the adder, subtractor, multiplier and divider. With these homomorphic operators, we can evaluate arbitrary polynomials without decryption.

4.6 Multi-key homomorphic distributed linear regression scheme

We first implement a multi-key fully homomorphic encryption scheme in which the secret key is distributed among the parties, while the corresponding collective public key $pk$ is known to all of them. Thus, each party can independently compute on ciphertexts encrypted under $pk$ but all parties have to collaborate to decrypt a ciphertext. This enables the participants to train a collectively encrypted model, that cannot be decrypted as long as one participant is honest and refuses to participate in the decryption. Then we implement two methods: calculation formula method and GD method to train and evaluate the linear regression model.

Calculation formula method: The advantage of this method is that the global optimal solution can be obtained without multiple iterative operation which can reduce the computational cost of time and space. We first estimate the size of the input data, and then select homomorphic operators with appropriate bits to calculate the above formula. Since our operators are integer operators, we choose to directly round off the input data under the condition of ensuring accuracy. We use Algorithm 7 to train the linear regression model under multi-key homomorphic encryption.

GD method: Considering that our homomorphic operators only support integer calculation, and the learning rate $\alpha$ is usually a floating-point number less than 1, we rewrite the GD iterative formula in Subsection 3.4 to zoom the $\alpha$ to integers. We bring the linear regression model into the origin GD iterative formula, the zooming multiple is $n$, and the practical integer GD iterative formula is below:

$$b' = b \cdot n - \alpha \cdot n \cdot \frac{2}{m} \sum_{i=1}^{m} [y_i \cdot n - (\frac{\omega}{n} \cdot x_i - \frac{b}{n})]$$

$$\omega' = \omega \cdot n - \alpha \cdot n \cdot \frac{2}{m} \sum_{i=1}^{m} [y_i \cdot n - (\frac{\omega}{n} \cdot x_i - \frac{b}{n})]$$

$$\text{loss} = \frac{1}{m} \cdot \sum_{i=1}^{m} |y_i - (\omega \cdot x_i + b)/n|^2$$

After we get the rewritten iterative formula, we estimate the size of the input data to select the zooming multiple, iterative times and homomorphic operators with appropriate numbers of bits.

We use Algorithm 8 to train the linear regression model under multi-key homomorphic encryption.

5 IMPLEMENTATION AND EXPERIMENTS

The test environment is Ubuntu 18.04 operation system, with Intel Xeon Gold 5220 CPU and 46512MiB memory.
The most time-consuming part of the gate circuit is gate bootstrapping, which determines the calculation time of the gate circuit specifically, the number of gate bootstrapping, determines the calculation time of the gate circuit. Specifically, the AND gate needs 2 NAND gates to implement, the OR gate needs 3, the NOT gate needs 1, the NOR gate needs 4, the XOR gate needs and the XNOR gate needs 5.

The result in Table 2 shows that our scheme is more efficient than the naive scheme, which connects the bootstrapped NAND gates to realize the operations. We have improved in efficiency, especially in bootstrapped OR gate, bootstrapped NOR gate, NOT gate, bootstrapped XOR gate, and bootstrapped XNOR gate: the time cost by a single bootstrapped XOR gate reduced by about 67% while the NOT gate is evaluated almost instantaneously for no bootstrapping is needed.

### Table 2: Expression of our basic bootstrapped gates and the average time of each gate in both our scheme and naive scheme.

| Gate    | Naive scheme(s) | NAND gates | Our scheme(s) |
|---------|-----------------|------------|---------------|
| AND    | 0.244689        | 2          | 0.12231       |
| OR     | 0.365409        | 3          | 0.12230       |
| NOT    | 0.1222991       | 1          | 3e-06         |
| NOR    | 0.483549        | 4          | 0.12227       |
| XOR    | 0.486477        | 4          | 0.12223       |
| XNOR   | 0.609807        | 5          | 0.12223       |

### 5.2 Experiments on k-bit homomorphic adder

We construct a k-bit homomorphic complement integer adder based on the basic bootstrapped gates we designed above. We create two participants for our experiment. Each of them creates its addend and secret key individually. For simplicity, we set $k = 1, 2, \ldots, 8$.

The server input multi-key parameters, the ciphertexts encrypted under different keys by two participants, and the public keys of two participants. Then the server extends the ciphertext, evaluates the k-bit adder homomorphically, and returns the ciphertext to the participants. The participants decrypt the ciphertext and get the result.

According to the structure of k-bit homomorphic adder, while computing the addition two k-bit addends, k 1-bit adder is required. As shown in subsection 4.2, it requires 5 bootstrapped gates to construct a 1-bit adder. That is to say, 5k basic bootstrapped gates are needed to construct a k-bit homomorphic adder. Obviously, the cost of the homomorphic evaluation grows linearly with the number of bootstrapped gates. So, the cost of the adder grows linearly with the bits of input numbers.

### Table 3: The average time of k-bit homomorphic adder.

| Bits  | Time(s) | Num of gates | Time/gate(s) |
|-------|---------|--------------|--------------|
| 1-bit | 1.19515 | 5            | 0.23903      |
| 2-bit | 2.39738 | 10           | 0.239738     |
| 3-bit | 3.60485 | 15           | 0.240523     |
| 4-bit | 4.80614 | 20           | 0.240307     |
| 5-bit | 6.01884 | 25           | 0.240754     |
| 6-bit | 7.25483 | 30           | 0.241828     |
| 7-bit | 8.43608 | 35           | 0.241031     |
| 8-bit | 9.69468 | 40           | 0.242367     |

### Table 4: The average time of k-bit homomorphic adder in both our schemes and naive scheme.

| Bits  | Our Schemes(s) | Naive Scheme(s) | Reduction Rate |
|-------|----------------|-----------------|----------------|
| 4-bit | 6.82           | 15.02           | 54.59%         |
| 8-bit | 13.90          | 29.96           | 53.60%         |

From Table 3 and Figure 5, we can learn that the number of binary bootstrapped gates grows linearly with the number of the bits of addends. As evaluation is the most expensive step in our scheme, we can say that the cost of the k-bit homomorphic complement adder grows linearly with the bits of input numbers. And from Table 4, we can find that compared to the naive scheme, our scheme reduces the consumption time of the k-bit homomorphic complement adder by 53~54%.
5.3 Experiments on \( k \)-bit homomorphic multiplier

To construct a \( k \)-bit homomorphic multiplier, we construct 3 kinds of homomorphic adder: 0-homadder, 1-homadder and 2-homadder. We also construct a \( k \)-bit homomorphic complement integer multiplier based on three kinds of adders and the basic bootstrapped gates we designed above. The structure of 3 kinds of homomorphic adder in section 4.3 shows that it requires 7 basic bootstrapped gates to construct a homomorphic adder: 5 gates in 1-bit adder and two AND gates to compute the input. It can be learned from the structure of the \( k \)-bit homomorphic multiplier in section 3.3 that it requires \( k \times (k - 1) \) homomorphic adders to construct a \( k \)-bit homomorphic multiplier. That is to say, it requires \( 7 \times k \times (k - 1) \) basic bootstrapped gates for a \( k \)-bit homomorphic multiplier. So, the cost of the multiplier grows quadratically with the bits of input numbers. Note that there is parallel optimization in the multiplier, so the average time consumed by a gate is much less than that consumed by a single gate.

We create two participants for our experiment, each of them creates its complement number and secret key individually. For simplicity, we set \( k = 2, 3, \ldots, 8 \). The server input multi-key parameters, the ciphertexts encrypted under different keys by two participants, and the public keys of two participants. Then the server extends the ciphertext, evaluates the \( k \)-bit multiplier homomorphically, and returns the ciphertext to the participants. The participants decrypt the ciphertext and get the result.

Table 5: The average time of each step of experiments on \( k \)-bit homomorphic adder, as well as the number of gates used in multiplier.

| Bits | Time(s) | Gates | Time/gate(s) |
|------|---------|-------|--------------|
| 2-bit| 1.14368 | 14    | 0.081691429  |
| 3-bit| 2.66844 | 42    | 0.063534286  |
| 4-bit| 4.83101 | 84    | 0.057512024  |
| 5-bit| 7.43843 | 140   | 0.053131643  |
| 6-bit| 10.762  | 210   | 0.051247619  |
| 7-bit| 14.6522 | 294   | 0.049837415  |
| 8-bit| 19.1052 | 392   | 0.048737755  |

Table 6: The average time of \( k \)-bit homomorphic multiplier in both our schemes and naive scheme.

| Bits | Our Schemes(s) | Naïve Scheme(s) | Reduction Rate |
|------|----------------|-----------------|----------------|
| 4-bit| 19.52          | 60.53           | 67.77%         |
| 8-bit| 86.11          | 268.17          | 67.89%         |

Figure 5: The cost of \( k \)-bit homomorphic complement array integer adder

Figure 6: The cost of \( k \)-bit homomorphic complement array integer multiplier

5.4 Experiments on \( k \)-bit homomorphic subtractor and divider

From Table 5 and Figure 6, we can learn that the number of binary bootstrapped gates grows quadratically with the number of the bits of addends. Similarly, we can say that the cost of the \( k \)-bit homomorphic complement multiplier grows quadratically with the bits of input numbers. And from Table 6, we can find that compared to the naive scheme, our scheme reduces the consumption time of the \( k \)-bit homomorphic complement multiplier by 67%.
Table 7: The average time of experiments on $k$-bit homomorphic subtractor

| Bits   | Time(s) | Gates | Time/gate(s) |
|--------|---------|-------|--------------|
| 1-bit  | 1.99209 | 6     | 0.332015     |
| 2-bit  | 2.90969 | 12    | 0.242474167 |
| 3-bit  | 4.45103 | 18    | 0.24729444  |
| 4-bit  | 5.84264 | 24    | 0.243443333 |
| 5-bit  | 7.42889 | 30    | 0.247629667 |
| 6-bit  | 8.93639 | 36    | 0.248233056 |
| 7-bit  | 10.35   | 42    | 0.246428571 |
| 8-bit  | 11.9013 | 48    | 0.24794375  |

Table 8: The average time of $k$-bit homomorphic subtractor in both our schemes and naïve scheme.

| Bits   | Our Schemes(s) | Naïve Scheme(s) | Reduction Rate |
|--------|----------------|-----------------|----------------|
| 4-bit  | 7.32           | 23.11           | 68.26%         |
| 8-bit  | 14.87          | 45.81           | 67.68%         |

Figure 7: The cost of $k$-bit homomorphic complement array integer subtractor

For each hom-CAS while there are $k^2$ hom-CASs in each homomorphic absolute value array divider. It takes $2k$ homomorphic gates to construct a compensation device. As a result, $7k^2 + 2k + 1$ basic bootstrapped gates are needed to construct a $k$-bit homomorphic divider. Note that there are more optimization in the divider than in the multiplier, and the CAS almost run in parallel, so a $k$-bit divider is faster than a $k$-bit multiplier, and the cost of the divider grows almost linearly with the number of layers.

Table 9: The average time of experiments on $k$-bit homomorphic divider

| Bits   | Time(s) | Layers | Time/layer(s) |
|--------|---------|--------|---------------|
| 1-bit  | 1.483   | 1      | 1.483         |
| 2-bit  | 4.09858 | 2      | 2.04929       |
| 3-bit  | 7.08719 | 3      | 2.362396667  |
| 4-bit  | 10.0022 | 4      | 2.50055       |
| 5-bit  | 13.1937 | 5      | 2.63874       |
| 6-bit  | 15.965  | 6      | 2.660833333  |
| 7-bit  | 19.1003 | 7      | 2.728614286  |
| 8-bit  | 22.2938 | 8      | 2.786725      |

Table 10: The average time of $k$-bit homomorphic divider in both our schemes and naïve scheme.

| Bits   | Our Schemes(s) | Naïve Scheme(s) | Reduction Rate |
|--------|----------------|-----------------|----------------|
| 4-bit  | 48.23          | 149.67          | 67.78%         |
| 8-bit  | 153.80         | 494.62          | 68.90%         |

Figure 8: The cost of $k$-bit homomorphic complement array integer divider

5.5 Experiments on multi-key homomorphic distributed linear regression

We implement a multi-key fully homomorphic encryption scheme in which the secret key is distributed among the parties, while the corresponding collective public key $pk$ is known to all of them. Thus, each party can independently compute on ciphertexts encrypted under $pk$ but all parties have to collaborate to decrypt a ciphertext. This enables the participants to train a collectively encrypted model, that cannot be decrypted as long as one participant is honest and refuses to participate in the decryption.

Our multi-key fully homomorphic linear regression scheme is composed by multiple parties which can be divided by three types of entities: participants, cloud server, and decryption party, we take two participants as examples. The whole scheme are shown in Figure 9, the steps are as follows:

(1) Participants have the need to outsource computing, and each participant offers its own part of data for model training. During the step of data encryption, all participants call MKTFHE.KeyGen($mkparams$) to generate their symmetric key, asymmetric key bootstrapping key and key-switching key independently, then call MKTFHE.SymEnc($\mu$) to encrypt the data. And finally upload the ciphertext of input data and public key to the cloud server.

(2) Cloud server is usually composed of one or more high-performance servers, and doesn’t have their own data. After receiving the ciphertext data from participants, the cloud server will use the participants’ key to generate a new bootstrapping key and extend the single-key ciphertext of the input data to multi-key ciphertext. Then cloud server uses our designed homomorphic operators to train the linear regression model. After the training is finished, the result of the model will be sent to the decryption party.
(3) When the decryption party receive the ciphertext of the trained model, they will unite all the participants to call MKTFHE.SymDec(c, \{sk_i\}) to decrypt the result.

![Figure 9: Multi-key fully homomorphic linear regression scheme](image)

We train a multi-key fully homomorphic linear regression model in two methods by using our proposed homomorphic operators. The input data is composed of several sets of linear data and some random noise. Considering the size of the data, we chose 8-bit homomorphic operators to train the model in formula method and 16-bit homomorphic operators in GD method. In addition, the learning rate and the zooming multiple of the GD method is 0.001 and 10,000. We make several experiments and find that the model will converge by performing up to 10 iterations. We also create m participants in our experiment. Each of them generates their own secret key to encrypt their own data. For simple, we set m = 2, 4, 8, and the 3, 4, 5, 6 columns record the time of each step.

Figure 9: Multi-key fully homomorphic linear regression scheme

In Table 11, the result shows that the running time including key reconstruction time, extension ciphertext time all grow linearly with the numbers of participants; due to the large zooming multiple of the GD method, the homomorphic operators with larger bits are selected.

In Table 12, the result shows that compared to the naïve scheme, our scheme reduces the training time of the multi-key fully homomorphic linear regression by 65~66%, reduces the evaluation time by 66~67%. At the same time, the GD method needs multiple iterations, and the calculation process is more complex, so the running time including training time and evaluation time is much more than that of the formula method.

Note that there is more optimization in our linear regression scheme such as running in parallel. Besides, with our homomorphic operators, we can implement more complex multi-key fully homomorphic machine learning scheme.

| Method   | Num of Parties | KeyGen(s) | Ciphertext extension(s) |
|----------|----------------|-----------|-------------------------|
| Formular | k = 2          | 1.984     | 0.0008                  |
| Formular | k = 4          | 4.002     | 0.0016                  |
| Formular | k = 8          | 8.916     | 0.0035                  |
| GD       | k = 2          | 2.025     | 0.0008                  |

Table 11: The result of experiments on multi-key fully homomorphic linear regression

**Security Analysis.** Our scheme is semantically secure under the (R)LWE assumption and we show our setting of the parameters in Table 8, following the notation of the MKTFHE library [8]. The achieved security level of our scheme is 110-bit while the dimension of the TLWE problem is k = 1.

**6 CONCLUSION AND DISCUSSION**

**6.1 Conclusion**

The scheme proposes a series of k-bit homomorphic complement operators based on MKTFHE, which narrows the gap between the original NAND gate and complicated machine learning functions. Experiment shows that the cost of the adder and the subtractor grows linearly with the bits of input numbers and the cost of the multiplier grows quadratically. Meanwhile, the cost of the divider grows almost linearly with the number of layers in it.

To narrow the gap between multi-key bootstrapped NAND gates and multi-key homomorphic mathematical operators, we construct other basic bootstrapped gates (AND, OR, NOT, NOR, XOR, and XNOR) in the same way as the bootstrapped NAND gate in MKTFHE and with the same efficiency as the NAND gate. Experiment shows that constructing basic binary gates in this way is much more efficient than building them by directly joining the bootstrapped NAND gates together, especially when constructing XOR gate and NOT gate. Then we construct a k-bit complement adder, a k-bit complement subtractor, a k-bit complement multiplier and a k-bit complement divider based on our basic binary bootstrapped gates. Finally, we train distributed linear regression model by utilizing our proposed multi-key homomorphic operators. The operators we designed can be directly used to achieve privacy-preserving distributed machine learning schemes in distributed communication system.

**6.2 Discussion**

The plaintext space of the MKTFHE scheme is \{0, 1\}, which is smaller than the plaintext space in other homomorphic encryption schemes like BGV [6] or CKKS [11, 12]. Extending the message space on torus may be helpful. It has been experimented that integers or fixed-point numbers can also be mapped to a ring on the torus [3, 4] as well as performing homomorphic evaluations. If the message space of MKTFHE is magnified to integers or fixed-point numbers, it may reduce the overhead in time and space in MKTFHE.

Common reference strings (CRS) are needed in this scheme for all the participants, and the computing server has to know the multi-key parameters in advance. If the CRS can be removed in MKTFHE, the scheme will no longer need a trusted third party to generate the CRS, making the scheme more secure.

We described only multi-key homomorphic linear regression model in our scheme, but more machine learning schemes (such as logistic regression, SVM, etc) can be evaluated in the same way, for the basic mathematical operators are provided in our scheme.

**ACKNOWLEDGMENTS**

This work is supported by National Natural Science Foundation of China (62272131), Shenzhen Basic Research Project, China (JCYJ2020 0109113405927) and National Science and Technology Major Project Carried on by Shenzhen, China.
Table 12: The result of multi-key fully homomorphic linear regression in both our schemes and naïve scheme.

| Method | Num of Parties | Our Schemes(s) | Training Naïve Scheme(s) | Reduction Rate | Our Schemes(s) | Evaluation Naïve Scheme(s) | Reduction Rate |
|--------|----------------|----------------|--------------------------|---------------|----------------|--------------------------|---------------|
| Formula | \( k = 2 \) | 515.77 | 1591.32 | 65.32% | 43.64 | 133.21 | 67.23% |
| Formula | \( k = 4 \) | 1105.21 | 3173.34 | 65.17% | 82.68 | 252.57 | 67.26% |
| Formula | \( k = 8 \) | 2203.78 | 6335.17 | 65.21% | 164.09 | 508.75 | 67.74% |
| GD     | \( k = 2 \) | 1717.61 | 5202.83 | 66.98% | 43.71 | 131.20 | 66.68% |

Table 13: Parameter sets of multi-key homomorphic linear regression

| \( LWE-n \) | \( LWE-\alpha \) | \( LWE-B' \) | \( LWE-d' \) | \( RLWE-N \) | \( RLWE-\beta \) | \( RLWE-B \) | \( RLWE-d' \) |
|-------------|---------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 560         | \( 3.05 \times 10^{-5} \) | \( 2^x \) | 8           | 1024        | \( 3.72 \times 10^{-3} \) | \( 2^y \) | 3           |

(0J2ZGD20200617103000001), Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (2022B1 212010005).
[33] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. 2012. 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. 1219–1234.

[34] Pratyay Mukherjee and Daniel Wichs. 2016. Two round multiparty computation via multi-key FHE. In Advances in Cryptology – EUROCRYPT 2016 (Lecture Notes in Computer Science, Vol. 9666). 735–763.

[35] Chris Peikert and Sina Shiehian. 2016. Multi-key FHE from LWE, Revisited. In Theory of Cryptography - 14th International Conference, TCC 2016-B (Lecture Notes in Computer Science, Vol. 9986). 217–238.

[36] Jerome Saltzer and M Frans Kaashoek. 2009. Principles of computer system design: an introduction. Morgan Kaufmann.

[37] Yifan Tian, Laurent Njilla, Jiawei Yuan, and Shucheng Yu. 2021. Low-Latency Privacy-Preserving Outsourcing of Deep Neural Network Inference. IEEE Internet Things J. 8, 5 (2021), 3300–3309.

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

[39] Haridimos T Vergos. 2007. An Efficient BIST Scheme for Non-Restoring Array Dividers. In 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007). IEEE, 664–667.

[40] Zhi-Hua Zhou. 2021. Machine learning. Springer Nature.