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_mrche@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.

Multi-key Fully Homomorphic Encryption Scheme with Compact Ciphertext
Tanping Zhou is with the Engineering University of People's Armed Police，Xi'an, China, E-mail: tanping2020@iscas.ac.cn

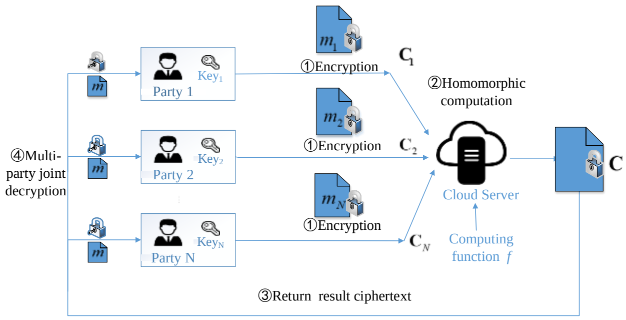

Figure 1: Multiparty data security computing model of MKFHE

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, Chillotti et al. constructed the full homomorphic scheme CGGI16[24] based on a variant of GSW13 on the T=(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_i , pk_i)

from the selected parameters. For example, $sk_i := \boldsymbol{s}_i$, $pk := [\boldsymbol{b}_i = \boldsymbol{s}_i\boldsymbol{B} + \boldsymbol{e}_i, \boldsymbol{B}] \in \mathbb{Z}_q^{m\times n}$. Then, the *jpk* of the parties set is generated by accumulating all the parties' *pk*. So the *jpk* is obtained as $\overline{\overline{pk}} := \overline{\overline{A}} \triangleq [\boldsymbol{b}_1 + ... + \boldsymbol{b}_k \quad \boldsymbol{B}] \in \mathbb{Z}_q^{m\times n}$. $\overline{\overline{pk}}$ can be used for all parties to encrypt their data, so that all the ciphertexts correspond the same *jsk* without performing the ciphertext extension program. 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.

Schemes	Bit-Size		
	ciphertext	evaluation key	accumulated switching key
LZY+19	$O(kn)$	$O(k^3n)$	$O(kn)$
CCS19	$O(kn)$	$O(k^2n^2)$	$O(kn)$
CDKS19	$O(kn)$	$O(kn)$	$O(kn)$
Our scheme	$O(n)$	$O(n)$	$O(n)$

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 $\mathcal{E} = (\text{Setup}, \text{KeyGen}, \text{Enc}, \text{Eval}, \text{Dec})$ is described as follows:

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

$\mathcal{E}.\text{Key Gen}(pp)$: 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 μ, outputs a ciphertext ct_i.

$\mathcal{E}.\text{Dec}((sk_{i_1}, sk_{i_2}, ..., sk_{i_k}), ct_S)$: Given a ciphertext ct_S corresponding to a set of parties $S = \{i_1, i_2, ..., i_k\} \subseteq [K]$, and their secret keys $sk_S = \{sk_{i_1}, sk_{i_2}, ..., sk_{i_k}\}$, outputs the message μ 。

$\mathcal{E}.\text{Eval}(\mathcal{C}, (ct_{S_1}, pk_{S_1}, evk_{S_1}), ..., (ct_{S_t}, pk_{S_t}, evk_{S_t}))$: On input a Boolean circuit $\mathcal{C}$ along with t tuples $(ct_{S_i}, pk_{S_i}, evk_{S_i})_{i=1,...,t}$, each tuple comprises of a ciphertext ct_{S_i} corresponding to a parties set S_i, a set of public keys $pk_{S_i} = \{pk_j, \forall j \in S_i\}$, and the evaluation keys evk_{S_i}, outputs a ciphertext ct_S 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 $\{(ct_{S_i}, pk_{S_i})\}_{i\in\{1,...,t\}}$, let $\mu_i = \text{Dec}(sk_{S_i}, ct_{S_i})$, where $sk_{S_i} = \{sk_j, \forall j \in S_i\}$, a leveled MKFHE scheme $\mathcal{E}$ is correct if it holds that

$$\Pr[\text{Dec}(sk_S, Eval(\mathcal{C}, (ct_{S_i}, pk_{S_i}, evk_{S_i})_{i\in[t]})) \neq \mathcal{C}(\mu_1, ..., \mu_t)] = negl(\lambda)$$

Definition 2.3 (Compactness of MKFHE)[21]. A leveled MKFHE scheme is compact if there exists a polynomial $poly(\cdot,\cdot,\cdot)$ such that $|ct| \leq poly(\lambda, K, L)$, which means that the length of ct is independent of the circuit $\mathcal{C}$, but depend on the security parameter λ, 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 λ 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 $(\mathbf{a}_i, b_i) \in R_q^{n+1}$ uniformly from

R_q^{n+1} . For the second distribution, one first draws $\boldsymbol{a}_i \leftarrow R_q^n$ uniformly, and samples $(\boldsymbol{a}_i, b_i) \in R_q^{n+1}$ by choosing $\boldsymbol{s} \leftarrow R_q^n$ and $e_i \leftarrow \chi$ uniformly, and set $b_i = <\boldsymbol{a}_i, \boldsymbol{s}> + e_i$. The GLWE assumption is that the GLWE problem is infeasible.

LWE problem. The LWE problem is simply GLWE problem instantiated with $d = 1$.

RLWE problem. The RLWE problem is GLWE problem instantiated with $n = 1$.

2.3 BitDecomp( ) and Powersof2( )[3]

Here we introduce two subroutines ($\text{BitDecomp}(\cdot)$ and $\text{Powersof }2(\cdot)$) which are widely used in FHE schemes. Let $\mathbf{x} \in R_q^n$ be a polynomial of dimension n over R_q , and let $\beta = \lfloor \log q \rfloor + 1$.

$\text{BitDecomp}(\mathbf{x} \in R_q^n, q)$: On input $\mathbf{x} = (x_1, ..., x_n) \in R_q^n$ and the modulus q , outputs $(x_{1,0}, ..., x_{1,\beta-1}, ..., x_{n,0}, ..., x_{n,\beta-1}) \in \{0,1\}^{n \cdot \beta}$ where $x_{i,j}$ is the *j*-th bit in x_i 's binary representation (ordered from least significant to most significant), namely $\mathbf{x} = (\sum_{j=0}^{\beta-1} 2^j x_{1,j}, ..., \sum_{j=0}^{\beta-1} 2^j x_{n,j})$.

$\text{Powersof }2(\mathbf{y} \in R_q^n, q)$: On input $\mathbf{y} = (y_1, ..., y_n) \in R_q^n$ and the modulus q , outputs $(y_1, 2y_1, ..., 2^{\beta-1} y_1, ..., y_n, 2y_n, ..., 2^{\beta-1} y_n) \in R_q^{n \cdot \beta}$.

It's straightforward to verify that for arbitrary $\mathbf{x}, \mathbf{y} \in R_q^n$, it holds that

$$\langle \text{BitDecomp}(\mathbf{x}, q), \text{Powersof }2(\mathbf{y}, q) \rangle = \langle \mathbf{x}, \mathbf{y} \rangle \bmod q .$$

3 General construction of MKFHE scheme with compact ciphertext

In this section, we introduce the general construction of fixed length ciphertext MKFHE (we call this kind of scheme FCMKFHE scheme for short). The ciphertext expansion algorithm plays an important role in MKFHE scheme. Its function is to expand one party's ciphertext to multi-parties' ciphertext. When all parties' ciphertexts correspond to the same joint secret key, the homomorphic computation can be performed just like single-party FHE scheme. The ciphertext expansion function is the core of the ciphertext expansion algorithm. It generates the corresponding extended ciphertext according to the form of the cascaded secret key. Usually, the generated joint ciphertext is a linear or a square relationship about the number of parties. When the number of parties increases, the efficiency of the scheme will drop sharply, which is exactly the bottleneck restricting the specific application of MKFHE. Therefore, for MKFHE, the form of joint secret key determines 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).

$\text{SMMK.Setup}(1^\lambda) : \text{FHE.Setup}(1^\lambda) \rightarrow \boldsymbol{params}$;

$\text{SMMK.KeyGen}(\boldsymbol{params}, i, \boldsymbol{B})$: $\text{FHE.KeyGen}(\boldsymbol{params}, \boldsymbol{B}) \rightarrow pk_i, sk_i$;After all parties have completed the process SMMK.KeyGen(), run the generation algorithm of evaluation key.

$\text{SMMK.EvalKeyGen}(\boldsymbol{params}, sk_i, \{pk_1, ..., pk_k\})$:

1) $\text{SMMK.SMPK}(\boldsymbol{params}, sk_i, \{pk_1, ..., pk_k\})$:

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 $\boldsymbol{B} \in \mathbb{Z}_q^{m \times (n-1)}$, the party's secret key $sk_i = \boldsymbol{s}_i$ and public key $\boldsymbol{b}_i = [\boldsymbol{s}_i \boldsymbol{B} + \boldsymbol{e}_i, \boldsymbol{B}] \in \mathbb{Z}_q^{m \times n}$ Output the *jpk* as $\overline{\overline{pk}} := \overline{\overline{\boldsymbol{A}}} \triangleq [\boldsymbol{b}_1 + ... + \boldsymbol{b}_k \quad \boldsymbol{B}] \in \mathbb{Z}_q^{m \times n}$.

2) $\text{FHE.SwitchKeyGen}(\boldsymbol{params}, sk_i, \overline{\overline{pk}})$: Input the party's *sk* and *jpk*, output the *accumulated evaluation key* (*aek*) of party *i* $\overline{\overline{\boldsymbol{KS}}}_i = \text{FHE.Enc}_{\overline{\overline{pk}}}(sk_i \otimes sk_i)$ and the *accumulated switching*

key (*awk*) of parties set $\overline{\overline{\boldsymbol{KS}_{\overline{\overline{Set}}}}}$. 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($\boldsymbol{params}$,sk_i, $\overline{\overline{pk}}$): Input the party's *sk* and *jpk* . Output the *bootstrapping key* (*bk*) of party *i* $\overline{\overline{\boldsymbol{BK}_i}} = \text{FHE.Enc}_{\overline{\overline{pk}}}(sk_i)$, and the *accumulated strapping key* (*abk*) of the accumulated ciphertext $\overline{\overline{\boldsymbol{BK}_{\overline{\overline{Set}}}}} = HomAdd_{\log(kB_\chi)}(\overline{\overline{\boldsymbol{BK}_1}},...,\overline{\overline{\boldsymbol{BK}_k}})$, where $HomAddk(*)$ is the homomorphic addition circuit for l bits.

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

SMMK.Dec($(sk_1,...,sk_N),\boldsymbol{C}$) . Like most MKFHE schemes, the decryption result consists of two parts: partial decryption and final decryption.

1) SMMK.PartDec($\boldsymbol{C}, i, sk_i$): Input the secret key of party *i* $sk_i = (-\boldsymbol{s}_i, 1)$ and the result ciphertext ***C***, and output the partial decryption. Taking the GSW-type FHE as an example, its decryption form is $\mu' = \boldsymbol{s}_i\boldsymbol{C}\boldsymbol{G}^{-1}(\hat{\boldsymbol{w}}^T)$. We only calculate $p_i' := \boldsymbol{s}_i\boldsymbol{C}_{[1,...,n-1;]}\boldsymbol{G}^{-1}(\hat{\boldsymbol{w}}^T) + e_i^{sm}$ and get the partial decryption p_i', where $\boldsymbol{C}_{[1,...,n-1;]}$ represents the first *n*-1 columns of ciphertext $\boldsymbol{C}$, and $e_i^{sm} \stackrel{R}{\leftarrow} [-B_{smdg}^{dec}, B_{smdg}^{dec}]$ is the generated error used to protect the security of partial decryption.

2) SMMK.FinDec($p_1', \ldots, p_N'$) : Input all the partial decryptions, and output the resulting plaintext $m' = \hat{\boldsymbol{C}}_{[n;]}\hat{\boldsymbol{G}}^{-1}(\hat{\boldsymbol{w}}^T) - \sum_{i=1}^N p_i'$.

The homomorphic computation is as follows.

SMMK.Add($\boldsymbol{C}_1,\boldsymbol{C}_2$): $\boldsymbol{C}_+ \leftarrow$ FHE.Add($\boldsymbol{C}_1,\boldsymbol{C}_2$) .

SMMK.Mult($\boldsymbol{C}_1,\boldsymbol{C}_2,\overline{\overline{\boldsymbol{EVK}_{\overline{\overline{Set}}}}}$):

$\boldsymbol{C}_\times \leftarrow \text{FHE.Mult}_{\overline{\overline{\boldsymbol{EVK}_{\overline{\overline{Set}}}}}}(\boldsymbol{C}_1,\boldsymbol{C}_2)$.

SMMK.Bootstrap($\overline{\overline{\boldsymbol{BK}_{\overline{\overline{Set}}}}},\boldsymbol{C}$): FHE.Bootstrap($\overline{\overline{\boldsymbol{BK}_{\overline{\overline{Set}}}}},\boldsymbol{C}$) .

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

$$\begin{aligned}&\hat{\boldsymbol{C}}_{[n;]}\hat{\boldsymbol{G}}^{-1}(\hat{\boldsymbol{w}}^T) - \sum_{i=1}^N p_i' = \sum_{i=1}^N \boldsymbol{e}_i \cdot \boldsymbol{R}\hat{\boldsymbol{G}}^{-1}(\hat{\boldsymbol{w}}^T)\\ &+\mu\hat{\boldsymbol{t}}(\hat{\boldsymbol{w}}^T) - \sum_{i=1}^N e_i^{sm} = \lfloor q/2 \rfloor + e^*\end{aligned} \tag{1}$$

For the above scheme, the ciphertexts of all parties are encrypted by the *joint public key* $\overline{\overline{pk}}$, 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 FCMKFHE and TFHE-type FCMKFHE. 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 FCMKFHE scheme, which supports the timely updating parties set without regenerating their ciphertexts and keys.

3.2 Authorized mode FCMKFHE 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 FCMKFHE 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^λ): FHE.Setup(1^λ) $\to$ $\boldsymbol{params}$;

AMMK.KeyGen($\boldsymbol{params}$): FHE.KeyGen($\boldsymbol{params},\boldsymbol{B}$) $\to (pk_i, sk_i)$;After all parties have completed the process AMMK.KeyGen(), run the generation algorithm of evaluation key.

AMMK.EvalKeyGen($\boldsymbol{params}$,sk_i,$\{pk_1,...,pk_N\}$):

$$\text{SMMK.EvalKeyGen}(\boldsymbol{params}, sk_i, \{pk_1,...,pk_N\}) \to \{\overline{\overline{pk}}, \overline{\overline{\boldsymbol{KS}_i}}, \overline{\overline{\boldsymbol{KS}_{\overline{\overline{Set}}}}}, \overline{\overline{\boldsymbol{BK}_i}}, \overline{\overline{\boldsymbol{BK}_{\overline{\overline{Set}}}}}\};$$

AMMK.Enc(pk,μ): FHE.Enc(pk,μ) $\to \boldsymbol{C}$.（Note: This is a single party's public key encryption, not a joint public key encryption）

AMMK.Dec($(sk_1,...,sk_N),\boldsymbol{C}$): SAMK.Dec($(sk_1,...,sk_N),\boldsymbol{C}$) $\to \mu'$.

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

AMMK.Eval($(\boldsymbol{C}_1,\boldsymbol{C}_2),\overline{\overline{\boldsymbol{BK}_i}},\overline{\overline{\boldsymbol{KS}_{\overline{\overline{Set}}}}}$):

1) $\boldsymbol{C}_i' = Hom_{\overline{\overline{pk}},\overline{\overline{\boldsymbol{BK}_{c_i}}}}\{\text{FHE.Dec}_{sk_{c_i}}(\boldsymbol{C}_i)\}$. This process can refresh different public keys. $\overline{\overline{\boldsymbol{BK}_{\boldsymbol{c}_i}}}$ is the *bootstrapping key* corresponding to $\boldsymbol{C}_i$. If $\boldsymbol{C}_i$ is the ciphertext of a single party, the *bootstrapping key* $\overline{\overline{\boldsymbol{BK}_{\boldsymbol{c}_i}}}$ is setted as $\overline{\overline{\boldsymbol{BK}_i}}$. If . $\boldsymbol{C}_i$ is the joint

ciphertext of all parties, the *bootstrapping key* $\overline{\overline{\boldsymbol{BK}}}_{c_i}$ is setted as $\overline{\overline{\boldsymbol{BK}}}_{\overline{\overline{set}}}$.

2）FHE.Eval$((\boldsymbol{C}_1', \boldsymbol{C}_2'), \overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}}})$. This process realizes the homomorphic computation of joint ciphertext, where $\overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}}}$ 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 $\{0,1\}^N$, 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$(1^\lambda) \to pp = (pp^{\text{LWE}}, pp^{\text{GSW}})$：

LWE.Setup$(1^\lambda) \to pp^{\text{LWE}} = (\eta, \chi, \alpha, B_{ks}, d_{ks}, \boldsymbol{B})$；

GSW.Setup$(1^\lambda) \to pp^{\text{GSW}} = (N, \phi, \alpha, B, d, \boldsymbol{y})$, Where $\boldsymbol{B}, \boldsymbol{y}$ are common random variables.

AMTMK.KeyGen$(pp) \to (pk_i, sk_i, pk_{BK,i}, sk_{BK,i})$:

LWE.KeyGen$(pp) \to \{pk_i = \boldsymbol{A}_i, sk_i = \boldsymbol{s}_i\}$；

RGSW.KeyGen$(pp) \to \{pk_{BK,i} = \boldsymbol{Z}_i, sk_{BK,i} = z_i\}$.

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

AMTMK.EvalKeyGen$(pp, sk_i, \{pk_1, ..., pk_k\}) \to \{\overline{\overline{pk}}, \overline{\overline{\boldsymbol{KS}}}_i, \overline{\overline{\boldsymbol{BK}}}_i\}$:

1) **Accumulate the public key.** Given the public keys $\boldsymbol{b}_1, ..., \boldsymbol{b}_k$ of k parties, we obtain the *joint public key* $\overline{\overline{pk}} := [\boldsymbol{b}_1 + ... + \boldsymbol{b}_k \quad \boldsymbol{B}] \in \mathbb{Z}_q^{m \times n}$.

Accumulate the bootstrapping public key. Given the bootstrapping public keys $\boldsymbol{d}_1, ..., \boldsymbol{d}_k$ of k parties, we obtain the *accumulated bootstrapping public key* $\overline{\overline{pk}}_{BK} := \overline{\overline{\boldsymbol{Z}}} = [\boldsymbol{d}_1 + ... + \boldsymbol{d}_k \quad \boldsymbol{y}] \in T_q^{2d \times 2}$。

2) **Accumulate the single-party bootstrapping key.** Input the *accumulated bootstrapping public key* $\overline{\overline{pk}}_{BK} = \overline{\overline{\boldsymbol{Z}}}$ and the secret key $\boldsymbol{s}_i \in \mathbb{Z}^n$ of LWE ciphertext. Output the single-party's *accumulated bootstrapping key* $\overline{\overline{\boldsymbol{BK}}}_i = \{\overline{\overline{\boldsymbol{BK}}}_{i,j}\}_{j \in [n]}$, where $\overline{\overline{\boldsymbol{BK}}}_{i,j} = \text{RGSW.Enc}(s_{i,j}, \overline{\overline{\boldsymbol{Z}}})$, $i \in [k]$, $j \in [n]$.

3) **Accumulate the evaluation key**. Input the accumulated public key $\overline{\overline{pk}}$ and the secret key z_i of the RGSW ciphertext, let $\boldsymbol{t}_i =: (z_{i,0}, -z_{i,w-1}, ..., -z_{i,1}) \in \mathbb{B}^N$, and output the *accumulated evaluation key* (*aek*) $\overline{\overline{\boldsymbol{KS}}}_i = \text{LWE.KSGen}(\boldsymbol{t}_i, \overline{\overline{pk}})$ of single party, where $i \in [k]$.

AMTMK.Enc(pk, μ): Input the plaintext μ, and single party's public key pk, run LWE.Enc$(pk, \mu) \to \boldsymbol{ct} = (b, \boldsymbol{a}) \in \mathbb{T}^{n+1}$.

AMTMK.Dec$((sk_1, ..., sk_k), \boldsymbol{ct})$: Input the ciphertext $\boldsymbol{ct} = (b, \boldsymbol{a}) \in \mathbb{T}^{n+1}$ and the secret key $(sk_1, ..., sk_k)$. Return the plaintext bit $\mu' \in \{0,1\}$ that makes $|b + \sum_{j=1}^{k} < \boldsymbol{a}, \boldsymbol{s}_j > -\frac{1}{4}m|$ be smallest.

AMTMK.Boot$(\boldsymbol{c}, \{\overline{\overline{\boldsymbol{BK}}}_i\}_{i \in [k]}, \{\overline{\overline{\boldsymbol{KS}}}_i\}_{i \in [k]})$: Input the ciphertext $\boldsymbol{ct} = (b', \boldsymbol{a}') \in \mathbb{T}^{n+1}$, the *bootstrapping key* set $\{\overline{\overline{\boldsymbol{BK}}}_i\}_{i \in [k]}$ and the *accumulated evaluation key* set $\{\overline{\overline{\boldsymbol{KS}}}_i\}_{i \in [k]}$. Then use bootstrapping process to realize homomorphic computation.

1) The cloud server uses $\overline{\overline{\boldsymbol{KS}}}_i$ to generate *accumulated switching key* $\overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}}} = \{\sum_{i=1}^{k} \overline{\overline{\boldsymbol{KS}}}_{i,j}\}_{j \in [N]}$. The cloud sever also uses the $\overline{\overline{\boldsymbol{BK}}}_i$ to generate the *accumulated bootstrapping key* $\overline{\overline{\boldsymbol{BK}}}_{bit_{l-1}(\overline{\overline{Set}}, j)}, ..., \overline{\overline{\boldsymbol{BK}}}_{bit_0(\overline{\overline{Set}}, j)} = HomAddk(\overline{\overline{\boldsymbol{BK}}}_{1,j}, ..., \overline{\overline{\boldsymbol{BK}}}_{k,j})$, where $j \in [n]$, $l = \lceil \log(k) \rceil$. $HomAddk(,)$ 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 $\overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}}}$ and $\overline{\overline{\boldsymbol{BK}}}_{\overline{\overline{Set}}}$ once, and then output them as public variables.

2) Ciphertext refresh. Given ciphertext $\boldsymbol{c} = (b', \boldsymbol{a}') \in \mathbb{T}^{n+1}$, and the evaluated key $\{\overline{\overline{\boldsymbol{BK}}}_{bit_{l-1}(\overline{\overline{Set}})}, ..., \overline{\overline{\boldsymbol{BK}}}_{bit_0(\overline{\overline{Set}})}\}$ or $\overline{\overline{\boldsymbol{BK}}}_i$. Run the following homomorphic accumulation algorithm [24]:

① Input the ciphertext $\boldsymbol{c}=(b',\boldsymbol{a}')\in\mathbb{T}^{n+1}$, output $\tilde{b}=\lfloor 2N\cdot b'\rceil$, $\tilde{\boldsymbol{a}}=\lfloor 2N\cdot\boldsymbol{a}'\rceil$ and the *bootstrapping key*

$$\overline{\overline{\boldsymbol{BK}}}=\begin{cases}\overline{\overline{\boldsymbol{BK}_i}} & \boldsymbol{c}\text{ corresponds to the secret key }\boldsymbol{s}_i\\ \overline{\overline{\boldsymbol{BK}_{\overline{\overline{Set}}=\{s_1,\dots,s_{k'}\}}}} & \boldsymbol{c}\text{ corresponds to the secret key }(s_1+\dots+s_{k'})\end{cases}$$

② Initialize the RLWE ciphertext $\boldsymbol{ACC}=(-\frac{1}{8}h(X)\cdot X^{\tilde{b}},\boldsymbol{0})$, where $h(X)=1+X+\dots+X^{\frac{N}{2}-1}-X^{\frac{N}{2}}-\dots-X^N$. Let $\tilde{\boldsymbol{a}}=(\tilde{a}_j)_{j\in[n]}$, for j=1 to n, run the following process.

(1).$\boldsymbol{ACC}$=CMux$(\overline{\overline{\boldsymbol{BK}_{bit_0(\overline{\overline{Set}},j)}}},X^{a_j}\boldsymbol{ACC},\boldsymbol{ACC})$;

(2).$\boldsymbol{ACC}$=CMux$(\overline{\overline{\boldsymbol{BK}_{bit_1(\overline{\overline{Set}},j)}}},X^{2a_j}\boldsymbol{ACC},\boldsymbol{ACC})$;

...

$(l-1)$.$\boldsymbol{ACC}$=CMux$(\overline{\overline{\boldsymbol{BK}_{bit_{l-1}(\overline{\overline{Set}},j)}}},X^{(2^{l-1})a_j}\boldsymbol{ACC},\boldsymbol{ACC})$.

We select the largest circuit $\mathrm{CMux}(\boldsymbol{C},\boldsymbol{d}_1,\boldsymbol{d}_0)$. Input one TGSW ciphertext $\boldsymbol{C}$ and two input RLWE ciphertexts $\boldsymbol{d}_1,\boldsymbol{d}_0$. Output RLWE ciphertext $\boldsymbol{C}\boxdot(\boldsymbol{d}_1-\boldsymbol{d}_0)+\boldsymbol{d}_0$, where $\boxdot$ is a hybrid homomorphic multiplication of GSW ciphertext and BGV ciphertext. The specific process is shown in [24].

③ Output $\boldsymbol{ACC}\leftarrow(\frac{1}{8},\boldsymbol{0})+\boldsymbol{ACC}(\mathrm{mod}1)$

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

① Input the ciphertext $\boldsymbol{ACC}=(c_0,c_1)\in T^2$. Set b'' be a constant term of polynomial c_0 and $\boldsymbol{a}''$ be a vector composed of coefficients of polynomial c_1. Output LWE ciphertext .

②Let $\overline{\overline{\boldsymbol{KS}_{\overline{\overline{Set}}}}}=\{\sum_{i=1}^{k}\overline{\overline{\boldsymbol{KS}_{i,j}}}\}_{j\in[N]}$, run the key switching algorithm and output the ciphertext $\overline{\overline{\boldsymbol{ct}}}\leftarrow\mathrm{LWE.MKSwitch}(\overline{\boldsymbol{ct}}'',\overline{\overline{\boldsymbol{KS}_{\overline{\overline{Set}}}}})$.

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

$\mathrm{AMTMK.NAND}(\boldsymbol{c}_1,\boldsymbol{c}_2)=\mathrm{HDTMK.Boot}((\boldsymbol{0},5/8)-\boldsymbol{c}_1-\boldsymbol{c}_2)$.

Security. Like most schemes, our scheme security also rely on cyclic security assumption. The semantic security of our scheme is based on (R)LWE assumption, and the parameters pp^{LWE} and pp^{GSW} make the (R)LWE assumption to be λ - bit secure.

Correctness. In this scheme, the error of ciphertext $(b',\boldsymbol{a}')$ in bootstrap process is $(b'+<\boldsymbol{a}',\overline{\overline{\boldsymbol{s}}}>)=\frac{m}{4}+e'$, where $|e'|<\frac{1}{16}$. 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.

AMCMK.Setup(1^λ)：Input the security parameters λ and select an integer N (where N is the power of 2). Let χ_{key}, χ_{err} and χ_{enc} be the distribution of secret key, error and encryption process on $R=\mathbb{Z}[X]/(X^N+1)$ respectively. Select prime P and p. L respects the circuit layers, the ciphertext modulus is $q_l=p^l$, where $1\le l\le L$. Select $\boldsymbol{a}\leftarrow U(R^d_{P\cdot q_L})$ and $a'\leftarrow U(R^1_{P\cdot q_L})$. Output common parameter $pp=(N,\chi_{key},\chi_{err},\chi_{enc},L,P,q_l,a,a')$.

AMCMK.KeyGen(pp)：Input the parameters pp. Select $s\leftarrow\chi_{key}$ and $e'\leftarrow\chi_{err}$, and output the public key $pk:=b'=-s\cdot a'+e'\in R^1_{P\cdot q_L}$. Select $\boldsymbol{e}\leftarrow\chi^d_{err}$ and generate the calculated public key $pk_{evk}:=\boldsymbol{b}=-s\cdot\boldsymbol{a}+\boldsymbol{e}\in R^d_{P\cdot q_L}$., generate the evaluation key evk for basic CKKS : Let $s'\leftarrow s^2$. Set the evaluation key as $evk\leftarrow(b'',\ a'')\in R^2_{Pq_L}$ where.

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

AMCMK.EvalKeyGen$(pp,sk_i,\{pk_i\}_{i\in[k]},\{pk_{evk,i}\}_{i\in[k]})\to\{\overline{\overline{pk}},\overline{\overline{\boldsymbol{ks}}},\overline{\overline{\boldsymbol{rk}}},\overline{\overline{\boldsymbol{ck}}}\}$:

1) Accumulate the public key. Given k parties' public key $\boldsymbol{b}_1,\dots,\boldsymbol{b}_k$, the CKKS-type *accumulated public key* is generated as $\overline{\overline{pk}}:=(\boldsymbol{b}_1+\dots+\boldsymbol{b}_k)\in R^d_{Pq_L}$, where $\overline{\overline{pk}}_{[,j]}$ represents the j-th element of $\overline{\overline{pk}}$.

2）Accumulate the evaluation key. Given the k parties' evaluation public key $b'_1,\dots,b'_k$, the CKKS-type accumulated evaluation public key is generated as $\overline{\overline{pk_{evk}}}:=(b'_1+\dots+b'_k)\in R^1_{Pq_L}$.

3）The accumulated evaluation key generation.

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

$(\overline{\overline{\boldsymbol{d}}}_0, \overline{\overline{\boldsymbol{d}}}_1) := \{(\overline{\overline{d_{0,[,j]}}}, \overline{\overline{d_{1,[,j]}}})\}_{j\in[d]}$, where $(\overline{\overline{d_{0,[,j]}}}, \overline{\overline{d_{1[,j]}}}) \leftarrow \text{CKKS.Enc}_{\overline{\overline{pk_{[,j]}}}, Pq_L}(r \cdot \boldsymbol{g}_{[,j]})$, $j \in [d]$ and $\boldsymbol{g} = (1, B_g, ..., B_g^{d-1})$. B_g is the decomposition basis. Set $\overline{\overline{d}}_2 = r \cdot a' + e_2 + P \cdot s(\bmod Pq_L)$, where $e_s \leftarrow \chi_{err}$.

Output:

$$\overline{\overline{\boldsymbol{ks}}}_{se} := \left[\boldsymbol{g}^{-1}(\overline{\overline{b'}})\right]_{1\times d} \cdot \left[\overline{\overline{\boldsymbol{d}}}_0 \mid \overline{\overline{\boldsymbol{d}}}_{i,1}\right]_{d\times 2} + \left[0 \mid \overline{\overline{d}}_2\right] \in \text{CKKS}_{\overline{\overline{s}}}(P \cdot s_i \overline{\overline{s}})$$

.

The *refresh key* of party's ciphertext set is obtained as $\overline{\overline{\boldsymbol{ks}}}_{s_i \to \overline{\overline{s}}} \leftarrow \text{CKKS.Enc}_{\overline{\overline{pk}}_{evk}, Pq_L}(P \cdot s_i)$. Then, output the *shift key* $\overline{\overline{\boldsymbol{rk}}}_{i,r} \leftarrow \text{CKKS.Enc}_{\overline{\overline{pk}}_{evk}, Pq_L}(\kappa_{5^r}(s_i))$ and *conjugate key* $\overline{\overline{\boldsymbol{ck}}}_i \leftarrow \text{CKKS.KSGen}_{\overline{\overline{pk}}_{evk}, Pq_L}(\kappa_{-1}(s_i))$.

4) Generate the evaluation key in cloud.

$$\overline{\overline{\boldsymbol{ks}}}_{set} := \sum_{i=1}^{k} \overline{\overline{\boldsymbol{ks}}}_{se,i} \in \text{CKKS}_{\overline{\overline{s}}}(P\overline{\overline{s}} \cdot \overline{\overline{s}})$$

$$\overline{\overline{\boldsymbol{rk}}}_{set,r} = \sum_{i=1}^{k} \overline{\overline{\boldsymbol{rk}}}_{i,r} \in \text{CKKS.Enc}_{\overline{\overline{pk}}, Pq_L}(\kappa_{5^r}(\overline{\overline{s}}))$$

$$\overline{\overline{\boldsymbol{ck}}}_{set} = \sum_{i=1}^{k} \overline{\overline{\boldsymbol{ck}}}_i \in \text{CKKS.Enc}_{\overline{\overline{pk}}, Pq_L}(\kappa_{-1}(\overline{\overline{s}})) .$$

When the parties set of the AMCMK scheme is updated, the bootstrapping process is no longer needed. The original 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) : $\boldsymbol{c} = \text{CKKS.Enc}_{pk}(m)$. The encrypted ciphertexts are modulo *P* to reduce their size.

AMCMK.Dec$((sk_1, ..., sk_k), \boldsymbol{c})$: Input the ciphertext $\boldsymbol{c}$ of *l*-th level. Output $m' =< \boldsymbol{c}, sk_1 + ... + sk_k > (\bmod q_l)$.

AMCMK.KeySwitchingKey$(\boldsymbol{c}, \{\overline{\overline{\boldsymbol{ks}}}_{s_i \to \overline{\overline{s}}}\}_{i\in[k]})$: Input the ciphertext $\boldsymbol{c}' = (b', a')$, output the corresponding accumulated switching key

$$\overline{\overline{\boldsymbol{ks}}}_{refresh} = \begin{cases} \boldsymbol{ks}_{s_i \to \overline{\overline{s}}} & \boldsymbol{c} \text{ corresponds to the secret key } \boldsymbol{s}_i \\ \sum_{i=1}^{k'} \boldsymbol{ks}_{s_i \to \overline{\overline{s}}} & \boldsymbol{c} \text{ corresponds to the secret key } (s_1 + ... + s_{k'}) \end{cases}$$

, 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$(\boldsymbol{c}, \{\overline{\overline{\boldsymbol{ks}}}_{s_i \to \overline{\overline{s}}}\}_{i\in[k]})$ 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$(\boldsymbol{ct}, \boldsymbol{ct}')$: CKKS.Add$(\boldsymbol{ct}, \boldsymbol{ct}')$;

AMCMK.CMult$(a, \boldsymbol{ct})$: CKKS.CMult$(a, \boldsymbol{ct})$;

AMCMK.Mult$_{\overline{\overline{\boldsymbol{ks}}}_{set}}(\boldsymbol{ct}, \boldsymbol{ct}')$: CKKS.Mult$_{\overline{\overline{\boldsymbol{ks}}}_{set}}(\boldsymbol{ct}, \boldsymbol{ct}')$;

AMCMK.Bootstrapping$_{\overline{\overline{\boldsymbol{ks}}}_{set}, \overline{\overline{\boldsymbol{rk}}}_{set,r}, \overline{\overline{\boldsymbol{ck}}}_{set}}(\boldsymbol{c})$:

CKKS.Bootstrapping$_{\overline{\overline{\boldsymbol{ks}}}_{set}, \overline{\overline{\boldsymbol{rk}}}_{set,r}, \overline{\overline{\boldsymbol{ck}}}_{set}}(\boldsymbol{c})$.

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 $\| a \|_\infty^{can}$ denote the infinite normal form of $a(\zeta)$ (the inner product of the coefficients of a and vectors $(1, \zeta_M, ..., \zeta_M^{N-1})$) obtained by normal embedding of polynomial $a(X) \in R = \mathbb{Z}[X]/(\Phi_M(X))$. According to the analysis in CKKS17, $\| a \|_\infty^{can} \le 6\sigma$, where σ^2 is the variance of $a(\zeta)$. $\| ab \|_\infty^{can} \le 16\sigma_1\sigma_2$, where σ_1^2 and σ_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 $Var(a(\zeta_M)) = q^2 N / 12$. If a is taken from the discrete Gaussian distribution $DG_q(\sigma^2)$ with variance σ^2 , then $Var(a(\zeta_M)) = \sigma^2 N$. If a is taken from the $\{0, \pm 1\}$ distribution $HWT(h)$ with Hamming weight h , then $Var(a(\zeta_M)) = h$. The CKKS17 scheme can encrypt plural vectors. Considering the accuracy, the scheme usually expands the data by Δ times before encryption, and Δ is called the increment factor. For a given ciphertext $\boldsymbol{ct} \in R_q^2$, the scheme can decrypt correctly if the increment factor $\Delta > N + 2B$, where $< \boldsymbol{ct}, \boldsymbol{sk} >= m + e(\bmod q_L)$, B is the upper bound of $\|e\|_\infty^{can}$. The error growth of important functions is shown in the following Lemmas.

Lemma 1[23]. Let $\boldsymbol{ct} \leftarrow Enc_{pk}(m)$ be an encryption of $m \in R$ and $e \in R$, then $< \boldsymbol{ct}, \boldsymbol{sk} >= m + e(\bmod q_L)$, where $\|e\|_\infty^{can} \le B_{clean}$, such that $B_{clean} = 8\sqrt{2}\sigma N + 6\sigma\sqrt{N} + 16\sigma\sqrt{hN}$.

Lemma 2. Let $\boldsymbol{ct} \leftarrow Enc_{\overline{\overline{pk}}}(m)$ denote the ciphertext of $m \in R$ encrypted by the accumulated public key $\overline{\overline{pk}}$, for a certain set $e \in R$, there is $< \boldsymbol{ct}, (1, \overline{\overline{s}}) >= m + e(\bmod q_L)$, where $\overline{\overline{pk}} = (b_1 + ... + b_k, a)$, $\|e\|_\infty^{can} \le B_{sclean}$ and $B_{sclean} = 8\sqrt{2k}\sigma N + 6\sigma\sqrt{N} + 16\sigma k\sqrt{hN}$.

See Appendix B for detail proof of lemma 2.

For the *refresh key* $\overline{\overline{\boldsymbol{ks}}}_{s_i \to \overline{\overline{s}}} \leftarrow \text{CKKS.Enc}_{\overline{\overline{pk}}_{evk}, Pq_L}(P \cdot s_i)$, the *shift key* $\overline{\overline{\boldsymbol{rk}}}_{i,r} \leftarrow \text{CKKS.Enc}_{\overline{\overline{pk}}_{evk}, Pq_L}(\kappa_{5^r}(s_i))$ and the *conjugate key* $\overline{\overline{\boldsymbol{ck}}}_i \leftarrow \text{CKKS.KSGen}_{\overline{\overline{pk}}_{evk}, Pq_L}(\kappa_{-1}(s_i))$, we have $\|e\|_\infty^{can} \le B_{sclean}$.

Lemma 3. Let $\overline{\overline{\boldsymbol{ks}}}_{set}$ be the accumulated switch-key, $\overline{\overline{\boldsymbol{ks}}}_{se,i}$ be one element of $\overline{\overline{\boldsymbol{ks}}}_{set}$, then, $<\overline{\overline{\boldsymbol{ks}}}_{se,i}, (1, \overline{\overline{s}})>= P^{-1} \cdot s_i \overline{\overline{s}} + e_{\overline{\overline{\boldsymbol{ks}}}_{se,i}} (\bmod q_L)$, where $\| e_{\overline{\overline{\boldsymbol{ks}}}_{set}} \|_\infty^{can} \le B_{\overline{\overline{\boldsymbol{ks}}}_{set}} = \sqrt{k} \| e_{\overline{\overline{\boldsymbol{ks}}}_{se,i}} \|_\infty^{can}$ and $\| e_{\overline{\overline{\boldsymbol{ks}}}_{se,i}} \|_\infty^{can} \le B_{\overline{\overline{\boldsymbol{ks}}}_{se}} = 8 B_{ks} B_{sclean} \sqrt{dN/3}$.

See Appendix B for detail proof of lemma 3.

Lemma 4[23]. Let $\boldsymbol{ct}' \leftarrow RS_{l \to l'}(\boldsymbol{ct})$ (where $\boldsymbol{ct} \in R_q^2$), for $e \in R$, there is $<\boldsymbol{ct}', \boldsymbol{sk}>= \frac{q_{l'}}{q_l} <\boldsymbol{ct}, \boldsymbol{sk}> + e (\bmod q_{l'})$, where $\|e\|_\infty^{can} \le B_{rs}$, $B_{rs} = \sqrt{N/3}(3 + 8\sqrt{h})$.

Lemma 5. Let $\boldsymbol{ct}_{mult} \leftarrow Mult_{\overline{\overline{\boldsymbol{ks}}}_{set}}(\boldsymbol{ct}_1, \boldsymbol{ct}_2)$ (where $\boldsymbol{ct}_1, \boldsymbol{ct}_2 \in R_q^2$), for $e \in R$, there is $<\boldsymbol{ct}_{mult}, \boldsymbol{sk}>=<\boldsymbol{ct}_1, \boldsymbol{sk}><\boldsymbol{ct}_2, \boldsymbol{sk}> + e_{mult} (\bmod q_{l'})$, where $\|e\|_\infty^{can} \le B_{mult}$, $B_{mult} = P^{-1} q_l \cdot B_{\overline{\overline{\boldsymbol{ks}}}_{set}} + B_{rs}$.

Lemma 5 can be obtained by taking the upper bound $\| \overline{\overline{\boldsymbol{ks}}}_{set} \|_\infty^{can} \le B_{\overline{\overline{\boldsymbol{ks}}}_{set}}$ of the *switching key* into Lemma 3. The specific proof is omitted.

Lemma 6. Let $\boldsymbol{ct}' \leftarrow KS_{\overline{\overline{\boldsymbol{ks}}}_{refresh}}(\boldsymbol{ct})$. $\boldsymbol{ct} \in R_q^2$ corresponds to the secret key $\boldsymbol{sk}$. Let

$$\overline{\overline{\boldsymbol{ks}}}_{refresh} = \begin{cases} \boldsymbol{ks}_{s_i \to \overline{\overline{s}}} & \boldsymbol{sk} = (1, s_i) \\ \sum_{i=1}^{k'} \boldsymbol{ks}_{s_i \to \overline{\overline{s}}} & \boldsymbol{sk} = (1, s_1 + \ldots + s_{k'}) \end{cases},$$

for $e \in R$, there is $<\boldsymbol{ct}', (1, \overline{\overline{s}})>=<\boldsymbol{ct}, \boldsymbol{sk}> + e_{ks} (\bmod q)$, where $\|e_{ks}\|_\infty^{can} \le P^{-1} q \cdot \sqrt{k} B_{sclean} + B_{rs}$.

Lemma 6 can also be obtained by taking the upper bound $\| \overline{\overline{\boldsymbol{ks}}}_{refresh} \|_\infty^{can} \le \sqrt{k} B_{sclean}$ of the *switching key* into Lemma 4. The specific proof is omitted.

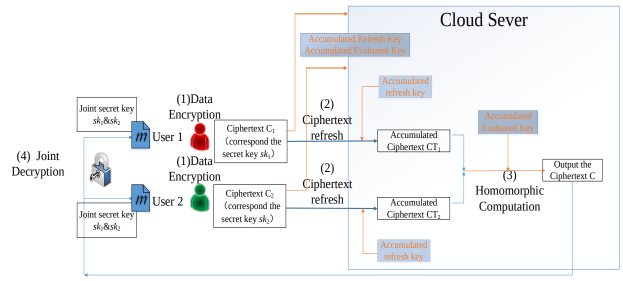

Figure 2: The homomorphic computation model of MKFHE with compact ciphertext

Figure 2 takes two parties as an example to introduce the steps of homomorphic operation.

Step 1. System initialization stage.

The two parties interact with the cloud twice to construct the public key. Parties publish their public key, obtain the accumulated public key from the cloud sever. They use the accumulated public key to generate their own accumulated evaluated key and refresh key, then upload them to the cloud sever. The cloud sever collects the accumulated calculation key and refresh key of all parties sets, and generates the accumulated evaluated key and accumulated refresh key.

Step 2. Data encryption.

The two parties use the public key or accumulated public key to encrypt the ciphertext, and upload the ciphertext to the cloud sever.

Step 3. Ciphertext refresh.

The cloud sever uses the accumulated refresh key to refresh parties' ciphertext.

Step 4. Homomorphic Computation.

The cloud sever uses the accumulated evaluated key to run homomorphic computation, and outputs ciphertext.

Step 5. Joint decryption.

The parties decrypt the ciphertext separately to get the final plaintext.

5 CONCLUSION

In this paper, firstly, we proposed a general construction of MKFHE scheme with compact ciphertext. Then, 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. Finally, we specifically propose the TFHE-type MKFHE and BGV-type MKFHE with compact ciphertext. The analysis shows that the ciphertext size of our schemes is independent of the number of parties, and the homomorphic computation efficiency is as high as the single-party FHE scheme.

ACKNOWLEDGMENTS

Insert paragraph text here.

REFERENCES

[1] Brakerski Z, Gentry C, Vaikuntanathan V. (Leveled) fully homomorphic encryption without bootstrapping[C]. Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. ACM, New York, 2012: 309–325.

[2] Doröz Y, Hu Y, Sunar B. Homomorphic AES evaluation using the modified LTV scheme[J]. Designs, Codes and Cryptography, 2016, 80(2): 333-358.

[3] Gentry C, Sahai A, Waters B. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based[C]. Advances in Cryptology—CRYPTO 2013. Springer, Berlin, Heidelberg, 2013: 75–92.

[4] Ducas L, Micciancio D. FHEW: Bootstrapping homomorphic encryption in less than a second[C]. Advances in Cryptology—EUROCRYPT 2015. Springer, Berlin, Heidelberg, 2015: 617–640.

[5] Alperin-Sheriff J, Peikert C. Faster bootstrapping with polynomial error[C]. In: Advances in Cryptology—CRYPTO 2014. Springer, Berlin, Heidelberg, 2014: 297–314.

[6] Chillotti I, Gama N, Georgieva M, et al. Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds[C]. International Conference on the Theory and Application of Cryptology and Information Security—ASIACRYPT 2016. Springer, Berlin, Heidelberg, 2016:3-33.

[7] Chillotti I, Gama N, Georgieva M, et al. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE[C]. International Conference on the Theory and Application of Cryptology and Information Security—ASIACRYPT 2017. Springer, Cham, 2017:377-408.

[8] Brakerski Z, Vaikuntanathan V. Fully homomorphic encryption from ring-LWE and security for key dependent messages[C]. In: Advances in Cryptology—CRYPTO 2011. Springer, Berlin, Heidelberg, 2011: 505–524.

[9] Brakerski Z, Vaikuntanathan V. Efficient fully homomorphic encryption from (standard) LWE[C]. IEEE 52nd Annual Symposium on Foundations of Computer Science—FOCS 2011. IEEE, 2011: 97–106.

[10] Brakerski Z. Fully homomorphic encryption without modulus switching from classical GapSVP[C]. In: Advances in Cryptology—CRYPTO 2012. Springer, Berlin, Heidelberg, 2012: 868–886.

[11] Gentry C, Halevi S, Smart N P. Fully homomorphic encryption with polylog overhead[C]. Advances in Cryptology—EUROCRYPT 2012. Springer, Berlin, Heidelberg, 2012: 465–482.

[12] Halevi S, Shoup V. Faster homomorphic linear transformations in HElib [EB/OL]. [2018-03-04]. https://eprint.iacr.org/2018/244.

[13] Cheon J H, Kim A , Kim M , et al. Homomorphic encryption for arithmetic of approximate numbers[C]. International Conference on the Theory and Application of Cryptology and Information Security—ASIACRYPT 2017. Springer, Cham, 2017:409-437.

[14] López-Alt A, Tromer E, Vaikuntanathan V. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption[C]. In Proceedings of the 44th annual ACM symposium on Theory of computing—STOC 2012. ACM, New York, 2012: 1219-1234.

[15] Hoffstein J, Pipher J, Silverman J H. NTRU: A ring-based public key cryptosystem[C]. International Algorithmic Number Theory Symposium. Springer, Berlin, Heidelberg, 1998: 267-288.

[16] Chongchitmate W, Ostrovsky R. Circuit-private Multi-Key fhe[C]. International Conference on Practice and Theory in Public Key Cryptography—PKC. Springer, Berlin, Heidelberg, 2017:241–270.

[17] Clear M, McGoldrick C. Multi-identity and Multi-Key leveled FHE from learning with errors[C]. Advances in Cryptology—CRYPTO 2015. Springer, Berlin, Heidelberg, 2015: 630-656.

[18] Mukherjee P, Wichs D. Two round multiparty computation via Multi-Key FHE[C]. Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 2016: 735-763.

[19] Peikert C, Shiehian S. Multi-Key FHE from LWE, revisited[C]. Theory of Cryptography Conference. Springer, Berlin, Heidelberg, 2016: 217-238.

[20] Brakerski Z, Perlman R. Lattice-based fully dynamic Multi-Key FHE with short ciphertexts[C]. Advances in Cryptology—CRYPTO 2016. Springer, Berlin, Heidelberg, 2016: 190-213.

[21] Chen L, Zhang Z, Wang X. Batched multi-hop Multi-Key FHE from ring-lwe with compact ciphertext extension[C]. Theory of Cryptography Conference. Springer, Cham, 2017: 597-627.

[22] Li N, Zhou T, Yang X, et al. Efficient Multi-Key FHE with short extended ciphertexts and directed decryption protocol[J]. IEEE Access, 2019(7): 56724-56732.

[23] Chen H, Dai W, Kim M, et al. Efficient Multi-Key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference [C]. Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019:395-412.

[24] Chillotti I, Gama N, Georgieva M, et al. Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds[C]. International Conference on the Theory and Application of Cryptology and Information Security—ASIACRYPT 2016. Springer, Berlin, Heidelberg, 2016:3-33.

[25] Chillotti I, Gama N, Georgieva M, et al. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE[C]. International Conference on the Theory and Application of Cryptology and Information Security—ASIACRYPT 2017. Springer, Cham, 2017:377-408.

[26] Chen H, Chillotti I, Song Y. Multi-Key homomorphic encryption from TFHE[C]. In: Advances in Cryptology – ASIACRYPT, 2019:446-472.

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.

LWE.Setup$(1^{\lambda}) \to pp^{\text{LWE}} = (n, \chi, \alpha, B_{ks}, d_{ks})$: Input the security parameters, generate the LWE dimension *n*, secret key distribution χ, error distribution parameter α, decomposition basis B_{ks}, decomposition degree d_{ks}, common matrix $\boldsymbol{B} \in \mathbb{T}^{m\times n}$ for all users, and output common parameter $pp^{\text{LWE}} = (n, \chi, \alpha, B_{ks}, d_{ks}, \boldsymbol{B})$.

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

LWE.Enc$(pk, m) \to \boldsymbol{ct} = (b, \boldsymbol{a}) \in \mathbb{T}^{n+1}$: Select $\boldsymbol{r} \in \mathbb{B}^m$ randomly, and calculate $(b, \boldsymbol{a}) := \boldsymbol{rA} + (\frac{1}{4}m, 0..., 0) + \boldsymbol{e}$.

LWE.KSGen$(\boldsymbol{t}, \overline{\overline{pk}}) \to \overline{\overline{\boldsymbol{KS}}}_j = [\boldsymbol{b}_j \mid \boldsymbol{A}_j] \in \mathbb{T}^{d_{ks}\times(n+1)}$: Input the LWE secret $\boldsymbol{t}_i \in \mathbb{Z}^N$, accumulated public key $\overline{\overline{pk}}$. $\overline{\overline{\boldsymbol{A}}} = [\overline{\overline{\boldsymbol{b}}} \mid \boldsymbol{B}] = [\boldsymbol{b}_1 + ... + \boldsymbol{b}_k \mid \boldsymbol{B}] \in \mathbb{T}^{m\times n}$ corresponds the secret key $\overline{\overline{\boldsymbol{s}}} = \boldsymbol{s}_1 + ... + \boldsymbol{s}_k$, where $\boldsymbol{s}_i$ is the secret key of party *i*. Output the switch key $\overline{\overline{\boldsymbol{KS}}}_i = \{\overline{\overline{\boldsymbol{KS}}}_{i,j}\}_{j\in[N]}$ from $\boldsymbol{t}_i \in \mathbb{Z}^N$ to $\overline{\overline{\boldsymbol{s}}}$, where $\overline{\overline{\boldsymbol{KS}}}_{i,j} = \boldsymbol{R}_{ks}\overline{\overline{\boldsymbol{A}}} + (\boldsymbol{e}_{i,j}, 0, ..., 0) + (t_{i,j}, 0, ..., 0) \cdot \boldsymbol{g}_{ks}$, $\boldsymbol{R}_{ks} \in \mathbb{Z}_q^{d_{ks}\times m}$, $\boldsymbol{g}_{ks} = (1/B_{ks}, 1/B_{ks}^2, ..., 1/B_{ks}^{d_{ks}})^T$, $j \in [N]$.

LWE.Switch$(\overline{\overline{\boldsymbol{ct}}} \in \mathbb{T}^{N+1}, \{\overline{\overline{\boldsymbol{KS}}}_{i,j}\}_{i\in[N], j\in[w]}) \to \overline{\overline{\boldsymbol{ct}'}} \in \mathbb{T}^{n+1}$: Input the accumulated ciphertext $\boldsymbol{ct} = (b, \boldsymbol{a}) \in \mathbb{T}^{N+1}$ and the switch key $\{\overline{\overline{\boldsymbol{KS}}}_{i,j}\}_{i\in[k], j\in[N]}$. Construct the accumulated switch key $\overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}}} = \{\sum_{i=1}^{k} \overline{\overline{\boldsymbol{KS}}}_{i,j}\}_{j\in[N]}$, where user $i \in [k]$ and dimension $j \in [N]$. Calculate $(b', \boldsymbol{a}') = \sum_{j=1}^{N} \boldsymbol{g}_{ks}^{-1}(a_j)\overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}}, j} (\text{mod} 1)$. Output the ciphertext $\overline{\overline{\boldsymbol{ct}'}} = (b' + b, \boldsymbol{a}') \in \mathbb{T}^{n+1}$.

Correctness. For $(b',\boldsymbol{a}')=\sum_{j=1}^{N}\sum_{i=1}^{k}\boldsymbol{g}_{ks}^{-1}(a_j)\overline{\overline{\boldsymbol{KS}}}_{i,j}$, we can obtain $<\overline{\overline{\boldsymbol{ct}'}},(1,\overline{\overline{\boldsymbol{s}}})>=<\overline{\overline{\boldsymbol{ct}}},(1,\overline{\overline{\boldsymbol{t}}})>+e_{ks}$, where $e_{ks}=\sum_{j=1}^{N}\overline{\overline{t}}_j e'_j+<\boldsymbol{g}_{ks}^{-1}(a_j),\sum_{i=1}^{k}\boldsymbol{R}_{ks}\overline{\overline{\boldsymbol{A}}}+(\boldsymbol{e}_{i,j},0,...,0)_j>$.

The correctness derivation process is shown in the following equation:

$$
\begin{aligned}
<\overline{\overline{\boldsymbol{ct}'}},(1,\overline{\overline{\boldsymbol{s}}})> &\overset{\overline{\overline{\boldsymbol{ct}'}}=(b'+b,\boldsymbol{a}')}{=} b+b'+\sum_{j=1}^{N}a'_j\overline{\overline{s}}_j \\
&=b+\sum_{j=1}^{N}<\boldsymbol{g}_{ks}^{-1}(a_j),\sum_{i=1}^{k}\boldsymbol{R}_{ks}\overline{\overline{\boldsymbol{A}}}+(\boldsymbol{e}_{i,j},0,...,0)+(t_{i,j},0,...,0)\cdot\boldsymbol{g}_{ks}> \\
&=b+\sum_{j=1}^{N}\begin{matrix}<\boldsymbol{g}_{ks}^{-1}(a_j),\sum_{i=1}^{k}\boldsymbol{R}_{ks}\overline{\overline{\boldsymbol{A}}}+(\boldsymbol{e}_{i,j},0,...,0)>+\\ <\boldsymbol{g}_{ks}^{-1}(a_j),\sum_{i=1}^{k}(t_{i,j},0,...,0)\cdot\boldsymbol{g}_{ks}>\end{matrix} \\
&=b+\sum_{j=1}^{N}<\boldsymbol{g}_{ks}^{-1}(a_j),\sum_{i=1}^{k}\boldsymbol{R}_{ks}\overline{\overline{\boldsymbol{A}}}+(\boldsymbol{e}_{i,j},0,...,0)+e'_{i,j}>+a_j\overline{\overline{t}}_j+\overline{\overline{t}}_j e'_j \\
&=<\overline{\overline{\boldsymbol{ct}}},(1,\overline{\overline{\boldsymbol{t}}})>+\sum_{j=1}^{N}\overline{\overline{t}}_j e'_j+<\boldsymbol{g}_{ks}^{-1}(a_j),\sum_{i=1}^{k}\boldsymbol{R}_{ks}\overline{\overline{\boldsymbol{A}}}+(\boldsymbol{e}_{i,j},0,...,0)_j>
\end{aligned}
$$

where

$$
\begin{aligned}
&(b',\boldsymbol{a}')=\sum_{j=1}^{N}\boldsymbol{g}_{ks}^{-1}(a_j)\overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}},j} \\
&\overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}}}=\{\sum_{i=1}^{k}\overline{\overline{\boldsymbol{KS}}}_{i,j}\}_{j\in[N]} \\
&\overline{\overline{\boldsymbol{KS}}}_{i,j}=\boldsymbol{R}_{ks}\overline{\overline{\boldsymbol{A}}}+(\boldsymbol{e}_{i,j},0,...,0)+(t_{i,j},0,...,0)\cdot\boldsymbol{g}_{ks}
\end{aligned}
.
$$

If $|e_{ks}|<\frac{1}{8}$, $\overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}}}=\{\sum_{i=1}^{N}\boldsymbol{KS}_{i,j}\}_{j\in[w]}$ can be seen as the valid switch key from $\overline{\overline{\boldsymbol{t}}}\in\mathbb{Z}^w$ to $\overline{\overline{\boldsymbol{s}}}\in\mathbb{Z}^n$.

B CKKS17 Scheme

CKKS.Setup$(1^\lambda)\to pp=(N,\chi_{key},\ \chi_{err},\ \chi_{enc},L,q_l)$: Input the parameters λ, and select an integer N to the power of 2. Let χ_{key}, χ_{err} and χ_{enc} be the distribution 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 $q_l=p^l$, where $1\le l\le L$. Output common parameter $pp=(N,\chi_{key},\ \chi_{err},\ \chi_{enc},L,P,q_l,a,a')$.

CKKS.KeyGen$(\boldsymbol{params})\to(\boldsymbol{pk},\boldsymbol{sk},\boldsymbol{ks},\boldsymbol{rk}_r,\boldsymbol{ck})$:

- CKKS.PSKeyGen$(\boldsymbol{params})\to(\boldsymbol{pk},\boldsymbol{sk})$: Select $s\leftarrow\chi_{key}$, and let the secret key $sk\leftarrow(1,\ s)$. Select $a\leftarrow U\left(R_{q_L}\right)$ and the error $e\leftarrow\chi_{err}$. Set the public key $pk\leftarrow(b,\ a)\in R_{q_L}^2$, where $b=-as\ +\ e\ \left(\bmod q_L\right)$.
- CKKS.KSGen(sk,s') : Input $s'\in R$, and select $a'\leftarrow R_{P\cdot q_L}$ and $e'\leftarrow\chi_{err}$ .let the evaluated key be $evk\leftarrow\left(b',\ a'\right)\in R_{P\cdot q_L}^2$, where $b'=-a's+e'+Ps'\ (\bmod P\cdot q_L)$.

Obtain the switch key $\boldsymbol{ks}\leftarrow$ CKKS.KSGen$_{\boldsymbol{sk}}(s^2)$;

Obtain the shift key $\boldsymbol{rk}_r\leftarrow$ CKKS.KSGen$_{\boldsymbol{sk}}(\kappa_{5^r}(s))$;

Obtain the conjugate key $\boldsymbol{ck}\leftarrow$ CKKS.KSGen$_{\boldsymbol{sk}}(\kappa_{-1}(s))$.

CKKS.Enc$_{pk,q}(m)$: Select $r\leftarrow\chi_{enc}$, $e_0,\ e_1\leftarrow\chi_{err}$ randomly. Output $\boldsymbol{ct}=r\cdot pk+\left(m+e_0,e_1\right)\ \left(\bmod q_L\right)$, such that $<\boldsymbol{ct},\boldsymbol{sk}>(mod\ q_L)\approx m$.

CKKS.Dec$_{sk}(\boldsymbol{ct})$: Input the ciphertext $\boldsymbol{ct}$ of the *l*-th level, and output the plaintext $m'=<\boldsymbol{ct},\boldsymbol{sk}>(\bmod q_l)$.

CKKS.Add$(\boldsymbol{ct},\boldsymbol{ct}')$: Input the ciphertexts $\boldsymbol{ct}$ and $\boldsymbol{ct}'$ of the *l*-th level, and output the ciphertext $\boldsymbol{ct}_{\text{add}}=\boldsymbol{ct}+\boldsymbol{ct}'(\bmod q_l)$.

CKKS.CMult$_{\boldsymbol{ks}}(a,\boldsymbol{ct})$: Input the constant $a\in R$ and the ciphertext $\boldsymbol{ct}$ of the l-th level. Output the ciphertext $\boldsymbol{ct}_{\text{cmult}}=a\cdot\boldsymbol{ct}(\bmod q_l)$.

CKKS.Mult$_{\boldsymbol{ks}}(\boldsymbol{ct},\boldsymbol{ct}')$: Input the ciphertext $\boldsymbol{ct}=(c_0,c_1),\ \boldsymbol{ct}'=(c'_0,c'_1)\in R_{q_l}^2$ of the *l*-th level, and output the ciphertext $\boldsymbol{ct}_{\text{mult}}=(d_0,d_1)+\lfloor P^{-1}\cdot d_2\cdot\boldsymbol{ks}\rceil(\bmod q_l)$.

KS$_{\boldsymbol{swk}}(\boldsymbol{ct})$: Input the evaluated key $\boldsymbol{swk}$ and the ciphertext $\boldsymbol{ct}$ of the *l*-th level. Output the ciphertext $\boldsymbol{ct}'\leftarrow\left(c_0,\ 0\right)\ +\ \lfloor P^{-1}\cdot c_1\cdot\boldsymbol{swk}\rceil\ (mod\ q_l)$.

CKKS.Rescale$_{l\to l'}\to(\boldsymbol{ct})$: Input the ciphertext $\boldsymbol{ct}$ of the *l*-th level and the next level label l'. Output the ciphertext $\boldsymbol{ct}'=\lfloor p^{l'-l}\cdot\boldsymbol{ct}\rceil\in(\bmod q_{l'})$.

CKKS.Bootstrapping$_{\boldsymbol{ks,rk,ck}}(\boldsymbol{c})$: Input the evaluated key $\boldsymbol{ks,rk,ck}$ and ciphertext $\boldsymbol{c}$. Output the refreshed ciphertext $\boldsymbol{c}'$. See the CHKKS and CCS18 schemes for the detail bootstrapping process.

CKKS.Rotate$_{\boldsymbol{rk}}(\boldsymbol{ct};k)$: Input the shift key $\boldsymbol{rk}$ and the ciphertext $\boldsymbol{ct}$. If the plaintext vector $m(Y)$ moves k bits, then output the ciphertext of $m(Y^{5^k})$.

CKKS.Conjugate$_{\boldsymbol{ck}}(\boldsymbol{ct})$: Input the shift key $\boldsymbol{ck}$ and the ciphertext $\boldsymbol{ct}$. If the plaintext vector $m(Y)$ is conjugated to a vector $m(Y^{-1})$, then output the ciphertext of $m(Y^{-1})$.

- GSW.PSKeyGen$(\boldsymbol{params})\to(\boldsymbol{pk},\boldsymbol{sk})$: Select $s\leftarrow\chi_{key}$, and set the secret key $sk\leftarrow(1,\ s)$. Select

$\boldsymbol{a} \xleftarrow{U} R_{P\cdot q_L}^{2d}$ and $\boldsymbol{e} \leftarrow \chi_{err}^{2\beta}$, output the public key $pk := [\boldsymbol{b} = \boldsymbol{a}z + \boldsymbol{e}, \boldsymbol{a}] \in R_{P\cdot q_L}^{2d\times 2}$, where

$$pk := [\boldsymbol{b} = \boldsymbol{a}z + \boldsymbol{e}, \boldsymbol{a}] = \begin{bmatrix} \boldsymbol{b}[1] & \boldsymbol{a}[1] \\ \vdots & \vdots \\ \boldsymbol{b}[2d] & \boldsymbol{a}[2d] \end{bmatrix} \in R_{P\cdot q_L}^{2d\times 2}.$$

GSW.Enc$_{pk}(\mu)$: Select $r \leftarrow \chi_{enc}$ and $\boldsymbol{E} := [\boldsymbol{e}_0 \mid \boldsymbol{e}_1] \leftarrow \chi_{err}^{2\beta\times 2}$. Output the ciphertext $\boldsymbol{C} = r[\boldsymbol{b}, \boldsymbol{a}] + p\boldsymbol{E} + \boldsymbol{s}\boldsymbol{G} \in R_{p\cdot q_L}^{2\beta_l \times 2}$, where $pk := [\boldsymbol{b} = \boldsymbol{a}z + \boldsymbol{e}, \boldsymbol{a}] \in R_{P\cdot q_L}^{2d\times 2}$. That is

$$\text{GSW.Enc}(\mu)_s = \begin{bmatrix} r\boldsymbol{b}[1] + p\boldsymbol{e}_1[1] + \mu & r\boldsymbol{a}[1] + p\boldsymbol{e}_2[1] \\ r\boldsymbol{b}[2] + p\boldsymbol{e}_1[2] & r\boldsymbol{a}[2] + p\boldsymbol{e}_2[2] + \mu \\ \ddots & \ddots \\ r\boldsymbol{b}[2\beta-1] + p\boldsymbol{e}_1[2\beta-1] + 2^{\lfloor \log q \rfloor}\mu & r\boldsymbol{a}[2\beta-1] + p\boldsymbol{e}_2[2\beta-1] \\ r\boldsymbol{b}[2\beta] + p\boldsymbol{e}_1[2\beta] & r\boldsymbol{a}[2\beta] + p\boldsymbol{e}_2[2\beta] + \mu 2^{\lfloor \log q \rfloor} \end{bmatrix}$$

.

B1 Proof of Lemma 2

Proof. Define $\overset{=}{e} = e_1 + \ldots + e_k$, $\overset{=}{s} = s_1 + \ldots + s_k$, and $\overset{=}{b} = b_1 + \ldots + b_k$. For the ciphertext $\boldsymbol{ct} = r\cdot(\overset{=}{b}, a) + (m + e_0', e_1') \pmod q$, where $b_i = -as_i + e_i \pmod{q_L}$. Select $r \leftarrow ZO(0.5)$, $e_0', e_1' \leftarrow DG_q(\sigma^2)$, $s_i \leftarrow HWT(h)$, $a \leftarrow \mathcal{U}_{q_L}$ and $e_i \leftarrow DG_{q_L}(\sigma^2)$, then calculate the expression of error e and the upper limit of $\|e\|_\infty^{can}$.

$$e =< \boldsymbol{ct}, (1, \overset{=}{s}) > -m =< r\cdot(\overset{=}{b}, a) + (m + e_0', e_1'), (1, \overset{=}{s}) > -m$$

$$=< r\cdot(\overset{=}{b}, a), (1, \overset{=}{s}) > + < (e_0', e_1'), (1, \overset{=}{s}) >$$

$$\overset{b_i = -as_i + e_i \pmod{q_L}}{=} \quad r\overset{=}{e} + e_0 + e_1\overset{=}{s}$$

$$\|e\|_\infty^{can} \overset{e = r\overset{=}{e} + e_0 + e_1\overset{=}{s}}{=} \|r\overset{=}{e} + e_0' + e_1'\overset{=}{s}\|_\infty^{can}$$

$$\le \|r\overset{=}{e}\|_\infty^{can} + \|e_0'\|_\infty^{can} + \|e_1'\overset{=}{s}\|_\infty^{can}.$$

$$\le 8\sqrt{2k}\sigma N + 6\sigma\sqrt{N} + 16\sigma k\sqrt{hN}$$

B2. Proof of Lemma 3.

Proof: （1）calculate the error of $e_{\overline{\overline{ks}}_{se,i}}$.

$$e_{\overline{\overline{ks}}_{se,i}} = \boldsymbol{c}_{\overline{\overline{ks}}_{se,i}} \begin{bmatrix} 1 \\ \overset{=}{s} \end{bmatrix} - P\cdot s_i \overset{=}{s}$$

$$= \left[0 \mid \overline{\overline{d}}_{i,2}\right] \begin{bmatrix} 1 \\ \overset{=}{s} \end{bmatrix} + \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot \left[\overline{\overline{\boldsymbol{d}}}_{i,0} \mid \overline{\overline{\boldsymbol{d}}}_{i,1}\right]_{d\times 2} \begin{bmatrix} 1 \\ \overset{=}{s} \end{bmatrix} - P\cdot s_i \overset{=}{s}$$

$$= r_i \overset{=}{e'} + e_{i,2}\cdot\overset{=}{s} + \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot \left[\boldsymbol{e}_{sclean}\right]_{d\times 1} \pmod{Pq_L}$$

where

$$\overline{\overline{\boldsymbol{ks}}}_{se,i} := \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot \left[\overline{\overline{\boldsymbol{d}}}_{i,0} \mid \overline{\overline{\boldsymbol{d}}}_{i,1}\right]_{d\times 2} + \left[0 \mid \overline{\overline{d}}_{i,2}\right] \in \text{CKKS}_{\overset{=}{s}}(P^{-1}\cdot \mu_i \overset{=}{s})$$

$$a. \left[0 \mid \overline{\overline{d}}_{i,2}\right] \begin{bmatrix} 1 \\ \overset{=}{s} \end{bmatrix} = -r_i\overset{=}{b'} + r_i\overset{=}{e'} + e_{i,2}\cdot\overset{=}{s} + P\cdot s_i\overset{=}{s} \pmod{Pq_L}.$$

$$b. \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot \left[\overline{\overline{\boldsymbol{d}}}_{i,0} \mid \overline{\overline{\boldsymbol{d}}}_{i,1}\right]_{d\times 2} \begin{bmatrix} 1 \\ \overset{=}{s} \end{bmatrix} = \overset{=}{b'} r_i + \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot \left[\boldsymbol{e}_{sclean}\right]_{d\times 1}$$

$$a. \left[0 \mid \overline{\overline{d}}_{i,2}\right] \begin{bmatrix} 1 \\ \overset{=}{s} \end{bmatrix} = \overline{\overline{d}}_{i,2}\cdot\overset{=}{s}$$

$$\overset{\overline{\overline{d}}_2 = r\cdot a' + e_2 + P\cdot s \pmod{Pq_L}}{=} \quad r\cdot a'\cdot\overset{=}{s} + e_{i,2}\cdot\overset{=}{s} + P\cdot s_i\overset{=}{s} \pmod{Pq_L}$$

$$\overset{r_i\overset{=}{b'} = -r_i\overset{=}{s}\cdot a' + r_i\overset{=}{e'} \in R^1_{P\cdot q_L}}{=} \quad -r_i\overset{=}{b'} + r_i\overset{=}{e'} + e_{i,2}\cdot\overset{=}{s} + P\cdot s_i\overset{=}{s} \pmod{Pq_L}$$

$$b. \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot \left[\overline{\overline{\boldsymbol{d}}}_{i,0} \mid \overline{\overline{\boldsymbol{d}}}_{i,1}\right]_{d\times 2} \begin{bmatrix} 1 \\ \overset{=}{s} \end{bmatrix}$$

$$\overset{\substack{(\overline{\overline{d}}_{0,[,j]}, \overline{\overline{d}}_{1[,j]}) \leftarrow \text{CKKS.Enc}_{\overline{\overline{pk}}[,j], Pq_L}(r\cdot \boldsymbol{g}_{[,j]}) \\ \boldsymbol{e}_{sclean} = (\boldsymbol{e}_{sclean[,j]})_{j\in[d]}}}{=} \quad \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot (r_i\cdot\boldsymbol{g} + \boldsymbol{e}_{sclean})$$

$$= \overset{=}{b'} r_i + \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot \left[\boldsymbol{e}_{sclean}\right]_{d\times 1}$$

（2）Calculate the upper limit of $\|e_{\overline{\overline{ks}}_{se,i}}\|_\infty^{can}$.

$$\|e_{\overline{\overline{ks}}_{se,i}}\|_\infty^{can} = \|r_i\overset{=}{e'} + e_{i,2}\cdot\overset{=}{s} + \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot \left[\boldsymbol{e}_{sclean}\right]_{d\times 1} \pmod{Pq_L}\|_\infty^{can}$$

$$\le 8\sqrt{2k}\sigma N + 16\sigma k\sqrt{hN} + 8B_{ks}B_{sclean}\sqrt{dN/3}$$

$$e_{\overline{\overline{ks}}_{se,i}} = r_i\overset{=}{e'} + e_{i,2}\cdot\overset{=}{s} + \left[\boldsymbol{g}^{-1}(\overset{=}{b'})\right]_{1\times d} \cdot \left[\boldsymbol{e}_{sclean}\right]_{d\times 1} \pmod{Pq_L}$$

where

$$e_{\overline{\overline{\boldsymbol{ks}}}_{se,i}} = r_i\overline{\overline{e'}} + e_{i,2}\cdot\overline{\overline{s}} + \left[\boldsymbol{g}^{-1}(\overline{\overline{b'}})\right]_{1\times d}\cdot\left[\boldsymbol{e}_{sclean}\right]_{d\times 1} (\bmod Pq_L)$$
$$\|r_i\overline{\overline{e'}}\|_\infty^{can} \le 8\sqrt{2k}\sigma N$$
$$\|e_{i,2}\cdot\overline{\overline{s}}\|_\infty^{can} \le 16\sigma k\sqrt{hN}$$
$$\|\left[\boldsymbol{g}^{-1}(\overline{\overline{b'}})\right]_{1\times d}\cdot\left[\boldsymbol{e}_{sclean}\right]_{d\times 1}\|_\infty^{can} \le 8B_{ks}B_{sclean}\sqrt{N/3}$$
$$\|\left[\boldsymbol{g}^{-1}(\overline{\overline{b'}})\right]_{1\times d}\cdot\boldsymbol{e}_{clean}\|_\infty^{can} \le \|\sum_{i=1}^{d}\boldsymbol{g}^{-1}(\overline{\overline{b'}})_{[i,]}\boldsymbol{e}_{clean,[,i]}\|_\infty^{can}$$
$$\le\sqrt{d}\,\|\boldsymbol{g}^{-1}(\overline{\overline{b'}})_{[i,]}\boldsymbol{e}_{clean,[,i]}\|_\infty^{can}$$
$$\le 8B_{ks}B_{sclean}\sqrt{dN/3}$$
where
$$\|\left[\boldsymbol{g}^{-1}(\overline{\overline{b'}})\right]_{1\times d}\cdot\left[\boldsymbol{e}_{sclean}\right]_{d\times 1}\|_\infty^{can} \le 8B_{ks}B_{sclean}\sqrt{N/3}$$

（3）Calculate the $e_{\overline{\overline{\boldsymbol{ks}}}_{set}}$, and the upper limit of $\|e_{\overline{\overline{\boldsymbol{ks}}}_{set}}\|_\infty^{can}$.

$$\overline{\overline{\boldsymbol{ks}}}_{set}\begin{bmatrix}1\\ \overline{\overline{s}}\end{bmatrix} \overset{\overline{\overline{\boldsymbol{ks}}}_{set}:=\sum_{i=1}^{k}\overline{\overline{\boldsymbol{ks}}}_{se,i}}{=} \sum_{i=1}^{k}\overline{\overline{\boldsymbol{ks}}}_{se,i}\begin{bmatrix}1\\ \overline{\overline{s}}\end{bmatrix},$$
$$=\sum_{i=1}^{k}P\cdot s_i\overline{\overline{s}} + e_{\overline{\overline{\boldsymbol{ks}}}_{se,i}} = P\overline{\overline{s}}\cdot\overline{\overline{s}} + \sum_{i=1}^{k}e_{\overline{\overline{\boldsymbol{ks}}}_{se,i}}$$
$$\|e_{\overline{\overline{\boldsymbol{ks}}}_{set}}\|_\infty^{can} = \|\sum_{i=1}^{k}e_{\overline{\overline{\boldsymbol{ks}}}_{se,i}}\|_\infty^{can} \le \sqrt{k}\|e_{\overline{\overline{\boldsymbol{ks}}}_{se,i}}\|_\infty^{can}.$$

C Homomorphic adder

Homomorphic adder is constructed by homomorphic addition and homomorphic multiplication of TGSW ciphertext

C.1 Mathematical expression of adder

C.1.1 Half-Adder：

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

Output：

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

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

C.1.2 Full-Adder(x,y,c)：

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

Output：

-Carry:

$$\mathrm{TGSW}(c_{out}) = \mathrm{TGSW}(x)\boxtimes\mathrm{TGSW}(y) + \mathrm{TGSW}(c_{in})\boxtimes\{\mathrm{TGSW}(x)+\mathrm{TGSW}(y)\}.$$

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

-Sum:

$$\mathrm{TGSW}(c_{out}) = \mathrm{TGSW}(x)+\mathrm{TGSW}(y)+\mathrm{TGSW}(c_{in}).$$

The corresponding plaintext is $c_{out} = x+y+c_{in}$.

Homomorphic addition algorithm for two l-bit TGSW ciphertexts HomAdd:

Input two groups of TGSW ciphertext $\{\mathrm{TGSW}(x_{l-1}),...,\mathrm{TGSW}(x_0)\}$ and $\{\mathrm{TGSW}(y_{l-1}),...,\mathrm{TGSW}(y_0)\}$ 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 $l-1$,

$$\begin{aligned}&(1)\text{:}\{\mathrm{TGSW}(c),\mathrm{TGSW}(s_0)\} = \\ &\mathrm{FullAdd}(\mathrm{TGSW}(x_0),\mathrm{TGSW}(y_0),\boldsymbol{0})\\ &(2)\text{:}\{\mathrm{TGSW}(c),\mathrm{TGSW}(s_1)\} = \\ &\mathrm{FullAdd}(\mathrm{TGSW}(x_1),\mathrm{TGSW}(y_1),\mathrm{TGSW}(c))\\ &\dots\\ &(l-1)\text{:}\{\mathrm{TGSW}(c),\mathrm{TGSW}(s_{l-1})\} = \\ &\mathrm{FullAdd}(\mathrm{TGSW}(x_{l-1}),\mathrm{TGSW}(y_{l-1}),\mathrm{TGSW}(c))\end{aligned}.$$

Output the ciphertext $\{\mathrm{TGSW}(c),\mathrm{TGSW}(s_{l-1}),...,\mathrm{TGSW}(s_0)\}$.

For the homomorphic addition $\mathrm{HomAddk}\{\mathrm{TGSW}(x_k),...,\mathrm{TGSW}(x_0)\}$ of k l -bit TGSW ciphertexts, we use HomAdd algorithm and binary tree to realize fast calculation.

C.2 Error analysis of adder

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

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

$$\begin{aligned}&Var(Err(A\boxtimes B))\le(k+1)lN\beta^2 Var(Err(A)) + \\ &(1+kN)(\mu_A\varepsilon)^2 + \mu_A^2 Var(Err(B))\\ &= 2dNV_B Var(Err(A)) + (1+N)\varepsilon^2 + Var(Err(B))\end{aligned}$$

where $k=1, l=d, \beta=\frac{B_g}{2}$,$V_B=\beta^2, \mu_A=\{0,1\}$,and ε^2 is the var of gap round.

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

$$\begin{aligned}&Var(Err(\overline{S}))\le Var(Err(\overline{X})) + Var(Err(\overline{Y}))\\ &+Var(Err(\overline{C_{in}})) = 4dkN\beta^2 + Var(Err(c_{in}))\end{aligned},$$

$Var(Err(\overline{C_{out}})) \le (6dNV_B+1)Var(Err(\overline{X}))+$
$2(1+N)\varepsilon^2+Var(Err(\overline{C_{in}}))$.
$=(6dNV_B+1)2dkN\beta^2+2(1+N)\varepsilon^2+Var(Err(\overline{C_{in}}))$

(3) The l-bit HomAdd algorithm is formed by continuously running l full-adders, the output has an error of variance：

$Var(Err(c)) \le l(6dNV_B+1)\sqrt{2dkN}\beta^2+2l(1+N)\varepsilon^2$,

$Var(Err(s_i)) \le 2\beta+(l-1)\cdot\{(6dNV_B+1)\sqrt{2dkN}\beta^2+$
$2(1+N)\varepsilon^2\}$
$\le l(6dNV_B+1)\sqrt{2dkN}\beta^2+2l(1+N)\varepsilon^2$

For convenience,

$Var(Err(\text{HomAdd}_l(output))) \le l(6dNV_B+1)\sqrt{2dkN}\beta^2+$
$2l(1+N)\varepsilon^2$

HomAddk algorithm can realize homomorphic addition of *k* *l*-bit TGSW ciphertexts. There are two methods as bellow.

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 $\sum_{i=1}^{l}2^{i-1}i$ times, the calculation speed is slow, but the error growth is small. The error variance of its output is

$$Var(Err(\text{HomAddk}(output))) \le \sum_{i=1}^{l}(2^{i-1}i)(2dNV_B\cdot 2dkN\beta^2+(1+N)\varepsilon^2)+2dkN\beta^2 .$$

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 $l=\lceil\log(k)\rceil$ times, so the calculation speed is fast, but the error growth is also large. Since the length of HomAdd addition is from 1 to $l=\lceil\log(k)\rceil$, the error variance of the output is

$$Var(Err(\text{HomAddk}(output))) \le (l!)(6dNV_B+1)^l\cdot 2dkN\beta^2 .$$

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 $\varepsilon^2=1/(12B^{2d})$ 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 ε_{ks}^2 , $V_{B_{ks}}$ and B_{ks} in the bootstrapping algorithm. Define the secret key distribution $\chi\in\{0,1\}^w$ and $\varphi\in\{0,1\}^n$ on RGSW and LWE. Let $\mathrm{Var}(e)$ be the variance of the random variable e over $\mathbb{R}$. If e is a vector composed of random variables, $\mathrm{Var}(\boldsymbol{e})$ is the maximum variance of the vector.

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

The initial error of the evaluation key.

The variance of error $\overline{\overline{\boldsymbol{KS}_{i,j}}}$ is

$$Var(Err(\overline{\overline{\boldsymbol{KS}_{i,j}}})) \overset{\overline{\overline{\boldsymbol{KS}_i}}=\text{LWE.KSGen}(\boldsymbol{t}_i,\overline{\overline{pk}})}{=} Var(Err(\boldsymbol{R}_{ks}\overline{\overline{\boldsymbol{A}}}+(t_{i,j},0,...,0)\cdot\boldsymbol{g}_{ks})) \le mk\cdot\alpha^2 .$$

The variance of error $\overline{\overline{\boldsymbol{BK}_{i,j}}}$ is

$$Var(Err(\overline{\overline{\boldsymbol{BK}_{i,j}}})) \overset{\overline{\overline{\boldsymbol{BK}_{i,j}}}=\text{RGSW.Enc}(s_{i,j},\overline{\overline{\boldsymbol{Z}}})}{=} Var(Err(\boldsymbol{R}\cdot\overline{\overline{\boldsymbol{Z}}}+s_{i,j}\boldsymbol{h})\le 2dkN\cdot\beta^2 .$$

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

令 Let $\boldsymbol{d}_0,\boldsymbol{d}_1$ be TRLWE samples and let $C\in\text{TGSW}_s(\{0,1\})$. Then, $msg(\text{CMux}(C,\boldsymbol{d}_1,\boldsymbol{d}_0))=msg(C)\cdot msg(\boldsymbol{d}_1)\cdot msg(\boldsymbol{d}_0)$. And we have

$$\|Err\left(\text{CMux}(C,\boldsymbol{d}_1,\boldsymbol{d}_0)\right)\|_\infty \le \max(\|Err(\boldsymbol{d}_0)\|_\infty,\|\text{Err}(\boldsymbol{d}_1)\|_\infty)+\eta(\boldsymbol{C}) ,$$

where $\eta(C) = 2dN\frac{B_g}{2}\|Err(C)\|_\infty + (k+1)\varepsilon$. So

$$Var\left(Err\left(\text{CMux}(C,\boldsymbol{d}_1,\boldsymbol{d}_0)\right)\right)\le \max\left(Var\left(Err\left(\boldsymbol{d}_0\right)\right),\ Var\left(Err\left(\boldsymbol{d}_1\right)\right)\right)+\vartheta(C) ,$$

where $\vartheta(C) = 2dNV_B Var\left(Err\left(C\right)\right)+ (N+1)\varepsilon^2$.

The accumulated process. The initial RLWE ciphertext is general, and its error is 0. All bootstrap keys $\{\overline{\overline{\boldsymbol{BK}_{bit_{l-1}(\overline{\overline{Set}},j)}}},...,\overline{\overline{\boldsymbol{BK}_{bit_0(\overline{\overline{Set}},j)}}}\}_{j\in[n]}$ are generated by HomAdd algorithm, and the variance of error is $\sum_{i=1}^{l}(2^{i-1}i)(2dNV_B\cdot 2dkN\beta^2+(1+N)\varepsilon^2)+2dkN\beta^2$. By recursively running Cmux circuit for $l\cdot n$ times, the error variance of accumulated process is

$$2dNV_B\cdot ln\{\sum_{i=1}^{l}(2^{i-1}i)(2dNV_B\cdot 2dkN\beta^2+(1+N)\varepsilon^2)+2dkN\beta^2\}+ln(N+1)\varepsilon^2 .$$

The key switching algorithm. Input accumulated ciphertext $\boldsymbol{ct}=(b,\boldsymbol{a})\in\mathbb{T}^{N+1}$ and accumulated key $\overline{\overline{\boldsymbol{KS}_{\overline{\overline{Set}}}}}=\{\sum_{i=1}^{k}\overline{\overline{\boldsymbol{KS}_{i,j}}}\}_{j\in[N]}$, where

$\overline{\overline{\boldsymbol{KS}}}_{i,j} = \boldsymbol{R}_{ks}\overline{\overline{\boldsymbol{A}}} + (\boldsymbol{e}_{i,j}, 0, ..., 0) + (t_{i,j}, 0, ..., 0) \cdot \boldsymbol{g}_{ks}$. Output the ciphertext $(b', \boldsymbol{a}') = \sum_{j=1}^{N} \boldsymbol{g}_{ks}^{-1}(a_j)\overline{\overline{\boldsymbol{KS}}}_{\overline{\overline{Set}},j} (\bmod 1)$ and its error variance

$$Var(Err(\overline{\overline{\boldsymbol{ct}'}})) = \tfrac{1}{2}\varepsilon^2 N + d_{ks}V_{B_{ks}}N\alpha^2(1+m) + Var(Err(\boldsymbol{ct})).$$

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

$$\tfrac{1}{2}\varepsilon^2 N + d_{ks}V_{B_{ks}}N\alpha^2(1+m) + 2dNV_B \cdot ln\{\sum_{i=1}^{l}(2^{i-1}i)(2dNV_B \cdot 2dkN\beta^2 + (1+N)\varepsilon^2) + 2dkN\beta^2\} + (N + 1)ln\varepsilon^2.$$