Multi-key Fully Homomorphic Encryption Scheme with Compact Ciphertext

Tanping Zhou*
Institute of Software,
Chinese Academy of Sciences
Beijing China
tanping2020@iscas.ac.cn

Zhenfeng Zhang
Institute of Software,
Chinese Academy of Sciences
Beijing China
zfzhang@tca.iscas.ac.cn

Long Chen
Institute of Software,
Chinese Academy of Sciences
Beijing China
longchen@njit.edu

Xiaoliang Che
Cryptography Engineering
Engineering University of PAP
Xi'an Shaanxi China
smo_mrcche@yeah.net

Wenchao Liu
Cryptography Engineering
Engineering University of PAP
Xi'an Shaanxi China
mr_yangxy@yeah.net

Xiaoyuan Yang*
Cryptography Engineering
Engineering University of PAP
Xi'an Shaanxi China
mr_yangxy@yeah.net

ABSTRACT
Multi-Key fully homomorphic encryption (MKFHE) allows computation on data encrypted under different and independent keys. The previous researches show that the ciphertext size of MKFHE scheme usually increases linearly or squarely with the number of parties, which restricts the application of the MKFHE scheme. In this paper, we propose a general construction of MKFHE scheme with compact ciphertext. Firstly, we construct the accumulated public key of the parties set with compact by accumulating every party’s public key under the CRS model. Secondly, all parties provide the ciphertext of their secret keys key which is encrypted by the accumulated public-key as the accumulated evaluation key. Thirdly, we run the bootstrapping process (or key switching process) on each party's ciphertext and accumulated evaluation key to refresh the ciphertext. Finally, We homomorphically calculate the refreshed ciphertext and decrypt it by the joint secret key. Furthermore, according to the advantages of TFHE-type scheme’s efficient bootstrapping and CKKS scheme supporting approximate data homomorphic computation, we improve the bootstrapping in our general scheme and specifically propose two efficient MKFHE schemes with compact ciphertext.

Our work has two advantages. The one is that the ciphertext size of the proposed general scheme is independent of the number of parties, and the homomorphic computation is as efficient as the single-party full homomorphic encryption scheme. When the parties' set is updated, the ciphertext of the original set can continue to be used for homomorphic computation of the new parties' set after refreshed. Another advantage is that only by authorization can a party’s data be used in the homomorphic operation of a set, i.e., all parties need to regenerate their accumulated evaluation key with the set. Compared with the fully dynamic MKFHE scheme, the authorized MKFHE scheme we proposed supports parties to effectively control which set their data.

KEYWORDS
Multi-key Fully homomorphic encryption, Lattice cipher, Bootstrapping process, Homomorphic decryption

1 INTRODUCTION
Single-party Fully homomorphic encryption (FHE) is a cryptographic scheme that enables homomorphic operations on encrypted data without decryption. Many of HE schemes (eg.1-13) have been suggested following Gentry’s blueprint [3]. The typical FHE schemes can only support homomorphic computation of ciphertext for a single party, that is, all ciphertexts participating in computation correspond to the one secret key. However, in many scenarios, it is usually necessary to calculate the data uploaded to the cloud by multi-party in the network. In 2012, López-Alt et al. [14] proposed a multi-key fully homomorphic encryption (MKFHE) scheme, which is a variant of FHE allowing computation on data encrypted under different and independent keys. One of the most appealing applications of MKFHE is to construct on-the-fly multiparty computation (MPC) protocols. The process of MKFHE is shown in Figure 1.
1.1 Background

The MKFHE schemes are mainly divided into four types: NTRU-type MKFHE, GSW-type MKFHE, BGV-type MKFHE and TFHE-type MKFHE.

In 2012, López-Alt et al. first proposed the NTRU-type MKFHE based on the NTRU cryptosystem[15], which was optimized later in DHS16[2]. In PKC2017, Chongchitmate et al. proposed a general transformation framework CO17[16] from MKFHE to MKFHE with circuit privacy, and constructed a three-round dynamic secure multi-party computation protocol. However, the security of this construction is based on a new and somewhat non-standard assumption on polynomial rings.

In CRYPTO2015, Clear and McGoldrick proposed the first GSW-type MKFHE scheme CM15 based on LWE problem [17], which proposes a transformation mode from FHE to MKFHE. The ciphertext of the single-party FHE is expanded to a new large ciphertext, which corresponds to the cascaded secret key of all parties. Then, the extended ciphertexts are used for homomorphic computation, and the final ciphertext is decrypted jointly by all parties. This transformation mode is widely adopted by most MKFHE schemes based on LWE or RLWE problems. In EUROCRYPT 2016, Mukherjee and Wichs presented a construction of MKFHE MW16[18] based on LWE that simplifies the scheme of CM15 and admits a simple 1-round threshold decryption protocol. Based on this threshold MKFHE, they successfully constructed a general two-round MPC protocol upon it in the common random string model. The schemes CM15 and MW16 need to determine all the involved parties before the homomorphic computation and do not allow any new party to join in, which is called single-hop MKFHE[19]. In TCC2016, Peikert and Shiehian proposed a notion of multi-hop MKFHE PS16[19], in which the result ciphertexts of homomorphic evaluations can be used in further homomorphic computations involving additional parties (secret keys). That is, any party can dynamically join the homomorphic computation at any time. However, the disadvantage is that the number of parties is limited. In CRYPTO2016, A similar notion named fully dynamic MKFHE BP16[20] was proposed by Brakerski and Perlman. A slight difference is that in fully dynamic MKFHE the bound of the number of parties does not need to be input during the setup procedure. The length of extended ciphertext only increases linearly with the number of parties. However, in the process of homomorphic computation, the scheme needs to use the parties’ joint public key to run the bootstrapping process, so the efficiency of ciphertext computation is low.

In TCC2017, Chen et al. proposed the first BGV-type multi-hop MKFHE scheme CZW17[21]. They used GSW-type expansion algorithm to encrypt the secret key to generate the joint evaluation key of the parties set. CZW17 supports the ciphertext packaging technology based on Chinese remainder theorem (CRT), and can be used to construct 2-round MPC protocol and threshold decryption protocol. In 2019, Li et al. put forward a nested ciphertext extension method LZY+19[22], which reduces the evaluation key and the expansion ciphertext size. In 2019, Chen et al. optimized the relinearization process and constructed an efficient MKFHE CDKS19 [23]. Because of its efficient homomorphic computation, it is applied to the neural network to perform the privacy computation.

In ASIACRYPT2016, Chilotti et al. constructed the full homomorphic scheme CGGI16[24] based on a variant of GSW13 on the $\mathbb{Z}=[0,1]$ ring TGSW. In the scheme, the external product of TGSW ciphertext (matrix) and TLWE ciphertext (vector) is used to replace the product of TGSW ciphertext (matrix) and TGSW ciphertext (matrix). Therefore, the addition operation on polynomial exponent is more efficient, such that the time of bootstrap process and the size of bootstrap key are greatly reduced. In ASIACRYPT2017, Chilotti et al. optimized the accumulation process in the CGGI16 scheme and proposed CGGI17[25], which reduced the bootstrapping time to 13ms. In the follow-up work, they wrote the FHE software library TFHE. In ASIACRYPT2019, Chen et al. designed an efficient ciphertext expansion algorithm based on CGGI17, realized the efficient expansion evaluation key, and proposed an MKFHE scheme CCS19[26]. The ciphertext length of the scheme increases linearly with the number of parties. And also, they compiled an MKFHE software library MKTFHE, which has important guiding significance for the application of MKFHE schemes.

1.2 Our Contributions

Throughout the paper, there are many definitions of each party. Here we give a simple description of them. For the party $i$, firstly he selects his secret key (sk) and generates the corresponding public key (pk). Then it provides the part of its public key to generate the joint public key (jpk). He uses the jpk to encrypt his ciphertext for generating the accumulated ciphertext. Party $i$ uses the jpk encrypt his secret key for generating the evaluation key (ek), uses jpk encrypt the joint ciphertext for generating the accumulated evaluation key (aek). Similarly, party $i$ generates switching key (wk), accumulated switching key (awk), bootstrapping key (bk), and accumulated bootstrapping key (abk) that participate in homomorphic operation according to jpk.

Single-party FHE uses the same key for encryption or decryption. To construct MKFHE like the FHE encryption mode, we need to construct a common public key of the parties set. So the ciphertext generated of each party in MKFHE scheme corresponds one joint secret key(jsk). For any party $i$, he generates its key pair (sk, pk)
Multi-key Fully Homomorphic Encryption Scheme with Fixed-length Ciphertext

from the selected parameters. For example, \( sk_i := s_i \), \( pk := [b_i = s_i B + e_i B] \in \mathbb{Z}_{q^{n_k}}^{\times} \). Then, the \( pk \) of the parties set is generated by accumulating all the parties’ \( pk \). So the \( pk \) is obtained as \( \text{pk} := A^\oplus [b_1 + ... + b_k] \in \mathbb{Z}_{q^{n_k}}^{\times} \). \( pk \) can be used for all parties to encrypt their data, so that all the ciphertexts correspond the same \( sk \) without performing the ciphertext extension process. When decrypting, each party gets his partial decryption result, and then integrates them into the final plaintext.

Our work is to generate the joint public key of the parties set by directly accumulating the public keys of multi-parties under the CRS model, introduce the bootstrapping or key switching process into the ciphertext extension process, and construct the compact extended ciphertext based on (R)LWE problem.

The result shows that the size of the ciphertext is independent of the number of parties. And the homomorphic computation is as efficient as the single party FHE scheme. When the parties set is updated, the original joint ciphertext can continue to be used to synthesize new joint ciphertext to participate in homomorphic computation, but each party needs to provide a new public key. The memory (bit-size) comparison between our scheme and LZY+19, CCS19 and CDKS schemes are shown in Table 1.

<table>
<thead>
<tr>
<th>Schemes</th>
<th>Bit-Size</th>
<th>Evaluation key</th>
<th>Accumulated switching key</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZY+19</td>
<td>O(kn)</td>
<td>O(k(^3)n)</td>
<td>O(kn)</td>
</tr>
<tr>
<td>CCS19</td>
<td>O(kn)</td>
<td>O(k(^2)n(^2))</td>
<td>O(kn)</td>
</tr>
<tr>
<td>CDKS19</td>
<td>O(kn)</td>
<td>O(kn)</td>
<td>O(kn)</td>
</tr>
<tr>
<td>Our scheme</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Table 1: The memory (bit-size) comparison between our scheme with LZY+19, CCS19 and CDKS19. \( k \) denotes the number of parties and \( n \) is the dimension of the (R)LWE assumption.

2 PRELIMINARIES

2.1 Definition of multi-key fully holomorphic encryption(MKFHE)

We now introduce the cryptographic definition of a leveled multi-key FHE, which is similar to the one defined in CZW17 with some modifications from LTV12.

Definition 2.1 (Multi-key FHE)[21]. Let \( \mathcal{C} \) be a class of circuits. A leveled multi-key FHE scheme \( \mathcal{E} = \langle \text{Setup}, \text{KeyGen}, \text{Enc}, \text{Eval}, \text{Dec} \rangle \) is described as follows:

\( \mathcal{E} \).\text{Setup}(\lambda, \lambda^\{1,1,1\} \lambda) \) : Given the security parameter \( \lambda \), the circuit depth \( L \), and the number of distinct parties \( K \) that can be tolerated in an evaluation, outputs the public parameters \( pp \).

\( \mathcal{E} \).\text{KeyGen}(pp) \) : Given the public parameters \( pp \), derives and outputs a public key \( pk_i \), a secret key \( sk_i \), and the evaluation keys \( evk_i \) of party \( i \) \( (i = 1, ..., K) \) .

\( \mathcal{E} \).\text{Enc}(pk_i, m) \) : Given a public key \( pk_i \) and message \( m \), outputs a ciphertext \( ct_i \).

\( \mathcal{E} \).\text{Eval}(\mathcal{C},(ct_1, pk_1, evk_1),..., (ct_K, pk_K, evk_K)) \) : On input a Boolean circuit \( \mathcal{C} \) along with \( t \) tuples \( (ct_s, pk_s, evk_s)_{i \in \{1,...,t}\} \), each tuple comprises of a ciphertext \( ct_s \) corresponding to a parties set \( S_i \), a set of public keys \( pk_s = \{ pk_s, \forall j \in S_i \} \), and the evaluation keys \( evk_s \), outputs a ciphertext \( ct_{S_i} \) corresponding to a set of secret keys indexed by \( S = \bigcup_{i=1}^t S_i \subseteq [K] \).

Definition 2.2 (Correctness of MKFHE)[21]. On input any circuit \( \mathcal{C} \) of depth at most \( L \) and a set of tuples \( \{(ct_s, pk_s, evk_s)_{i \in \{1,...,t}\}\} \), let \( \mu_i = \text{Dec}(sk_i, ct_i) \) , where \( sk_i = \{ sk_j, \forall j \in S_i \} \) , a leveled MKFHE scheme \( \mathcal{E} \) is correct if it holds that

\[ \Pr[\text{Dec}(sk_1, \text{Eval}(\mathcal{C},(ct_1, pk_1, evk_1),..., (ct_K, pk_K, evk_K))) \neq \mathcal{C}(\mu_1, ..., \mu_t)] = \text{negl}(\lambda) \]

Definition 2.3 (Compactness of MKFHE)[21]. A leveled MKFHE scheme is compact if there exists a polynomial \( \text{poly}(\cdot, \cdot) \) such that \( |ct| \leq \text{poly}(\lambda, K, L) \), which means that the length of \( ct \) is independent of the circuit \( \mathcal{C} \) , but depend on the security parameter \( \lambda \), the number of parties \( K \) and the circuit depth \( L \).

2.2 The general learning with errors (GLWE) problem

The learning with errors (LWE) problem and the ring learning with errors (RLWE) problem are syntactically identical, aside from different rings, and these two problems are summarized as GLWE problem in [BGV12].

Definition 2.4 (GLWE problem)[22]. Let \( \lambda \) be a security parameter. For the polynomial ring \( R = \mathbb{Z}[x]/x^d + 1 \) and \( R_q = R/qR \), and an error distribution \( \chi = \chi(\lambda) \) over \( R \), the GLWE problem is to distinguish the following two distributions: In the first distribution, one samples \( (a_i, b_i) \in R^{d+1} \) uniformly from
denotes the size of ciphertext. In this paper, we would to construct the joint secret key whose length is independent of the number of parties by accumulating all parties’ secret keys. We called it the compact secret key. Starting from the compact secret key, we design a new ciphertext expansion algorithm to obtain the joint ciphertext whose length is also independent of the number of parties. Furthermore, we propose two general FCMKFHE schemes—the static mode FCMKFHE scheme (SMMK) and authorized mode FCMKFHE scheme (AMMK), which are suitable for different scenarios.

3.1 Static mode FCMKFHE scheme

We can construct MKFHE by imitating the form of FHE, that is, every party uses the same joint public key for encryption, so that all ciphertexts correspond to one joint secret key. As long as the joint public key is short enough, the corresponding joint secret key will also be short, so the size of generated ciphertext will be small. Therefore, homomorphic computation can be performed directly without ciphertext expansion program. In this section, we construct a joint private key and public key by accumulating all the parties’ keys, so we can get a MKFHE scheme with fixed ciphertext length. Because when the parties participating in the calculation is updated, the original ciphertext needs to be regenerated, so the scheme constructed in this way does not support the dynamic update of parties’ information. We call it static mode MKFHE scheme—SMMK.

Since the size of ciphertext and joint secret key of SMMK scheme are in the same magnitude as that of single-party FHE. So, their homomorphic computation mode is same, which makes the multi-key homomorphic computation of SMMK scheme very efficient. Taking the party i as an example, the calculation process of SMMK scheme is as follows. (Like most MKFHE schemes, SMMK is based on the CRS model, and all parties use some shared parameters).

SMMK.Setup(1'): FHE.Setup(1') \rightarrow params

SMMK.KeyGen((params, i), B) \rightarrow pk_i, sk_i

After all parties have completed the process SMMK.KeyGen(), run the generation algorithm of evaluation key.

SMMK.EvalKeyGen(params, sk_i, {pk_{i_1}, ..., pk_{i_l}}):

1) SMMK.SMPK(params, sk_i, {pk_{i_1}, ..., pk_{i_l}}):

This is the public key accumulation function used to generate the jpk. Take the GSW-type MKFHE scheme as an example, input the public parameter B \in \mathbb{Z}_q^{mn \times (n-1)} , the party's secret key s_{k_i} = s_i and public key b_i = [s_iB + e_i, B] \in \mathbb{Z}_q^{m \times n} Output the jpk as \( pk := \prod_{i=1}^{l} b_i \) \in \mathbb{Z}_q^{m \times n}.

2) FHE.SwitchKeyGen(params, sk_i, pk) : Input the party’s sk and jpk, output the accumulated evaluation key (aek) of party i K_{S_i} = FHE.Enc_{pk}(sk_i \otimes sk_i) and the accumulated switching
Multi-key Fully Homomorphic Encryption Scheme with Fixed-length Ciphertext

deadlock (awk) of parties set KS_{Set}. Due to the different structures of the schemes (like the GSW-type MKFHE does not need to run this key switching process), the generation process of the awk is slightly different. See the specific scheme in Section 4 for details.

3) FHE.BootKeyGen(params, sk, pk) : Input the party’s sk and jpk. Output the bootstrapping key (bk) of party iBK_{i} = FHE.Enc_{pk}(sk_{i}), and the accumulated strapping key (abk) of the accumulated ciphertext

BK_{Set} = \text{HomAdd}_{\text{log}_{B}/q}(BK_{1}, ..., BK_{i}), where \text{HomAdd}(\ast) is the homomorphic addition circuit for 1 bits.

SMMK.Enc(pk, \mu) : FHE.Enc((pk, \mu). The encryption is the same as the single-party FHE schemes.

SMMK.Dec((sk_{1}, ..., sk_{n}), C)\. Like most MKFHE schemes, the decryption result consists of two parts: partial decryption and final decryption.

1) SMMK.PartDec(C, i, sk_{i}) : Input the secret key of party i sk_{i} = (-s, 1) and the result ciphertext C, and output the partial decryption. Taking the GSW-type FHE as an example, its decryption form is \mu' = sCG^{-1}(\hat{w}^T)\. We only calculate \hat{p}_{i}' := s_{C_{[1, n-k]}}G^{-1}(\hat{w}^T) + e_{i}^{m} and get the partial decryption \hat{p}_{i}', where C_{[1, n-k]} represents the first n-1 columns of ciphertext C, and e_{i}^{m} \in [-B_{\text{madd}^{-1}}^{-}, B_{\text{madd}^{-1}}] is the generated error used to protect the security of partial decryption.

2) SMMK.FinDec(p_{1}', ..., p_{n}') : Input all the partial decryptions, and output the resulting plaintext m' = \hat{C}_{[1, n-k]}G^{-1}(\hat{w}^T) - \sum_{i=1}^{n}p_{i}'.

The homomorphic computation is as follows.

SMMK.Add(C_{1}, C_{2}) : C_{1} \leftarrow FHE.Add(C_{1}, C_{2}).

SMMK.Mult(C_{1}, C_{2}, EVK_{Set}) : C_{1} \leftarrow FHE.Mult_{EVK}(C_{1}, C_{2}).

SMMK.Bootstrap(BK_{Set}, C) : FHE.Bootstrap(BK_{Set}, C).

The correctness of the decryption process above can be verified as following.

\hat{C}_{[1, n-k]}G^{-1}(\hat{w}^T) = \sum_{i=1}^{n}p_{i}' = \sum_{i=1}^{N}e_{i}^{m} + RCG^{-1}(\hat{w}^T) + \mu(\hat{w}^T) - \sum_{i=1}^{N}e_{i}^{m} = \lceil q/2 \rceil + \epsilon^{*} (1)

For the above scheme, the ciphertexts of all parties are encrypted by the joint public key pk, and the homomorphic computation is the same as single-party FHE scheme, so the efficiency of the scheme is better than the previous MKFHE schemes. By simply changing the form of encryption and decryption, we can construct BGV-type FCMKfeito and TATE-type FCMKfeito. However, the SMMK scheme also has some defects. When some new parties join, the original ciphertext and evaluated key are unavailable. We must regenerate the new ciphertext and evaluated key for the updated parties set. We aim to construct a new FCMKfeito scheme, which supports the timely updating parties set without regenerating their ciphertexts and keys.

3.2 Authorized mode FCMKfeito scheme

The ciphertext and evaluated key of the SMMK scheme are all for a constant parties set. In this section, we focus on constructing an authorized mode FCMKfeito scheme (we call it AMMK scheme), which has the following advantages: the size of ciphertext is independent of the number of parties, and all ciphertexts continue to be used in the updated set. The idea of the construction is: Party i uses his pk encrypt his data and obtains his own ciphertext. Then, by using the optimized bootstrapping process (or key switching process), his own ciphertext is converted to the joint ciphertext corresponding to the apk. So that the joint ciphertext can be reused. Different from SMMK scheme, the scheme needs to adjust the public key corresponding to the ciphertext to a new parties set before homomorphic computation, and the parties set needs to interact to generate a new evaluation key when updating.

The operation process is as follows.

AMMK.Setup(1^{s}) : FHE.Setup(1^{s}) \rightarrow params ;

AMMK.KeyGen(params) : FHE.KeyGen(params, B) \rightarrow (pk_{i}, sk_{i}) ; After all parties have completed the process AMMK.KeyGen(), run the generation algorithm of evaluation key.

AMMK.EvalKeyGen(params, sk_{i}, {pk_{i}, ..., pk_{N}}) :

SMMK.EvalKeyGen(params, sk_{i}, {pk_{i}, ..., pk_{N}}) :

AMMK.Enc(pk, \mu) : FHE.Enc(pk, \mu) \rightarrow C\. (Note: This is a single party’s public key encryption, not a joint public key encryption.)

AMMK.Dec((sk_{1}, ..., sk_{n}), C) : SMMK.Dec((sk_{1}, ..., sk_{n}), C) \rightarrow \mu'.

Similar to scheme BP16, this scheme uses bootstrapping process to implement homomorphic computation.

AMMK.Eval((C_{1}, C_{2}), BK_{i}, KS_{Set}) :

1) C'_{i} = Hom_{\text{log}_{B}/q}(\text{FHE.Dec}_{\text{sk}_{i}}(C_{i})). This process can refresh different public keys. BK_{i} is the bootstrapping key corresponding to C_{i}. If C_{i} is the ciphertext of a single party, the bootstrapping key BK_{i} is set as BK_{i}. If C_{i} is the joint
ciphertext of all parties, the bootstrapping key $\overline{BK}_i$ is set as $\overline{BK}_i = \overline{BK}_m$.

2) FHE.Eval($\langle C'_1, C'_2 \rangle, \overline{KS}_{\overline{pk}}$). This process realizes the homomorphic computation of joint ciphertext, where $\overline{KS}_{\overline{pk}}$ is the accumulated evaluation key of the joint ciphertext.

The drawback of the AMMK scheme is that when the parties set is updated, all parties need to update the evaluation key and bootstrapping key. That is, if party $i$ wants to updated his $ek$ and $bk$, he must obtain other parties’ authorization in updated set. So, the scheme needs 3-rounds of interaction to construct MPC.

4 Specific structure of FCMKFHE scheme

The general SMMK and AMMK schemes need to perform the bootstrapping process to refresh the ciphertext, so their efficiency is low. In this section, relying on the efficient TFHE-type MKFHE and BGV-type MKFHE, we propose a targeted optimization method and construct two efficient FCMKFHE schemes.

4.1 Construction of TFHE-type FCMKFHE

TFHE-type scheme is the fastest bootstrapping scheme at present, but its secret key vectors are only taken from $\{0,1\}^y$, and the value of accumulated bootstrapping secret key is larger, so it can’t be directly applied to AMMK. To combine the FCMKFHE scheme with the TFHE-type scheme better, we design a secret key extension algorithm, and construct an efficient TFHE-type FCMKFHE scheme--AMTMK.

AMTMK.Setup($1^i$) → $pp = (pp_{LWE}, pp_{GSW})$:
LWE.Setup($1^i$) → $pp_{LWE} = (\eta, \chi, \alpha, B_x, d_x, B)$;
GSW.Setup($1^i$) → $pp_{GSW} = (N, \phi, \alpha, B, d, y)$, Where $B, y$ are common random variables.

AMTMK.KeyGen($pp$) → $\langle pk, sk, pk_{BK}, sk_{BK} \rangle$:
LWE.KeyGen($pp$) → $\{ pk_i = A_i, sk_i = s_i \}$;
RGSW.KeyGen($pp$) → $\{ pk_{BK,i} = Z_i, sk_{BK,i} = z_i \}$.

After all parties have completed the program AMTMK.KeyGen($params$), run the algorithm of evaluation key generation. If the parties set is updated, rerun the key generation algorithm.

AMTMK.EvalKeyGen($pp, sk_i, \{ pk, \ldots, pk \}$) → $\langle pk, KS_{\overline{pk}}, BK \rangle$:

1) Accumulate the public key. Given the public keys $b_1, \ldots, b_k$ of $k$ parties, we obtain the joint public key $pk := [b_1 + \ldots + b_k] \in \mathbb{Z}_{q}^{\max}$.

Accumulate the bootstrapping public key. Given the bootstrapping public keys $d_1, \ldots, d_l$ of $k$ parties, we obtain the accumulated bootstrapping public key $pk_{BK} := \mathbb{Z} = [d_1 + \ldots + d_l] \in \mathbb{Z}_{q}^{2 \times l}$.

2) Accumulate the single-party bootstrapping key. Input the accumulated bootstrapping public key $pk_{BK} = \mathbb{Z}$ and the secret key $s_i \in \mathbb{Z}^n$ of LWE ciphertext. Output the single-party’s accumulated bootstrapping key $BK_i = \{ BK_{i,j} \}_{j \in \mathbb{N}}$, where $BK_{i,j} = \text{RGSW.Enc}(s_i, Z)$, $i \in [k]$, $j \in [n]$. Run the following homomorphic accumulation algorithm [24]:

3) Accumulate the evaluation key. Input the accumulated public key $pk$ and the secret key $z_i$ of the RGSW ciphertext, let $t_i = (z_{i,0}, -z_{i,1}, \ldots, -z_{i,l}) \in \mathbb{B}^n$, and output the accumulated evaluation key (aek) $KS_i = \text{LWE.KSGen}(t_i, pk)$ of single party, where $i \in [k]$.

AMTMK.Enc($pk, \mu$) : Input the plaintext $\mu$, and single party’s public key $pk$ run LWE.Enc($pk, \mu$) → $ct = (b,a) \in \mathbb{T}^{+1}$.
AMTMK.Dec($ct$) : Input the ciphertext $ct = (b,a) \in \mathbb{T}^{+1}$ and the secret key $(sk_1, \ldots, sk_k)$. Return the plaintext bit $\mu' \in [0,1]$ that makes $|b + \sum_{i}^{k} < a, s_i > - \frac{1}{m} |$ be smallest.

AMTMK.Boot($c, \{ BK_{i,j} \}_{i \in \mathbb{N}}, \{ KS_i \}_{i \in \mathbb{N}}$) : Input the ciphertext $ct = (b',a') \in \mathbb{T}^{+1}$, the bootstrapping key set $\{ BK_{i,j} \}_{i \in \mathbb{N}}$ and the accumulated evaluation key set $\{ KS_i \}_{i \in \mathbb{N}}$. Then use bootstrapping process to realize homomorphic computation.

1) The cloud server uses $KS_i$ to generate accumulated switching key $KS_{\overline{sw}} = \{ KS_{\overline{sw},i} \}_{i \in \mathbb{N}}$. The cloud sever also uses the $BK_i$ to generate the accumulated bootstrapping key $BK_{\overline{sw},i} = \text{HomAddk}(BK_{i,1}, \ldots, BK_{i,l})$, where $j \in [n], l = \lceil \log(k) \rceil$. $\text{HomAddk}(\cdot)$ is a homomorphic addition algorithm for $k$ 1-bit TGSW ciphertexts, which can be constructed by homomorphic multiplication and homomorphic addition of TGSW ciphertexts. See Annex C for details. For a constant parties set, the cloud sever only needs to calculate $KS_{\overline{sw}}$ and $BK_{\overline{sw}}$ once, and then output them as public variables.

2) Ciphertext refresh. Given ciphertext $c = (b',a') \in \mathbb{T}^{+1}$, and the evaluated key $\{ BK_{\overline{sw},i} \}_{i \in \mathbb{N}}$ or $BK_i$. Run the following homomorphic accumulation algorithm [24]:
Multi-key Fully Homomorphic Encryption Scheme with Fixed-length Ciphertext

1. Input the ciphertext \( c = (b',a') \in \mathbb{T}^{n+1} \), output \( \hat{b} = [2N \cdot b'] \), \( \hat{a} = [2N \cdot a'] \) and the bootstrapping key

\[
\overline{BK} = \begin{cases} BK_i & \text{c corresponds to the secret key } s_i \\ \overline{BK}_{\text{err}} = (s_1, \ldots, s_l) & \text{c corresponds to the secret key } (s_1 + \ldots + s_l) \end{cases}
\]

2. Initialize the RLWE ciphertext \( ACC = (-\frac{1}{a} h(X) \cdot X^s, \theta) \), where \( h(X) = 1 + X + \ldots + X^{s-1} - X^s - \ldots - X^{N} \). Let \( \hat{a} = (\hat{a}_j)_{j=|n|} \), for \( j = 1 \) to \( n \), run the following process.

\[
\begin{align*}
(1) & ACC = \text{CMux}(\overline{BK}_{\text{err}}, X^a ACC, ACC) \\
(2) & ACC = \text{CMux}(\overline{BK}_{\text{err}}, X^{2a} ACC, ACC) \\
& \vdots \\
(l-1) & ACC = \text{CMux}(\overline{BK}_{\text{err}}, X^{(2^{l-1})w} ACC, ACC)
\end{align*}
\]

We select the largest circuit \( \text{CMux}(C,d_1,d_0) \). Input one TGSW ciphertext \( C \) and two input RLWE ciphertexts \( d_1, d_0 \). Output RLWE ciphertext \( C \otimes (d_1 - d_0) + d_0 \), where \( \Box \) is a hybrid homomorphic multiplication of GSW ciphertext and BGV ciphertext. The specific process is shown in [24].

3. Output \( ACC \leftarrow (\frac{1}{a}, \theta) + ACC \mod (1) \)

3.1. Key switching process. The last step is to convert ACC into LWE ciphertext and run the key switching algorithm.

3.2. Input the ciphertext \( ACC = (c_0, a^n) \in T^2 \). Set \( b^* \) be a constant term of polynomial \( c_0 \) and \( a^n \) be a vector composed of coefficients of polynomial \( c_1 \). Output LWE ciphertext .

\[
\begin{align*}
2 & \text{Let } \overline{KS}_{\text{err}} = \{\sum_{i=1}^{\text{l}} \overline{KS}_{\text{err},i} \}_{j=|N|} \text{, run the key switching algorithm and output the ciphertext} \\
& \overline{ct} \leftarrow \text{LWE.MKSwitch}(a^n, \overline{KS}_{\text{err}})
\end{align*}
\]

The NAND circuit of homomorphic NAND gate is constructed by the bootstrapping process.

\text{AMTK.NAND}(c_1, c_2) = \text{HTMK.Boot}((0.5/8) - c_1 - c_2) \cdot \text{AMCMK.EvalKeyGen( } \overline{ct}, \{ pk_i \}_{i=1}, \{ pk_{e, i} \}_{i=1} \text{ )} \rightarrow \{ pk, sk, rk, ek \}.

1) Accumulate the public key. Given \( k \) parties’ public key \( b_1, \ldots, b_k \), the CKKS-type accumulated public key is generated as \( pk := (b_1 + \ldots + b_k) \in R_{p_0}^d \), where \( pk_{i,j} \) represents the \( j \)-th element of \( pk \).

2) Accumulate the evaluation key. Given the \( k \) parties’ evaluation public key \( b_1', \ldots, b_k' \), the CKKS-type accumulated evaluation public key is generated as \( pk_{e, i} := (b_1' + \ldots + b_k') \in R_{p_0}^i \).

3) The accumulated evaluation key generation.

\text{AMCMK.SEvalKey}(pk, a', s) \cdot Select \ r \leftarrow ZO(0.5) \text{ randomly, and the partial switching key is obtained as}

So the error magnitude \( e \) of the output LWE ciphertext is small. The detailed process of noise analysis is shown in Appendix D.

4.2 Construction of BGV-type FCMKFHE scheme

CKKS17 scheme is an efficient and concerned BGV-type FHE scheme. It can calculate floating-point data efficiently and is widely used in secure neural network et al. According to the characteristics of CKKS scheme, we construct an effect BGV-type FCMKFHE scheme AMCMK in this section.

\text{AMCMK.KeyGen}(pp) : Input the parameters \( pp \). Select \( s \leftarrow \text{KeyGen} \) and \( e' \leftarrow \text{KeyGen} \), and output the public key \( pk = (b', a') \leftarrow U(R_{p_0}^d) \). Output common parameter \( pp = (N, \chi_{\text{key}}, \chi_{\text{err}}, \chi_{\text{enc}}, L, P, q_1, a, a') \).

\text{AMCMK.EvalKeyGen}(pp) : Input the parameters \( pp \). Select \( \text{AMCMK.EvalKeyGen}(pp) \) run the algorithm of evaluation key generation. If the parties set is updated, rerun the generation algorithm.


\[(d_j, \overline{d}_j) := \left((d_{\overline{1}, j}, d_{\overline{1}, j})\right)_{j \in [d]} \quad \text{where} \]
\[(d_{\overline{1}, j}) := \text{CKKS.Enc}_{\text{pk}_M, \text{pk}_d}(r \cdot g_{\overline{1}, j}), \quad j \in [d] \quad \text{and} \]
\[g = (l, B_1, \ldots, B_{d-1}) \quad B_j \quad \text{is the decomposition basis. Set} \]
\[d_2 = r \cdot a' + e_2 + P \cdot s \pmod{P q_1}, \quad \text{where} \ e_2 \in \mathcal{Z}_{m, k}. \]

Output:
\[\overline{\text{ks}}_{\overline{1}, j} \in [g^{-1}(b')] \cdot \left[d_{\overline{1}, i} \cdot d_{\overline{1}, j}^{-1}\right]_{j \in [d]} + \left[0 \cdot d_2\right]_{j \in [d]} \in \text{CKKS}_M(P \cdot s, s) \]

The refresh key of party's ciphertext set is obtained as
\[\overline{\text{ks}}_{\overline{1}, j} \leftarrow \text{CKKS.Enc}_{\text{pk}_M, \text{pk}_d}(P \cdot s). \quad \text{Then, output the shift key} \]
\[\overline{\text{rk}}_{\overline{1}, j} \leftarrow \text{CKKS.Enc}_{\text{pk}_M, \text{pk}_d}(\overline{\text{ks}}_{\overline{1}, j}) \quad \text{and conjugate key} \]
\[\overline{\text{ck}}_{\overline{1}, j} \leftarrow \text{CKKS.KSGen}_{\text{pk}_M, \text{pk}_d}(\overline{\text{ks}}_{\overline{1}, j}). \]

4) Generate the evaluation key in cloud.
\[\overline{\text{ks}}_{\overline{1}, j} := \sum_{i=1}^{d} \overline{\text{ks}}_{\overline{1}, i} \in \text{CKKS}_M(P \cdot s) \]
\[\overline{\text{rk}}_{\overline{1}, j} := \sum_{i=1}^{d} \overline{\text{rk}}_{\overline{1}, i} \in \text{CKKS.Enc}_{\text{pk}_M, \text{pk}_d}(\overline{\text{ks}}_{\overline{1}, i}) \]
\[\overline{\text{ck}}_{\overline{1}, j} := \sum_{i=1}^{d} \overline{\text{ck}}_{\overline{1}, i} \in \text{CKKS.Enc}_{\text{pk}_M, \text{pk}_d}(\overline{\text{ks}}_{\overline{1}, i}). \]

When the parties set the AMCMK scheme is updated, the bootstrapping process is no longer needed. The ciphertexts are converted to the ciphertexts of the new set through the accumulated key switching process. Compared with BP16 scheme, AMCMK can improve efficiency.

AMCMK.Enc(pk, m) : c = CKKS.Enc(pk)(m). The encrypted ciphertexts are modulo P to reduce their size.

AMCMK.Dec((sk_1, ..., sk_{\overline{1} \cdot j}), c) : Input the ciphertext c of l-th level. Output m = c, s_k, + ... + s_k, > (mod q_1).

AMCMK.KeySwitchingKey(c, \{ks_{\overline{1} \cdot j}\}) : Input the ciphertext c' = (b', a'), output the corresponding accumulated switching key

\[\overline{\text{ks}}_{\overline{1}, j} \quad \text{c corresponds to the secret key s}_j \]
\[\sum_{i=1}^{d} \overline{\text{ks}}_{\overline{1}, i} \quad \text{c corresponds to the secret key (s_1 + ... + s_{\overline{1} \cdot j})} \]

, where k' represent the original parties set.

Homomorphic computation. If the public keys corresponding to the ciphertexts participating in homomorphic operation are different, we use the process

AMCMK.KeySwitchingKey(c, \{ks_{\overline{1} \cdot j}\}) to convert them to the same. The homomorphic computation process and bootstrapping process of the AMCMK are the same as CKKS17. We just replace the evaluation key with the accumulated evaluation key, so the calculation efficiency is the same as CKKS17.

AMCMK.Add(ct, ct') : CKKS.Add(\text{ct}, \text{ct'})

AMCMK.CMult(a, ct) : CKKS.CMult(a, ct)

AMCMK.Mult(\text{ct}, \text{ct'}) : CKKS.Mult(\text{ct}, \text{ct'})

AMCMK.Bootstrapping(\text{ct}, \text{ct'})

Whether the ciphertext can be decrypted correctly depends on the size of the error in the ciphertext. Following the expression of CKKS17, in this section, we analyze the works of the main functions and growth of the error.

Let \(\|a\|_e\) denote the infinite normal form of a(\(\zeta\)) (the inner product of the coefficients of a and vectors (1, \(\zeta, \zeta^2, \ldots, \zeta^{N-1}\)) obtained by normal embedding of polynomial \(a(X) \in R = \mathbb{Z}[X] / (\Phi_M(X))\). According to the analysis in CKKS17, \(\|a\|_e \leq 6\sigma\) where \(\sigma^2\) is the variance of a(\(\zeta\)) and \(ab\|e\|_e \leq 16\sigma_1\sigma_2\), where \(\sigma_1^2\) and \(\sigma_2^2\) are the variance of a(\(\zeta\)) and b(\(\zeta\)) respectively. If the coefficient of a is taken from the uniform distribution of \([0, q]\), then \(\text{Var}(a(\zeta_M)) = q^3 N / 12 \). If a is taken from the discrete Gaussian distribution \(DG_\zeta(\sigma^2)\) with variance \(\sigma^2\), then \(\text{Var}(a(\zeta_M)) = \sigma^2 N\). If a is taken from the \([0, z]\) distribution \(HWT(h)\) with Hamming weight h, then \(\text{Var}(a(\zeta_M)) = h\). The CKKS17 scheme can encrypt plural vectors. Considering the accuracy, the scheme usually expands the data by \(\Delta\) times before encryption, and \(\Delta\) is called the increment factor. For a given ciphertext ct \(\in R_\zeta\), the scheme can decrypt correctly if the increment factor \(\Delta > N + 2B\), where \(\text{ct, sk} := m + e \pmod{q_1}\), B is the upper bound of \(\|e\|_e\).

The error growth of important functions is shown in the following Lemmas.

Lemma 1[23]. Let ct \(\leftarrow \text{Enc}_{p_\text{m}}(m)\) be an encryption of \(m \in R\) and \(e \in R\), then \(\text{ct, sk} := m + e \pmod{q_1}\), where \(\|e\|_e \leq B_{\text{clean}}\), such that
\[B_{\text{clean}} = 8\sqrt{2\sigma N} + 6\sigma \sqrt{N} + 16\sigma \sqrt{hN} \]

Lemma 2. Let ct \(\leftarrow \text{Enc}_{p_k}(m)\) denote the ciphertext of \(m \in R\) encrypted by the accumulated public key pk, for a certain set \(e \in R\), there is \(\text{ct, sk} := m + e \pmod{q_1}\), where \(\text{pk} = (b_1 + ... + b_i, a)\), \(\|e\|_e \leq B_{\text{clean}}\) and \(B_{\text{clean}} = 8\sqrt{2k\sigma N} + 6\sigma \sqrt{N} + 16\sigma k \sqrt{hN}\).
Multi-key Fully Homomorphic Encryption Scheme with Fixed-length Ciphertext

See Appendix B for detail proof of lemma 2.

For the refresh key \( \overline{ks}_{\overline{ref}_{\overline{s}}e} \leftarrow \text{CKKS.Enc}_{\overline{pk}_{\overline{ev}}_{\overline{s}}} (P \cdot s_{i}) \), the shift key \( \overline{rk}_{\overline{s}} \leftarrow \text{CKKS.Enc}_{\overline{pk}_{\overline{ev}}_{\overline{s}}} (\overline{c}_{\overline{s}} (s_{i})) \) and the conjugate key \( \overline{ck}_{\overline{s}} \leftarrow \text{CKKS.KSGen}_{\overline{pk}_{\overline{ev}}_{\overline{s}}} (\overline{c}_{\overline{1}} (s_{i})) \), we have

\[
\|e\|_{\infty} \leq B_{\text{clean}}.
\]

**Lemma 3.** Let \( \overline{ks}_{\overline{s}} \) be the accumulated switch-key, \( \overline{ks}_{\overline{s}}e_{\overline{s}} \), be one element of \( \overline{ks}_{\overline{s}} \), then

\[
<\overline{ks}_{\overline{s}}e_{\overline{s}}, (1, s) >= P_{1} \cdot s_{1} + e_{\overline{ks}_{\overline{s}}e_{\overline{s}}} (\text{mod} \, q_{1})
\]

\[
\|e_{\overline{ks}_{\overline{s}}e_{\overline{s}}}\|_{\infty} \leq B_{\text{ks}} \cdot B_{\text{clean}} = \sqrt{q_{1}} \cdot \|e_{\overline{ks}_{\overline{s}}e_{\overline{s}}}\|_{\infty}
\]

and

\[
\|e_{\overline{ks}_{\overline{s}}e_{\overline{s}}}\|_{\infty} \leq B_{\text{ks}} \cdot B_{\text{clean}} = 8B_{\text{ks}} \cdot B_{\text{clean}} = \sqrt{dN} / 3.
\]

See Appendix B for detail proof of lemma 3.

**Lemma 4**[23]. Let \( \overline{ct} \leftarrow \text{RS}^{t}_{\text{ref}} (\overline{ct}) \) (where \( \overline{ct} \in R^{2}_{q} \)), for \( e \in R \), there is \( <\overline{ct}^{t}, \overline{sk} >>> \overline{ct}, \overline{sk} >>> \overline{ct}, \overline{sk} >>> \overline{ct}, \overline{sk} >>> \overline{ct}, \overline{sk} + e_{\overline{ct}} (\text{mod} \, q_{e}) \), where

\[
\|e\|_{\infty} \leq B_{\text{ct}} \cdot B_{\text{ct}} = P_{1} \cdot q_{1} \cdot B_{\text{ct}} + B_{\text{ct}}.
\]

**Lemma 5.** Let \( \overline{ct}_{\text{mul}} \leftarrow \text{Mult}_{\overline{w}_{\text{ref}}} (\overline{ct}_{1}, \overline{ct}_{2}) \) (where \( \overline{ct}_{1}, \overline{ct}_{2} \in R^{2}_{q} \)), for \( e \in R \), there is

\[
<\overline{ct}_{\text{mul}}, \overline{sk} >>> \overline{ct}_{1}, \overline{sk} >>> \overline{ct}_{2}, \overline{sk} >>> \overline{ct}, \overline{sk} >>> \overline{ct}, \overline{sk} + e_{\overline{ct}_{\text{mul}}} (\text{mod} \, q_{e}) ,
\]

where

\[
\|e\|_{\infty} \leq B_{\text{ct}_{\text{mul}}} \cdot B_{\text{ct}_{\text{mul}}} = P_{1} \cdot q_{1} \cdot B_{\text{ct}} + B_{\text{ct}}.
\]

**Lemma 6.** Let \( \overline{ct} \leftarrow \text{KS}^{t}_{\text{ref}_{\text{ref}}} (\overline{ct}) \). \( \overline{ct} \in R^{2}_{q} \) corresponds to the secret key \( \overline{sk} \). Let

\[
\overline{ks}_{\text{ref}_{\text{ref}}} = \left\{ \begin{array}{ll}
\overline{ks}_{\overline{s}}e_{\overline{s}} & \text{if} \overline{s}_{i} = 1,
\overline{ks}_{\overline{s}}e_{\overline{s}} = \sum_{i=1}^{e} \overline{ks}_{\overline{s}}e_{\overline{s}} & \text{if} \overline{s}_{i} < 1.
\end{array} \right.
\]

For \( e \in R \), there is \( <\overline{ct}, (1, s) >>> <\overline{ct}, \overline{sk} >>> <\overline{ct}, \overline{sk} + e_{\overline{ct}} (\text{mod} \, q_{e}) ,
\]

where

\[
\|e_{\overline{ct}_{t}}\|_{\infty} \leq P_{1} \cdot q_{1} \cdot \sqrt{B_{\text{clean}}} + B_{\text{ct}}.
\]

**ACKNOWLEDGMENTS**

Insert paragraph text here.

**REFERENCES**
A TFHE Scheme

The TFHE scheme based on LWE is usually symmetric mode, but the asymmetric mode is usually used in practical application. To show the accumulated public key conveniently, we show the asymmetric mode of TFHE scheme as bellow.

\[
\text{LWE.Setup}(1^n) \rightarrow pp^{\text{LWE}} = (n, \chi, \alpha, B_{\alpha}, d_{\alpha})
\]

Input the security parameters, generate the LWE dimension \( n \), secret key distribution \( \chi \), error distribution parameter \( \alpha \), decomposition basis \( B_{\alpha} \), decomposition degree \( d_{\alpha} \), common matrix \( B \in \mathbb{M}_{n \times n}^{\text{sym}} \) for all users, and output common parameter \( pp^{\text{LWE}} = (n, \chi, \alpha, B_{\alpha}, d_{\alpha}, B) \).

\[
\text{LWE.KeyGen}(pp) \rightarrow \{pk, sk\} : \text{ Select } s \in \mathbb{Z}^n \leftarrow \chi , \text{ and generate the public key } A = [b \ B], \text{ where } b = -Bs + e . \text{ Output } pk = A , \ sk = s .
\]

\[
\text{LWE.Enc}(pk, m) \rightarrow ct = (b, a) \in \mathbb{M}_{n^2}^{\text{sym}} : \text{ Select } r \in \mathbb{Z}^m \text{ randomly, and calculate } (b, a') := rA + (\frac{e}{m}, 0, \ldots, 0) + e
\]

\[
\text{LWE.KSGen}(t, pk) \rightarrow KS_{t} = [b_j | A_j] \in \mathbb{M}_{d_{\alpha}^{n+1}}^{n} : \text{ Input the LWE secret } t \in \mathbb{Z}^n , \text{ accumulated public key } pk .
\]

\[
\text{LWE.Switch}(ct) \rightarrow ct' = (b', a') \in \mathbb{M}_{n^2}^{\text{sym}} : \text{ Input the accumulated ciphertext } ct = (b, a) \in \mathbb{M}_{n^2}^{\text{sym}} \text{ and the switch key } \{KS_{t_j}\}_{j \in [N]} \text{ with dimension } j \in [N] . \text{ Calculate the ciphertext }
\]

\[
ct' = (b' + b, a') \in \mathbb{M}_{n^2}^{\text{sym}}
\]
Multi-key Fully Homomorphic Encryption Scheme with Fixed-length Ciphertext

Correctness. For \( (b', a') = \sum_{j=0}^{N-1} \sum_{i=0}^{k-1} g_{i,j}^{-1}(a_j) KS_{i,j} \), we can obtain
\[
\langle ct', (1, t) \rangle = \langle ct, (1, t) \rangle + e_{ts},
\]
where
\[
e_{ts} = \sum_{j=0}^{N-1} t_{i,j} e_j' + g_{i,j}^{-1}(a_j), \quad \sum_{j=0}^{N-1} R_{i,j} A + (e_j, 0, \ldots, 0) >.
\]
The correctness derivation process is shown in the following equation:
\[
\langle ct', (1, t) \rangle = b + b' + \sum_{j=0}^{N-1} a_j' s_j
\]
\[
= b + \sum_{j=0}^{N-1} g_{i,j}^{-1}(a_j) \sum_{i=0}^{k-1} R_{i,j} A + (e_j, 0, \ldots, 0) + (t_{i,j}, 0, \ldots, 0) \cdot g_{i,j}
\]
\[
= b + \sum_{j=0}^{N-1} g_{i,j}^{-1}(a_j) \sum_{i=0}^{k-1} R_{i,j} A + (e_j, 0, \ldots, 0) > +
\]
\[
= b + \sum_{j=0}^{N-1} g_{i,j}^{-1}(a_j) \sum_{i=0}^{k-1} (t_{i,j}, 0, \ldots, 0) \cdot g_{i,j}
\]
\[
< ct', (1, t) > + \sum_{j=0}^{N-1} t_{i,j} e_j' + g_{i,j}^{-1}(a_j), \sum_{j=0}^{N-1} R_{i,j} A + (e_j, 0, \ldots, 0) >
\]
where
\[
(b', a') = \sum_{j=0}^{N-1} \sum_{i=0}^{k-1} g_{i,j}^{-1}(a_j) KS_{i,j}
\]
\[
KS_{i,j} = \{ \sum_{j=0}^{N-1} KS_{i,j} \}_{j=0}^{k-1},
\]
\[
KS_{i,j} = R_{i,j} A + (e_j, 0, \ldots, 0) + (t_{i,j}, 0, \ldots, 0) \cdot g_{i,j}
\]
If \( |e_{ts}| < \frac{1}{5}, KS_{i,j} = \{ \sum_{j=0}^{N-1} KS_{i,j} \}_{j=0}^{k-1} \) can be seen as the valid switch key from \( t \in \mathbb{Z}^n \) to \( s \in \mathbb{Z}^n \).

B CKKS17 Scheme

CKKS.Setup\((1^{\lambda}) \rightarrow pp = (N, X_{key}, X_{err}, X_{enc}, L, \rho_i) \) : Input the parameters \( \lambda \) and select an integer \( N \) to the power of 2. Let \( X_{key}, X_{err} \) and \( X_{enc} \) be the degree of secret key, error and encryption process on \( R = \mathbb{Z}[X]/(X^N + 1) \) respectively. Select prime \( p \) and the circuit layer \( L \). The ciphertext modulus is \( \rho_i = p^i \), where \( 1 \leq l \leq L \). Output common parameter
\[
pp = (N, X_{key}, X_{err}, X_{enc}, L, P, q, a, a').
\]
CKKS.KeyGen\((params) \rightarrow (pk, sk, ks, rk, ck, evk) \) : Select \( s \leftarrow X_{key} \) and let the secret key \( sk \leftarrow (1, s) \). Select \( a \leftarrow U(R_{\rho_i}) \) and the error \( e \leftarrow X_{err} \). Set the public key
\[
\langle b, a \rangle \in \mathbb{R}^2, \text{ where } b = -as + e \pmod{q_{\rho_i}}.
\]
- CKKS.Enc\((pk, m) \rightarrow ct \) : Input \( m \in \mathbb{Z}^n \), and select \( a' \leftarrow R_{p,\rho_i} \) and \( e' \leftarrow X_{err} \). Let the evaluated key be
\[
evk \leftarrow \langle b', a' \rangle \in R_{p,\rho_i}^2, \text{ where } b' = -a's + e' + Ps' \pmod{P \cdot q_{\rho_i}}.
\]
Obtain the switch key \( ks \leftarrow \text{CKKS.KSGen}_d(s^2) \) : Obtain the shift key \( rk \leftarrow \text{CKKS.KSGen}_d(k_s(s)) \) : Obtain the conjugate key \( ck \leftarrow \text{CKKS.KSGen}_d(k_s(s)) \).

CKKS.Enc\_(pk, s) : Input \( r \leftarrow X_{enc}, e_0, e_1 \leftarrow X_{err} \) randomly. Output \( ct = pk + (m + e_0, e_1) \pmod{q_{\rho_i}} \), such that
\[
\langle ct, sk \rangle > (mod \ q_{\rho_i}) \approx m.
\]
CKKS.Dec\_(ct) : Input the ciphertext \( ct \) of the \( l \)-th level, and output the plaintext \( m' = \langle ct, sk \rangle > (mod \ q_{\rho_i}) \).
CKKS.Add\((ct, ct') \) : Input the ciphertexts \( ct \) and \( ct' \) of the \( l \)-th level, and output the ciphertext \( ct_{\text{add}} = 1 \cdot ct + 1 \cdot ct' \pmod{q_{\rho_i}} \).
CKKS.Mult\_(a, ct) : Input the constant \( a \in \mathbb{R} \) and the ciphertext \( ct \) of the \( l \)-th level. Output the ciphertext \( ct_{\text{mult}} = a \cdot ct \pmod{q_{\rho_i}} \).

CKKS.Bootstrapping\( (ks, rk, ck, \text{ct}) \) : Input the evaluated key \( swk \) and the ciphertext \( ct \) of the \( l \)-th level. Output the ciphertext
\[
ct' \leftarrow (c_0, c_1, c'_{c_1}, c_{c_1}) \in\mathbb{R}^2, \text{ where } ct' = \left[ p^{l-1} \cdot c_1, swk \right] \pmod{q_{\rho_i}}.
\]
CKKS.Rotate\_(ct) : Input the ciphertext \( ct \) of the \( l \)-th level and the next level label \( l' \). Output the ciphertext
\[
ct' = \left[ p^{l-1} \cdot ct \right] \in (mod \ q_{\rho_i}) \).
CKKS.Rotate\_(ct) : Input the shift key \( rk \) and the ciphertext \( ct \). If the plaintext vector \( m(Y) \) moves \( k \) bits, then output the ciphertext of \( m(Y') \).

- GSW.PSKeyGen\((params) \rightarrow (pk, sk) \) : Select \( s \leftarrow X_{key} \), and set the secret key \( sk \leftarrow (1, s) \). Select
\[ a \leftarrow R_{p,q_1}^{2d}, \quad e \leftarrow \chi_{x_e}^{2\beta}, \text{ output the public key} \]
\[ pk := \{ b = az + e, a \} \in R_{p,q_1}^{2d+2}, \text{ where} \]
\[ pk := \{ b = az + e, a \} \in R_{p,q_1}^{2d+2}. \]

\[ \text{GSW.Enc}_\mu (\mu) = \begin{cases} r b[1] + pe[1][1] + \mu \\ rb[2] + pe[2][2] \\ \\
\end{cases} \]

**B1 Proof of Lemma 2**

Proof. Define \( e = e_1 + \ldots + e_k, \quad s = s_1 + \ldots + s_k \), and \( b = b_1 + \ldots + b_k \). For the ciphertext
\[ ct = r \cdot (b, a) + (m + e', e') \mod q \], where
\[ b_i = -a s_i + e_i \mod q_i \]. Select \( r \leftarrow ZO(0.5) \), \( e', e' \leftarrow DG_q (\sigma^2) \), \( s \leftarrow HWT(h) \), \( a \leftarrow U_{\mathbb{Z}} \), and \( e_i \leftarrow DG_q (\sigma^2) \), then calculate the expression of error \( e \) and the upper limit of \( ||e||^\infty \).

\[ e = r \cdot (a, s) + (m + e', e') (1, s) > -m \]
\[ = r \cdot (b, a), (1, s) + + (e', e') (1, s) > -m \]
\[ b_i - a e_i (\mod q_i) = re + e_0 + e_i \cdot s \]
\[ \]
Multi-key Fully Homomorphic Encryption Scheme with Fixed-length Ciphertext

e_{\text{ring}} = e_{\text{ring}} + e_{\text{val}} \cdot \left[ g^{-1}(b') \right]_{\text{val}} \cdot e_{\text{clean}}_{\text{val},l} \mod PqL \)

\| e_{\text{ring}} \|_{\infty} \leq 8\sqrt{2k}\sigma N 

\| e_{\text{val}} \|_{\infty} \leq 16\sigma k\sqrt{N} 

\left\| g^{-1}(b') \right\|_{\text{val},l} \cdot \| e_{\text{clean}}_{\text{val},l} \|_{\infty} \leq 8B_{\text{ck}}B_{\text{clean}}\sqrt{\frac{N}{3}} 

\| g^{-1}(b') \|_{\text{val},l} \cdot e_{\text{clean}}_{\text{val},l} \|_{\infty} \leq 8B_{\text{ck}}B_{\text{clean}}\sqrt{\frac{N}{3}} 

where

\left[ \begin{array}{c} ks_{\text{ck}} \\ s \end{array} \right] = \sum_{i=1}^{k} \left[ \begin{array}{c} ks_{\text{ck},i} \\ s \end{array} \right] = \sum_{i=1}^{k} \left[ \begin{array}{c} 1 \\ s \end{array} \right] 

= \sum_{i=1}^{k} P \cdot s + e_{\text{ck},s} = P \cdot s + \sum_{i=1}^{k} e_{\text{ck},s} 

\| e_{\text{ck}} \|_{\infty} \leq \sqrt{k} \| e_{\text{ck},s} \|_{\infty} 

\text{C Homomorphic adder}

Homomorphic adder is constructed by homomorphic addition and homomorphic multiplication of TGSW ciphertexts.

\text{C.1 Mathematical expression of adder}

\text{C.1.1 Half-Adder:}

Input two single bit binary numbers \( x, y \), corresponding to GSW ciphertexts TGSW(x) and TGSW(y).

Output:

- Carry: TGSW(c_{\text{out}}) = TGSW(x) \oplus TGSW(y). The corresponding plaintext is \( c_{\text{out}} = x \cdot y \).

- Sum: TGSW(c_{\text{out}}) = TGSW(x) + TGSW(y). The corresponding plaintext is \( c_{\text{out}} = x + y \).

\text{C.1.2 Full-Adder(x,y,c):}

Input two single bit binary numbers \( x, y \) and the carry \( c_{\text{in}} \) corresponding to GSW ciphertexts TGSW(x), TGSW(y) and TGSW(c_{\text{in}}).

Output:

- Carry: TGSW(c_{\text{out}}) = TGSW(x) \oplus TGSW(y) + TGSW(c_{\text{in}}) \oplus \{ TGSW(x) + TGSW(y) \}.

The corresponding plaintext is \( c_{\text{out}} = x \cdot y + c_{\text{in}}(x + y) \).

- Sum: TGSW(c_{\text{out}}) = TGSW(x) + TGSW(y) + TGSW(c_{\text{in}}).

The corresponding plaintext is \( c_{\text{out}} = x + y + c_{\text{in}} \).

Homomorphic addition algorithm for two l-bit TGSW ciphertexts:

\text{HomAdd:}

Input two groups of TGSW ciphertexts \{TGSW(x_1),...,TGSW(x_l)\} and \{TGSW(y_1),...,TGSW(y_l)\} with length of l. The ripple-carry adder is used to calculate the homomorphic addition of two l-bit TGSW ciphertexts.

For \( i = 0 \) to \( l-1 \),

\( (1)_i \{ TGSW(c_{\text{out}}), TGSW(s_{\text{out}}) \} = \text{FullAdd}(TGSW(x_i), TGSW(y_i), 0) \)

\( (2)_i \{ TGSW(c_{\text{out}}), TGSW(s_{\text{out}}) \} = \text{FullAdd}(TGSW(x_i), TGSW(y_i), TGSW(c_{\text{in}})) \)

\( (l-1)_i \{ TGSW(c_{\text{out}}), TGSW(s_{\text{out}}) \} = \text{FullAdd}(TGSW(x_{i-1}), TGSW(y_{i-1}), TGSW(c_{\text{in}})) \).

Output the ciphertext \{ TGSW(c_{\text{out}}), TGSW(s_{\text{out}}),...,TGSW(s_{\text{out}}) \}.

For the homomorphic addition \text{HomAddk}\{TGSW(x_1),...,TGSW(x_l)\} of \( k \cdot l \)-bit TGSW ciphertexts, we use HomAdd algorithm and binary tree to realize fast calculation.

\text{C.2 Error analysis of adder}

For convenience, let \( \overline{X}, \overline{Y}, \overline{C_{\text{in}}}, \overline{s} \) and \( \overline{C_{\text{out}}} \) represent TGSW(x), TGSW(y), TGSW(c_{\text{in}}), and TGSW(c_{\text{out}}), respectively. TGSW(x) and TGSW(y) have the same error variance.

(1) For the homomorphic multiplication between TGSW ciphertexts, we have

\[ \text{Var}(\text{Err}(A \otimes B)) \leq (k + 1)N \beta^2 \text{Var}(\text{Err}(A)) + (1 + kN)\mu(\gamma)^2 + \mu^2 \text{Var}(\text{Err}(B)) \]

where \( k = 1, l = d, \beta = \frac{\beta}{\xi} \), \( V_B = \beta^2 \), \( \mu = \{0, 1\} \) and \( \epsilon \) is the var of gap round.

(2) For the full-adder based on homomorphic multiplication between TGSW ciphertexts, we have

\[ \text{Var}(\text{Err}(\overline{S})) \leq \text{Var}(\text{Err}(\overline{X})) + \text{Var}(\text{Err}(\overline{Y})) + \text{Var}(\text{Err}(\overline{C_{\text{out}}})) = 4dKN \beta^2 + \text{Var}(\text{Err}(c_{\text{in}})) \].
\[
\text{Var}(\overline{\text{Err}}) \leq (6dN_b + 1)\text{Var}(\overline{\text{Err}}(X)) + 2(1 + N)e^2 + \text{Var}(\overline{\text{Err}}(C_{out})) = (6dN_b + 1)2dkN\beta^2 + 2(1 + N)e^2 + \text{Var}(\overline{\text{Err}}(C_{out}))
\]

(3) The 1-bit HomAdd algorithm is formed by continuously running 1 full-adders, the output has an error of variance:
\[
\text{Var}(\text{Err}(c)) \leq 1(6dN_b + 1)\sqrt{2dkN\beta^2 + 2(1 + N)e^2}.
\]
\[
\text{Var}(\text{Err}(s_i)) \leq 2\beta + (l - 1) \cdot (6dN_b + 1)\sqrt{2dkN\beta^2 + 2(1 + N)e^2}
\]
\[
\leq l(6dN_b + 1)\sqrt{2dkN\beta^2 + 2l(1 + N)e^2}.
\]

For convenience,\[\text{Var}(\text{Err}(\text{HomAdd}(\text{output}))) \leq l(6dN_b + 1)\sqrt{2dkN\beta^2 + 2(1 + N)e^2}.
\]

HomAdd algorithm can realize homomorphic addition of k-bit TGSW ciphertexts. There are two methods as below:

1) When the number of users is large, we can use the serial mode to add one addend once. Because this method needs to run halfAdd algorithm \(i\) times, the calculation speed is slow, but the error growth is small. The error variance of its output is
\[
\text{Var}(\text{Err}(\text{HomAdd}(\text{output}))) \leq \sum_{i=1}^{l}(2^{i-1}) \cdot 2dkN\beta^2 + (1 + N)e^2 + 2dkN\beta^2.
\]

2) When the number of users is small, we can run HomAdd algorithm in the form of binary tree. This method needs to operate HomAdd algorithm for \(l = \left\lceil \log(k) \right\rceil\) times, so the calculation speed is fast, but the error growth is also large. Since the length of HomAdd addition is from 1 to \(l = \left\lceil \log(k) \right\rceil\), the error variance of the output is
\[
\text{Var}(\text{Err}(\text{HomAdd}(\text{output}))) \leq (l)(6dN_b + 1)\cdot 2dkN\beta^2.
\]

We can use the above two methods to balance the computational complexity and noise to achieve better results.

Appendix D: Error analysis of AMTMK scheme.

The decomposition basis is defined as \(B\), and the decomposition degree is defined as \(d\). Let \(e^2 = 1/(12B^{2d})\) be the variance of a uniform distribution over \((-1/2B^d, 1/2B^d)\).

Define \(V_B = \begin{cases} \frac{1}{12}(B^2 - 1) & \text{if } B & \text{is odd} \\ \frac{1}{12}(B^2 + 2) & \text{if } B & \text{is even} \end{cases}\) as the uniformly distributed variance over \((-1/2B^d, 1/2B^d)\). Also, define the parameters \(e^2, V_B, B\), and \(B_k\) in the bootstrapping algorithm.

Define the secret key distribution \(\chi \in \{0, 1\}^n\) and \(\varphi \in \{0, 1\}^n\) on RGSW and LWE. Let \(\text{Var}(\phi)\) be the variance of the random variable \(e\) over \(\mathbb{R}\). If \(e\) is a vector composed of random variables, \(\text{Var}(\phi)\) is the maximum variance of the vector.

**Rounding error.** Given \(\tilde{b} = [2N \cdot b']\) and \(\tilde{a} = [2N \cdot a']\), assuming that the each rounding of error obeys the random uniform distribution of \(\mathbb{R}(\text{mod} 1) = (-0.5, 0.5]\), then the variance of the overall rounding error of expression \((\tilde{b} - [2N \cdot b']) + <\tilde{a} - [2N \cdot a'], s >\) is \(\frac{1}{12}(1 + n/2)\).

**The initial error of the evaluation key.**

The variance of error \(KS_{ij}\) is
\[
\text{Var}(\text{Err}(KS_{ij})) = \text{Var}(\text{Err}(TGSW({0,1}))) = (2N \cdot \varepsilon) \cdot \text{Var}(\text{Err}(\text{Ring})) = \text{Var}(\text{Err}(\text{Ring})) + \epsilon^2.
\]

**The variance error \(BK_{ij}\) is**
\[
\text{Var}(\text{Err}(BK_{ij})) = (2N \cdot \varepsilon) \cdot \text{Var}(\text{Err}(\text{Ring})) + \epsilon^2.
\]

According to CGGI17 scheme, the bootstrap error of AMTMK scheme is analyzed as follows.

\(\triangleright\) Let \(d_0,d_1\) be TRLWE samples and let \(C \in \text{TGSW}((0,1))\). Then, \(\text{msg}(\text{CMux}(C,d_i,d_j)) = \text{msg}(C) \cdot \text{msg}(d_i) \cdot \text{msg}(d_j)\). And we have
\[
\|\text{Err}(\text{CMux}(C,d_i,d_j))\|_\infty \leq \max (\|\text{Err}(d_i)\|_\infty + \eta(C)),
\]
where \(\eta(C) = 2dN \cdot \frac{\varepsilon}{2} \cdot \|\text{Err}(C)\|_\infty + (k + 1)\varepsilon\) so
\[
\text{Var}(\text{Err}(\text{CMux}(C,d_i,d_j))) \leq \max (\text{Var}(\text{Err}(d_i)), \text{Var}(\text{Err}(d_j)) + G(C)),
\]
where \(G(C) = 2dN^2 \cdot \text{Var}(\text{Err}(C)) + (N + 1)\varepsilon^2\).

**The accumulated process.** The initial RLWE ciphertext is general, and its error is 0. All bootstrap keys \(\{BK_{ij}, i \leq j \}\) are generated by HomAdd algorithm, and the variance of error is
\[
\sum_{i=1}^{l}(2^{i-1}) \cdot 2dkN\beta^2 + (1 + N)\varepsilon^2 + 2dkN\beta^2.
\]
By recursively running Cmux circuit for \(l \cdot n\) times, the error variance of accumulated process is
\[
2dN^2 \cdot \text{ln}(\sum_{i=1}^{l}(2^{i-1}) \cdot 2dkN\beta^2 + (1 + N)\varepsilon^2 + 2dkN\beta^2) + \text{ln}(N + 1)\varepsilon^2.
\]

**The key switching algorithm.** Input accumulated ciphertext \(ct = (b,a) \in \mathbb{T}^{n \cdot 1}\) and accumulated key \(KS_{ji} = \sum_{i=1}^{l} KS_{ij}\) for \(j \in [N]\) and
Multi-key Fully Homomorphic Encryption Scheme with Fixed-length Ciphertext

\[ KS_{i,j} = R_{i,j}A + (e_{i,j}, 0, ..., 0) + (t_{i,j}, 0, ..., 0) \cdot g_{bs} . \]

Output the ciphertext \( (b', a') = \sum_{j=1}^{N} g_{bs}^{-1}(a_j) \overline{KS}_{st,j} \) (mod 1) and its error variance

\[ Var(Err(c')) = \frac{1}{2} \varepsilon^2 N + d_{bs} V_{B_{bs}} N \alpha^2 (1 + m) + \]

\[ Var(Err(c)) \]

**The bootstrapping process.** The error of bootstrap process can be obtained from the accumulated process and the key switching process, so the error variance is

\[ \frac{1}{2} \varepsilon^2 N + d_{bs} V_{B_{bs}} N \alpha^2 (1 + m) + \]

\[ 2dN_{V_{B}} \cdot \ln\{ \sum_{i=1}^{j} (2^{j-1}) \cdot (2dN_{V_{B}} \cdot 2dkN \beta^2) + \]

\[ (1 + N)\varepsilon^2 + 2dkN \beta^2 \} + (N + 1)\ln\varepsilon^2 \]