




# Pure Multi Key BGV Implementation\*

Justine Paillet<sup>1</sup><sup>a</sup>, Olive Chakraborty<sup>2</sup><sup>b</sup> and Marina Checri<sup>2</sup><sup>c</sup>

<sup>1</sup>*Hensoldt SAS France, 115 avenue de Dreux, Plaisir, France*

<sup>2</sup>*Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France*

**Keywords:** Homomorphic Encryption, Pure Multi-Key, Multi-User Frameworks, Cloud Computing.

**Abstract:** This paper offers an in-depth exploration of a Pure Multi-Key BGV Implementation. It provides a detailed analysis of the calculations involved for both the server and parties in the scheme, specifically focusing on the challenging and specific relinearisation of terms. Particular emphasis is given towards the extended RGSW multiplication and the complexity of addition, multiplication, and keyswitch key generation.

## 1 INTRODUCTION

In the rapidly evolving landscape of cryptography, ensuring the security and privacy of sensitive data has become paramount. With the rise of cloud computing, secure data processing techniques have garnered significant attention. Homomorphic encryption, a groundbreaking concept in the field of cryptography, allows computations to be performed on encrypted data without the need for decryption, thereby ensuring end-to-end security and privacy. Among the various homomorphic encryption schemes, the BGV scheme (Brakerski et al., 2014) stands out for its efficiency and versatility.

This paper delves into the realm of pure multi-key BGV implementation, a topic of great significance given the increasing reliance on secure computations in various domains such as finance, healthcare, and data analytics. A pure multi-key BGV encryption allows multiple parties to jointly perform computations on encrypted data without sharing their private keys. This collaborative approach is essential in scenarios where data privacy is non-negotiable, and parties must work together while preserving the confidentiality of their individual inputs.


### 1.1 Motivation


One of the modern questions linked to FHE is the possibility of creating multi-user cryptosystems. If it is possible to perform calculations on encrypted data, it could be interesting to be able to modify the data collectively. This is why thinking about a multi-key system could prove interesting.


A homomorphic multi-key system is one that allows several users, each with a pair of keys, to interact on an encrypted message. There are two main methods: threshold and multi-key. While each user has a private key, the number of keys used differs according to the type of scheme chosen. In the case of threshold, there is one public key common to all participants<sup>1</sup>.

As in asymmetric cryptography, encryption is done with the common public key. The direct consequence will be the size of the ciphertext, which will be said to be "normal", i.e. the size of a conventional ciphertext. With pure multi-key – the second multi-key approach – each party uses his or her own public key to encrypt, meaning there will be as many ciphertexts as public keys for encrypting a single piece of data. Although the global ciphertext is now heavier than the threshold ciphertext, the presence of a public key for each party means that the pure multi-key system is more scalable. Unlike the threshold approach, it is not necessary to rebuild the entire system to add or remove a party. Threshold schemes are normally designed to be decryptable if at least  $m$  parties participate in the decryption. This is not the case with pure multi-key schemes, which require all the parties

<sup>1</sup>Built using participants' public keys during a pre-calculation phase

<sup>a</sup> <https://orcid.org/0009-0009-6056-7766>

<sup>b</sup> <https://orcid.org/0000-0003-0396-908X>

<sup>c</sup> <https://orcid.org/0000-0002-0473-1164>

\* The first author contribution to this work was done while at CEA List. The third author's work was supported by the France 2030 ANR project ANR-22-PECY-003 SecureCompute.

to participate in the decryption. However, it is not uncommon to see the threshold requiring all parties.

## 1.2 Our Contributions

We have implemented a Multi-Key Homomorphic Encryption scheme using OpenFHE. This work presents a detailed exposition of the necessary theory and calculations, focusing mainly on refining the methodology for performing multiplication on extended RGSW ciphertexts by choosing the needed rows. Building upon this implementation, we present a complexity analysis and report computational time results.

## 1.3 Organization

The paper is organized as follows. Sec. 2 presents related work on existing FHE multi-user approaches. The next section, Sec 3 presents some preliminaries on FHE and the MKHE scheme. Details on our contributions are in Sec. 4. We conclude the paper with experimental results in Sec. 5.

## 2 RELATED WORKS

Multi-user approaches for implementing FHE schemes refer to Threshold Homomorphic Encryption (ThHE, also known as MultiParty HE) and Multi-Key Homomorphic Encryption (MKHE). They involve multiple keys during the decryption of ciphertexts and ensure that no single entity holds the decryption key (i.e. the private key).

ThHE schemes were introduced in (Asharov et al., 2012; Boneh et al., 2018; Chowdhury et al., 2022). These schemes allow a subset of the private key owners to decrypt (collectively) a message encrypted with a collective public key (computed from all the individual ones in a public way). The subset size is fixed and defined at setup. However, ThHE has a static setup and needs to remove and re-encrypt all the data each time a user is removed.

In contrast, MKHE schemes (described in (López-Alt et al., 2013; Doröz et al., 2016; Chen et al., 2017; Chen et al., 2019)) remove the need for a key setup by allowing the evaluation server to dynamically extend ciphertexts from encryption under individual keys to ones under the concatenation of several users keys. Then, all the private key owners have to collaborate for decryption. However, the increase in ciphertext size in the number of users results in a considerable increase in homomorphic operators' computational costs. For instance, a multi-key version of

TFHE (Chen et al., 2019) turns 2.19kB ciphertexts into 17.32kB ciphertexts for the same clear data when the number of users increases from 2 to 8. The computation time for homomorphic evaluation on ciphertexts increases even faster. The time for evaluating a NAND gate is 0.27 seconds for two users and becomes 7.16 seconds for eight users.

Recently, hybrid approaches have been proposed (Aloufi and Hu, 2019) to combine the advantages of both ThHE and MKHE. However, to the best of the authors' knowledge, hybrid approaches have so far been built only on BGV (Brakerski et al., 2014), which restricts the set of functionalities that may be implemented.

## 3 PRELIMINARIES

### 3.1 Notations

Let  $R$  be a ring. For example,  $R = \mathbb{Z}[X]/X^{d+1}$ , where  $d$  is a power of two. We let  $\lambda$  denote the security parameter throughout the paper. We denote  $K$  as a bound on the number of keys and  $L$  as a bound on the depth of the circuit, which defines the number of levels.  $S$  will denote the set of all parties such that  $|S| = k \leq K$ .

**Cryptosystem Parameters.** We also need to create parameters specific to BGV:

- Let  $\chi = \chi(\lambda, K, L)$  be a random distribution,  $B$ -bounded (i.e. in the interval  $]-B, B[$ ) on  $R$  – defined in the RLWE problem (Lyubashevsky et al., 2010);
- Let  $q_L \gg q_{L-1} \gg \dots \gg q_0$  be  $L + 1$  decreasing moduli.
- Let  $p$  be a small integer, such that  $\forall \ell \in \llbracket 0, L \rrbracket, p \wedge q_\ell = 1$ .
- For each level  $\ell \in \llbracket 0, L \rrbracket$ , let  $\beta_\ell = \lfloor \log(q_\ell) \rfloor + 1$ ; and let  $a_\ell \in R_{q_\ell}^{2\beta_\ell}$  be a public vector.

Therefore each algorithm will take the following parameters  $pp = (R, \chi, B, \{q_i, a_i\}_{i \in \{L, \dots, 0\}}, p)$ .

**Extended Object.** Let  $S'$  be a subset of  $S$ . An extended object  $\bar{o}$  is the concatenation of a set of object  $\bar{o} = (o'_1 \parallel \dots \parallel o'_k)$ , where

$$o'_j = \begin{cases} o_j & \text{if } j \in S' \\ 0 & \text{otherwise} \end{cases}$$

Note that this definition is different than the RGSW extended ciphertext. We will define and explain it later.

**Subroutines.** We define two useful subroutines.  
 $\text{POWERSOF2}(s) := (s \cdot 2^0, \dots, s \cdot 2^i, \dots, s \cdot 2^{\beta_\ell - 1})$ .  
 $\text{BITDECOMP}(U) := (D_0, \dots, D_i, \dots, D_{\beta_\ell - 1})$ , where  
 $U = \sum_{i=0}^{\beta_\ell - 1} 2^i \cdot D_i$ .

### 3.2 BGV

This subsection defines the construction of the BGV system. The notations introduced here are essential for understanding the multi-key part. The evaluation of homomorphic operations is described later, especially in the multi-key case (which is quite similar to the single-key case).

**Secret Key.** For each level  $\ell \in \llbracket 0, L \rrbracket$ , party  $j$  randomly chooses  $z_{\ell,j} \leftarrow \chi$  and sets  $s_{\ell,j} = (1, z_{\ell,j}) \in R_{q_\ell}^2$ .

**Public Key.** For each level  $\ell \in \llbracket 0, L \rrbracket$ , party  $j$  computes  $pk_{\ell,j} = [a_\ell \cdot z_{\ell,j} + p_j \cdot e_{\ell,j}, -a_\ell] =: [b_{\ell,j}, -a_\ell]$ , from the  $L + 1$  public vectors  $a_\ell$ , and where  $e_{\ell,j} \leftarrow \chi$ . Note that  $pk_{\ell,j} \cdot sk_{\ell,j} = a_\ell \cdot z_{\ell,j} + p \cdot e_{\ell,j} - a_\ell \cdot z_{\ell,j} = p \cdot e_{\ell,j}$ .

**Encryption.** To encrypt  $m \in R_p$ , party  $j$  chooses  $u \xleftarrow{\$} R_p$  and  $e_1, e_2 \leftarrow \chi$ . Then, party  $j$  computes and outputs a level- $L$  BGV ciphertext

$$c = [u \cdot pk_{L,j}[0] + m + p \cdot e_1, u \cdot pk_{L,j}[1] + p \cdot e_2] \in R_{q_L}^2$$

**Decryption.** Given a level- $\ell$  BGV ciphertext  $c = (c_0, c_1) \in R_{q_\ell}^2$  encrypted under  $pk_{\ell,j}$ , party  $j$  computes  $m = \langle c, s_{\ell,j} \rangle \pmod{q_\ell} \pmod{p}$ , where  $m$  is the plaintext.

## 3.3 Multi-Key

In 2017, Long Chen, Zhenfeng Zhang and Xueqing Wang published a paper describing for the first time a multi-key implementation of BGV: Batched Multi-hop Multi-key FHE from Ring-LWE with Compact Ciphertext Extension. (Chen et al., 2017).

### 3.3.1 Key Generation and Encryption

**Extended Public and Secret Keys.** Multi-key encryption involves several parties, each holding its own public/private key pair. From the individual public keys, we can construct an extended public key  $\bar{p}_k$ , which will be their concatenation. Similarly, we can construct the extended secret key  $\bar{s}_k$  from the individual private keys. Still, it must be underlined that this extended key is only a theoretical object, mainly used to explain decryption.

**Extended Ciphertext.** Below, we define an extended ciphertext  $\bar{c}$ . This ciphertext corresponds to a message  $m$  encrypted under the extended public key  $\bar{p}_k$ . Moreover, for any  $c_i \in \bar{c}$  such that  $i \in S$ , we have a message  $m_i$  encrypted under the key  $sk_i$ , such that:

$$m = \sum_{i \in S} m_i$$

where  $m$  is the message shared between the users.

**Decryption.** As for classic BGV decryption, the pure multi-key cryptosystems' decryption is based on a scalar product. For  $k$  parties,  $\bar{c}_k = (c_1 \parallel \dots \parallel c_k)$  and  $\bar{s}_k = (sk_1 \parallel \dots \parallel sk_k) = ((1, z_1) \parallel \dots \parallel (1, z_k))$ , where  $c_t$  and  $sk_t$  are respectively the ciphertext and the secret key of party  $t \in \llbracket 1, k \rrbracket$ . Thus,

$$\langle \bar{c}_k, \bar{s}_k \rangle = \sum_{t=1}^k \langle c_t, sk_t \rangle = \sum_{i \in S} \langle c_i, sk_i \rangle = \sum_{i \in S} m_i = m$$

Since the extended private key is a theoretical object, all the participants must work collectively to decrypt the message.

### 3.3.2 Operations

Multi-Key addition is simply a vector addition. The case of the multiplication is quite different. As in the single-key multiplication, the Multi-Key multiplication uses the tensor product, denoted  $\otimes$ . Due to the increase of the ciphertext's length (because of the tensor product), we need to reduce it to its initial length by using an *evaluation key*. The *evaluation key* is specific to FHE schemes and is another public element constructed from the secret key. This key is necessary for ciphertext relinearisation (dimension reduction) after multiplication. To create such a key in a pure multi-key scheme, we must introduce the RGSW cryptosystem (Gentry et al., 2013).

## 3.4 RGSW Scheme for the Generation of Evaluation Key

Multiplication is the critical point in the BGV Multi-Key cryptographic scheme, so we need to focus on constructing the evaluation key, also known as the relinearisation key in the BGV scheme. To do so, we need to use all the parties' keys, which is precisely this point that poses a problem and makes multiplication difficult. Sharing a secret key is a major security flaw. That is why we need to find a solution to remedy this. We use RGSW to encrypt users' private keys and obtain indices of keys to construct the relinearisation key we need.

### 3.4.1 Key Generation and Encryption

The RGSW key pair of party  $j$  is the same as that generated with BGV. We denote  $\mu$  as the plaintext in those algorithms and recall that  $p_k = [b_{l,j}, -a_l]$ .

**Encryption.** RGSW scheme uses two encryption algorithms, described by Algorithms 1 and 2.

**Data:**  $\mu$   
**for**  $i = 0$  to  $\beta_\ell$  **do**  
      $r_i \leftarrow \mathcal{X}$ ;  
      $e_1, e_2 \leftarrow \mathcal{X}$ ;  
      $f_1 = b_{\ell,j}[i]r_i + te_1 + \text{POWERSOF2}(\mu)[i]$ ;  
      $f_2 = b_{\ell,j}[i]r_i + te_1$ ;  
      $F[i] = \{f_1, f_2\}$ ;  
**end**  
**Result:**  $F$

Algorithm 1: RGSW.RANDENC.

**Data:**  $\mu$   
 $E \leftarrow \mathcal{X}^{2\beta_\ell}$ , error vector;  
 $r \leftarrow \mathcal{X}^2$ ;  
 $G := (\mathbf{I}_2, \dots, 2^i \mathbf{I}_2, \dots, 2^{\beta_\ell-1} \mathbf{I}_2)$ ;  
 $C := vP + pE + G^T \mu$ ,  $P$  the public key;  
**Result:**  $(C, F = \text{RGSW.RANDENC}(r))$

Algorithm 2: RGSW.ENC.

Algorithms 1 and 2 have two important properties:

- $F \cdot s = pe' + \text{POWERSOF2}(\mu)^T$ ,  
with  $e' = er + e_1 - e_2 \cdot z$
- $C \cdot s = pe' + \mu Gs \in R_q^{2\beta_\ell}$

**Decryption.** To decrypt, perform a standard BGV decryption on the first row of  $C$ .

### 3.4.2 Operations

**Addition.** To perform a Multi-Key addition, perform a classic matrix addition, i.e. add the rows one by one:

$$C + C' := (v + v')P + t(E + E') + G(\mu + \mu')$$

**Multiplication.** Unlike addition, multiplication is a more complex operation. In the following, the RGSW multiplication is denoted by  $C_1 \odot C_2$ . To compute it, first calculate the bit decomposition of  $C_1$ :  $\text{BITDECOMP}(C_1) = \{D_0, \dots, D_{\beta_\ell-1}\} \in (R^{(2 \times \beta_\ell)})^{\beta_\ell}$ , such that  $C_i = \sum_{j=0}^{\beta_\ell-1} 2^j \cdot D_j$ . Then, calculate the following classical matrix product:

$$C_1 \odot C_2 := \text{BITDECOMP}(C_1) \cdot C_2 \quad (1)$$

**Extended RGSW Ciphertext.** The extended RGSW ciphertext is the last object we need to construct the evaluation key. To extend an RGSW ciphertext  $C_i$  produced with the function RGSW.ENC (Alg. 2), we construct  $X_j$  from the result  $F$  of RGSW.RANDENC (Alg. 1), and  $\tilde{b}_j := b_j - b_i \in R_q^{2\beta}$  (where  $b_i, b_j$  are the first elements of the corresponding public keys). For each  $u \in \{0, \dots, 2\beta_\ell - 1\}$ , and for  $v \in \{0, \dots, \beta_\ell - 1\}$ ,

$$X_j[u] := \text{BITDECOMP}(\tilde{b}_j[u]) \cdot F_i = \begin{pmatrix} f_{1(0)} & f_{2(0)} \\ \vdots & \vdots \\ f_{1(v)} & f_{2(v)} \\ \vdots & \vdots \\ f_{1(\beta_\ell-1)} & f_{2(\beta_\ell-1)} \end{pmatrix} \quad (2)$$

Finally, put the  $X_j$  blocks obtained in the  $i^{\text{th}}$  column of a matrix, and  $C_i$  on the diagonal. Initialise the rest of the matrix at 0. We obtain the following result:

$$\bar{C} = \begin{pmatrix} C_i & X_1 & 0 \\ & \ddots & \vdots \\ \vdots & & C_i \\ 0 & X_k & C_i \end{pmatrix} \in R_q^{2k\beta_\ell \times 2k} \quad (3)$$

We define the extended RGSW ciphertext construction as  $\text{RGSW.CTEXT}(C_i, pk_S, F_i)$ .

### 3.5 Evaluation/Relinearisation Key

In pure Multi-Key BGV, the relinearisation key stands for the evaluation key. There are three main steps to create the evaluation key:

- Pre-calculations on secret keys (each party);
- Computations from the encrypted data (server);
- Selection of the rows for the Key Switch key.

**Pre-Calculations:** Each user must perform pre-calculations to transmit encrypted information about their secret keys. To do this, each party  $j$  computes the RGSW ciphertexts for powers of 2 of its secret keys (4) and (5) and encrypts the bit decomposition of its secret keys (6) and (7). The idea is to multiply the powers of two of one key with the bit decomposition of another to obtain the multiplication of these two keys.

$$\begin{aligned}\Phi_{i,\ell,j} &= \text{RGSW.ENC}_{s_{\ell-1,j}}(\text{POWERSOF2}(s_{\ell,j})[i]) \\ &= r_{i,\ell,j} \cdot [b_{\ell-1}, a_{\ell-1}] + p \cdot E_{i,\ell,j} \\ &\quad + \text{POWERSOF2}(s_{\ell,j})[i] \cdot G\end{aligned}\quad (4)$$

$$F_{i,\ell,j} = \text{RGSW.ENCRAND}(r_{i,\ell,j}, pt_{\ell-1,j}) \in R_{q_\ell}^{\beta_\ell \times 2} \quad (5)$$

$$\begin{aligned}\Psi_{i,\ell,j} &= \text{RGSW.ENC}_{s_{\ell-1,j}}(\text{BITDECOMP}(s_{\ell,j})[i]) \\ &= r_{i,\ell,j} \cdot [b_{\ell-1}, a_{\ell-1}] + p \cdot E_{i,\ell,j} \\ &\quad + \text{BITDECOMP}(s_{\ell,j})[i] \cdot G\end{aligned}\quad (6)$$

$$F'_{i,\ell,j} = \text{RGSW.ENCRAND}(r_{i,\ell,j}, pt_{\ell-1,j}) \in R_{q_\ell}^{\beta_\ell \times 2} \quad (7)$$

These individual pre-computations are then used to construct the following extended RGSW ciphertexts:

$$\begin{aligned}\bar{\Phi}_{i,\ell,j^*} &= \text{RGSW.CTEXT}(\Phi_{i,\ell,j^*}, pk_S, F_{i,\ell,j^*}) \\ &= \text{RGSW.ENC}_{\bar{s}_{\ell-1}}(\text{POWERSOF2}(s_{\ell,j^*})[i])\end{aligned}\quad (8)$$

$$\begin{aligned}\bar{\Psi}_{i,\ell,j^*} &= \text{RGSW.CTEXT}(\Psi_{i,\ell,j^*}, pk_S, F'_{i,\ell,j^*}) \\ &= \text{RGSW.ENC}_{\bar{s}_{\ell-1}}(\text{BITDECOMP}(s_{\ell,j^*})[i])\end{aligned}\quad (9)$$

The last calculations (the extended RGSW ciphertexts (8) and (9)) can be performed by the server, although it is still technically possible to let each client perform them.

**Server Calculations.** First, remember the writing of the (theoretical) level- $\ell$  extended secret key:

$$\bar{sk}_\ell = (s_{\ell,j_1} \parallel \dots \parallel s_{\ell,j_k}) = (1, z_{\ell,j_1}, \dots, 1, z_{\ell,j_k}) \in R_{q_\ell}^{2k}$$

From the previous pre-calculations, the server has to create a ciphertext of the product of the secret key  $\bar{sk}_{\ell,j}$  of party  $j$  at level  $\ell$  and the secret key  $\bar{sk}_{\ell,j'}$  of party  $j'$  at the same level. In other words, the server computes  $\text{RGSW.ENC}(\bar{sk}_{\ell,j}[t] \cdot \bar{sk}_{\ell,j'}[t'])$  from  $\bar{\Phi}_j$  and  $\bar{\Psi}_{j'}$ , where  $t, t' \in \{0, 1\}$  are the indexes of the key  $s_{\ell,j}$ . The server performs the following calculations for  $t = t' = 1$ .

$$\begin{aligned}& \text{RGSW.ENC}_{\bar{s}_{\ell-1}}(\bar{sk}_{\ell,j}[t] \cdot \bar{sk}_{\ell,j'}[t']) \\ &= \sum_{i=1}^{\beta_\ell} (\text{RGSW.ENC}_{\bar{s}_{\ell-1}}[\text{POWERSOF2}(sk_{\ell,j}[t])[i]] \\ &\quad \odot \text{RGSW.ENC}_{\bar{s}_{\ell-1}}(\text{BITDECOMP}(sk_{\ell,j'}[t'])[i])) \\ &= \sum_{i=1}^{\beta_\ell} \bar{\Phi}_j[i] \odot \bar{\Psi}_{j'}[i]\end{aligned}\quad (10)$$

**Key Construction.** In the previous section, we saw how to calculate  $C_\delta := \text{RGSW.ENC}_{\bar{s}_{\ell-1}}(\bar{s}'_\ell[\delta]) \in R_{q_\ell}^{2k\beta_\ell \times 2k}$ , where  $\bar{s}'_\ell = \bar{s}_\ell \otimes \bar{s}_\ell$  and  $\delta \in \{1, \dots, 4k^2\}$ . Constructing the evaluation key involves choosing the right rows in the matrix to perform a Key Switch. Remember that to create a Key Switch, we must have rows  $c_{d,\delta}$  for  $d \in \{1, \dots, \beta_\ell\}$  such as:

$$\langle c_{d,\delta}, \bar{s}_\ell \rangle = p \cdot e_{d,\delta} + 2^{d-1} \cdot \bar{s}'_\ell[\delta]$$

For  $d' \in \{1, \dots, \beta_\ell\}$ , we choose the rows with indices  $2 \cdot k \cdot (d' - 1)$  if the first index of the rows starts at 0, and we add 1 to this number if the index starts at 1. Contrary to the original article (Chen et al., 2017), where the authors use the  $d$  above to select the rows, we introduce the notation  $d'$  to choose them. We make this slight correction because the powers do not match otherwise, and the rows are incorrectly indexed.

## 4 CONTRIBUTIONS

This section will focus mainly on the paper presenting implementation ideas: Batched Multi-hop Multi-Key FHE from ring-LWE with Compact Ciphertext Extension (Chen et al., 2017).

### 4.1 Explanations of Multi-Key BGV Operations

Let  $k$  be the number of parties.

**Multiplication.** When multiplying two ciphertexts encrypted under two different keys  $pk_1$  and  $pk_2$ , the result ciphertext is then decryptable by the corresponding secret keys' tensor product  $s_1 \otimes s_2 = (1, s_1, s_2, s_1 \cdot s_2)$ . Therefore, this new ciphertext should be relinearised to put it back under a key of the usual form.

### 4.2 Multiplication on Extended RGSW Ciphertexts

In (Chen et al., 2017), the authors do not explain how to perform multiplication on extended RGSW ciphertexts. Instead, the article implies that the calculations are the same as for non-extended multiplication – that the operation is analogous – which is not the case. This section explains why it differs from multiplication on classic RGSW ciphertexts and how to perform this operation on extended ciphertexts.

#### 4.2.1 Explanation of the Product on Extended RGSW Ciphertexts

*Decryption of an Extended Ciphertext.* An RGSW extended ciphertext  $C$ , extended from the ciphertext  $C_i$  of party  $i$  to  $k$  participants, is a matrix of  $k$  rows, with  $C_i$  on the diagonal and  $X_j$  blocks in the  $i^{\text{th}}$  column (cf eq. (3)). To decrypt such a ciphertext, each line of blocks must be considered (remembering that correct decryption is achieved by examining every other line). Given  $j \in \{1, \dots, k\}$ , there are two possibilities:

- ( $i = j$ ) On line  $i$ , there is only one non-zero element,  $C_i$ , in column  $i$ . Therefore,  $\langle l_j, \bar{s}_\ell \rangle = C_i \cdot s_i$ . The result is correct since the decryption is done with the right key.
- ( $i \neq j$ ) On line  $j$ , there are two non-zero elements,  $C_i$  and  $X_j$ . Hence,

$$\begin{aligned} \langle l_j, \bar{s}_\ell \rangle &= X_j \cdot s_i + C_i \cdot s_j = pe' + \tilde{b}_j r + C_i s_j \\ &= pe' + pe_j r - as_j r - pe_i r + as_i r + rpe_i \\ &\quad - as_i r + pe_{01} + \mu + ras_j + ps_j e_{02} \\ &= p(e' + e_j r + e_{01} + s_j e_{02}) + G\mu s_j \quad (11) \end{aligned}$$

Which is a right decryption.  $\square$

**Applying RGSW Multiplication.** This section explains applying the multiplication described above (see eq. (1)) to an extended ciphertext. First, let us start by doing it naively on a simple case.

**Example 1.** We want to compute

$$C \odot \mathbf{1} = \begin{pmatrix} 5 & 2 & 2 & 1 \\ 3 & 4 & 6 & 0 \end{pmatrix} \odot \begin{pmatrix} G & 0 \\ 0 & G \end{pmatrix}$$

with  $G = (I_2 \mid 2I_2 \mid 4I_2)$ .

$$\text{BITDECOMP}(C) =$$

$$\left( \begin{array}{cccc|cccc|cccc} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{array} \right)$$

If we perform matrix multiplication, the first element is different from the expected result (1 instead of 5).

So, there is a problem applying the multiplication using the earlier definition. On the other hand, if we change this product into a matrix product and then a multitude of RGSW products, we get a correct result.

#### 4.2.2 Extended RGSW Multiplication Algorithm

As explained above, this multiplication is essential for constructing an evaluation key. This section focuses on performing an RGSW-extended multiplication, avoiding the calculation of duplicate multiplications, which are time-consuming operations.

As previously, there are two cases: where  $i = j$  and where  $i \neq j$ . The first one is less useful because it allows the calculation of Key Switch keys of the form  $s_i \times s_i$ . Thus, we focus on the second one: for  $i \neq j$ , the matrix product is calculated as follows<sup>2</sup>.

- (1) Initialize a block matrix to 0.
- (2) Set the matrix diagonal to  $C_i \odot C_j$ ;
- (3) On column  $i$ , if  $v \neq i$ , add  $X_v \odot C_j$  to location  $v$ ;
- (4) On column  $j$ , if  $v \neq i$ , add  $X_v \odot X'_i$  to location  $v$ ;
- (5) On column  $j$ , if  $v \neq j$ , add  $C_i \odot X'_v$  to location  $v$ .

Regarding the number of internal multiplications, step (2) requires only one multiplication, while steps (3), (4) and (5) each require two times  $(k - 1)$  multiplications. This algorithm requires a total of  $6(k - 1) + 1$  internal multiplications to create this type of Key Switch key.

**Remarks on the Choice of Rows.** By constructing matrices this way and choosing rows within each block, we compute many costly multiplications. That increases considerably with the number of parties. Recall that the choice of rows only justifies the condition  $\langle c_{d,\delta}, \bar{s}_\ell \rangle = p \cdot e_{d,\delta} + 2^{d-1} \cdot \bar{s}'_\ell[\delta]$ .

Consequently, there are two interesting remarks. The first one is that only  $\beta_\ell$  rows are required for the computation, instead of  $2k\beta_\ell$ . The second one comes from the fact that all blocks of rows encrypt the same information. This implies that the information is calculated  $k$  times. Thus, choosing the rows in a different way is possible by considering only one block of rows.

### 4.3 Multi-Key Key Switch

The Key Switch used for relinearisation is similar to the one used in BGV. We will, therefore, describe a relinearisation based on a Key Switch protocol.

**Recall.** Let  $c = (c_0, c_1, c_{1,2} \parallel c'_0, c_2, c_{2,1})$ . We assume that we have already split the elements to be multiplied by  $s_1, s_2$  and that we have already relinearised the terms into  $s_1 s_1$  and  $s_2 s_2$ . In this way, we only<sup>3</sup> have to relinearise the terms to be multiplied by  $s_1 s_2$ . We also need to recall that

$$m_1 \cdot m_2 = (c_0 + c'_0) + c_1 \cdot s_1 + c_2 \cdot s_2 + (c_{1,2} + c_{2,1}) \cdot s_1 s_2$$

However, as explained above, it is neither reasonable nor secure to share one's secret key to create a key product. We will therefore have to use

<sup>2</sup>Remember that a matrix product is not always commutative

<sup>3</sup>This operation will be the same in all cases

an evaluation key  $\kappa = \{\kappa_i\}_{i \in [0, \beta_\ell - 1]}$  such that for all  $i \in [0, \beta_\ell - 1]$ , remembering that  $\bar{s}_\ell$  is the extended key of the system:

$$\begin{aligned} 2^i \cdot s_1 s_2 &= \langle \kappa_i, \bar{s}_\ell \rangle \\ &= \kappa_i[0] \cdot 1 + \kappa_i[1] \cdot s_1 + \kappa_i[2] \cdot 1 + \kappa_i[3] \cdot s_2 \end{aligned} \quad (12)$$

**Relinearisation.** To relinearise the term  $c_{1,2} + c_{2,1}$ , denoted  $\eta$ , the server first adds the two elements  $c_{1,2}$  and  $c_{2,1}$ <sup>4</sup>, so that there is only one relinearisation step. Then, the server computes for each  $i$  and for each  $j \in \{0, 1, 2, 3\}$ :

$$\eta \cdot \kappa_{i,j} = \text{BITDECOMP}(c_{1,2} + c_{2,1})[i] \cdot \kappa_i[j] \quad (13)$$

We remark, by using (12) and (13), that

$$\begin{aligned} \mathbf{c} \cdot \kappa[i] &= \langle (\eta \cdot \kappa_{i,0}, \eta \cdot \kappa_{i,1}, \eta \cdot \kappa_{i,2}, \eta \cdot \kappa_{i,3}), \bar{s}_\ell \rangle \\ &= \text{BITDECOMP}(\eta)[i] \cdot 2^i s_1 s_2 \end{aligned} \quad (14)$$

$$= \text{BITDECOMP}(c_{1,2} + c_{2,1})[i] \cdot 2^i s_1 s_2 \quad (15)$$

Having said all this, we can now carry out the relinearisation and demonstrate that it is a correct operation. The next step is to transform the initial ciphertext  $ct$  into the relinearised ciphertext  $ct'$ , as follows:

$$\begin{aligned} c' := (c_0 + \sum_{i=0}^{\beta_\ell - 1} \eta \cdot \kappa_{i,0}, \quad c_1 + \sum_{i=0}^{\beta_\ell - 1} \eta \cdot \kappa_{i,1}, \\ c'_0 + \sum_{i=0}^{\beta_\ell - 1} \eta \cdot \kappa_{i,2}, \quad c_2 + \sum_{i=0}^{\beta_\ell - 1} \eta \cdot \kappa_{i,3}) \end{aligned} \quad (16)$$

**Decryption.** The desired result of decrypting  $c'$  is  $m_1 \cdot m_2$ . To decrypt  $c'$ , calculate the scalar product of  $c'$  with the extended key  $\bar{s}_\ell$ .

$$\begin{aligned} \langle c', \bar{s}_\ell \rangle &= (c_0 + c'_0) \cdot 1 + c_1 \cdot s_1 + c_2 \cdot s_2 \\ &\quad + \sum_{i=0}^{\beta_\ell - 1} \eta \cdot (\kappa_{i,0} \cdot 1 + \kappa_{i,1} \cdot s_1 + \kappa_{i,2} \cdot 1 + \kappa_{i,3} \cdot s_2) \end{aligned}$$

By using this transformation,

$$\begin{aligned} \langle c', \bar{s}_\ell \rangle &= (c_0 + c'_0) \cdot 1 + c_1 \cdot s_1 + c_2 \cdot s_2 + \sum_{i=0}^{\beta_\ell - 1} \eta \cdot \kappa[i] \\ &= (c_0 + c'_0) \cdot 1 + c_1 \cdot s_1 + c_2 \cdot s_2 \\ &\quad + \sum_{i=0}^{\beta_\ell - 1} \text{BITDECOMP}(c_{1,2} + c_{2,1})[i] \cdot 2^i s_1 s_2 \\ &\text{by (15)} \end{aligned}$$

<sup>4</sup>The operation can be performed separately, but the relinearisation operation is much more costly than the addition operation.

$$\begin{aligned} &= (c_0 + c'_0) \cdot 1 + c_1 \cdot s_1 + c_2 \cdot s_2 + (c_{1,2} + c_{2,1}) s_1 s_2 \\ &= m_1 \cdot m_2 \end{aligned}$$

□

## 4.4 Multiplication Complexity

**Parameters.** To compute some complexity calculus, we need to define some parameters:

- p: plaintext modulus;
- n: cyclotomic order/2;
- $Q = \prod q_i$ : ciphertext modulus;
- k: number of party;
- d: multiplicative depth.

### Multiplication on $\mathbb{Z}/q\mathbb{Z}$ .

1. Multiplication complexity: naive ( $O(\log(q)^2)$ ) and karatsuba ( $O(\log(q)^{\log(3)})$ )
2. Reduction modulus  $q$ :  $O((2 \log(q) - \log(q) + 1) \log(q)) = O(\log(q)^2)$ . Due to the complexity of this division, the result for the addition in  $\mathbb{Z}/q\mathbb{Z}$  will be the same as the multiplication.

**Global complexity:**  $C_{\text{mult}} = O(\log(q)^2)$

**Multiplication of Two DCRTPoly.** A DCRTPoly is a vector of  $(d+2)$  elements. These are vectors of  $n$  elements in  $\mathbb{Z}/q\mathbb{Z}$ , each associated with a modulus  $q$ .

To multiply it, we need to perform the multiplication term by term, i.e. there will be  $n$  multiplications in  $\mathbb{Z}/q\mathbb{Z}$  per vector. Let  $\log(q_i) \sim \log(q)$ .

**Global complexity:**  $C_{\text{multDCRTPoly}} = O(n \cdot d \cdot C_{\text{mult}}) = O(n \cdot d \cdot \log(q)^2)$ .

**BitDecomp(NUM).** To find one bit  $i$ , we need to apply NUM &  $0 \dots 0 \underbrace{1}_i 0 \dots 0$ . This operation

complexity is  $O(\log(Q))$ . We need to repeat it for each bit and number in interpolating the DCRTPoly (only one vector left).

**Global complexity<sup>5</sup>:**  $C_{\text{BitDecomp}} = O(\log(Q)^2 \cdot n)$

### RGSW Multiplication.

1. Loop from 0 to  $2k \log(Q)/2k$  – the number of rows we want to compute here is  $\log(Q)$ . At this level we know the complexity bound as:  $C_{\text{RGSWmult}} = O(\log(Q) \cdot C_{\text{firstLoop}})$

<sup>5</sup>This complexity is valid for the version of OpenFHE we used. It could be improved in future versions of the library.

2. Let's compute  $C_{\text{firstLoop}}$ . It consists of two-bit decompositions and a loop from 0 to  $2\log(Q)$  – this time, we need all the rows.  $C_{\text{firstLoop}} = O(C_{\text{BitDecomp}} + \log(Q) \cdot C_{\text{secondLoop}}) = O(\log(Q)^2 \cdot n + \log(Q) \cdot C_{\text{secondLoop}})$
3. Determining the complexity of the second loop. Let us enumerate all the operations in this loop:
  - 4 multiplications of DCRTPoly;
  - 4 additions of DCRTPoly.
 Consequently,  $C_{\text{secondLoop}} = O(4(C_{\text{multDCRTPoly}} + C_{\text{addDCRTPoly}})) = O(n \cdot d \cdot \log(q)^2)$

**Global complexity:**  $C_{\text{RGSWmult}} = O(\log(Q)^2 \cdot n \cdot d \cdot \log(q)^2)$ .

## 5 RESULTS AND CONCLUSION

For these results, the computer used has an Intel Core processor i7-12800H, 14 cores (6 performance-cores – max turbo frequency cores 4.80 GHz and 8 efficient-cores – max turbo frequency cores 3.70 GHz) for a total of 20 threads. The library is OpenFHE version 1.1.1 (1.0.3 was used for comparison). We used a plaintext modulus of 65537, and the multiplicative depth was 3. The other parameters were OpenFHE default ones.

Table 1: Addition.

parties	2	3	4	5	8	10
time (ms)	98	157	219	279	460	579

Table 2: Multiplication.

parties	2	3	4	5	8	10
time (ms)	115	252	397	550	1136	1642

We did the two operations (addition and multiplication) for 2, 3, 4, 5, 8 and 10 parties. The corresponding results can be found in the tables 1 and 2. Multiplication time is to be understood without relinearisation: we just decrypted the ciphertext to verify the correctness of the algorithm.

The last step was to create the relinearisation key. The time result (described in 3 and 4) is only for one element of the evaluation key and does not include the pre-calculation time (which must be done independently by every party because it is the secret key's encryption).

As shown by these last results, the implementation of the library is crucial to evaluation key's calculation time. Refinement of core functions can very likely lead to substantial improvements in calculation time

Table 3: RGSW multiplication v1.1.1, One row.

threads	1	2	4	8	16	20
time (min)	3:38	3:43	3:25	3:38	3:40	3:23

Table 4: RGSW multiplication v1.0.3, One row.

threads	1	2	4	8	16	20
time (min)	6:23	5:51	5:40	5:25	4:53	6:13

<sup>6</sup>, and it will hopefully be enhanced in future versions of OpenFHE.

## REFERENCES

- Aloufi, A. and Hu, P. (2019). Collaborative Homomorphic Computation on Data Encrypted under Multiple Keys. *IWPE'19*.
- Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., and Wichs, D. (2012). Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE. In *EUROCRYPT 2012*, volume 7237, pages 483–501. Springer.
- Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P. M. R., and Sahai, A. (2018). Threshold Cryptosystems from Threshold Fully Homomorphic Encryption. In *CRYPTO 2018*, volume 10991, pages 565–596. Springer.
- Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2014). (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36.
- Chen, H., Chillotti, I., and Song, Y. (2019). Multi-key homomorphic encryption from TFHE. In *ASIACRYPT 2019*, volume 11922, pages 446–472. Springer.
- Chen, L., Zhang, Z., and Wang, X. (2017). Batched Multi-hop Multi-key FHE from ring-LWE with Compact Ciphertext Extension. *TCC 2017*, 10678:17.
- Chowdhury, S., Sinha, S., Singh, A., Mishra, S., Chaudhary, C., Patranabis, S., Mukherjee, P., Chatterjee, A., and Mukhopadhyay, D. (2022). Efficient threshold FHE with application to real-time systems. *IACR Cryptol. ePrint Arch.*, page 1625.
- Doröz, Y., Hu, Y., and Sunar, B. (2016). Homomorphic AES evaluation using the modified LTV scheme. *Designs, Codes and Cryptography*, 80(2):333–358.
- Gentry, C., Sahai, A., and Waters, B. (2013). Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based.
- López-Alt, A., Tromer, E., and Vaikuntanathan, V. (2013). On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 94.
- Lyubashevsky, V., Peikert, C., and Regev, O. (2010). On ideal lattices and learning with errors over rings. In *EUROCRYPT 2010*, pages 1–23.

<sup>6</sup>For instance, the bit decomposition could still be significantly optimized.