Multi-key Fully Homomorphic Encryption Scheme with Compact Ciphertexts

Tanping Zhou[1,2], Long Chen[2], and Xiaoliang Che[1] Wenchao Liu[1] Zhenfeng Zhang[2] Xiaoyuan Yang[1]

[1] College of Cryptographic Engineering, Engineering University of PAP, Xi'an 710086, China
[2] Institute of Software, Chinese Academy of Sciences, Beijing 100080, China

Abstract. Multi-Key fully homomorphic encryption (MKFHE) allows computations on data encrypted by different parties. One disadvantage of previous MKFHE schemes is that the ciphertext size increases linearly or squarely with respect to the number of parties. It incurs a heavy communication and computation burden for the homomorphic evaluation, especially when the number of involved parties is large. In this paper, we propose the first method to construct MKFHE scheme while keeping the size of the ciphertext and corresponding evaluation key to be independent of the number of parties during the homomorphic evaluation. Specifically, we construct efficient compact MKFHE schemes with various advantages. On the one hand, we show how to construct compact MKFHE schemes which support the homomorphic encryption of ring elements and are friendly to floating point numbers. On the other hand, we give a compact MKFHE scheme that supports high efficient bootstrapping. In our paper, we show a novel method to reduce the cost of generating these evaluation keys from a quadratic time to a linear time with respect to the number of parties.

Keywords: Multi-key fully homomorphic encryption · Compact ciphertext · Compact evaluation key.

1 INTRODUCTION

Fully homomorphic encryption (FHE) is a cryptographic scheme that enables homomorphic operations on encrypted data without decryption [21,7,19]. Traditional FHE schemes can only support homomorphic computation of ciphertext from a single party, i.e., all ciphertexts participating in the computation are encrypted by one secret key. However, in many scenarios, it is usually required to calculate the data uploaded to the cloud by multiple parties in the network. In 2012, López-Alt et al. [24] 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 multi-party computation (MPC) protocols, where the circuit to be evaluated can be dynamically decided after

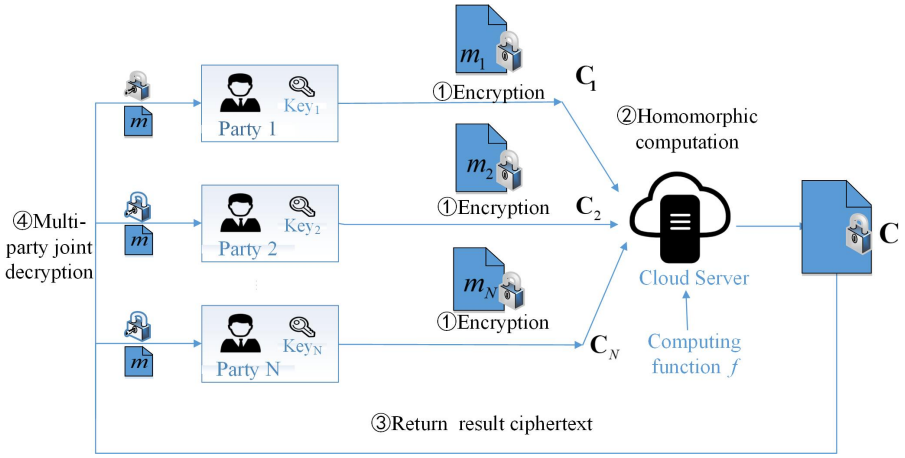

Fig. 1. multi-party data security computing model of MKFHE

the data providers upload their encrypted data. The typical application model of MKFHE is shown in Figure 1.

The original MKFHE proposed by López-Alt et al. is based a variant of NTRU assumption [24], which may be valuable to potential attacks [1]. Recently, a series of MKFHE constructions based on LWE [24,17,16,25,26,8,9] and its ring variant [11,10] has been proposed. However, a crucial issue for these LWE/Ring-LWE based MKFHE schemes is that the size of the ciphertext increase linearly with respect to the number of parties. Therefore, when we consider the application scenarios such that data are collected from a large group of parties, the communication and the homomorphic operations will be far from practical. To handle this practical issue, we are trying to design Ring-LWE based MKFHE schemes with compact ciphertexts.

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[24], which was optimized later in DHS16[17]. In PKC2017, Chongchitmate et al. proposed a general transformation framework CO17[15] 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 CRYPTO15, Clear and McGoldrick proposed the first GSW-type MKFHE scheme CM15 based on the LWE problem[16], which proposed 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 (R)LWE problems. In EUROCRYPT16, Mukherjee and Wichs presented the construction of MKFHE MW16[25] based on LWE that simplifies the scheme of CM15 and admited a simple one-round threshold decryption protocol. Based on this threshold MKFHE(ThMKFHE), 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[26]. In TCC16, Peikert and Shiehian proposed a notion of multi-hop MKFHE PS16[26], 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 CRYPTO16, A similar notion named fully dynamic MKFHE BP16[8] 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, the scheme needs to use the parties' joint public key to run the bootstrapping process, in the process of homomorphic computation, so the efficiency of ciphertext computation is low.

In TCC17, Chen et al. proposed the first BGV-type multi-hop MKFHE scheme CZW17[11]. 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 two-round MPC protocol and threshold decryption protocol. In 2019, Li et al. put forward a nested ciphertext extension method LZY+19[23], 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 [10]. Because of its efficient homomorphic computation, it is applied to the neural network to perform the privacy computation. In ASIACRYPT19, Chen et al. designed an efficient ciphertext expansion algorithm based on CGGI17[14], which is an FHE that has efficient bootstrapping, and proposed an MKFHE scheme CCS19[9]. 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.

Except for the MKFHE, threshold FHE(ThFHE) also involves many parties. The difference lies in that: in threshold FHE, all ciphertexts are encrypted in the same public key(joint public key); while in MKFHE, the ciphertexts are encrypted in different public keys. In threshold FHE, the joint public key is always the accumulation of the users' public key, which induce the compact ciphertext. While the difficulty of ThFHE lies in the generation of the evaluation key for bootstrapping or homomorphic multiplication. In TCC12, Asharov et

al. first proposed the concept of threshold FHE and constructed an N-out-of-N threshold FHE scheme in AJL+12[2]. Based on a common reference string(CRS), AJL+12 scheme uses the idea of public key accumulation to generate the joint public key, and uses secret sharing to decompose the secret key. One of their main contributions lies in the interactive generation of evaluation key using a key homomorphism symmetric encryption scheme. While AJL+12 scheme has some defects: It is incompatible with single-key homomorphic encryption since each user needs to use the joint public key for encryption; the amount of data broadcast by each user is linearly related to the number of users; when the user set is updated, the ciphertext needs to be regenerated (the ciphertext cannot be reused). At CRYPTO15, Gordon et al. designed a three-round MPC protocol that can resist abort attackers, and constructed the (N/2+1) out-of-N Threshold FHE scheme in GLS15[18]. One of their main contributions lies in the efficient generation of joint GSW ciphertext. The defects of GLS15 are: their scheme is based on the GSW13 scheme, which has the natural defect of a large ciphertext scale; moreover, the ciphertext scale of GLS15 is linearly related to the number of users. At CRYPTO18, Boneh et al. designed a threshold FHE BGG+18[4] with a center based on the linear secret sharing scheme, which reduced the noise of joint decryption in the TFHE scheme. Then, the decentralized TFHE is proposed, by taking the cryptographer as the center and sharing the private key, then generating a temporary joint public key and private key. All the other users in the scheme need to use the temporary public key for encryption. The defects of the scheme are: the size of the ciphertext is related to the number of users; it is difficult to select a suitable user to act as the center when there are multiple data encryptors. In ASIACRYPT20, Badrinarayanan et al.[3] followed the idea of the BGG+18 scheme and constructed a threshold MKFHE scheme. The size of the ciphertext is related to the number of users, since the scheme uses a cascaded method to generate the ciphertext. Very recently, Lee et al.[22] designed a new blind rotation using ring automorphisms, and constructed efficient FHEW-type FHE support for arbitrary secret key distributions at no additional runtime costs. Lee et al. applicate the FHE to threshold homomorphic encryption and generate the joint evaluation key by encrypting the secret key with the joint public key, the method is similar to ours. The drawback of the scheme is that it does not analyze how to update the ciphertext when the user set is updated.

1.2 Our Contributions

In this paper, we propose constructions of MKFHE scheme with compact ciphertexts. The sizes of the ciphertexts, as well as the size of the evaluation key, are independent of the number of parties. Our core technique is a methodology to obtain the joint ciphertexts from the ciphertexts encrypted with a single key, as well as the corresponding evaluation keys for the joint ciphertext.

Moreover, we studied how to optimize the ciphertext extension procedure according to different underlying FHE schemes. On the one hand, we adapt our ciphertext extension technique to the CKKS scheme[12] and obtain an MKFHE

scheme that can deal with the approximate number. In the meantime, we successfully reduce the cost of generating the evaluation key from a quadratic time for the naive approach to a linear time with respect to the number of parties. On the other hand, we adapt our ciphertext extension technique to the TFHE scheme [13,14] and obtain an MKFHE with fast bootstrapping.

1.3 Technique Overview

To make the size of ciphertext to be independent of the number of parties, we need to solve problems.

First of all, we need to perform homomorphic evaluation between the ciphertexts corresponding to two different public keys. In the previous MKFHE schemes, the ciphertexts encrypted under one secret key need to be extended to the ones encrypted under joint public keys. Particularly, this kind of extension must be proceeded by the semi-honest evaluator who only knows the ciphertext and the public key. Since all parties have not coordinated before, it is extremely hard to make the ciphertext compact.

Our intuition is nature: We let the evaluator first generate the joint public key from all the public keys of the users participating the evaluation by accumulating them. Secondly, the user uses the joint public key to encrypt his private key to generate a partial evaluation key. Finally, the evaluator uses the key switching technology to convert the ciphertexts corresponding to different public keys to the joint ciphertext corresponding to the user set, and then perform the homomorphic operation on the joint ciphertext.

However, the above final procedure is not trivial, since the homomorphic operation needs the evaluation key. Intuitively and informally, the evaluation key of MKFHE can be viewed as the encryption of the joint secret key with the joint public key. But in our setting, no party alone holds the joint secret key, so it is doomed to require more complex techniques to deal with the evaluation key generation. In this work, we let the user generate a partial evaluation key which is the encryption of his private key and the evaluation key of the joint ciphertexts is derived from the partial evaluation keys of all users. Specifically, we design different approaches to compute the evaluation key of the joint ciphertexts for different FHE schemes.

Another issue is to update the ciphertext. When the user set is updated, we need to convert the ciphertexts of the old user group into the ciphertext corresponding to the new user set. The solution is that the old user set, as the owner of the old ciphertext, should authorize the new user set. Firstly, the user in old user set encrypts his secret by the new joint public key corresponding to the new user set, and generates the partial evaluation key. Secondly, The cloud uses the partial evaluation key to generate the joint evaluation key of the new user set. Finally, the cloud runs the key switching process or bootstrapping process to convent the old joint ciphertext to ciphertext corresponding to the new joint public key.

1.4 Related works

Similar to MKFHE, threshold FHE also involves multiple parties. However, in threshold FHE, all ciphertexts are encrypted with the same public key(joint public key), while in MKFHE, the ciphertexts are encrypted with different public keys.

The comparison between our schemes and the most current MKFHE and threshold FHE(ThMKFHE) schemes is shown in Table 1. As we know, we construct the first MKFHE scheme with compact ciphertext and evaluation key, which makes it almost as efficient as FHE. Moreover, our scheme supports the authentication of the user.

Compared with the ThFHE scheme, our scheme has the advantages: the data is encrypted by the user's public key; when the user set is updated, the ciphertext can be reused by regenerating the new joint public key and new joint evaluation key. Compared with the MKFHE scheme, our scheme has the advantages: Users can decide whether to participate in a computation task with a given user set by authorizing the user set; the scheme is compact and the joint evaluation key is also independent of the number of users, and the homomorphic operation is as efficient as single-party FHE scheme.

Table 1. The comparison between our scheme and the most current MKFHE and Threshold FHE schemes, k denotes the number of parties,l denotes the number of ciphertexts and n is the dimension of the (R)LWE assumption

Scheme	Ciphertext	Joint evaluation key (jek)	Computational complex of generating jek	Encrypt with personal pk	Ciphertext reusable	Depth of bootstrapping
AJL+12	$O(n)$	$O(n)$	$O(kn)$	No	No	$> \log(n)$
GLS15	$O(kn^2 \log n)$	0(GSW-type)	0(GSW-type)	No	No	$> \log(n)$
BGG+18 centralized version	$O(n)$	$O(n)$	$O(n)$	No	No	$> \log(n)$
BGG+18 decentralized version instance with CDKS19	$O(ln)$	$O(ln)$	$O(l^2n)$	No	No	$> \log(n)$
BGM+20 instance with CDKS19	$O(ln)$	$O(ln)$	$O(l^2n)$	Yes	No	$> \log(n)$
LZY+19	$O(kn)$	$O(k^3n)$	$O(k^3n)$	Yes	Yes	$> \log(n)$
CCS19	$O(kn)$	$O(k^2n^2)$	$O(k^2n^2)$	Yes	Yes	$O(1)$
CDKS19	$O(kn)$	$O(kn)$	$O(k^2n)$	Yes	Yes	$> \log(n)$
Our AMCMK	$O(n)$	$O(n)$	$O(kn)$	Yes	Yes	$> \log(n)$
Our AMTMK	$O(n)$	$O(n)$	$O(kn)$	Yes	Yes	$O(1)$

2 PRELIMINARIES

Throughout the paper, there are many definitions of each party. Here we give a simple description of them. For the party i, he selects his *secret key (sk)* and generates the corresponding *public key (pk)*. The public key corresponding to a user set is *joint public key (jpk)*. The ciphertext corresponding to *joint public key*

(jpk) is called *joint ciphertext*. The ciphertexts about user's secret is called *partial evaluation key (pevk)*, the *pevk* contains of *partial switching key (psk)* and *partial bootstrapping key (pbk)*. The difference between *psk* and *pbk* is that they have a different form of the encrypted secret key. The evaluation key corresponding to the joint ciphertext is called *joint evaluation key (jek)* , the *jek* contains of *joint switching key (jsk)* and *joint bootstrapping key (jbk)*.

2.1 Definition of multi-key fully homomorphic encryption

We now introduce the cryptographic definition of a leveled multi-key FHE, which is similar to the definitions in CZW17[11,24] with some modifications.

The one modification is that we separate the evaluation key generation from the key generation process. In the previous MKFHE schemes, the user cannot decide which user set is involved in the evaluation of his ciphertext, which makes it easy for users to lose control over data and their privacy. We separate the evaluation key generation process to let the users decide whether to participate in a computation task with a given user set, by generating the corresponding partial evaluation key of the set.

The other modification is that when the user set is updated, we need to regenerate the evaluation key, and convert the ciphertext of the old user set into the ciphertext corresponding to the new user set, which is done in $\mathcal{C}.\mathsf{Eval}$ process. This requirement is reasonable in real-world scenarios. Because when the user set is updated, the old user set, as the owner of the old ciphertext, should authorize the new user set.

Definition 1 (Multi-key FHE). *Let $\mathcal{C}$ be a class of circuits. A leveled multi-key FHE scheme $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{EvalKeyGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ is described as follows:*

- $\mathcal{E}.\mathsf{Setup}(1^\lambda, 1^K, 1^L)$*: Given the security parameter λ, the circuit depth L, and the number of distinct parties K that involving in the evaluation, outputs the public parameters pp.*
- $\mathcal{E}.\mathsf{KeyGen}(pp)$*: 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 P_i, where $i = \{1, ..., K\}$.*
- $\mathcal{E}.\mathsf{EvalKeyGen}(pp, \{sk_i\}_{i\in[K]}, \{pk_i\}_{i\in[K]})$*:*
  - $\mathsf{JPKGen}(pp, \{pk_i\}_{i\in[K]})$*: Given the public key $\{pk_i\}_{i\in[K]}$ of all user to evaluator, the evaluator outputs the joint public key $\overline{pk}$, and send them to all users.*
  - $\mathsf{PEvalKeyGen}(pp, sk_i, \overline{pk})$*: Input the $\overline{pk}$ and secret key sk_i, the user P_i outputs his partial evaluation key $pevk_i$.*
  - $\mathsf{AggEvalKey}(pp, \{pevk_i\}_{i\in[k]})$*: Given $\{pevk_i\}_{i\in[K]}$, the evaluator (usually the cloud) aggregate them to generate and output the joint evaluation key $\overline{evk_{set}}$. The evaluation key may consist bootstrapping key, switching key, which is different for different schemes. If the parties set is updated, rerun $\mathcal{E}.\mathsf{EvalKeyGen}()$.*

- $\mathcal{E}.\mathsf{Enc}(pk_i, m)$*: Given a public key* pk_i *and message* μ*, outputs a ciphertext* ct_i*.*
- $\mathcal{E}.\mathsf{Dec}((sk_1, \cdots, sk_K), ct_S)$*: Given a ciphertext* ct_S *corresponding to a set of parties* $S = \{i_1, i_2, ..., i_k\} \subseteq [K]$*, and their secret keys* $sk_S = \{sk_1, sk_2, ..., sk_K\}$*, outputs the message* μ*.*
- $\mathcal{E}.\mathsf{Eval}(\mathcal{C}, ct_1, \cdots, ct_t, \{pevk_i\}_{i\in[K]}, \overline{evk_{set}})$*: On input a Boolean circuit* $\mathcal{C}$ *along with* t *ciphertext* $\{ct_i\}_{i\in[t]}$ *corresponding to the user* i *or a user set* S'*, and the joint evaluation key* $\overline{evk_{set}}$*, output a ciphertext* ct_S *corresponding to the user set* S*.*

Definition 2 (Correctness of MKFHE[11]). *On input any circuit of depth at most* L *and a set of ciphertexts* $\{ct_i\}_{i\in[t]}$*, let* $\mu_i = \mathrm{Dec}(sk_i, ct_i)$*, a leveled MKFHE scheme* $\mathcal{E}$ *is correct if it holds that*

$$\Pr[\mathrm{Dec}(\{sk_i\}_{i\in[K]}, \mathsf{Eval}(\mathcal{C}, ct_1, \cdots, ct_t, evk_{set})) \neq \mathcal{C}(\mu_1, ..., \mu_t)] = \mathsf{negl}(\lambda)$$

In general, the ciphertext length of the compact MKFHE scheme is related to the security parameter λ, the number of participants K, and the circuit depth L polynomial level. While the compact defined in this paper is more strict, we require the ciphertext length is independent of K.

Definition 3 (Compact Multi-key FHE). *For a leveled MKFHE scheme, the ciphertext of the MKFHE scheme is compact if there is a polynomial function* $poly(\cdot)$ *such that the length of the ciphertext* $|c| \leq poly(\lambda, L)$*, and the length of the ciphertext are independent of the operation circuit* $\mathcal{C}$ *and the number of participants* K*.*

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[6].

Definition 4. *GLWE problem[6]. Let* λ *be a security parameter. For the polynomial ring* $R = \mathbb{Z}[X]/x^d + 1$ *and* $\mathcal{R}_q = R/qR$*, 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* $(\mathbf{a}_i, b_i) \in \mathcal{R}_q^{n+1}$ *uniformly from* $\mathcal{R}_q^{n+1}$*. For the second distribution, one first draws* $\mathbf{a}_i \leftarrow \mathcal{R}_q^n$ *uniformly, and samples* $(\mathbf{a}_i, b_i) \in \mathcal{R}_q^{n+1}$ *by choosing* $\mathbf{s} \leftarrow \mathcal{R}_q^n$ *and* $e_i \leftarrow \chi$ *uniformly, and set* $b_i =< \mathbf{a}_i, \mathbf{s} > +e_i$*.*

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

2.3 Gadget decomposition

Homomorphic multiplication of two ciphertexts will make the noise increase sharply. One method proposed by BGV12 is to reduce the noise by bit decomposite one ciphertext.

Let $\mathbf{g} = (g_i) \in \mathbb{Z}^d$ be a gadget vector and q an integer. The gadget decomposition, denoted by $\mathbf{g}^{-1}$, is a function from $\mathcal{R}_q$ to R^d which transforms an element $a \in \mathcal{R}_q$ into a vector $u = (u_0, \cdots, u_{d-1}) \in R^d$ of small polynomials such that $a = \sum_{d_i=0}^{d-1} g_i \cdot u_i(\pmod q)$[10].

2.4 Static mode CMKFHE scheme

To solve the problem that the length of ciphertext usually increases linearly or squarely with the increase of the number of users, which leads to the low efficiency of the scheme. Many researchers have made related attempts, among which threshold fully homomorphic encryption is a typical representative[2]. N-out-of-N threshold fully homomorphic encryption is a special case of the threshold fully homomorphic encryption scheme. Those schemes are usually compact in ciphertext, simple in structure . While in those schemes the ciphertext and the user set are bounded,i.e., when the parties participating involved in the calculation are updated, the ciphertext needs to be regenerated. In order to be consistent with the authorized mode compact MKFHE scheme in section 3, this paper refers to N-out-of-N threshold fully homomorphic encryption with compact ciphertext as static mode CMKFHE scheme—SMMK.

Based on the CRS model , we present the general construction of static mode CMKFHE scheme based on FHE, called SMMK. We take the GSW-type FHE as an example to illustrate.

$\mathsf{SMMK.Setup}(1^\lambda)$: Run $\mathsf{FHE.Setup}(1^\lambda) \rightarrow \mathsf{params}$, and all users jointly generate a common reference string $\mathbf{B} \in \mathbb{Z}_q^{m\times(n-1)}$.

$\mathsf{SMMK.KeyGen}(\mathbf{params}, i, \mathbf{B})$: Run $\mathsf{FHE.KeyGen}(\mathbf{params}, \mathbf{B}) \rightarrow (pk_i, sk_i)$. For all the user $i \in [k]$, run the key generation algorithm : The user selects $s_i \leftarrow \chi_{\text{key}}$, sets the secret key $sk_i := (1, -s_i)$; and generate the public key $\mathbf{b}_i = [\mathbf{s}_i\mathbf{B}+\mathbf{e}_i, \mathbf{B}] \in \mathbb{Z}_q^{m\times n}$. Output the public key and private key pair (pk_i, sk_i).

After all parties have completed the process $\mathsf{SMMK.KeyGen}()$, run the generation algorithm to generate the joint evaluation key.

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

- $\mathsf{SMMK.JPKGen}(\mathsf{params}, \{pk_1, ..., pk_k\})$: Generate the *joint public key*. Input public keys $\{pk_1, ..., pk_k\}$ of all users, output the joint public key $\overline{pk} := \overline{\mathbf{A}} = \left[\mathbf{b}_1 + ... + \mathbf{b}_k \ \mathbf{B}\right] \in \mathbb{Z}_q^{m\times n}$.
- $\mathsf{FHE.SwitchKeyGen}(\mathsf{params}, sk_i, \overline{pk})$: Generate *partial switching key* and *joint switching key*. Input the party i's sk_i and joint public key, the user i generate the *partial switching key (psk)* $\overline{\mathbf{KS}_i} = \mathsf{Enc}_{\overline{pk}}(sk_i)$, which is used to translate the ciphertext of user i to the user set. Generate the *joint switching key (jsk)* $\overline{\mathbf{KS}_{Set}} = \mathsf{Enc}_{\overline{pk}}(\overline{sk_i} \otimes \overline{sk_i})$ for the user set, which is used to deal with

the homomorphic multiplication for the joint ciphertext. Due to the different structures of the schemes, the generation process of the *psk* is slightly different(for example, the GSW-type MKFHE does not need to run this key switching process).

– $\mathsf{FHE.BootKeyGen}(\mathsf{params}, sk_i, \overline{pk})$: Generate the *partial bootstrapping key* and *joint bootstrapping key*. Input the party's sk_i and *jpk* $\overline{pk}$. Firstly, according to the single-key fully homomorphic encryption scheme, the user i generate the *partial bootstrapping key pbk* $\overline{\mathbf{BK}_i} = \mathsf{Enc}_{\overline{pk}}(sk_i)$ of party i (Due to the different structures of the schemes, the *pbk* and *psk* can be the same or not). Then, generate the *joint bootstrapping key* by the *partial bootstrapping key pbk* $\overline{\mathbf{BS}_{\overline{Set}}} = HomAdd_{\log(kB_\chi)}(\overline{\mathbf{BK}_1}, \cdots, \overline{\mathbf{BK}_k})$ which is used to bootstrap the joint ciphertext.

Due to the different types of bootstrapping, the HomAdd operation is slightly different. If the bootstrapping is implemented by the arithmetic operation, then the HomAdd operation is an arithmetic operation; if the bootstrapping is implemented by the boolean circuit, the HomAdd operation is the boolean circuit.

$\mathsf{SMMK.Enc}(\overline{pk}, \mu)$: $\mathsf{FHE.Enc}(\overline{pk}, \mu)$. The encryption is the same as the single-party FHE schemes. Input the plaintext μ and the accumulated public key $\overline{pk}$, and call the single-key FHE scheme encryption algorithm to generate the ciphertext $\mathbf{C}$.

$\mathsf{SMMK.Dec}((sk_1, ..., sk_N), \mathbf{C})$: The decryption result consists of two parts: partial decryption and final decryption.

– $\mathsf{SMMK.PartDec}(\mathbf{C}, i, sk_i)$: Input the secret key $sk_i = (-\mathbf{s}_i, 1)$ of party i and the ciphertext $\mathbf{C}$, and output the partial decryption result p'_i. In GSW-type FHE, its decryption form is $\mu' = \mathbf{s}_i\mathbf{C}\mathbf{G}^{-1}(\hat{\mathbf{w}}^T)$. We calculate $p'_i := \mathbf{s}_i\mathbf{C}_{[1,\cdots,n-1;]}\mathbf{G}^{-1}(\hat{\mathbf{w}}^T) + e_i{}^{sm}$ and get the partial decryption p'_i where $\mathbf{C}_{[1,\cdots,n-1;]}$ represents the first n-1 columns of ciphertext $\mathbf{C}$, $\hat{\mathbf{w}} = (0, \cdots, \lceil q/2 \rceil) \in \mathbb{Z}^{kn}$, and $e_i^{sm} \xleftarrow{R} [-B^{dec}_{smdg}, B^{dec}_{smdg}]$ is the generated error used to protect the security of partial decryption.
– $\mathsf{SMMK.FinDec}(p'_1, \cdots p'_N)$: Input all the partial decryption results p'_i, and output the plaintext $m' = \hat{\mathbf{C}}_{[n;]}\hat{\mathbf{G}}^{-1}(\hat{\mathbf{w}}^T) - \sum_{i=1}^{N} p'_i$.

The homomorphic computation is just like the FHE with the joint keys.

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

$\mathsf{SMMK.Mult}(\mathbf{C}_1, \mathbf{C}_2, \overline{\mathsf{KS}_{\overline{Set}}})$: $\mathbf{C}_\times \leftarrow \mathsf{FHE.Mult}_{\overline{\mathsf{KS}_{\overline{Set}}}}(\mathbf{C}_1, \mathbf{C}_2)$.

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

The advantages of SMMK are: (1) The scheme is efficient. The ciphertexts of all parties are encrypted by the joint public key $\overline{pk}$, the scheme is strongly compact, and the homomorphic operation is as efficient as single-party FHE scheme. (2) The construction idea of the scheme can be applied to various MKFHE schemes, such as BGV-type and GSW-type FCMKFHE. (3) Users can decide whether to participate in a computation task with a given user set. The participants of this scheme can control the set of users whose data participate in the

evaluation (by encrypting the data with the joint public key of this specific user set).

However, the SMMK scheme also has some defects: (1) The data encrypted by the users' public key pk_i cannot be homomorphically evaluated with the ciphertext data of other users, which limits the application of the scheme; (2) When the user set is updated, the user needs to regenerate the ciphertext.

3 The General construction of authorized mode compact MKFHE scheme

The ciphertext and evaluated key of the SMMK are all corresponding to a particular party set. In this section, we focus on constructing an authorized mode compact MKFHE scheme (we call it AMMK scheme). Our target is: the size of ciphertext is independent of the number of parties; the data encrypted by the user's public key pk_i can also be calculated with the ciphertext of other users; all ciphertexts can be reused after authorized by the old user set and new user set. To construct an AMMK scheme that satisfies the above conditions, we need to solve three problems.

- How to perform homomorphic evaluation between the ciphertexts corresponding to two different public keys. The solution in this paper is: different from the ciphertext expansion algorithms of other MKFHE schemes, we first accumulate the public keys of the users participating in the evaluation to generate the joint public key. Secondly, the user uses the joint public key to encrypt his private key to generate a partial evaluation key. Finally, the computer uses the key switching technology to convert the ciphertexts corresponding to different public keys to the joint ciphertext corresponding to the user set, and then perform the homomorphic operation on the joint ciphertext.
- How to update the ciphertext. When the user set is updated, how to convert the ciphertext of the old user set into the ciphertext corresponding to the new user set. The solution is that the old user set, as the owner of the old ciphertext, should authorize the new user set, i.e. firstly, the user in old user set encrypts his secret by the new joint public key corresponding to the new user set, and generate the partial bootstrapping key. Secondly, The cloud uses the partial bootstrapping key to generate the joint bootstrapping key of the new user set. Finally, the cloud runs the bootstrapping process (or key switching process in section 4) to convent the old joint ciphertext to ciphertext corresponding to the new joint public key.
- How to update the evaluation keys (bootstrapping keys and conversion keys). That is, when the user set is updated, how to update the calculation key of the user set. The solution in this paper is to use the SMMK method to generate an evaluation key for a new set of users.

The general construction of AMMK is as follows.

$\mathsf{AMMK.Setup}(1^\lambda)$: Run $\mathsf{FHE.Setup}(1^\lambda) \to \mathbf{params}$.

$\mathsf{AMMK.KeyGen}(\mathbf{params})$: Run $\mathsf{FHE.KeyGen}(\mathbf{params}, \mathbf{B}) \to (pk_i, sk_i)$.

After all parties have completed the process $\mathsf{AMMK.KeyGen}(\mathbf{params})$, run the generation algorithm of the evaluation key.

$\mathsf{AMMK.EvalKeyGen}(\mathbf{params}, sk_i, \{pk_1, ..., pk_N\})$: This step is for the user to authorize the user set. Run $\mathsf{SMMK.EvalKeyGen}(\mathbf{params}, sk_i, \{pk_1, ..., pk_N\}) \to \{\overline{pk}, \overline{\mathbf{KS}}_i, \overline{\mathbf{KS}_{\overline{Set}}}, \overline{\mathbf{BK}}_i, \overline{\mathbf{BS}_{\overline{Set}}}\}$.

$\mathsf{AMMK.Enc}(pk_i, \mu)$: Run $\mathsf{FHE.Enc}(pk_i, \mu) \to \mathbf{C}$.

$\mathsf{AMMK.Dec}((sk_1, \cdots, sk_N), \mathbf{C})$: Run $\mathsf{SAMK.Dec}((sk_1, \cdots, sk_N), \mathbf{C}) \to \mu'$.

$\mathsf{AMMK.Eval}((\mathbf{C}_1, \mathbf{C}_2), \overline{\mathbf{BK}}_i, \overline{\mathbf{BK}_{\overline{Set}}}, \overline{\mathbf{KS}_{\overline{Set}}})$: Different from SMMK scheme, the $\mathsf{AMMK.Eval}$ has two steps: adjuste the public key corresponding to the ciphertext to the same user set; call the homomorphic evaluation function of FHE.

- $\mathbf{C'}_i = Hom_{\overline{pk}, \overline{\mathbf{BK}_{c_i}}}\{\mathsf{FHE.Dec}_{sk_{\mathbf{c}_i}}(\mathbf{C}_i)\}$. This process can switch the public key of the ciphertexts to the public key of usesr set. $\overline{\mathbf{BK}_{\mathbf{c}_i}}$ is the *bootstrapping key* corresponding to $\mathbf{C}_i$. If $\mathbf{C}_i$ is the ciphertext of a single party, the $\overline{\mathbf{BK}_{\mathbf{c}_i}}$ is setted as *partial bootstrapping key* $\overline{\mathbf{BK}}_i$. If $\mathbf{C}_i$ is the joint ciphertext of all parties, the $\overline{\mathbf{BK}_{\mathbf{c}_i}}$ is setted as *joint bootstrapping key* $\overline{\mathbf{BK}_{\overline{Set}}}$
- $\mathsf{FHE.Eval}((\mathbf{C'}_1, \mathbf{C'}_2), \overline{\mathbf{KS}_{\overline{Set}}})$. This process realizes the homomorphic computation of joint ciphertext, where $\overline{\mathbf{KS}_{\overline{Set}}}$ is the *accumulated evaluation key* of the joint ciphertext.

Compared with the ThFHE scheme, our scheme has the advantages: the data is encrypted by the user's public key; when the user set is updated, the ciphertext can be reused by regenerating the new joint public key and new joint evaluation key. Compared with the MKFHE scheme, our scheme has the advantages: Users can decide whether to participate in a computation task with a given user set by authorizing the user set; the scheme is strongly compact, and the homomorphic operation is as efficient as single-party FHE scheme.

4 Construction of BGV-type CMKFHE scheme

CKKS17 [11] scheme is an efficient BGV-type FHE scheme, which can deal with approximate numbers and is friendly to floating point calculations. In this section, we construct an effective compact MKFHE scheme based on CKKS17, called AMCMK. AMCMK scheme can also be extended to the BGV/BFV[5,20] schemes trivially. Our contributions are as follows:

- We reduce the number of multiplication of constructing the joint switching key from $O(k^2)$ to $O(k)$. The joint switching key of joint ciphertext is $\mathsf{Enc}_{\overline{pk}}(\overline{s} \otimes \overline{s})$(see BGV for more details), where $\overline{s} = s_1 + s_2 + \cdots + s_k$.The general way to generate the evaluation key is as follows: user i encrypts s_i with to obtain the partial switching key $\mathsf{Enc}_{\overline{pk}}(s_i)$; the calculator then obtains the joint switching key $\mathsf{Enc}_{\overline{pk}}(\overline{s} \otimes \overline{s})$ by implement $O(k^2)$ homomorphic multiplications and homomorphic additions on $\mathsf{Enc}_{\overline{pk}}(s_i)$. In this paper, we

present an algorithm, where the user P_i can generate $\mathsf{Enc}_{\overline{pk}}(\overline{s} \cdot s_i)$ with 1 multiplication. Then by sum the items $\mathsf{Enc}_{\overline{pk}}(\overline{s} \cdot s_i)$ for different users, we can get $\mathsf{Enc}_{\overline{pk}}(\overline{s} \otimes \overline{s}) = \sum_{i=1}^{k} \mathsf{Enc}_{\overline{pk}}(\overline{s} \cdot s_i)$ with only $O(k)$ multiplications.
- We reduce the size of switching key (relinearization key) of CDKS19 by raising the ciphertext modulus. Homomorphic multiplication of two ciphertexts will make the noise increase sharply. There are two common solutions: one method proposed by BGV12 is to reduce the noise by bit decomposite one ciphertext. Another method proposed by CKKS17 scheme is to raise the modulus of one ciphertext. The defect of the bit decomposition method is that the ciphertext size increases dramatically. The CDKS19 scheme uses twice consecutive bit decomposition to generate the switching key , which makes the size of switching key grows quadratically with the decompose parameters. We suggest mixed-use of bit decomposition and raising modulus to reduce the size of switching key.

4.1 Generation of joint switching key

In the BGV-type FHE scheme, the homomorphic multiplication operation of the (joint) ciphertext $\mathsf{Enc}_{\overline{s}}(a)$ needs to input the (joint) switching key in the form of $\mathsf{Enc}_{\overline{pk}}(\overline{s} \otimes \overline{s})$. The generation of the joint switching key is the core of the CMKFHE. We present an efficient way to generate the joint switching key, by optimizing the relinearization algorithm of CDKS19 to adjust the joint secret key. For the convenience of description, this section takes two users P_1, P_2 as examples to introduce our idea of generating the joint switching key.

We denote the secret/public key pair of users P_1 and P_2 as $sk = (1, -s)$, $pk = (b = -s \cdot a_1 + e_1 \pmod q, a_1)$, $sk' = (1, -s')$, $pk' = (b' = -s' \cdot a_1 + e_1' \pmod q, a_1)$. Then the joint secret/public key pair is $\overline{sk} = (1, -\overline{s}) = (1, -s - s')$, $\overline{pk} = (\overline{b} = -\overline{s} \cdot a_1 + \overline{e} \pmod q, a_1)$. The problem is: how to efficiently generate joint switching key $\mathsf{Enc}_{\overline{pk}}(\overline{s} \otimes \overline{s})$, where $\hat{s} = \overline{s} \otimes \overline{s} = (1; -\overline{s}; -\overline{s}; \overline{s}^2)$.

The solution in this paper is as follows: Define the four ciphertext components of the joint switching key as $K_1 = \mathsf{Enc}_{\overline{pk}}(1), \overline{K_2} = \overline{K_3} = \mathsf{Enc}_{\overline{pk}}(-\overline{s}), \overline{K_4} = \mathsf{Enc}_{\overline{pk}}(\overline{s}^2)$. The generation process of $\overline{K_4} = \mathsf{Enc}_{\overline{pk}}(\overline{s}^2)$ is the most complicated since the multiplication operation is involved. This section takes the process of generating $\overline{K_4}$ as an example to illustrate.

- Decompose $\overline{K_4}$ into two parts: $\overline{K_4} = \mathsf{Enc}_{\overline{pk}}(\overline{s}^2) = \mathsf{Enc}_{\overline{pk}}(s\overline{s}) + \mathsf{Enc}_{\overline{pk}}(s'\overline{s})$.
- User P_1 generates the corresponding component $\mathsf{Enc}_{\overline{pk}}(s\overline{s})$, user P_2 generates the corresponding component as follows: $\mathsf{Enc}_{\overline{pk}}(s'\overline{s})$

  - P_1 and P_2 jointly generate a public random polynomial $a \in \mathcal{R}_q$.
  - P_1 and P_2 separately generate the matrix $\mathbf{D}$ and $\mathbf{D}'$. We take the generation $\mathbf{D} = [d_0|d_1|d_2]$ of P_1 as an example to illustration.
    * Generate the RLWE ciphertext $d_2 = r \cdot a + e_2 + s \pmod q$ of s, where $e_2 \leftarrow \psi$, $a \in \mathcal{R}_q$ is the public random polynomial.

* Encrypt r with the public key $\bar{b}$ and get the ciphertext $(\bar{d_1}, \bar{d_0}) = r'(\bar{b}, \bar{a}) + e + r$, such that $\bar{d_0} = -\bar{s} \cdot \bar{d_1} + e_1 + r \pmod q$, for a random $\bar{d_1} \in \mathcal{R}_q$,and small e_1(The idea behind this step is: Use $\bar{s}$ to force decrypt d_2, $\bar{s}d_2 = r \cdot \bar{s}a + \bar{s}e_2 + \bar{s}s \pmod q \approx r \cdot \bar{b} + \bar{s}s$, then makeup $r \cdot \bar{b}$ with $(\bar{d_0} - \bar{s}\bar{d_1}) \cdot \bar{b}$.)

The equation holds:$(1, -s - s') \cdot \begin{pmatrix} \bar{b}d_0 \\ \bar{b}d_1 + d_2 \end{pmatrix} \approx s\bar{s}$.

– Compute and output $\bar{K_4} = \mathsf{Enc}_{\bar{pk}}(\bar{s}^2) = \begin{pmatrix} \bar{b}d_0 \\ \bar{b}d_1 + d_2 \end{pmatrix} + \begin{pmatrix} \bar{b}d_0' \\ \bar{b}d_1' + d_2' \end{pmatrix}$.

Note: The above generation process involves the multiplication of ciphertext $\bar{b}d_0$, $\bar{b}d_1$. We use the method of bit decomposition and modulus raising to solve this problem.

4.2 Construction

Our scheme is based on CKKS17, see Appendix A for more details. The constructions of AMCMK are as follows:

$\mathsf{AMCMK.Setup}(1^\lambda)$**:** 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 numbers P and p, the max circuit layers L, the ciphertext modulus $q_l = p^l$, where $1 \leq l \leq L$. Select common reference strings $\mathbf{a} \leftarrow U(\mathcal{R}^d_{P \cdot q_L})$ and $a' \leftarrow U(\mathcal{R}^1_{P \cdot q_L})$. Output public parameter $pp = \{N, \chi_{key}, \chi_{err}, \chi_{enc}, L, P, q_l, \mathbf{a}, a'\}$.

In this step, we generate some common reference strings, which can be used to generate the public keys.

$\mathsf{AMCMK.KeyGen}(pp)$**:** Given the public parameters $pp = \{N, \chi_{key}, \chi_{err}, \chi_{enc}, L, P, q_l, \mathsf{a}, a'\}$, the party i outputs the public key $pk_i := \{b_i' = -s_i \cdot a + e_i'\} \in \mathcal{R}^1_{P \cdot q_L}$ where $s_i \leftarrow \chi_{key}$, $e_i' \leftarrow \chi_{err}$, the evaluation public key $epk_i := \{\mathbf{b}_i = -s \cdot \mathbf{a}_i + \mathbf{e}_i\} \in \mathcal{R}^d_{P \cdot q_L}$ where $\mathbf{e}_i \leftarrow \chi^d_{err}$.

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

$\mathsf{AMCMK.EvalKeyGen}(pp, \{sk_i\}_{i \in [k]}, \{pk_i, epk_i\}_{i \in [k]}) \rightarrow \{\overline{pk}, evk\}$**:** Our (joint) evaluation key $evk = \{\overline{\mathbf{ks}_{\overline{set}}}, \overline{\mathbf{rk}_{\overline{set},r}}, \overline{\mathbf{ck}_{\overline{set}}}, \overline{\mathbf{ks}_{s_i \to \bar{s}}}\}$ consist of 4 parts: the joint switching key, joint rotation key, joint conjugation key, and the partial switching key, which is used to refresh the public key of single-user ciphertext to user set.

– $\mathsf{JPKGen}(pp, pk_1, \cdots, pk_k)$: Evaluator accumulate the public keys and evaluation public keys. Given k parties' public key $\mathbf{b}_1, ..., \mathbf{b}_k$, the joint public key is generated as $\overline{epk} := \overline{\mathbf{b}} = (\mathbf{b}_1 + ... + \mathbf{b}_k) \in \mathcal{R}^d_{Pq_L}$. Denote $\overline{epk}_{[,j]}$ as the j-th element of $\overline{epk}$. Given the k parties' evaluation public key $b_1', ..., b_k'$, the joint evaluation public key is generated as $\overline{pk} := \overline{b'} = (b_1' + ... + b_k') \in \mathcal{R}^1_{Pq_L}$.

– $\mathsf{PEvalKeyGen}(pp, sk_i, \overline{pk})$: Given joint public key $\overline{pk}$ and secret key sk_i, the user P_i outputs the $pevk_i = \{\overline{\mathbf{ks}_{set,i}}, \overline{\mathbf{rk}_{i,r}}, \overline{\mathbf{ks}_{s_i\to\overline{s}}}, \overline{\mathbf{ks}_{s_i\to\overline{s}}}\}$.
  - $\mathsf{PSwitchKeyGen}(pp, sk_i, \overline{pk})$: User P_i generates the partial switching key. Select $r \leftarrow ZO(0.5)$ randomly, and the partial switching key is obtained as $(\overline{\mathbf{d}_0}, \overline{\mathbf{d}_1}) := \{(\overline{d_{0,[,j]}}, \overline{d_{1,[,j]}})\}_{j\in[d]}$, where $(\overline{d_{0,[,j]}}, \overline{d_{1[,j]}}) \leftarrow \mathsf{CKKS.Enc}_{\overline{epk_{[,j]}}, Pq_L}(r \cdot \mathbf{g}_{[,j]})$, $j \in [d]$ and $\mathbf{g} = (1, B_g, ..., B_g^{d-1})$. B_g is the decomposition basis. We denote $\mathsf{CKKS.Enc}_{pk,q}(m)$ as the CKKS ciphertext of m under the public key pk and modulus q, see Appendix A for details. Set $\overline{d_2} = r \cdot a' + e_2 + P \cdot s (\bmod Pq_L)$, where $e_2 \leftarrow \chi_{err}$. Output partial switching key:

$$\overline{\mathbf{ks}_{set,i}} := \left[\mathbf{g}^{-1}(\overline{b'})\right]_{1\times d} \cdot \left[\overline{\mathbf{d}_0} | \overline{\mathbf{d}_{i,1}}\right]_{d\times 2} + \left[0 | \overline{d_2}\right] \in \mathsf{CKKS.Enc}_{\overline{pk}, P\cdot q_L}(P \cdot s_i \overline{s}).$$

  - $\mathsf{PBootKeyGen}(pp, sk_i, \overline{pk})$: Generate and output the partial bootstrapping key: the partial refresh key $\overline{\mathbf{ks}_{s_i\to\overline{s}}} \leftarrow \mathrm{CKKS.Enc}_{\overline{pk}, Pq_L}(P \cdot s_i)$, the partial rotation key $\overline{\mathbf{rk}_{i,r}} \leftarrow \mathsf{CKKS.Enc}_{\overline{pk}, Pq_L}(\kappa_{5^r}(s_i))$, the partial conjugate key $\overline{\mathbf{ks}_{s_i\to\overline{s}}} \leftarrow \mathsf{CKKS.Enc}_{\overline{pk}, Pq_L}(P \cdot s_i)$.

– $\mathsf{AggEvalKey}(pp, \{pevk_i\}_{i\in[k]})$: Given the partial evaluation key $\{pevk_i\}_{i\in[k]}$, the evaluator output $evk = \{\overline{\mathbf{ks}_{\overline{set}}}, \overline{\mathbf{rk}_{\overline{set},r}}, \overline{\mathbf{ck}_{\overline{set}}}, \overline{\mathbf{ks}_{s_i\to\overline{s}}}\}$.:
  - $\mathsf{AggSwitchKey}(pp, sk_i, \overline{pk})$: The evaluator generates the joint switching key $\overline{\mathbf{ks}_{set}} := \{\overline{K_1}, \overline{K_2}, \overline{K_3}, \overline{K_4}\}$, where $\overline{K_1} = (1, 0) \in \mathsf{CKKS.Enc}_{\overline{pk}, P\cdot q_L}(1)$, $\overline{K_2} = \overline{K_3} = -\sum_{i=1}^{k} \overline{\mathbf{ks}}_{s_i\to\overline{s}} \in \mathsf{CKKS.Enc}_{\overline{pk}, P\cdot q_L}(\overline{-s})$, $\overline{K_4} = \sum_{i=1}^{k} \overline{\mathbf{ks}_{\overline{set},i}} \in \mathsf{CKKS.Enc}_{\overline{pk}, P\cdot q_L}(P\overline{s} \cdot \overline{s})$.
  - $\mathsf{AggBootKey}(pp, \overline{\mathbf{ks}_{s_i\to\overline{s}}}, \overline{\mathbf{rk}_{i,r}})$: The evaluator generates the joint bootstrapping key:

$$\overline{\mathbf{rk}_{\overline{set},r}} = \sum\nolimits_{i=1}^{k} \overline{\mathbf{rk}_{i,r}} \in \mathsf{CKKS.Enc}_{\overline{pk}, Pq_L}(\kappa_{5^r}(\overline{s}))$$

$$\overline{\mathbf{ck}_{\overline{set}}} = \sum\nolimits_{i=1}^{k} \overline{\mathbf{ck}_i} \in \mathsf{CKKS.Enc}_{\overline{pk}, Pq_L}(\kappa_{-1}(\overline{s})).$$

$\mathsf{AMCMK.Enc}(pk, m)$**:** Given the public key $pk := \{b' = -s \cdot a' + e'\} \in \mathcal{R}^1_{P\cdot q_L}$. Select $r \leftarrow \chi_{enc}, e_0, e_1 \leftarrow \chi_{err}$ randomly. Output $\mathbf{ct} = r \cdot (b', a') + (m + e_0, e_1) (\bmod q_L)$, such that $\langle \mathbf{ct}, \mathbf{sk}\rangle (\bmod q_L) \approx m$.
The public key $pk \in \mathcal{R}^1_{P\cdot q_L}$ has two purposes, one is to encrypt plaintext, and the other is to generate joint evaluation keys.

$\mathsf{AMCMK.Eval}(\overline{\mathbf{ks}_{\overline{set}}}, \overline{\mathbf{rk}_{\overline{set},r}}, \overline{\mathbf{ck}_{\overline{set}}}, \mathbf{c})$**:** This procedure consists of two subroutines: one is to refresh public key of the ciphertexts to the same user set, the other is homomorphic computation.
– Generate the refreshing key $\overline{\mathbf{ks}_{refresh}}$, we denote $[k']$ as the original parties set.

$$\overline{\mathbf{ks}_{refresh}} = \begin{cases} \mathbf{ks}_{s_i\to\overline{s}} & \mathbf{c} \text{ is corresponds to the secret key } \mathbf{s}_i \\ \sum_{i=1}^{k'} \mathbf{ks}_{s_i\to\overline{s}} & \mathbf{c} \text{ is corresponds to the secret key } (\mathbf{s}_1 + ... + \mathbf{s}_{k'}). \end{cases}$$

When the secret key corresponding to c is $\mathbf{s}_i$, the refreshing switching key is used to refresh the ciphertext of a single user; when the secret key

corresponding to c is $\mathbf{s}_1 + \cdots + \mathbf{s}_{k'}$, refreshing switching key is used to refresh the ciphertext of the old user set.

- Run the key switching process for all ciphertexts $\mathsf{CKKS.KeySwitching}(\mathbf{c}, \overline{\mathbf{ks}_{refresh}})$. The original single-user or old set ciphertexts are converted to the ciphertexts of the new set through the key switching process. Compared with using bootstrapping in BP16 [8], key switching is much more efficient.

  The homomorphic computation process is the same as CKKS17, while replacing the evaluation key with the joint evaluation key.
- $\mathsf{AMCMK.Add}(\mathbf{ct}, \mathbf{ct}')$: Input the ciphertexts $\mathbf{ct}$ and $\mathbf{ct}'$ of the l-th level, and output the ciphertext $\mathbf{ct}_{\text{add}} = \mathbf{ct} + \mathbf{ct}' (\text{mod} q_l)$.
- $\mathsf{AMCMK.Mult}_{\overline{\mathbf{ks}_{set}}}(\mathbf{ct}, \mathbf{ct}')$: Run $\mathsf{CKKS.Mult}_{\overline{\mathbf{ks}_{set}}}(\mathbf{ct}, \mathbf{ct}')$.

$\mathsf{AMCMK.Dec}((sk_1, ..., sk_k), \mathbf{c})$**:** Input the ciphertext $\mathbf{c}$ of l-th level. Output $m' = \langle \mathbf{c}, sk_1 + ... + sk_k \rangle (\text{mod} q_l)$.

4.3 Scheme analysis

Security analysis. Based on the LWE assumption, this scheme can achieve IND-CPA security, under the cyclic security assumption. The biggest difference between this scheme and the CKKS17 scheme lies in the generation process of the joint public key $\overline{\mathbf{b}} = \mathbf{b}_1 + ... + \mathbf{b}_k$. The joint public key is the accumulation of multiple user public keys $\mathbf{b}_i$. According to the LWE assumption, $\mathbf{b}_i$ is indistinguishable from a uniform distribution, so joint public keys $\overline{\mathbf{b}}$ is indistinguishable from a uniform distribution. Moreover, the encryption and decryption operations and homomorphic operation functions of this scheme are the same as the TFHE. So our scheme is IND-CPA security like CKKS17.

Correctness analysis. Whether the ciphertext can be decrypted correctly depends on the size of the error in the ciphertext. Following the expression of CKKS17, we analyze the works of the main functions and the growth of the error . Analysis shows that the noise of our scheme is about $\sqrt{k}$ times large than CKKS17, which is much less than about k^2 times in CDKS19. The details are as follows:

Let $||a||_\infty^{can}$ denote the infinite normal form of $a(\zeta)$ (the inner product of the coefficients of and vectors $(1, \zeta_M, ..., \zeta_M^{N-1})$) obtained by normal embedding of polynomial $a(X) \in R = \mathbb{Z}[X]/(\Phi_M(X))$. 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 $\mathbf{ct} \in \mathcal{R}_q^2$, the scheme can decrypt correctly if the increment factor $\Delta > N + 2B$, where $< \mathbf{ct}, \mathbf{sk} >= m + e (\text{mod} q_L)$, B is the upper bound of $||e||_\infty^{can}$. The error growth of important functions is shown in the following lemmas.

Lemma 1 ([12]). *Let* $\mathbf{ct} \leftarrow \mathsf{CKKS.Enc}_{pk}(m)$ *be an encryption of* $m \in \mathcal{R}$ *with public key pk and* $e \in \mathcal{R}$*, then* $< \mathbf{ct}, \mathbf{sk} >= m + e (\text{mod} q_L)$*, where* $||e||_\infty^{can} \leqslant B_{clean}$*, such that* $B_{clean} = 8\sqrt{2}\sigma N + 6\sigma\sqrt{N} + 16\sigma\sqrt{hN}$.

Lemma 2. *Let* $\mathbf{ct} \leftarrow \mathsf{CKKS.Enc}_{\overline{pk}}(m)$ *denote the ciphertext of* $m \in R$ *encrypted by the accumulated public key* $\overline{pk}$*, for a certain set* $e \in \mathcal{R}$*, there is* $< \mathbf{ct}, (1, \overline{s}) >=$

$m + e(\mathrm{mod} q_L)$, *where* $\overline{pk} = (b_1 + ... + b_k, a)$, $||e||_\infty^{can} \leqslant B_{sclean}$ *and* $B_{sclean} = 8\sqrt{2k}\sigma N + 6\sigma\sqrt{N} + 16\sigma k\sqrt{hN}$.

Proof. Define $\overline{e} = e_1 + ... + e_k$, $\overline{s} = s_1 + \cdots s_k$, and $\overline{b} = b_1 + ... + b_k$. For the ciphertext $\mathbf{ct} = r \cdot (\overline{b}, a) + (m + e'_0, e'_1) (\mathrm{mod} q)$, where $b_i = -as_i + e_i (\mathrm{mod} q_L)$. 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 $e_i \leftarrow DG_{q_L}(\sigma^2)$, the expression of error e and the upper limit of $||e||_\infty^{can}$ is as follows:

$e =< \mathbf{ct}, (1, \overline{s}) > -m =< r \cdot (\overline{b}, a) + (m + e'_0, e'_1) , (1, \overline{s}) > -m$

$=< r \cdot (\overline{b}, a), (1, \overline{s}) > + < (e'_0, e'_1), (1, \overline{s}) >$

$= r\overline{e} + e'_0 + e'_1\overline{s},$

$||e||_\infty^{can} = ||r\overline{e} + e'_0 + e'_1\overline{s}||_\infty^{can}$

$\leqslant ||r\overline{e}||_\infty^{can} + ||e'_0||_\infty^{can} + ||e'_1\overline{s}||_\infty^{can}$

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

Lemma 3. *Let* $\overline{\mathbf{ks}_{set}}$ *be the accumulated switch-key,* $\overline{\mathbf{ks}_{set,i}}$ *be one element of* $\overline{\mathbf{ks}_{set}}$*, then ,* $< \overline{\mathbf{ks}_{set,i}}, (1, \overline{s}) >= P^{-1} \cdot s_i\overline{s} + e_{\overline{\mathbf{ks}_{set,i}}} (\mathrm{mod}\, q_L)$*, where* $||e_{\overline{\mathbf{ks}_{set}}}||_\infty^{can} \leqslant B_{\overline{\mathbf{ks}_{set}}} = \sqrt{k}||e_{\overline{\mathbf{ks}_{set,i}}}||_\infty^{can}$ *and* $||e_{\overline{\mathbf{ks}_{set,i}}}||_\infty^{can} \leqslant\leqslant 8\sqrt{2k}\sigma N + 16\sigma k\sqrt{hN} + B_{\overline{\mathbf{ks}_{se}}} = 8B_{ks}B_{sclean}\sqrt{dN/3}$.

Proof. For the partial refresh key $\overline{\mathbf{ks}_{s_i \to \overline{s}}} \leftarrow \mathsf{CKKS.Enc}_{\overline{pk}_{evk}, Pq_L}(P \cdot s_i)$, the partial rotation key $\overline{\mathbf{rk}_{i,r}} \leftarrow \mathsf{CKKS.Enc}_{\overline{pk}_{evk}, Pq_L}(\kappa_{5^r}(s_i))$ and the partial conjugate key $\overline{\mathbf{ck}_i} \leftarrow \mathsf{CKKS.KSGen}_{\overline{pk}_{evk}, Pq_L}(\kappa_{-1}(s_i))$, we suppose $||e||_\infty^{can} \leqslant B_{sclean}$.

(1) calculate the error of $e_{\overline{\mathbf{ks}_{set,i}}}$.

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

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

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

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

$$||e_{\overline{\mathbf{ks}_{set,i}}}||_\infty^{can} = ||r_i\overline{e'} + e_{i,2} \cdot \overline{s} + \left[\mathbf{g}^{-1}(\overline{b'})\right]_{1\times d} \cdot [\mathbf{e}_{sclean}]_{d\times 1} (\mathrm{mod} Pq_L)||_\infty^{can}$$
$$\leqslant 8\sqrt{2k}\sigma N + 16\sigma k\sqrt{hN} + 8B_{ks}B_{sclean}\sqrt{dN/3}$$

.

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

$$\overline{\mathbf{ks}_{set}} \begin{bmatrix} 1 \\ \overline{s} \end{bmatrix} \overset{\overline{\mathbf{ks}_{set}} := \sum_{i=1}^k \overline{\mathbf{ks}_{set,i}}}{=\!=} \sum\nolimits_{i=1}^k \overline{\mathbf{ks}_{set,i}} \begin{bmatrix} 1 \\ \overline{s} \end{bmatrix}$$

,

$$= \sum\nolimits_{i=1}^k P \cdot s_i\overline{s} + e_{\overline{\mathbf{ks}_{set,i}}} = P\overline{s} \cdot \overline{s} + \sum\nolimits_{i=1}^k e_{\overline{\mathbf{ks}_{set,i}}}$$

$$||e_{\overline{\mathbf{ks}_{set}}}||_\infty^{can} = ||\sum\nolimits_{i=1}^k e_{\overline{\mathbf{ks}_{set,i}}}||_\infty^{can} \leqslant \sqrt{k}||e_{\overline{\mathbf{ks}_{set,i}}}||_\infty^{can}.$$

Lemma 4 ([12]). *Let* $\mathbf{ct}' \leftarrow RS_{l\to l'}(\mathbf{ct})$*(where* $\mathbf{ct} \in \mathcal{R}_q^2$ *), for* $e \in \mathcal{R}$*, there is* $< \mathbf{ct}', \mathbf{sk} >= \frac{q_{l'}}{q_l} < \mathbf{ct}, \mathbf{sk} > +e(\mathrm{mod} q_{l'})$*, where* $||e||_\infty^{can} \leqslant B_{rs}$*,* $B_{rs} = \sqrt{N/3}(3 + 8\sqrt{h})$.

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

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

Lemma 6. *Let* $\mathbf{ct}' \leftarrow KS_{\overline{\mathbf{ks}_{refresh}}}(\mathbf{ct})$*,* $\mathbf{ct} \in \mathcal{R}_q^2$ *corresponds to the secret key* $\mathbf{sk}$*. Let*

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

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

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

These lemmas show that the noise is about $\sqrt{k}$ times large than CKKS17, which is much less than about k^2 times in CDKS19.

5 Construction of TFHE-type FCMKFHE

TFHE-type FHE scheme has the fastest bootstrapping operation, which is suitable for homomorphic logic operations, and has important applications in privacy computing systems, privacy neural networks, etc. In this chapter, we will construct an efficient compact TFHE-type CMKFHE, called AMTMK. Our contributions are as follows:

- We solve the problem of joint secret keyspace being mismatched with TFHE by proposing a variant of TFHE which support non-binary secret. TFHE-type FHE scheme requires that the secret keys should be taken from $\{0,1\}^n$, but the joint secret key is the accumulation of secret keys in the user set, which does not match the requirement. Our solution is: On the one hand, since the accumulation operation of binary secret keys in $\mathbb{T} := \mathcal{R}/\mathbb{Z}$ is meaningless, we extend $\mathbb{T} := \mathcal{R}/\mathbb{Z}$ to $\mathbb{Z}_q$, which is following the implementation of TFHE. On the other hand, we use the homomorphic addition boolean circuit to realize the accumulation of the secret key $\overline{\mathbf{s}} = \mathbf{s}_1 + ... + \mathbf{s}_N$, where $\mathbf{s}_i \in \mathbb{B}^n$, denote $\overline{\mathbf{s}}_{bin}$ as the binary representation of the joint private key, which meets the requirement of TFHE scheme. The homomorphic addition boolean circuit can be construct by constructing Half-Adder, Full-Adder and k bits homomorphic adder. In addition to the above methods, the very recent TFHE variant scheme[22] that supports arbitrary secret key distributions can be adapted to solve the problem of mismatching secret keyspace.

– We propose a method to generate the joint bootstrapping key by the public key. The bootstrapping key $\mathsf{RGSW.Enc}_{\mathbf{s}''}(\mathbf{s}_i)$ is encrypted by a special designed symmetric RGSW scheme, whose ciphertext structure of the scheme does not match well with the public key encryption scheme. While in CMKFHE scheme, bootstrapping key is encrypted by the joint public key, then a public key RGSW scheme is needed. We adapt the public key Ring-GSW scheme proposed by the CZW17, and the corresponding hybrid homomorphic multiplication operation in the LZY+19[23] to fulfill the bootstrapping in TFHE.

5.1 Basic scheme tools of AMTMK

This section introduces the underlying LWE scheme and Ring-GSW scheme of the AMTMK scheme.

$\mathsf{LWE.Setup}(1^\lambda) \to pp^{\mathsf{LWE}} = (n, \chi, \alpha, B_{ks}, d_{ks}, q, \mathbf{B})$: Input security parameter λ, generate LWE dimension n, key distribution χ_{key}, noise distribution χ_{err} with parameter α, decomposition base B_{ks}, decomposition degree d_{ks}, modulus q, matrix $\mathbf{B} \in \mathbb{Z}_q^{m\times n}$ shared by all users, output $pp^{\mathsf{LWE}} = (n, \chi_{key}, \alpha, B_{ks}, d_{ks}, q, \mathbf{B})$.

$\mathsf{LWE.KeyGen}(pp) \to \{pk, sk\}$: Select the private key $\mathbf{s} \in \mathbb{B}^n$, and calculate the public key $\mathbf{A} = \left[\mathbf{b} = -\mathbf{Bs} + \mathbf{e} \,|\mathbf{B}\right] \in \mathbb{Z}_q^{m\times n}$,where $\mathbf{e} \leftarrow \chi_{err}$, output $pk = \mathbf{A}$, $sk = (1, \mathbf{s}) \in \mathbb{B}^{n+1}$. We denote the joint public key as $\overline{pk} := \overline{\mathbf{A}} = [\overline{\mathbf{b}}|\mathbf{B}] = [\mathbf{b}_1 + ... + \mathbf{b}_k|\mathbf{B}]$, $m = O(n \log n)$.

$\mathsf{LWE.Enc}(pk, m) \to \mathbf{ct} = (b, \mathbf{a}) \in \mathbb{Z}_q^{n+1}$: Select $\mathbf{r} \in \mathbb{B}^m$ at random and output $(b, \mathbf{a}) := \mathbf{rA} + (\frac{q}{4}m, 0..., 0) + \mathbf{e}'$, where $\mathbf{e}' \leftarrow \chi_{err}$.

To meet the requirement of public key encryption with homomorphic operation, we adapt the Ring-GSW scheme given by the CZW17.

$\mathsf{GSW.Setup}(1^\lambda) \to pp^{\mathsf{GSW}} = (N, \chi_{key}, \alpha, B, d, \mathbf{y}, q)$: Input security parameter λ, generate a polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$, where the dimension is N, key distribution χ_{key}, noise distribution χ_{err} with parameter α, decomposition base B, decomposition degree d, matrix $\mathbf{y} \in \mathbb{Z}_q^{m\times n}$ shared by all users, output common parameter $pp^{\mathsf{GSW}} = (N, \chi_{key}, \alpha, B, d, \mathbf{y}, q)$.

$\mathsf{GSW.KeyGen}(\mathbf{params}) \to (pk, sk)$: Select $z \leftarrow \chi_{key}$, generate the private key $sk \leftarrow (1, -z)$. Randomly select $\mathbf{a} \xleftarrow{U} \mathcal{R}_q^{2d}$ and $\mathbf{e} \leftarrow \chi_{err}^{2d}$, denote the public key as $pk := [\mathbf{b} = \mathbf{a}z + \mathbf{e}, \mathbf{a}] \in \mathcal{R}_q^{2d\times 2}$.

$\mathsf{GSW.Enc}_{pk}(\mu)$: Select $r \leftarrow \chi_{enc}$ and $\mathbf{E} := [\mathbf{e}_0|\mathbf{e}_1] \leftarrow \chi_{err}^{2d\times 2}$. Output ciphertext $\mathbf{C} = r[\mathbf{b}, \mathbf{a}] + 2\mathbf{E} + \mu\mathbf{G} \in \mathcal{R}_q^{2d\times 2}$.

To meet the requirement of hyper homomorphic operation, we adapt the hyper homomorphic multiplication operation in LZY+19.

$\mathsf{RLWE.Enc}(z, \mu)$: Input the plaintext $\mu \in \mathcal{R}_2$, secret key $sk = (1, -z)$, output ciphertext $\mathbf{c} = (az + 2e + \mu, a) \in \mathcal{R}_q^2$.

$\mathsf{LWE.SampleExtract}(\mathbf{a}'', b'') \to (\mathbf{a}', b') \in \mathbb{Z}_q^{kN+1}$: $(\mathbf{a}'', b'')$ is an instance of $\mathsf{TLWE}_{\mathbf{s}''}(\mu)$ with secret key $\mathbf{s}'' \in R^k$, let $s' = \mathsf{KeyExtract}(\mathbf{s}'' \in R^k) := (coefs(\mathbf{s''}_1(X)$ $, ..., coefs(\mathbf{s''}_k(X))) \in \mathbb{Z}^{kN}$ and LWE instance $(\mathbf{a}', b') \in \mathbb{Z}_q^{kN+1}$, where $\mathbf{a}' = \{coefs(\mathbf{a}''_\mathbf{1}(\frac{1}{X}), ..., coefs(\mathbf{a}''_\mathbf{k}(\frac{1}{X}))\}$, $b' = b''_0$ is the constant term of b''.

$\mathsf{LWE.KSGen}(\mathbf{t}_i, \overline{pk}) \rightarrow \overline{\mathbf{KS}_j} = [\mathbf{b}_j|\mathbf{A}_j] \in \mathbb{Z}_q^{d_{ks}\times(n+1)}, j \in [N]$: Input LWE private key $\mathbf{t}_i \in \mathbb{Z}^N$, the joint public key $\overline{pk} = \overline{\mathbf{A}} = [\overline{\mathbf{b}}|\mathbf{B}] = [\mathbf{b}_1 + ... + \mathbf{b}_k|\mathbf{B}] \in \mathbb{Z}_q{}^{m\times n}$, output the switching key $\overline{\mathbf{KS}_i} = \{\overline{\mathbf{KS}_{i,j}}\}_{j\in[N]}$ from $\mathbf{t}_i \in \mathbb{Z}^N$ to $\bar{\mathbf{s}}$, where $\overline{\mathbf{KS}_{i,j}} = \mathsf{LWE.Enc}_{\overline{pk}}(t_{i,j}B_{ks}{}^j)$, i.e. $\overline{\mathbf{KS}_{i,j}} = \mathbf{R}_{ks}\overline{\mathbf{A}} + (\mathbf{e}_{i,j}, 0, ..., 0) + (t_{i,j}, 0, ..., 0) \cdot \mathbf{g}_{ks}$, $\mathbf{R}_{ks} \in \mathbb{Z}_q^{d_{ks}\times m}$, $\mathbf{g}_{ks} = (B_{ks}, B_{ks}^2, ..., B_{ks}^{d_{ks}})^T$, $j \in [N]$.

$\mathsf{LWE.Switch}(\overline{\mathbf{ct}} \in \mathbb{Z}_q^{N+1}, \{\overline{\mathbf{KS}_{i,j}}\}_{i\in[N],j\in[w]}) \rightarrow \overline{\mathbf{ct}'} \in \mathbb{Z}_q^{n+1}$: Input joint ciphertext $\mathbf{ct} = (b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$ and joint switching key $\overline{\mathbf{KS}_{\overline{Set}}}$. Run $(b', \mathbf{a}') = \sum_{j=1}^{N} \mathbf{g}_{ks}^{-1}(a_j)\overline{\mathbf{KS}_{\overline{Set},j}}$, where $\mathbf{g}_{ks}^{-1}(a_j)$ is the decomposition of a_j based on B_{ks}. Output ciphertext $\overline{\mathbf{ct}'} = (b' + b, \mathbf{a}') \in \mathbb{Z}_q^{n+1}$.

5.2 Homomorphic adder

The principle of the homomorphic adder circuit is the same as that of the common ripple carry adder circuit, except that the plaintexts in the adder are replaced by the ciphertexts, and the operations are replaced by the homomorphic operations on the ciphertext. In our scheme, the homomorphic adder is constructed by homomorphic addition and multiplication of RGSW ciphertexts. We construct the k bits homomorphic addition boolean circuit by constructing Half-Adder, Full-Adder. The noise analysisi of Homomorphic adder is present in appendix B.

Half-Adder: Input two ciphertexts $\mathsf{RGSW}(x)$, $\mathsf{RGSW}(x)$ corresponding to the bit x and y. Output carry ciphertext $\mathsf{RGSW}(c) = \mathsf{RGSW}(x) \cdot \mathsf{RGSW}(y)$, and the sum ciphertext $\mathsf{RGSW}(s) = \mathsf{RGSW}(x) + \mathsf{RGSW}(y)$, where $\cdot$ means homomorphic multiplication and $+$ means homomorphic addition.

Full-Adder: Input two ciphertexts $\mathsf{RGSW}(x)$, $\mathsf{RGSW}(x)$ corresponding to the bit x and y, and a carry ciphertext $\mathsf{RGSW}(c_{in})$. Output carry ciphertext $\mathsf{RGSW}(c_{out}) = \mathsf{RGSW}(x) \cdot \mathsf{RGSW}(y) + \mathsf{RGSW}(c_{in}) \cdot (\mathsf{RGSW}(x) + \mathsf{RGSW}(y))$, sum ciphertext $\mathsf{RGSW}(s_{out}) = \mathsf{RGSW}(c_{in}) + \mathsf{RGSW}(x) + \mathsf{RGSW}(y)$.

HomAdd$_l$: Input two sequence ciphertexts $\mathsf{RGSW}(x_{l-1}), \ldots, \mathsf{RGSW}(x_0)$ and $\mathsf{RGSW}(y_{l-1}), \ldots, \mathsf{RGSW}(y_0)$, corresponding to the binary representation of two number x, y. Firstly, compute $(\mathsf{RGSW}(c_0), \mathsf{RGSW}(s_0)) = \mathsf{Full-Adder}(\mathsf{RGSW}(x_0), \mathsf{RGSW}(y_0), 0)$; Secondly, for $i = 0$ to $l-2$, run $\mathsf{RGSW}(c_{i+1}), \mathsf{RGSW}(s_{i+1}) = Full-Adder(\mathsf{RGSW}(x_i), \mathsf{RGSW}(y_i), \mathsf{RGSW}(s_i))$; Finally, output ciphertext $(\mathsf{RGSW}(c_{l-1}), \mathsf{RGSW}(s_{l-1}), \ldots, \mathsf{RGSW}(s_0))$.

5.3 Construction

The underlying LWE scheme and Ring-GSW scheme of the AMTMK scheme is in [27]. We construction AMTMK as follow:

$\mathsf{AMTMK.Setup}(1^\lambda) \rightarrow pp = (pp^{\mathsf{LWE}}, pp^{\mathsf{GSW}})$: Run $\mathsf{LWE.Setup}(1^\lambda) \rightarrow pp^{\mathsf{LWE}} = (\eta, \chi, \alpha, B_{ks}, d_{ks}, \mathbf{B}, q)$,$\mathsf{GSW.Setup}(1^\lambda) \rightarrow pp^{\mathsf{GSW}} = (N, \chi_{key}, \alpha, \mathbf{B}, \mathbf{d}, \mathbf{y}, \mathbf{q})$, where $\mathbf{B}$, $\mathbf{y}$ are common random variables.

$\mathsf{AMTMK.KeyGen}(pp) \rightarrow (pk_i, sk_i, pk_{BK,i}, sk_{BK,i})$: User i generates public key and secret key. Run $\mathsf{LWE.KeyGen}(pp) \rightarrow \{pk_i = \mathbf{A}_i, sk_i = \mathbf{s}_i\}$,$\mathsf{GSW.KeyGen}(pp) \rightarrow \{pk_{BK,i} = \mathbf{Z}_i, sk_{BK,i} = z_i\}$.

When all parties have completed the program $\mathsf{AMTMK.KeyGen}(pp)$, the evaluator runs the algorithm of evaluation key generation.

$\mathsf{AMTMK.EvalKeyGen}(pp, sk_i, \{pk_1, \cdots, pk_k\}) \rightarrow \{\overline{pk}, \overline{\mathbf{KS}_i}, \overline{\mathbf{BK}_i}\}$: User authorization to the user set (generating the evaluation keys).

- $\mathsf{JPKGen}(pp, pk_1, , pk_k)$: Evaluator accumulate the public key. Given the public keys $\{\mathbf{b}_1, ..., \mathbf{b}_k\}$ and bootstrapping public keys $\{\mathbf{d}_1, ..., \mathbf{d}_k\}$ of k parties, evaluator generate and send the *joint public key* $\overline{pk} := \left[\mathbf{b}_1 + ... + \mathbf{b}_k \ \mathbf{B}\right] \in \mathbb{Z}_q^{m\times n}$ and the *joint bootstrapping public key* $\overline{pk_{BK}} := \overline{\mathbf{Z}} = \left[\mathbf{d}_1 + ... + \mathbf{d}_k \ \mathbf{y}\right] \in \mathbb{Z}_q^{2d\times 2}$ to all user.
- $\mathsf{PEvalKeyGen}(pp, \{sk_i, sk_{BK,i}\}, \{\overline{pk}, \overline{pk_{BK}}\})$: Given joint public key $\overline{pk}$ and secret key sk_i, the user P_i output the $pevk_i = \{\overline{\mathbf{KS}_i}, \overline{\mathbf{ks}_{s_i\rightarrow\overline{s}}}\}$.
  - $\mathsf{PSwitchKeyGen}(pp, sk_{BK,i}, \overline{pk})$: P_i generate the partial switching key. Input $\overline{pk}$ and the secret key z_i of the RGSW ciphertext, let $\mathbf{t}_i =: (z_{i,0}, -z_{i,w-1}, ..., -z_{i,1}) \in \mathbb{B}^N$, and output the the *partial switching key (psk)* $\overline{\mathbf{KS}_i} = \mathsf{LWE.KSGen}(\mathbf{t}_i, \overline{pk})$ of user i.
  - $\mathsf{PBootKeyGen}(pp, sk_i, \overline{pk_{BK}})$: P_i generate the partial bootstrapping key. Input $\overline{pk_{BK}} = \overline{\mathbf{Z}}$ and the secret key $\mathbf{s}_i \in \mathbb{Z}^n$ of user i for LWE ciphertext. Output the *partial bootstrapping key (pbk)* $\overline{\mathbf{BK}_i} = \{\overline{\mathbf{BK}_{i,j}}\}_{j\in[n]}$, where $\overline{\mathbf{BK}_{i,j}} = \mathsf{RGSW.Enc}(s_{i,j}, \overline{\mathbf{Z}})$, $i \in [k]$, $j \in [n]$.
- $\mathsf{AggEvalKey}(pp, \{pevk_i\}_{i\in[k]})$: Given the partial evaluation key $\{pevk_i\}_{i\in[k]}$, the evaluator output $evk = \{\overline{\mathbf{KS}_{\overline{Set}}}, \overline{\mathbf{BK}_{\overline{Set}}}, \overline{\mathbf{BK}_{i,j}}\}$.:
  - $\mathsf{AggSwitchKey}(pp, \overline{\mathbf{KS}_i})$: Evaluator generate the joint switching key $\overline{\mathbf{KS}_{\overline{Set}}} = \{\sum_{i=1}^{k} \overline{\mathbf{KS}_{i,j}}\}_{j\in[N]}$.
  - $\mathsf{AggBootKey}(pp, \overline{\mathbf{ks}_{s_i\rightarrow\overline{s}}})$: Evaluator generate the joint bootstrapping keys in cloud. Run The cloud sever uses the $\overline{\mathbf{BK}_i}$ to generate the *joint bootstrapping key* $\overline{\mathbf{BK}_{\overline{Set}}} = \{\overline{\mathbf{BK}_{bit_{l-1}(\overline{Set},j)}}, \cdots, \overline{\mathbf{BK}_{bit_0(\overline{Set},j)}}\} = \mathsf{HomAddk}(\overline{\mathbf{BK}_{1,j}}, ..., \overline{\mathbf{BK}_{k,j}})$, where $j \in [n]$, $l = \lceil \log(k) \rceil$. $\mathsf{HomAddk}(,)$ is a homomorphic addition circuit for k-bit. The details are shown in [27].

$\mathsf{AMTMK.Enc}(pk, \mu)$: Run $\mathsf{LWE.Enc}(pk, \mu) \rightarrow \mathbf{ct} = (b, \mathbf{a}) \in \mathbb{Z}_q^{n+1}$.

$\mathsf{AMTMK.Dec}((sk_1, ..., sk_k), \mathbf{ct})$: Input the ciphertext $\mathbf{ct}$. Output m' such that $= \langle \mathbf{ct}, (1, sk_1 + \cdots + sk_k)\rangle - \frac{q}{4}m (\mathrm{mod} q)$ be smallest.

The bootstrapping process has two purposes: one is to refresh the noise in the ciphertext, and the other is to use the evaluation key to refresh the public key of the ciphertexts to the user set.

$\mathsf{AMTMK.Boot}(\mathbf{c}, evk)$:

- Ciphertext refresh. Input ciphertext $\mathbf{c} = (b', \mathbf{a}') \in \mathbb{Z}_q^{n+1}$, and the bootstrapping key $\{\overline{\mathbf{BK}_{bit_{l-1}(\overline{Set})}}, ..., \overline{\mathbf{BK}_{bit_0(\overline{Set})}}\}$ or $\overline{\mathbf{BK}_i}$, the evaluator runs:
  - Input the ciphertext $\mathbf{c} = (b', \mathbf{a}') \in \mathbb{Z}_q^{n+1}$, output $\tilde{b} = \lfloor 2N \cdot b'/q \rceil$, $\tilde{\mathbf{a}} = \lfloor 2N \cdot \mathbf{a}'/q \rceil$ and the (joint) *bootstrapping key*

$$\overline{\mathbf{BK}} = \begin{cases} \overline{\mathbf{BK}_i} & \mathbf{c} \text{ corresponds to the secret key } \mathbf{s}_i \\ \overline{\mathbf{BK}_{\overline{Set'}=\{s_1,...,s_{k'}\}}} & \mathbf{c} \text{ corresponds to the secret key } (\mathbf{s}_1 + ... + s_{k'}). \end{cases}$$

When the secret key corresponding to $\mathbf{c}$ is $\mathbf{s}_i$, the *bootstrapping key* can refresh the ciphertext of a single user; when the secret key corresponding to $\mathbf{c}$ is $\mathbf{s}_1 + ... + \mathbf{s}_{k'}$, *bootstrapping key* can refresh the ciphertext of the old user set.

- Initialize the RLWE ciphertext $\mathbf{ACC} = (-\frac{q}{8}h(X)\cdot X^{\tilde{b}}, \mathbf{0})$, where $h(X) = 1 + X + ... + X^{\frac{N}{2}-1} - X^{\frac{N}{2}} - ... - X^N$. Let $\tilde{\mathbf{a}} = (\tilde{a}_j)_{j\in[n]}$, for j=1 to n, run the following process.
  - $\mathbf{ACC} = \mathrm{CMux}(\overline{\mathbf{BK}_{bit_0(\overline{Set},j)}}, X^{a_j}\mathbf{ACC}, \mathbf{ACC})$;
  - $\cdots$
  - $\mathbf{ACC} = \mathrm{CMux}(\overline{\mathbf{BK}_{bit_{l-1}(\overline{Set},j)}}, X^{(2^{l-1})a_j}\mathbf{ACC}, \mathbf{ACC})$.

  The $\mathrm{CMux}(\mathbf{C}, \mathbf{d}_1, \mathbf{d}_0) = \mathbf{C} \boxdot (\mathbf{d}_1 - \mathbf{d}_0) + \mathbf{d}_0$. The $\boxdot$ is a hybrid homomorphic multiplication of RGSW ciphertext and BGV ciphertext(see CGGI16/LZY+19 for more details).
- Output $\mathbf{ACC} \leftarrow (\frac{q}{8}, \mathbf{0}) + \mathbf{ACC}(\mathrm{mod}q)$.

– Switching the RLWE ciphertext $\mathbf{ACC}$ to the LWE ciphertext.
- Input the ciphertext $\mathbf{ACC} = (c_0, c_1) \in \mathcal{R}_q^2$, output LWE ciphertext $(b'', \mathbf{a}'') = \mathsf{LWE.SampleExtract}(\mathbf{ACC}) \in \mathbb{Z}_q^{kN+1}$.
- computer and output the ciphertext $\overline{\mathbf{ct}} \leftarrow \mathsf{LWE.Switch}(\overline{\mathbf{ct}}'', \overline{\mathbf{KS}_{\overline{Set}}})$.

Analysis. Based on the LWE assumption, this scheme can achieve IND-CPA security, under the cyclic security assumption. The difference between this scheme and the TFHE scheme lies in the generation process of the joint public key $\overline{\mathbf{b}} = \mathbf{b}_1 + ... + \mathbf{b}_k$. According to the LWE assumption, $\mathbf{b}_i$ is indistinguishable from a uniform distribution, so joint public keys $\overline{\mathbf{b}}$ is indistinguishable from a uniform distribution. The correctness follows the CGGI17. The detailed analysis is shown in Appendix C. Our AMTMK scheme has the advantages: The size of ciphertext, evaluation key, and joint switching key are independent of the number of users; the bootstrapping operation is almost as efficient as TFHE; users of our scheme can decide whether to participate in a computation task with a given user set, by authentication to the user set(generate the corresponding partial evaluation key).

6 CONCLUSION

In this paper, we propose a general construction of MKFHE scheme with compact ciphertext. We show how to construct a compact MKFHE scheme that supports the homomorphic encryption of ring elements and is friendly to floating-point numbers, and construct a compact MKFHE scheme that supports efficient bootstrapping.

References

1. Albrecht, M., Bai, S., Ducas, L.: A subfield lattice attack on overstretched ntru assumptions. In: Annual International Cryptology Conference. pp. 153–178. Springer (2016)

2. Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., Wichs, D.: Multiparty computation with low communication, computation and interaction via threshold fhe. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 483–501. Springer (2012)
3. Badrinarayanan, S., Jain, A., Manohar, N., Sahai, A.: Secure mpc: laziness leads to god. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 120–150. Springer (2020)
4. Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: Annual International Cryptology Conference. pp. 565–596. Springer (2018)
5. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: Annual Cryptology Conference. pp. 868–886. Springer (2012)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325 (2012)
7. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)
8. Brakerski, Z., Perlman, R.: Lattice-based fully dynamic multi-key fhe with short ciphertexts. In: Annual International Cryptology Conference. pp. 190–213. Springer (2016)
9. Chen, H., Chillotti, I., Song, Y.: Multi-key homomorphic encryption from tfhe. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 446–472. Springer (2019)
10. Chen, H., Dai, W., Kim, M., Song, Y.: Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 395–412 (2019)
11. Chen, L., Zhang, Z., Wang, X.: Batched multi-hop multi-key fhe from ring-lwe with compact ciphertext extension. In: Theory of Cryptography Conference. pp. 597–627. Springer (2017)
12. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 409–437. Springer (2017)
13. Chillotti, I., Gama, N., Georgieva, M., Izabachene, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: international conference on the theory and application of cryptology and information security. pp. 3–33. Springer (2016)
14. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 377–408. Springer (2017)
15. Chongchitmate, W., Ostrovsky, R.: Circuit-private multi-key fhe. In: IACR International Workshop on Public Key Cryptography. pp. 241–270. Springer (2017)
16. Clear, M., McGoldrick, C.: Multi-identity and multi-key leveled fhe from learning with errors. In: Annual Cryptology Conference. pp. 630–656. Springer (2015)
17. Doröz, Y., Hu, Y., Sunar, B.: Homomorphic aes evaluation using the modified ltv scheme. Designs, Codes and Cryptography **80**(2), 333–358 (2016)
18. Dov Gordon, S., Liu, F.H., Shi, E.: Constant-round mpc with fairness and guarantee of output delivery. In: Annual Cryptology Conference. pp. 63–82. Springer (2015)

19. Ducas, L., Micciancio, D.: Fhew: bootstrapping homomorphic encryption in less than a second. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 617–640. Springer (2015)
20. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive (2012)
21. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 169–178 (2009)
22. Lee, Y., Micciancio, D., Kim, A., Choi, R., Deryabin, M., Eom, J., Yoo, D.: Efficient fhew bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. Cryptology ePrint Archive (2022)
23. Li, N., Zhou, T., Yang, X., Han, Y., Liu, W., Tu, G.: Efficient multi-key fhe with short extended ciphertexts and directed decryption protocol. IEEE Access **7**, 56724–56732 (2019)
24. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Proceedings of the forty-fourth annual ACM symposium on Theory of computing. pp. 1219–1234 (2012)
25. Mukherjee, P., Wichs, D.: Two round multiparty computation via multi-key fhe. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 735–763. Springer (2016)
26. Peikert, C., Shiehian, S.: Multi-key fhe from lwe, revisited. In: Theory of Cryptography Conference. pp. 217–238. Springer (2016)
27. Zhou, T., Zhang, Z., Chen, L., Che, X., Liu, W., Yang, X.: Multi-key fully homomorphic encryption scheme with compact ciphertext. Cryptology ePrint Archive (2021)

Appendix

A CKKS17 scheme

$\mathsf{CKKS.Setup}(1^\lambda) \to pp = (N, \chi_{key}, \chi_{err}, \chi_{enc}, L, q_l)$: Input the parameters λ, and select an integer N to the power of 2. Let $\chi_{key}, \chi_{err}, \chi_{enc}$ be the distribution of secret key, error and encryption process on $\mathcal{R} = \mathbb{Z}/(X^N + 1)$ respectively. Select prime p and the circuit layer L. The ciphertext modulus is $q_l = p^l$, where $1 \leq l \leq L$. Output common parameter $pp = (N, \chi_{key}, \chi_{err}, \chi_{enc}, L, q_l)$.

$\mathsf{CKKS.KeyGen}(params) \to (pk, sk, ks, rk_r, ck)$:

-$\mathsf{CKKS.PSKeyGen}(\mathbf{params}) \to (\mathbf{pk}, \mathbf{sk})$: Select $s \leftarrow \chi_{key}$, and let the secret key $sk \leftarrow (1, s)$. Select $a \leftarrow U(\mathcal{R}_{q_L})$ and the error $e \leftarrow \chi_{err}$. Set the public key $pk \leftarrow (b, a) \in \mathcal{R}^2_{q_L}$, where $b = -as + e \pmod{q_L}$.

-$\mathsf{CKKS.KSGen}(sk, s')$: Input $s' \in R$, and select $a' \leftarrow \mathcal{R}_{P \cdot q_L}$ and $e' \leftarrow \chi_{err}$. let the evaluated key be $evk \leftarrow (b', a') \in \mathcal{R}^2_{P \cdot q_L}$, where $b' = -a's + e' + Ps' (\mathrm{mod}\, P \cdot q_L)$.

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

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

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

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

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

$\mathsf{CKKS.Add}(\mathbf{ct}, \mathbf{ct}')$: Input the ciphertexts $\mathbf{ct}$ and $\mathbf{ct}'$ of the l-th level, and output the ciphertext $\mathbf{ct}_{\mathrm{add}} = \mathbf{ct} + \mathbf{ct}'(\mathrm{mod} q_l)$.

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

$\mathsf{CKKS.Mult}_{\mathbf{ks}}(\mathbf{ct}, \mathbf{ct}')$: Input the ciphertext $\mathbf{ct} = (c_0, c_1), \mathbf{ct}' = (c_0', c_1') \in \mathcal{R}_{q_l}^2$ of the l-th level, and output the ciphertext $\mathbf{ct}_{\mathrm{mult}} = (d_0, d_1) + \left\lfloor P^{-1} \cdot d_2 \cdot \mathbf{ks} \right\rceil$ (mod q_l).

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

$\mathsf{CKKS.Rescale}_{l \to l'} \to (\mathbf{ct})$: Input the ciphertext $\mathbf{ct}$ of the l-th level and the next level label l'. Output the ciphertext $\mathbf{ct}' = \left\lfloor p^{l'-l} \cdot \mathbf{ct} \right\rceil \in (\mathrm{mod} q_{l'})$.

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

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

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

B Noise analysis of Homomorphic adder

For convenience, we denote $\mathsf{RGSW}x, \overline{y}, \overline{c_in}, \overline{s}, \overline{c_out}$ as $\mathsf{RGSW}x, \mathsf{RGSW}y, \mathsf{RGSW}c_in, \mathsf{RGSW}s$, $\mathsf{RGSW}c_out$, respectively.

1. For the homomorphic multiplication between TGSW ciphertexts, we have

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

   where

$$k = 1, l = d, \beta = \tfrac{B_g}{2}, V_B = \beta^2, \mu_A \in 0, 1$$

   and ϵ is the var of gap round.
2. For the full-adder based on homomorphic multiplication between RGSW ciphertexts, we have

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

$$\begin{aligned} &Var(Err(\overline{C_{out}})) \leq (6dNV_B + 1)Var(Err(\overline{X})) + 2(1+N)\varepsilon^2 + Var(Err(\overline{C_{in}})) \\ &= (6dNV_B + 1)2dkN\beta^2 + 2(1+N)\varepsilon^2 + Var(Err(\overline{C_{in}})) \end{aligned}$$

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

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

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

For convenience of expression, we express it uniformly as:

$$Var(Err(HomAdd_l(output))) \leq l(6dNV_B+1)\sqrt{2dkN}\beta^2 + 2l(1+N)\varepsilon^2$$

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

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

Method 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

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

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

Output carry ciphertext $\mathsf{RGSW}(c_{out}) = \mathsf{RGSW}(x)\cdot\mathsf{RGSW}(y) + \mathsf{RGSW}(c_{in})\cdot(\mathsf{RGSW}(x)+\mathsf{RGSW}(y))$, sum ciphertext $\mathsf{RGSW}(s_{out}) = \mathsf{RGSW}(c_{in})+\mathsf{RGSW}(x)+\mathsf{RGSW}(y)$.

C Error analysis of AMTMK scheme

The decomposition basis is defined as B, and the decomposition degree is defined as d. Let $\epsilon^2 = \frac{1}{12B^2d}$ be the variance of a uniform distribution over $(-\frac{1}{2B^d}, \frac{1}{2B^d}]$. Define $|V_B| = \begin{cases} \frac{1}{12(B-1)} \text{ if } B \text{ is odd,} \\ \frac{1}{12(B+2)} \text{ if } B \text{ is even} \end{cases}$ as the uniformly distributed variance over $(-\frac{1}{2B^d}, \frac{1}{2B^d}]$. Also, define the parameters ϵ_{ks}^2, $V_{B_{ks}}$ and B_{ks} in the bootstrapping algorithm. Define the secret key distribution $\chi \in \{0,1\}^\omega$ and $\phi \in \{0,1\}^n$ on RGSW and LWE. Let $Var(e)$ be the variance of the random variable e over $\mathbb{Z}_q$. If e is a vector composed of random variables, then $Var(e)$ is the maximum variance of the vector.

Rounding error. Given $\tilde{b} = \lfloor 2N\cdot b' \rceil$ and $\tilde{\mathbf{a}} = \lfloor 2N\cdot \mathbf{a}' \rceil$, assuming that the each rounding of error obeys the random uniform distribution of Z, then the

variance of the overall rounding error of expression $(\tilde{b}-2N\cdot b')+<\tilde{\mathbf{a}}-2N\cdot\mathbf{a}',\bar{\bar{\mathbf{s}}}>$ is $\frac{1}{12}(1+n/2)$. **The initial error of the evaluation key**. The variance of error $\overline{\mathbf{KS_{i,j}}}$ is

$$Var(Err(\overline{\mathbf{KS}_{i,j}})) = Var(Err(\mathbf{R}_{ks}\overline{\mathbf{A}} + (t_{i,j},0,...,0)\cdot\mathbf{g}_{ks})) \leq mk\cdot\alpha^2.$$

The variance of error $\overline{\mathbf{BK_{i,j}}}$ is

$$Var(Err(\overline{\mathbf{BK}_{i,j}})) = Var(Err(\mathbf{R}\cdot\overline{\mathbf{Z}} + s_{i,j}\mathbf{h}) \leq 2dkN\cdot\beta^2.$$

According to CGGI16 scheme, the bootstrap error of AMTMK scheme is analyzed as follows. Let $\mathbf{d}_0$, $\mathbf{d}_1$ be TRLWE samples and let $C\in\mathsf{RGSW_s}(m)$. Then $msg(CMux(C,\mathbf{d}_1,\mathbf{d}_0)) = m\cdot msg(\mathbf{d}_1)+(1-m)\cdot msg(\mathbf{d}_0)$. And we have $||Err\left(CMux(C,\mathbf{d}_1,\mathbf{d}_0)\right)||_\infty \leq max(||Err(\mathbf{d}_0)||_\infty,||(\mathbf{d}_1)||_\infty)+\eta(\mathbf{C})$ where $\eta(C)=2dN\frac{B_g}{2}||Err(C)||_\infty+(k+1)\varepsilon$. So

$$Var(Err(CMux(C,\mathbf{d}_1,\mathbf{d}_0))) \leq \mathsf{max}(Var(Err(\mathbf{d}_0)),Var(Err(\mathbf{d}_1)))+\vartheta(C),$$

where $\vartheta(C) = 2dNV_BVar(Err(C))+(N+1)\varepsilon^2$. The accumulated process. The initial RLWE ciphertext is general, and its error is 0. All bootstrap keys $\{\overline{\mathbf{BK}_{bit_{l-1}(\overline{set},j)}},...,\overline{\mathbf{BK}_{bit_0(\overline{set},j)}}\}_{j\in[n]}$ 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 ln 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 $\mathbf{ct}=(b,\mathbf{a})\in\mathbb{Z}_q^{N+1}$ and accumulated key $\overline{\mathbf{KS}_{\overline{Set}}}=\{\sum_{i=1}^{k}\overline{\mathbf{KS}_{i,j}}\}_{j\in[N]}$, where $\overline{\mathbf{KS}_{i,j}}=\mathbf{R}_{ks}\overline{\mathbf{A}}+(\mathbf{e}_{i,j},0,...,0)+(t_{i,j},0,...,0)\cdot\mathbf{g}_{ks}$. Output the ciphertext $(b',\mathbf{a}')=\sum_{j=1}^{N}\mathbf{g}_{ks}^{-1}(a_j)\overline{\mathbf{KS}_{\overline{Set},j}}(mod1)$ and its error variance $Var(Err(\overline{\mathbf{ct}'}))=\frac{1}{2}\varepsilon^2N+d_{ks}V_{B_{ks}}N\alpha^2(1+m)+Var(Err(\mathbf{ct}))$. 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 $\frac{1}{2}\varepsilon^2N+d_{ks}V_{B_{ks}}N\alpha^2(1+m)+2dNV_B\cdot ln\{\sum_{i=1}^{l}(2^{i-1}i)(2dNV_B\cdot 2dkN\beta^2+(1+N)\varepsilon^2)+2dkN\beta^2\}+(N+1)ln\varepsilon^2$.