

## RESEARCH ARTICLE

# Efficient TFHE Bootstrapping in the Multiparty Setting

JEONGEUN PARK<sup>1</sup> AND SERGI ROVIRA<sup>2</sup> <sup>1</sup>imec-COSIC, KU Leuven, Heverlee, 3001 Leuven, Belgium<sup>2</sup>WiSeCom, Universitat Pompeu Fabra, 08018 Barcelona, Spain

Corresponding authors: Jeongeun Park (jeongeun.park@esat.kuleuven.be) and Sergi Rovira (sergi.rovira@upf.edu)

The work of Jeongeun Park was supported by CyberSecurity Research Flanders under Grant VR20192203. The work of Sergi Rovira was supported in part by the Spanish Ministry of Economy and Competitiveness under Grant RTI2018-102112-B-I00.

**ABSTRACT** TFHE is a practical fully homomorphic encryption scheme (FHE) capable of computing any boolean gate or non-linear function. The scheme was originally designed to work for the single key setting. To implement realistic application scenarios, it is necessary to extend it to handle multiple users. In this paper, we introduce a new approach to generate TFHE bootstrapping keys for (predefined) multiple users. Hence, a fixed number of users can enjoy the same level of efficiency as in the single key setting, keeping their individual input privacy. Our construction relies on a novel algorithm called homomorphic indicator, which can be of independent interest. We provide a detailed analysis of the noise growth and a set of secure parameters suitable to be used in practice. Moreover, we compare the complexity of our technique with other state-of-the-art constructions and show which method performs better depending on the parameter sets. We also provide a prototype implementation of our technique. To the best of our knowledge, this is the first implementation of TFHE in the multiparty setting.

**INDEX TERMS** Homomorphic encryption, multiparty TFHE, post-quantum cryptography.


## I. INTRODUCTION

Many privacy-preserving protocols are efficiently designed in a non-interactive way with practical implementation results thanks to recent improvements on FHE. However, some problems arise when it comes to handling multiple users. The easiest way to handle multiple users is to share the same secret key among them and then ask a third party (a server in general) to compute a function of the given encrypted data of each user under the same key. However, in this case, some dishonest parties can easily intercept others' encrypted inputs through the public channel and decrypt them. Therefore, this naive approach does not guarantee input privacy [1], [2].

If numerous users want to compute a common function keeping their input privacy, multi-key homomorphic encryption (MKHE) [3] and multiparty homomorphic encryption (MPHE) (a.k.a. threshold-multikey FHE) [4] are the best

solutions for non-interactive protocols.<sup>1</sup> That is, both designs allow users to keep partial information of the master secret key without the need to involve a third trusted party during key distribution. In other words, no one knows the master secret key, so decrypting other's ciphertexts becomes impossible. Decryption is possible only when all the participants (users) agree to do so by the definition of MKHE/MPHE.

Despite providing user privacy, MKHE has a serious practical problem when the number of users is large, since the ciphertext size increases depending on the number of users. On the other hand, MPHE does not have such ciphertext expansion, so it can achieve (asymptotically) the same computational complexity as single key FHE. However, it requires a setup phase where users must interact at least once to generate their common public key, which is not necessary in MKHE. Therefore, both approaches have

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Masucci .

<sup>1</sup>Each user can protect its input from other parties in both designs since the secret key is never shared.

distinct advantages and there are different applications where each can have better performance than the other, as analyzed in [5]. For example, if there are already predefined inputs for a function which a server is going to compute, such as the training of a machine learning model, MPHE outperforms MKHE.

The main technical difficulty of designing MPHE is to create the common bootstrapping key. Mouchet et al. [4], introduce a MPHE construction requiring two rounds between participants. In [5], the author improved upon [4] by designing a non-interactive algorithm, once the public keys are already known. The setup phase (generation of the common keys) of the above works has practical computation time and memory consumption. Once the common public keys (including bootstrapping keys) are generated during the setup phase, most of the algorithms that the server runs have the same performance as those of the underlying single key scheme. Indeed, the asymptotic complexity of previously designed MPHE is the same as their single key FHE versions (BGV [6], FV [7], and CKKS [8]).

MPHE designs based on TFHE have not been well studied even though TFHE is widely used in privacy-preserving protocols [9], [10], [11].<sup>2</sup> TFHE enables to compute a gate operation over bits homomorphically, whereas the other schemes aim for (approximate or exact) arithmetic operations. After each gate operation (which only consists of a few additions), TFHE needs to run a bootstrapping algorithm to refresh the noise level to allow further operation without decryption failure. Therefore, constructing TFHE in the multiparty setting boils down to designing an efficient TFHE bootstrapping algorithm handling multiple keys.

Recently, Lee et al. [12] briefly introduced (a theoretical version of) an MPHE protocol based on TFHE. They first design the protocol for single key TFHE with larger coefficients for the secret key and then, they naturally extend it to the MPHE version, introducing a simple (global) bootstrapping key generation algorithm which requires all parties' local bootstrapping keys. This makes sense because the common secret key of MPHE corresponds to the sum  $s := s_1 + \dots + s_k$ , where  $s_i$ 's is the binary secret key of the  $i$ -th user, which is viewed as a secret key of a single user, corresponding to a secret  $s$  with large entries. However, a direct extension without implementation might hide the actual computation overhead caused by the noise contained in the global bootstrapping keys. The main difference between single key FHE with larger secret entries and its MPHE version is the noise contained in the same types of keys. Since the global keys are made of all parties' keys, the noise contained in the keys grows proportional to the total number of parties (which we denote by  $k$ ). Therefore, in practice, we cannot directly use the same parameters which are recommended for the single key setting. In other words,

<sup>2</sup>TFHE outperforms other schemes such as BGV, FV and CKKS when the computation is over bits and for non-linear functions such as ReLU, max/min functions.

we must consider the noise contained in the global keys, when it comes to choosing parameters.

A concurrent paper [13] gives a protocol for TFHE with larger key sizes with a different technique. We observe that their technique requires only  $k$  multiplications in the FFT domain among bootstrapping keys in the main loop of the bootstrapping algorithm. Whereas Lee et al. [12] makes use of key switching including two FFT conversions and an external product, which is more expensive than the  $k$  FFT multiplications in every loop. However, a detailed comparison of both techniques, even in the single key setting, has not appeared in the literature just yet.

## A. OUR CONTRIBUTIONS

In this work, we extend [13]'s scheme to a version of MPHE, introducing an efficient bootstrapping key generation algorithm. To do this, we introduce an algorithm called *homomorphic indicator*, which homomorphically computes a unit vector of a suitable dimension. That is, the algorithm takes as input a bit string and outputs a unit vector with an encryption of 1 in the position corresponding to the Hamming weight of the input string. This algorithm is the main building block of our bootstrapping key generation to indicate which key is selected during bootstrapping. The server can therefore run the bootstrapping algorithm of [13] with the indicated bootstrapping key, without revealing the corresponding secret value because the the algorithm's output is the same as their bootstrapping key. We believe that our technique for constructing the bootstrapping key can be of independent interest.

Furthermore, we compare the complexity of the bootstrapping technique of [12] with our extension of [13] and we show which scheme performs better in what parameter sets, based on our noise analysis. We provide a prototype implementation of our key generation algorithm using the Concrete library [14] with exact parameters yielding at least 110 bits security. With a number of parties of up to 16, we are able to perform bootstrapping in less than a second. To the best of our knowledge, this is the first implementation of TFHE in the multiparty setting. Since we also analyze the practicality of both our approach and [12] based on different metrics including noise analysis and complexity, this work can serve as a guide for readers to choose the right one for their use cases.

## B. A BRIEF NOTE ON APPLICATIONS

A direct application of MPHE is privacy preserving machine learning [10], [11], [15] where a model provider holding its private machine learning model encrypts the model under the common public key, and the client encrypts its private input data under the common public key, then both party sends ciphertexts to the server (which is the (semi-honest) third party) for a computation (see Fig. 1). At the end of the server's computation and partial decryption [16] interacting with the model provider, the client only gets the inference/predicted

value from the server. The whole protocol is non-interactive thanks to homomorphic encryption on the client's side, therefore the client can go offline after the query phase until the computation is over. More applications are discussed in Section IX-A.

## C. PAPER ORGANIZATION

In Section II we establish notation and give background on FHE. In Section III, we detail the bootstrapping process of single-key TFHE. In Section IV, we detail the blind rotation algorithms of [12] and [13]. In Section V we explain the homomorphic indicator algorithm and we detail how to use it to construct suitable bootstrapping keys to be used with the blind rotation of [13]. In Section VI, we explain how MPHE TFHE based on [13] works using the keys constructed in the previous section. In Section VII we study the noise growth of our construction. In Section VIII we provide a comparison between the MPHE approach of [12] and our approach, we also provide the implementation details and our parameter recommendations. Finally, in Section IX we conclude the paper, providing a few extra comments on applications.

## II. PRELIMINARIES

In this section, we fix notation for the rest of the paper and provide the necessary background on FHE and multiparty FHE to understand our contributions. The interested reader can find more background in [17].

### A. NOTATIONS

We denote the security parameter of the FHE scheme by  $\lambda$ . The dot product of two vectors  $\mathbf{v}, \mathbf{w}$  is denoted by  $\langle \mathbf{v}, \mathbf{w} \rangle$ . For a vector  $\mathbf{x}$ , both  $x_i$  and  $\mathbf{x}[i]$  denote either the  $i$ -th scalar component or the  $i$ -th element of an ordered finite set. The  $i$ -th element of an array  $\mathbf{A}$  is denoted by  $\mathbf{A}[i]$ . We denote the logarithm function in base two by  $\log(\cdot)$ . Let  $\mathcal{R}$  and  $\mathcal{R}_q$  denote the rings of polynomials  $\mathbb{Z}[X]/(X^N + 1)$  and  $\mathbb{Z}_q[X]/(X^N + 1)$ , respectively, for positive integers  $q$  and  $N$ . Let  $\theta, \theta'$  denote the standard deviation of the noise contained in a fresh RLWE ciphertext and a fresh LWE ciphertext, respectively. All the noise is sampled from a distribution denoted by  $\chi$  according to a suitable standard deviation. Let  $\ell = O(\log q)$  be the parameters of the TFHE scheme [18]. We denote the number of users by  $k$ . The LWE (resp. RLWE) secret key of each user is denoted by  $\mathbf{s}_i = (s_{i,0}, \dots, s_{i,n-1})$ , where  $s_{i,j} \in \{0, 1\}$  and  $n \geq 1$  (resp.  $s_i \in \mathcal{R}$ ). The (master) secret key corresponding to the common public key for an LWE ciphertext is denoted by  $S = (S_0, S_1, \dots, S_{n-1})$ , where  $S_j = \sum_{i=1}^k s_{i,j} \leq k$  for all  $j \in \{0, \dots, n-1\}$ . Similarly,  $s := s_1 + s_2 + \dots + s_k$  is the master secret key for an RLWE/RGSW ciphertext. Given a base  $g$  and  $\ell = O(\log q)$ , we define a gadget vector  $\mathbf{g} = (1, g, \dots, g^{\ell-1})^t$ . An RGSW ciphertext is of the form  $\text{RGSW}_s(m) := (\mathbf{a}, \mathbf{b}) \in \mathcal{R}_q^{2\ell \times 2}$ . Let  $\mathbf{G}$  be a gadget matrix defined as  $\mathbf{G} := \mathbf{I}_2 \otimes \mathbf{g}$ . Let  $\mathbf{G}^{-1}(\cdot)$  be the gadget decomposition function, which satisfies  $\mathbf{G}^{-1}(a) \cdot \mathbf{G} \approx a \bmod q$  for any  $a \in \mathcal{R}_q^2$ . We define

$\mathbf{G}^{-1}(\mathbf{c}) := (\mathbf{G}^{-1}(c_1), \dots, \mathbf{G}^{-1}(c_\delta))$ , for a vector  $\mathbf{c} = (c_1, \dots, c_\delta) \in \mathcal{R}_q^{\delta \times 2}$ . In this paper, we use the terms *parties* and *users* interchangeably to refer to the entities who join the homomorphic computation.

### B. BASIC BACKGROUND ON FHE

Informally, a public key *Fully Homomorphic Encryption* (FHE) scheme is an encryption scheme which allows us to perform arbitrary computations over encrypted inputs without decrypting them first. That is, given a ciphertext  $\text{ct}$  encrypting a plaintext  $m$  and an arbitrary function  $f$ , we can obtain a new ciphertext  $\text{ct}'$  encrypting  $f(m)$  without decrypting  $\text{ct}$ .

*Definition 1 (Public Key Homomorphic Encryption Scheme):* A public key homomorphic encryption scheme  $\mathcal{E}$  consists of a set of probabilistic polynomial-time algorithms (KeyGen, Enc, Dec, Eval) such that:

- **KeyGen**( $1^\lambda$ ): outputs the secret key  $\text{sk}$ , the public key  $\text{pk}$  and the evaluation key  $\text{evk}$  given the security parameter  $\lambda$ . The evaluation key is also public, and it is used to perform the homomorphic operations over ciphertexts.
- **Enc**( $\text{pk}, m$ ): Outputs a ciphertext  $\text{ct}$  encrypting  $m$  under the public key  $\text{pk}$ .
- **Dec**( $\text{sk}, \text{ct}$ ): Outputs a message  $m$ . If the algorithm cannot recover  $m$  from  $\text{ct}$ , the output is  $\perp$ .
- **Eval**( $\text{evk}, f, \text{ct}_1, \dots, \text{ct}_n$ ): Outputs a ciphertext  $\text{ct}'_f$  such that  $\text{Dec}(\text{sk}, \text{ct}'_f) = f(m_1, \dots, m_n)$ , where  $\text{ct}_i \leftarrow \text{Enc}(\text{pk}, m_i)$  for all  $i \in \{1, \dots, n\}$ .

#### 1) BOOTSTRAPPING

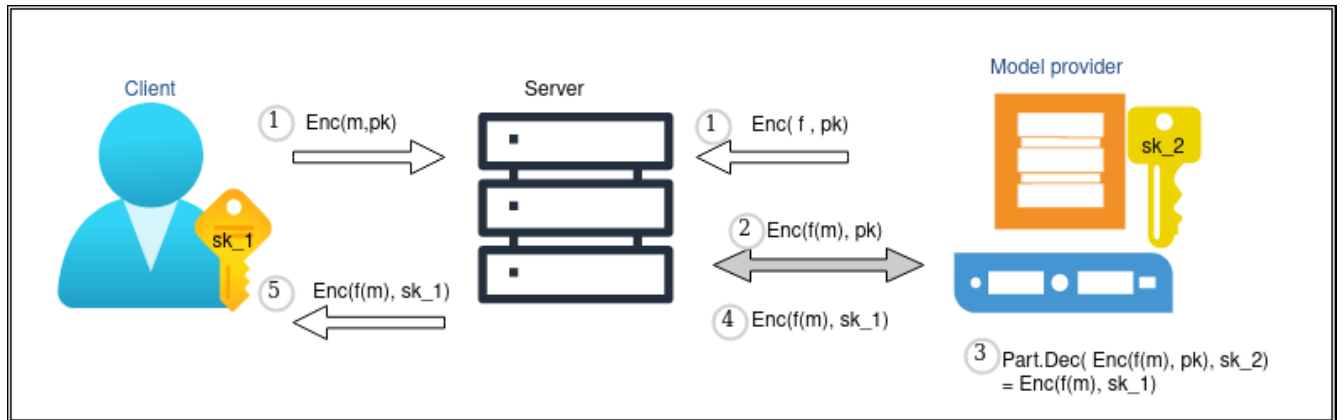
If a homomorphic encryption (HE) scheme has an additional algorithm called *bootstrapping*, we say that it is a *fully* HE scheme.

Bootstrapping is a technique which reduces the error within a ciphertext, which enables the computation of arbitrary functions. In Section III-D, we will describe in detail the bootstrapping procedure of TFHE, now we give a high-level overview of the general template of this technique.

Let  $(\text{pk}_1, \text{sk}_1)$  and  $(\text{pk}_2, \text{sk}_2)$  be two different pairs of keys produced by KeyGen and let  $\text{ct} \leftarrow \text{Enc}(\text{pk}_1, m)$ . To bootstrap  $\text{ct}$ , we first need to encrypt  $\text{sk}_1$  under  $\text{pk}_2$ , obtaining  $\text{sk}'_1 \leftarrow \text{Enc}(\text{pk}_2, \text{sk}_1)$ . Then we encrypt  $\text{ct}$  under  $\text{pk}_2$ , obtaining  $\text{ct}' \leftarrow \text{Enc}(\text{pk}_2, \text{ct})$ . Finally, the crucial step is to decrypt homomorphically  $\text{ct}'$  using  $\text{sk}'_1$  via the Eval procedure. That is:  $\text{ct}'' \leftarrow \text{Eval}(\text{evk}, \text{Dec}, \text{ct}', \text{sk}'_1)$ . By construction, we have that  $m \leftarrow \text{Dec}(\text{sk}_2, \text{ct}'')$ . The error within  $\text{ct}''$  consists of the error of a freshly encrypted ciphertext plus the error introduced by homomorphically evaluating the decryption circuit. If the error of  $\text{ct}$  corresponds to an evaluation of a circuit of less depth than the decryption circuit, then the error of  $\text{ct}''$  will be less than the one contained in  $\text{ct}$ .

#### 2) SECURITY ASSUMPTIONS

The security of the most well-known FHE schemes currently in use is based on the *learning with errors* (LWE) or the



**FIGURE 1.** Illustration of an application scenario of MPHE (two-party case). (1) Once a client and a service provider generate a common key ( $pk$ ) during the setup phase, both can send their item ( $m$  : client’s data,  $f$  : machine learning model) encrypted under the key to the server. (2) After the computation is over, the server sends the output ( $Enc(f(m), pk)$ ) to the model provider so that (3) he can run partial decryption with his own secret key  $sk_2$  and (4) sends the output which is an encryption of  $Enc(f(m), sk_1)$  to the server. (5) The server sends the given value from the model provider to the client, so that he can obtain  $f(m)$  by decrypting the ciphertext with its own secret key  $sk_1$ .

ring learning with errors (RLWE) problems. We describe the decisional version of both problems below.

**Definition 2 (Decisional Learning with Errors (LWE)):** Let  $n$  be a power of 2, let  $q \geq 3$  be an odd integer and let  $\chi$  be a distribution over  $\mathbb{Z}_q^n$ . The  $LWE_{n,q,\chi}$  problem is to distinguish the following two distributions over  $\{(a_i, b_i)\}_i \subset \mathbb{Z}_q^n \times \mathbb{Z}_q$ : In the first distribution one samples  $(a_i, b_i)$  uniformly at random from  $\mathbb{Z}_q^n \times \mathbb{Z}_q$  and, in the second distribution, one secretly samples  $s \leftarrow \mathbb{Z}_q^n$  uniformly and then defines  $(a_i, b_i = \langle a_i, s \rangle + e_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$  by sampling  $a_i \leftarrow \mathbb{Z}_q^n$  uniformly and  $e_i \leftarrow \chi$ .

**Definition 3 (Decisional Ring Learning with Errors (RLWE)):** Let  $f(x) = x^n + 1$  where  $n$  is a power of 2. Let  $q \geq 3$  be an odd integer. Let  $\chi$  be a distribution over  $\mathcal{R}$ . The  $RLWE_{n,q,\chi}$  problem is to distinguish the following two distributions over  $\{(a_i, b_i)\}_i \subset \mathcal{R}_q^2$  defined as follows: In the first distribution one samples  $(a_i, b_i)$  uniformly at random from  $\mathcal{R}_q^2$  and, in the second distribution, one secretly samples  $s \leftarrow \mathcal{R}_q$  uniformly and then defines  $(a_i, b_i = a_i s + e_i) \in \mathcal{R}_q^2$  by sampling  $a_i \leftarrow \mathcal{R}_q$  uniformly and  $e_i \leftarrow \chi$ .

We use the fact that the TFHE scheme is IND-CPA secure under the assumption that (R)LWE (defined above) is hard.

### C. MULTIPARTY FHE

In this paper, we are concerned with a generalization of FHE to a setting where we have multiple parties. Let  $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_k\}$  be a set of  $k$  parties. Informally, a *multiparty homomorphic encryption scheme* (MPHE) over  $\mathcal{P}$  is a homomorphic encryption scheme where the secret key is a function of the  $k$  individual secret keys of each party  $s_1, \dots, s_k$ . In this paper, we are interested in the case where the secret key  $s$  of the MPHE is computed as  $s = s_1 + \dots + s_k$ . Notice that obtaining this secret key structure from existing lattice-based HE schemes is already well-known and simple in a common random string (CRS) model [4], [5], [19]. For example, following the same notation as in subsection III-A,

the ciphertext  $LWE_{s_1}(0) + \dots + LWE_{s_k}(0)$  can be decrypted using the secret key  $s$  provided that the same mask  $a$  was used for all parties as the masks of these  $k$  LWE ciphertexts.<sup>3</sup> We follow the most communication efficient MPHE protocol [5], which can be described as follows:

- 1) Each party generates its own (local) public key using a common random string and distributes the keys to all the predefined users  $\mathcal{P}_i$ 's.
- 2) All parties locally generate (public) bootstrapping keys and ciphertexts by aggregating the given  $k$  different keys, obtained in 1). Then, they send them to a computing party (server).
- 3) The computing party generates the global evaluation key to perform the necessary operations over a set of given ciphertexts.
- 4) The computing party can now perform homomorphic operations over ciphertexts given by the  $k$  parties using the global keys computed in step 3).

We note that step 1) is a (one-round) setup for  $k$  parties, and step 3) is a (non-interactive) setup for the computing party. Therefore, for the same party set  $\mathcal{P}$ , the outcomes of steps 1) and 2) are reused over and over. The difficulty of building *practically efficient* MPHE schemes stems from the fact that more complex types of keys also need to be *condensed* to a global key from the individual keys produced from the parties in  $\mathcal{P}$  (step 3).

### III. BACKGROUND ON TFHE

The goal of this section is to provide the reader with a basic understanding of TFHE and its bootstrapping procedure. We refer the interested reader to [14], [17], [18], and [20] for further details on the topics exposed in this section.

<sup>3</sup>However, due to security issues, this technique is not directly used for generating ciphertexts, only for generating keys.

### A. TFHE CIPHERTEXTS

We denote the ciphertext modulus as  $q$  and encode a message  $m \in \{0, 1\}$  as  $\Delta m$ , where  $\Delta = \lfloor q/8 \rfloor$ . In our design, we only focus on binary messages and binary secret keys for each party.

- An LWE ciphertext is defined as  $\text{LWE}_s(m) := (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ , where  $\mathbf{a}$  is a random vector,  $b = \langle \mathbf{a}, \mathbf{s} \rangle + \Delta m + e$  for a message  $m \in \{0, 1\}$ , a secret key  $\mathbf{s} \in \mathbb{Z}^n$ , and a noise term  $e \leftarrow \chi_\theta$ . We call  $\mathbf{a}$  the *mask* and  $b$  the *body* of an LWE ciphertext. To decrypt, we compute  $b - \langle \mathbf{a}, \mathbf{s} \rangle = \Delta m + e$  and then apply a rounding operation to obtain the message  $m$ .
- A RLWE ciphertext is defined as  $\text{RLWE}_s(m) := (a, b) \in \mathcal{R}_q^2$ , where  $b = as + \Delta m + e'$  for a random polynomial  $a \in \mathcal{R}_q$ , a message polynomial  $m \in \mathcal{R}_2$ , a secret key  $s \in \mathcal{R}$ , and  $e' \leftarrow \chi_\theta$ . Similarly,  $a$  and  $b$  are called the mask and the body of an RLWE ciphertext, respectively. To decrypt, we compute  $b - as = \Delta m + e'$  and then apply a rounding operation to obtain the message  $m$ .
- Fix a base  $B_g$  and  $\ell = O(\log q)$ , and the gadget vector  $\mathbf{g} = (1, g, \dots, g^{\ell-1})^t$ . An RGSW ciphertext is of the form  $\text{RGSW}_s(m) := (\mathbf{a}, \mathbf{b}) \in \mathcal{R}_q^{2\ell \times 2}$ , where  $\mathcal{R}_q = \text{and } \mathbf{b} = \mathbf{H} + m\mathbf{G}$ , each row of  $\mathbf{H}$  is a  $\text{RLWE}_s(0)$  and  $\mathbf{G}$  is a gadget matrix defined as  $\mathbf{G} := \mathbf{I}_2 \otimes \mathbf{g}$ . The decryption process of a RGSW is not relevant to this work and we omit it.
- $\text{TrivialNoiseless}(\text{tag}, \mu)$ : it takes as input a tag  $\text{tag} \in \{\text{LWE}, \text{RLWE}, \text{RGSW}\}$  which indicates the ciphertext form, and a message  $\mu$ , and outputs a ciphertext with the form according to  $\text{tag}$  with a noiseless mask.

### B. EXTERNAL AND INTERNAL PRODUCTS

Given  $\mathbf{C}_1 = \text{RGSW}_s(m)$  and  $\mathbf{c}_2 = \text{RLWE}_s(\mu)$ , the *external product* produces the ciphertext  $\text{RLWE}_s(m\mu)$ . It is defined as follows:

$$\begin{aligned} \square : \text{RGSW} \times \text{RLWE} &\rightarrow \text{RLWE} \\ (\mathbf{C}_1, \mathbf{c}_2) &\mapsto \mathbf{C}_1 \square \mathbf{c}_2 = \mathbf{G}^{-1}(\mathbf{c}_2) \cdot \mathbf{C}_1. \end{aligned}$$

The *internal product* of two RGSW ciphertexts  $\mathbf{C}_1$  and  $\mathbf{C}_2$  is equivalent to computing the external product of  $\mathbf{C}_1$  with all  $2\ell$  RLWE rows  $(\mathbf{c}_1, \dots, \mathbf{c}_{2\ell})^t$  of  $\mathbf{C}_2$ . It is defined as follows:

$$\begin{aligned} \boxtimes : \text{RGSW} \times \text{RGSW} &\rightarrow \text{RGSW} \\ (\mathbf{C}_1, \mathbf{C}_2) &\mapsto (\mathbf{C}_1 \square \mathbf{c}_1, \dots, \mathbf{C}_1 \square \mathbf{c}_{2\ell}). \end{aligned}$$

### C. CMUX GATE

The main use of external products in TFHE is in the instantiation of the *controlled multiplexer* (CMUX) gate. Given two RLWE ciphertexts  $\mathbf{c}_0$  and  $\mathbf{c}_1$  encrypting messages  $m_1$  and  $m_2$  respectively and a RGSW ciphertext  $\mathbf{C}$  encrypting a (controller) bit  $b$ , the CMUX gate returns a RLWE ciphertext encrypting  $m_b$ . That is, the CMUX essentially selects either  $\mathbf{c}_0$  or  $\mathbf{c}_1$  depending on the controller bit encrypted in  $\mathbf{C}$ . The CMUX gate is instantiated as follows:

$$\text{CMUX}_{\square}(\mathbf{C}, \mathbf{c}_0, \mathbf{c}_1) \leftarrow \mathbf{C} \square (\mathbf{c}_1 - \mathbf{c}_0) + \mathbf{c}_0.$$

It can also be instantiated with internal products, producing a RGSW ciphertext encrypting  $m_b$ , as we use in our bootstrapping key generation. In this case, it is computed as follows:

$$\text{CMUX}_{\boxtimes}(\mathbf{C}, \mathbf{C}_0, \mathbf{C}_1) \leftarrow (\mathbf{C}_1 - \mathbf{C}_0) \boxtimes \mathbf{C} + \mathbf{C}_0.$$

### D. TFHE BOOTSTRAPPING

The bootstrapping of TFHE consists of four algorithms: modulus switching, blind rotation, sample extraction and key switching. In what follows, we describe how these algorithms work and their role in TFHE bootstrapping.

#### 1) MODULUS SWITCHING

*Modulus switching* is the first step of the bootstrapping algorithm and compresses the information of the input LWE ciphertext to a smaller space. Consider a ciphertext  $\text{ct} = \text{LWE}_s = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ . The goal of modulus switching is to change the modulus of  $\text{ct}$  from  $q$  to another modulus  $h$ . This is done by changing the modulus of all components of  $\text{ct}$  (let us say that  $a_n := b$ ) as follows

$$\tilde{a}_i = \left\lfloor \frac{(ha_i \bmod q)}{q} \right\rfloor \in \mathbb{Z}_h.$$

In TFHE bootstrapping, the key switching is done from  $q$  to the modulus  $2N$ , where  $N$  corresponds to the polynomial degree defining  $\mathcal{R}_q$ . Notice that  $2N$  is much smaller than  $q$ , for example  $N = 2^{10}$  and  $q = 2^{32}$ .

#### 2) BLIND ROTATION

The core and most computationally expensive procedure of TFHE bootstrapping is called *blind rotation*. This algorithm takes a trivial RLWE encryption of a vector  $v$  and rotates it by a ciphertext  $\text{LWE}_s(m) = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ . That is, the result of the blind rotation is a ciphertext  $\text{RLWE}_s(X^{-\bar{m}}v) \in \mathcal{R}_q^2$ , where  $\bar{m}$  is a rounded approximation of  $b - \langle \mathbf{a}, \mathbf{s} \rangle$ . In more detail, notice that we can write

$$\bar{m} = \bar{b} - \sum_{j=1}^n s_j \bar{a}_j,$$

where  $\bar{b}$  and  $\bar{a}_j$  are rounded approximations of  $b$  and  $\mathbf{a}_j$  respectively. The algorithm starts by generating a trivial RLWE encryption of  $X^{\bar{b}}v$  (referred to as *accumulator* in the literature). The goal now is to obtain an RLWE encryption of  $X^{\bar{m}}v$ . Notice that we only need to multiply the trivial encryption by  $X^{-a_j}$  for all  $j$  if  $s_j = 1$ . This conditional is instantiated using a CMUX gate and the bootstrapping keys (or *bootstrapping keys*), which are RGSW encryptions of the secret key components. That is,  $\text{bsk}[j] \leftarrow \text{RGSW}_s(s_j)$  for  $1 \leq j \leq n$ .

**Algorithm 1** Blind Rotation in the Binary Case

```

1: acc ← (0, . . . , 0, X-b̄v)
2: for j = 1 to n do
3:   acc ← acc + bsk[j] □ ((X-aj - 1) · acc)
4: end for
5: return acc
    
```

This blind rotation procedure only works when the secret keys are binary. In Section IV, we will see two approaches to perform a blind rotation where the secret keys are sampled from an arbitrary distribution.

3) SAMPLE EXTRACTION

The output of the bootstrapping procedure should be an LWE ciphertext, however, the output of the blind rotation procedure is an RLWE ciphertext. For this reason, after running the blind rotation, it is necessary to extract a LWE ciphertext from the output RLWE. Assume that the output RLWE ciphertext encrypts a plaintext polynomial with constant term  $\mu_0$ . The extraction process consists in carefully selecting some of the coefficients of the RLWE ciphertext and then building the output LWE ciphertext corresponding to  $\mu_0$ . How this selection is done is not relevant to this work, and we refer the reader to the literature provided at the beginning of the section for further details.

4) KEY SWITCHING

At this point, we have a ciphertext under the secret key  $s'$  but we need a ciphertext under  $s$ . To obtain the desired ciphertext, we can apply *key switching*, which is a common technique used in homomorphic encryption. Consider a LWE ciphertext  $c' = (a, b)$  encrypting a plaintext  $m$  under  $s'$ . The basic idea of key switching is to compute  $b - \langle a, s' \rangle$  homomorphically providing  $s'$  as an encryption under  $s$ . This will cancel out  $s'$  and allow us to decrypt the resulting ciphertext with  $s$ . The encryptions of the bits of  $s$  under  $s'$  correspond to the *key switching keys*, which are constructed as follows. Let  $B_{ks} < \frac{q}{2}$  be the key switching basis and let  $l_{ks}$  be such that  $\log_B(l_{ks}) = q$ . For every bit  $s'_i$  of  $s'$ , the corresponding key switching key will be

$$ksk_i = \{LWE_s(s'_i), LWE_s(s'_i B_{ks}), \dots, LWE_s(s'_i B_{ks}^{l_{ks}-1})\}.$$

We can now compute

$$c = (0, \dots, 0, b) - \sum_{i=1}^n \text{Decomp}(a_i, ksk_i),$$

where  $\text{Decomp}(a_i) = (a_{i,0}, \dots, a_{i,l_{ks}-1})$  is the decomposition of  $a_i$  in base  $B_{ks}$ . It is easy to see that  $c$  is a correct LWE ciphertext encrypting  $m$  under the secret key  $s$ .

**IV. TFHE BLIND ROTATION WITH ARBITRARY SECRET KEY DISTRIBUTION**

In the following subsections, we explain the two most efficient state-of-the-art approaches of TFHE blind rotation which can work with an arbitrary secret key distribution.

**A. BLIND ROTATION OF LEE ET AL**

The work of Lee et al. [12] introduces a new efficient blind rotation algorithm that works with arbitrary secret key distributions and requires small bootstrapping keys. Very briefly and omitting a lot of technicalities, each loop of their blind rotation algorithm consists of a number of external products followed by one automorphism (key switching). That is, starting with an accumulator  $acc = RLWE_s(f'(X))$  and a collection of bootstrapping keys  $bsk_j := RGSW_s(X^{s_j})$  for  $j \in J$ , for some indexing set  $J$ , their blind rotation algorithm proceeds by multiplying  $acc$  by all  $bsk_j$ , obtaining  $acc = RLWE_s(f'(X)X^{\sum_{j \in J} s_j})$ . Then, applying an automorphism they obtain

$$acc = RLWE_s(f'(X^g)X^{g \sum_{j \in J} s_j}),$$

for a suitable integer  $g$  depending on the polynomial degree  $N$ . After repeating this process with other indexing sets they finally obtain  $acc = RLWE_s(f(X)X^{\beta + \langle \alpha, s \rangle})$ .

This blind rotation technique requires a total of at least  $1.5n$  external products, where  $n$  is the dimension of the input LWE ciphertext.

To use this blind rotation algorithm in the MPHE setting, they require the construction of a common blind rotation key ( $bsk$ ) and a common automorphism key ( $ak$ ). We only give a short summary of how a computing party can obtain  $bsk$  and refer the interested reader to the original paper for the construction of  $ak$ . Let  $s_1, \dots, s_k$  be the secret keys of a set of  $k$  parties. Each party  $j$  computes  $bsk_{j,i} = RGSW_s(X^{s_{j,i}})$  where  $s_{j,i}$  is the  $i$ -th component of  $s_j$  and sends this to the computing party. Then, the computing party obtains

$$bsk_i = \boxtimes_{j \in J} bsk_{j,i} = RGSW_s(X^{S_{*,i}}).$$

**B. BLIND ROTATION OF JOYE AND PAILLIER**

As mentioned in the introduction, each loop of the blind rotation in the work of Joye and Paillier [13] consists of  $k$  RGSW ciphertext additions followed by an external product. Their bootstrapping key is a set of RGSW ciphertexts which encrypt either 0 or 1. Basically, a client generates  $k$  RGSW ciphertexts, denoted by  $\{\text{key}_{j,i}\}_{i \in [k]}$ , per secret key component  $S_j$  ( $j \in \{0, \dots, n\}$ ) such that all of them are encryptions of 0 or only one of them encrypts 1 and the rest encrypt 0. Then, the computing party runs a selection algorithm to choose the corresponding  $X^{a_j S_j}$ , as follows:

$$1 + (X^{a_j} - 1) \text{key}_{j,1} + (X^{2a_j} - 1) \text{key}_{j,2} + \dots + (X^{ka_j} - 1) \text{key}_{j,k}.$$

For example, if  $S_j = 3$ , then  $\text{key}_{j,3}$  is an encryption of 1 and the rest are encryptions of 0. Here, we note that the  $X^{ia_j}$ 's for  $i \in [k]$  are precomputed and the product with  $\text{key}_i$  is a multiplication in the FFT domain which is the most expensive part in this selection algorithm.

For the interested reader, we now provide a more detailed description of how the bootstrapping keys are constructed and how to use them during the blind rotation step. For

simplicity, we will assume that we sample keys from the set  $\mathcal{S} = \{0, \dots, k\}$ . Let  $\mathbf{I}$  be the indicator function, that is,  $\mathbf{I}\{t = S_j\} = 1$  when  $t = S_j$  and  $\mathbf{I}\{t = S_j\} = 0$  otherwise. Notice that we can write

$$X^{a_j S_j} = \sum_{i=0}^k \mathbf{I}\{i = S_j\} X^{ia_j} = \sum_{i=1}^k \mathbf{I}\{i = S_j\} (X^{ia_j} - 1).$$

Therefore, we can compute acc by setting

$$\text{key}_{j,i} \leftarrow \text{RGSW}(\mathbf{I}\{i = S_j\}),$$

and iterating

$$\text{acc} \leftarrow \text{acc} + \left( \sum_{i=1}^k (X^{ia_j} - 1) \text{key}_{j,i} \right) \boxtimes \text{acc}.$$

### C. CHOOSING A BLIND ROTATION ALGORITHM FOR MPHE TFHE

As we have mentioned in the introduction, the goal of this work is to extend [13] to the MPHE setting by constructing suitable bootstrapping keys and see under which conditions this approach is better than that of [12]. This analysis is done in Section VIII-A.

### V. BOOTSTRAPPING KEY GENERATION FOR MPHE TFHE

Since we wish to use Joye and Paillier’s approach for TFHE bootstrapping, our main goal in this section is to create on the fly a common bootstrapping key compatible with their blind rotation algorithm. This new key will be created from the locally generated keys sent by all users. To do this, we introduce our novel algorithm called *homomorphic indicator* and we explain how to use it to build the MPHE bootstrapping keys.

#### A. HOMOMORPHIC INDICATOR

Given an array  $\mathbf{A}$  of encryptions of 0’s and 1’s, the goal of the homomorphic indicator algorithm is to homomorphically produce a new array with an encryption of 1 in the position corresponding to the Hamming weight ( $W_H$ ) of  $\mathbf{A}$ , that is, the output array will have an encryption of 1 in the  $W_H(\mathbf{A})$ -th position and encryptions of 0 everywhere else.

Let us start by explaining how a plaintext version of the homomorphic indicator works. The algorithm takes as input an array  $\mathbf{A}$  of  $k$  slots<sup>4</sup> filled with 0’s and another array  $\mathbf{C}$  with also  $k$  slots but filled with 0’s and 1’s, which we will call *controller bits*. The algorithm proceeds by iterating over  $\mathbf{C}$  as follows. If the first controller bit is 1, the first slot of  $\mathbf{A}$  becomes 1, and the rest of  $\mathbf{A}$  is untouched. If the second controller bit is also 1,  $\mathbf{A}$  remains the same except the first and second slots, where the first slot becomes 0, and the second slot becomes 1. After repeating this  $k$  times, the array  $\mathbf{A}$  will have a 1 at the position corresponding to the number of 1’s in  $\mathbf{C}$ .

<sup>4</sup>A slot of an array refers to an element of an array in this paper, we use these terms interchangeably.

In the ciphertext case, we start with two arrays of  $k+1$  slots, denoted by  $\mathbf{A}^{old}$  and  $\mathbf{A}^{new}$ , where the 0-th slot of  $\mathbf{A}^{old}$  contains an encryption of 1 and the rest are set to encryptions of 0. Additionally, we need  $k$  controller bits  $\{\mathbf{C}_i\}_{i \in [k]}$ , each of which encrypts a bit. We note that the last  $k$  slots of  $\mathbf{A}^{old}$  are what we have at the beginning in the cleartext case. We store the desired value (which is 1) at the 0-th slot. In each loop,  $\mathbf{A}^{new}$  is updated based on the corresponding controller bit and the values in the slots of  $\mathbf{A}^{old}$ , as follows. For each  $j \in [k]$ ,  $\mathbf{A}^{new}[j] \leftarrow \mathbf{A}^{old}[j-1]$  if  $\mathbf{C}_i$  is an encryption of 1, otherwise,  $\mathbf{A}^{new}[j] \leftarrow \mathbf{A}^{old}[j]$ . This can be instantiated as

$$\mathbf{A}^{new}[j] := (\mathbf{A}^{old}[j-1] - \mathbf{A}^{old}[j]) \boxtimes \mathbf{C}_i + \mathbf{A}^{old}[j].$$

In order to update the first slot, we need to deal with the 0-th slot as well since it influences on the first slot of  $\mathbf{A}^{new}$ . The 0-th slot should also be updated to 0 or remain 1. That is,  $\mathbf{A}^{new}[0]$  is updated to  $\mathbf{A}^{old}[0]$  if  $\mathbf{C}_i$  is an encryption of 0, otherwise,  $\mathbf{A}^{new}[0]$  becomes an encryption of 0. This is instantiated as

$$\mathbf{A}^{new}[0] := \mathbf{A}^{old}[0] \boxtimes (1 - \mathbf{C}_i).$$

After all the slots of  $\mathbf{A}^{new}$  are updated,  $\mathbf{A}^{old}$  is set as  $\mathbf{A}^{new}$  and we repeat the protocol until all  $k$  controller bits are used. At the end of the protocol, we will take the last  $k$  slots of  $\mathbf{A}^{new}$  as the desired result.

---

#### Algorithm 2 Homomorphic Indicator (Hom.Indicator)

---

**Input:**  $\{\mathbf{C}_i\}_{i \in [m]}$ ,  $\mathbf{A}^{new}$  and  $\mathbf{A}^{old}$ .

**Output:**  $\mathbf{A}^{old}$ .

```

for  $i \leftarrow 1$  to  $k$  do
  for  $j \leftarrow 1$  to  $k$  do
     $\mathbf{A}^{new}[j] := \text{CMUX}_{\boxtimes}(\mathbf{C}_i, \mathbf{A}^{old}[j], \mathbf{A}^{old}[j-1])$ 
  end for
   $\mathbf{A}^{new}[0] := \mathbf{A}^{old}[0] \boxtimes (1 - \mathbf{C}_i)$ 
  for  $j \leftarrow 0$  to  $k$  do
     $\mathbf{A}^{old}[j] := \mathbf{A}^{new}[j]$ 
  end for
end for

```

---

Note that we use the internal product to instantiate the CMUX gate and the multiplication between two ciphertexts. However, one can use the internal product or any type of homomorphic multiplication. Since the multiplicative depth of the homomorphic indicator algorithm is linear in the size of the output (unit vector), the use of internal/external product is highly recommended if the size of the output is large, in order to manage the noise growth.

#### B. INSTANTIATION OF BOOTSTRAPPING KEY GENERATION

In this subsection, we explain how to use the homomorphic indicator to construct suitable bootstrapping keys for the blind rotation algorithm of [13]. We start by explaining how the users need to prepare their local keys, and then we detail how the server can build the bootstrapping keys from the local keys.

### 1) LOCAL BOOTSTRAPPING KEY GENERATION

We generate the local bootstrapping key similarly to [5], however, the instantiation is a bit different in order to minimize the noise. We adapt the typical technique, which transforms a private-key ciphertext into a public-key ciphertext. Our goal is to turn single key ciphertexts encrypted under a user's public key into a ciphertext encrypted under the global public key via one interaction. We specifically follow the way of [20] which is designed for TFHE ciphertexts. Each user does the following:

- 1) Generates a vector consisting of size  $m = O(n \log q)$  LWE/RLWE/RGSW encryptions of 0 under its own key, and distributes the vector of the ciphertexts to the other predefined  $k - 1$  users. In other words, each component of the vector looks like  $(a, b_i)$ , where  $b_i = as_i + e_i$ , with  $a$  being a common random string (CRS) which has already been provided to all the users. Once this step is done, the rest can be run multiple times without interactions among users.
- 2) Generates  $k$  random bit vectors of dimension  $m$ .
- 3) Computes  $k$  dot products between the bit vectors and the given  $k$  ciphertext vectors, including its own vector, and adds up the results. The outcome is an LWE/RLWE/RGSW encryption of 0 under the master secret key, which is  $s := (s_1 + s_2 + \dots + s_k)$ .

Every time all  $k$  users generate ciphertexts/local bootstrapping keys, each user does the second and the third step described above and add the message term on the desired spot at the end. We give an algorithm for this procedure below. In practice, we set  $m = \lceil 3 \log q \rceil$  as [6] analyzed.

---

#### Algorithm 3 Local Encryption (Local.Enc)

---

**Input:** A vector of dimension  $m$  consisting of ciphertexts (all of which can be one of LWE/RLWE/RGSW forms), denoted by  $\mathbf{V}$ , and a message  $\mu$ , a tag  $\text{tag} \in \{\text{LWE}, \text{RLWE}, \text{RGSW}\}$ .

**Output:** an LWE/RLWE/RGSW ciphertext  $c$  encrypting a message  $\mu$ .

Sample a random vector  $\mathbf{R} \leftarrow \{0, 1\}^m$

Computes  $\bar{\mu} := \text{TrivialNoiseless}(\text{tag}, \mu)$ .

Computes  $c := \langle \mathbf{V}, \mathbf{R} \rangle + \bar{\mu}$

---

### 2) GLOBAL BOOTSTRAPPING KEY GENERATION

The main goal of this algorithm is to create the global bootstrapping key of [13] *on the fly* using all clients' keys which have already been given at the beginning of the protocol. As we explained in Section II, the global bootstrapping key is an array of  $k$  RGSW ciphertexts per coefficient. We denote this key as  $\widehat{\text{bsk}} := [\widehat{\text{bsk}}_0, \dots, \widehat{\text{bsk}}_{n-1}]$ , where each component consists of  $k$  RGSW ciphertexts, that is  $\widehat{\text{bsk}}_i = [\text{RGSW}_s(b_{i,1}), \dots, \text{RGSW}_s(b_{i,k})]$ , where  $b_{i,j} \in \{0, 1\}$  for and  $j \in [k]$  and  $i \in [0, \dots, n - 1]$ .

To generate  $\widehat{\text{bsk}}$ , the following actions are required by each user and by the server (or a computing party).

**Users.** Each user  $i$  sends its (local) bootstrapping key  $\text{bsk}_i := (\text{RGSW}_s(s_{i,0}), \text{RGSW}_s(s_{i,1}), \dots, \text{RGSW}_s(s_{i,n-1}))$  to the server.

**Server.**

- Initialization: The server creates two arrays of  $k + 1$  slots per coefficient, denoted by  $\mathbf{A}^{\text{old}}$  and  $\mathbf{A}^{\text{new}}$ , respectively, where the first slot of  $\mathbf{A}^{\text{old}}$  is 1 and the rest are zeros.
- As soon as it receives the (local) bootstrapping keys  $\text{bsk}_1, \dots, \text{bsk}_k$  from the predefined  $k$  users, it executes algorithm 4, running Hom.Indicator as a subroutine.

---

#### Algorithm 4 Global Bootstrapping Key Generation

---

**Input:**  $\{\text{bsk}_i\}_{i \in [k]}$ ,  $\mathbf{A}^{\text{new}}$  and  $\mathbf{A}^{\text{old}}$ .

**Output:**  $\widehat{\text{bsk}}$ .

**for**  $t \leftarrow 0$  to  $n - 1$  **do**

**for**  $i \leftarrow 1$  to  $k$  **do**

    Parse  $\mathbf{C}_{i,t} := \text{bsk}_i[t]$

**end for**

$\mathbf{A} := \text{Hom.Indicator}(\{\mathbf{C}_{i,t}\}_{i \in [k]}, \mathbf{A}^{\text{new}}, \mathbf{A}^{\text{old}})$

$\widehat{\text{bsk}}[t] := [\mathbf{A}[1], \dots, \mathbf{A}[k]]$

  Refresh  $\mathbf{A}^{\text{new}}$  and  $\mathbf{A}^{\text{old}}$

**end for**

---

### C. A TOY EXAMPLE FOR 4 PARTIES

Assume that we have four users ( $k = 4$ ), and each has a secret key  $s_i$  for  $i \in [4]$ . For ease of exposition, we only focus on the constant term of each  $s_i$ , the process is the same for the rest of the coefficients. Let us say that  $s_{1,0} = 1, s_{2,0} = 1, s_{3,0} = 1$ , and  $s_{4,0} = 0$ , hence the constant term of the master secret key is 3 ( $S_0 = 3$ ). The server creates two arrays of  $k + 1 = 5$  elements, as we explained in the above section, called  $\mathbf{A}^{\text{old}}, \mathbf{A}^{\text{new}}$ . The array  $\mathbf{A}^{\text{old}}$  has an RGSW encryption of 1 at the first position and an RGSW encryption of 0 in the rest while  $\mathbf{A}^{\text{new}}$  consists of RGSW encryptions of 0.

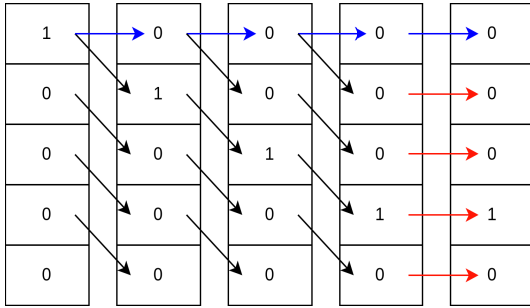
In the first iteration, since  $s_{1,0} = 1$ , we will set  $\mathbf{A}^{\text{new}}[1] = \text{RGSW}(1)$  and  $\mathbf{A}^{\text{new}}[j] = \text{RGSW}(0)$  for all  $j \neq 1$ . We then have an encryption of 1 in position 1. The same would happen in the next two iterations until we reach  $s_{4,0}$ . In this case, since  $s_{4,0} = 0$ , the algorithm does not change the plaintexts of all elements in the output array (but the noise is increased). At the end, we get an array with an RGSW encryption of 1 in position 3, and a RGSW encryption of 0 in the rest. The last 4 elements of the final array correspond to the bootstrapping keys needed for 4 parties in the blind rotation protocol of [13]. This example is depicted in Figure 2.

## VI. TFHE BOOTSTRAPPING IN THE MULTIPARTY SETTING

Once the predefined  $k$  users (parties) generate their own local bootstrapping key as discussed in Section V-B and upload them to server, the server runs Algorithm 4 to create the common/global bootstrapping key. The global bootstrapping key is stored, and the server can reuse it as long as the computation is for the same set of users.

After running the blind rotation algorithm of Joye and Paillier with the global bootstrapping key, the output is an





**FIGURE 2.** Example for  $k = 4$  of our bootstrapping key generation. The blue arrow shows the direction of moving the first slot. The black arrows show that the elements in the new array come from the previous slots in the old array when the corresponding secret key component is 1. The red arrow shows that the elements in the new array come from the same slots of the old array when the corresponding secret key component is 0.

RLWE ciphertext under the  $k$  “RLWE keys” defined over a polynomial ring of degree  $N$ . However, the result of a TFHE bootstrapping should be an LWE ciphertext encrypted under the  $k$  “LWE keys” to enable further computations.

As explained in Section III-D, in single key TFHE, we switch the ciphertext format from an RLWE to an LWE ciphertext by running an algorithm called *sample extraction*. However, the outcome has still dimension  $N$  which is larger than the original input dimension  $n$ . Therefore, TFHE introduced a key-switching algorithm to switch an LWE from dimension  $N$  to dimension  $n$ . We can adapt the single key version of this algorithm to the multiparty setting as follows.

### A. KEY SWITCHING IN THE MULTIPARTY SETTING

The key switching algorithm for the MPHE case is the same as in the single key TFHE setting, the only change is in the key switching keys. Assume that we want to switch from  $LWE_{s'}(m)$  to  $LWE_s(m)$  where  $s = s_1 + s_2 + \dots + s_k$  and  $s' = s'_1 + s'_2 + \dots + s'_k$  with  $s_i$  and  $s'_i$  being the secrets of the  $i$ -th party. Each user only needs to produce  $ksk_i := LWE_{s'}(s'_i)$  as their key switching key by running the function  $Local.Enc$ . To obtain the global key switching key, the server only needs to add up all  $ksk_i$ .

### B. OVERALL DESCRIPTION OF OUR BOOTSTRAPPING

We now have all the ingredients to perform MPHE bootstrapping using the blind rotation of [13]. We describe the process below.

**Users.** Each  $i$ -th user does the following:

- **Local public keys:** generates its own local public key; a vector of ciphertexts of dimension  $m (= O(\log q))$  per ciphertext format (LWE, RLWE, RGSW), where all the masks of the ciphertexts are set to  $a$  which is a common random string (CRS). We denote the vectors by  $V_{LWE}, V_{RLWE}, V_{RGSW}$ .
- **Local bootstrapping keys:** generates local bootstrapping key  $bsk_i$  by running  $Local.Enc(V_{RGSW}, RGSW, s_{i,j})$  for all  $j \in [n]$ .

Similarly, generates local key switching key  $ksk_i$  by running  $Local.Enc(V_{LWE}, LWE, s_i)$ .

- **Ciphertexts:** generates ciphertexts for homomorphic evaluation by running  $Local.Enc(V_{LWE}, LWE, m)$ , where  $m \in \{0, 1\}$ .

**Server.**

- **Setup:** runs Algorithm 4 to generate a global bootstrapping key  $bsk$ . Generates a global key switching key denoted by  $ksk$  by summing up all given  $ksk_i$  given by  $k$  users.
- **After a gate operation:** run Joye and Paillier’s bootstrapping with  $bsk$  and  $ksk$ .

### C. SECURITY

We rely on the fact that TFHE is IND-CPA secure (due to the (R)LWE assumption (see Section II) for one key) and the security of ciphertexts under multiple keys discussed in [2] and [5].

- **Local Public key:** each local key is encrypted under the owner’s key, therefore the other users cannot decrypt the secret key from the public key (due to IND-CPA of TFHE).
- **Common public keys:** common public keys (including bootstrapping keys and key-switching keys) are encrypted under  $sk_1 + sk_2 + \dots + sk_k$ . Even if  $k-1$  keys are compromised, the ciphertext is still encrypted under one valid and uncompromised key, hence the IND-CPA of TFHE guarantees the security of the common public keys.
- **Security model:** We present our construction under the assumption that both the users and the server are semi-honest. Lattice-based commitment schemes and zero-knowledge proofs can be used to transform a semi-honest MPHE protocol into a maliciously secure one, see for example [21].

### VII. NOISE ANALYSIS

In this section, we analyze the noise growth of the setup phase and the final noise after bootstrapping. We start by briefly recalling the noise growth of TFHE ciphertexts [18]. Let  $C_1$  and  $C_2$  be two RGSW ciphertexts encrypting  $m_1$  and  $m_2$ , respectively. After one internal product between  $C_1$  and  $C_2$ , denoted by  $C' := C_1 \boxtimes C_2$ , the variance of the noise contained in the output is

$$\begin{aligned} \text{Var}(\text{Err}(C')) &\leq \ell \cdot N \cdot g^2 \cdot \text{Var}(\text{Err}(C_1)) + (1 + N) \\ &\quad + \text{Var}(\text{Err}(C_2)). \end{aligned} \tag{1}$$

Let  $e$  be the initial noise in each user’s (local) public key, that is,  $\text{Var}(e) = \theta^2$  as we defined in Section II. After receiving all the public keys from the predefined users, the noise contained in each user’s bootstrapping key becomes  $k \cdot \iota \cdot e$  due to additions, where  $\iota = \lceil 3 \log q \rceil$  to guarantee LWE security [6] (as we mentioned in Section V-B). Therefore,

$$\text{Var}(\text{Err}(bsk_i)) \leq k \cdot \iota \cdot \theta^2.$$

The noise becomes larger during the generation of the global bootstrapping key. Each element in the initial array is a plaintext which has *no noise*. We can upper-bound the variance of the  $i$ -th output array  $A_i$  of one homomorphic indicator operation as

$$\text{Var}(\text{Err}(A_i)) \leq k \cdot \ell \cdot N \cdot g^2 \cdot \text{Var}(\text{Err}(\text{bsk}_i)) + k(1 + N),$$

from Equation (1). Each element of  $\widehat{\text{bsk}}$  is a vector of RGSW ciphertexts (denoted by  $[A_0, \dots, A_{n-1}]$ ). Each component of such vectors is created via homomorphic indicator which consists of  $k$  consecutive CMUX gates (internal products). We define the variance of  $\widehat{\text{bsk}}$  to be the maximum variance of  $A_i$ 's. Hence, the noise contained in  $\widehat{\text{bsk}}$  can be bounded as follows:

$$\begin{aligned} \text{Var}(\text{Err}(\widehat{\text{bsk}})) &= \max_i(\text{Var}(\text{Err}(A_0)), \dots, \text{Var}(\text{Err}(A_{n-1}))) \\ &\leq k \cdot \ell \cdot N \cdot g^2 \cdot \text{Var}(\text{Err}(\text{bsk}_i)) \\ &\quad + k \cdot (1 + N) \cdot \epsilon^2. \end{aligned}$$

Now, we run bootstrapping as specified in [13] with the constructed  $\widehat{\text{bsk}}$ . In the blind rotation algorithm of [13], there are  $k$  additions among the corresponding  $\widehat{\text{bsk}}$  components in every loop. After the additions, the output, say  $C_{add}$ , contains the noise of which the variance is

$$\text{Var}(\text{Err}(C_{add})) \leq 2 \cdot k \cdot \text{Var}(\text{Err}(\widehat{\text{bsk}})).$$

In terms of noise growth, the blind rotation of [13] is viewed as TFHE bootstrapping with  $C_{add}$  as the bootstrapping key. After blind rotation, the output RLWE ciphertext  $c$  has a noise that can be bounded as follows:

$$\text{Var}(\text{Err}(c)) \leq n \cdot \ell \cdot N \cdot g^2 \cdot \text{Var}(\text{Err}(C_{add})) + n \cdot (1 + N) \epsilon^2.$$

Performing a key switching to convert  $c$  to an LWE ciphertext, denoted by  $\mathbf{c}$ , only adds a small noise  $N \cdot \ell \cdot g^2 \cdot \text{Var}(\text{Err}(\text{evk}_i))$  which is not a dominant term compared to the noise in  $c$ . Therefore, the variance of the final noise in our case has complexity

$$O(k^3 \cdot \iota \cdot n \cdot \ell^2 \cdot N^2 \cdot g^4 \cdot \theta^2),$$

whereas [12] has  $O(k^2 \cdot \iota \cdot n \cdot \ell^2 \cdot N^2 \cdot g^4 \cdot \theta^2)$ . Under the central limit heuristic, the noise contained in the output LWE ciphertext  $\mathbf{c}$  has the following bound with overwhelming probability:

$$\|\text{Err}(\mathbf{c})\|_\infty \leq 6\sqrt{\text{Var}(\text{Err}(\mathbf{c}))}.$$

As a result, we can find the relation among parameters (mainly  $q, N, n$  and  $k$ ) from the following bound in order to guarantee the correctness,

$$\|\text{Err}(\mathbf{c})\|_\infty \leq q/16.$$

Overall, our bootstrapping key generation algorithm gives the same noise propagation as Lee et al.'s approach, however, our choice of bootstrapping adds more noise depending on the number of users during blind rotation. Therefore, the final noise contained in the output of bootstrapping using our technique has a bigger factor ( $k^{1.5}$ ) than the approach of Lee et al., where the noise grows linear in  $k$ .

**TABLE 1. Comparison in terms of the number of expensive operations such as external products denoted by  $T_{mult}$  and a point-wise multiplication between two polynomials of degree  $N$  in FFT domain, denoted by  $T_{PM}$  used in blind rotation.  $w$  is a small constant.**

Scheme	blind rotation
[12]	$(1.5n + w) \cdot T_{mult}$
[13]	$n \cdot T_{mult} + k \cdot n \cdot (4 \cdot \ell \cdot T_{PM})$

### VIII. PERFORMANCE EVALUATION

In this section, we detail our implementation choices and provide the best parameter sets together with their benchmarks with respect to the actual running time of the main homomorphic operation and noise growth. We came across some difficulties when comparing our implementation numbers with the existing design of [12] since they did not implement their multi-key extension. Therefore, for a fair comparison, we analyze the noise growth of both designs in terms of crucial parameters such as  $k, N, n, \ell$ , and provide benchmarks with respect to the running time of the dominant operation of both cases on the same machine.

#### A. COMPLEXITY COMPARISON

##### 1) BOOTSTRAPPING RUNNING TIME

In Table 1 we provide the computational cost of the blind rotation algorithms of [12] and [13] in terms of the cost of one point-wise multiplication in the FFT domain (denoted by  $T_{PM}$ ) and one external product (denoted by  $T_{mult}$ ). An external product consists of  $4\ell$  point-wise multiplications in the FFT domain and  $2\ell + 2$  FFT conversions of a polynomial (denoted by  $T_{FFT}$ ) (from the standard domain to FFT domain and vice versa). Since the dominant part of one external product is  $(2\ell + 2)$  FFT conversions, we can consider  $T_{FFT}$  as the dominant factor. The actual computational cost of blind rotation will, of course, depend on the chosen parameters for the schemes.

We want to know up to which  $k$  our MPHE approach based on [13] outperforms that of [12]. For this, we can use Table 1 and upper bound  $k$  as follows:

$$k \leq \frac{0.5 \cdot T_{mult}}{4 \cdot \ell \cdot T_{PM}} = \frac{1}{2} + \frac{(\ell + 1)}{4\ell} \cdot \frac{T_{FFT}}{T_{PM}}. \quad (2)$$

Theoretically, the ratio between  $T_{FFT}$  and  $T_{PM}$  (denoted by  $r$ ) is  $O(\log N)$ , however, the hidden constant in the complexity varies depending on machines and FFT libraries the server runs. Moreover, with the bound on the noise after blind rotation derived in Section VII, we obtain the following relation among parameters

$$k \leq \left( \frac{q}{96 \cdot \sqrt{\iota} \cdot \sqrt{n} \cdot \ell \cdot N \cdot g^2 \cdot \theta} \right)^{\frac{2}{3}} \quad (3)$$

to guarantee correctness in the multiparty setting with our global bootstrapping key.

With the parameters that we used in our implementation (see Table 2), we can handle up to  $k \leq 2^{13.3}$  parties according to (3). However, the practical bound for  $k$  depends on  $r$  and

$\ell$ , as described by (2). For example, let us fix  $\ell = 3$ . If  $r < 4$ , it is better to use single-key TFHE directly, if  $5 \leq r \leq 7$  then the optimal approach is to use our technique for  $k \leq 2$  and the approach of [12] for  $k \geq 3$ . Similarly, for  $8 \leq r \leq 10$  the optimal is to use our technique when  $k \leq 3$  and the approach of [12] for  $k \geq 4$ . When  $r \approx 64$  in our implementation, we can expect the upper bound of  $k$  which guarantees that our approach is better up to 21 parties for  $N = 2^{11}$ ,  $\ell = 3$ . Therefore, our approach is better the higher the ratio  $r$  is. This is the case when we want to handle larger message spaces in TFHE. That is, in order to handle message spaces larger than bits, TFHE offers functional bootstrapping, which uses  $N$  up to  $2^{14}$  [22]. In this case, our approach provides faster bootstrapping than that of [12] since  $r$  increases significantly compared to the case  $N = 2^{11}$ .

## 2) SETUP PHASE

Once the global keys are generated, the server can reuse the keys multiple times. Therefore, in the MPHE case, the global key generation is considered to be part of the setup phase. The setup phase of Lee et al. consists of  $nk$  internal products with multiplicative depth  $k$ . Similarly, our homomorphic indicator consists of  $k$  CMUX instantiated with internal products per secret key element. Therefore, our approach also requires  $kn$  internal products, in total, with multiplicative depth  $k$ . As a result, both approaches have the same computation complexity and the same noise propagation during this phase.

## 3) MEMORY BLOWUP

The main downside of our approach is linear memory blowup as the number of parties grows, whereas the other existing approach has constant memory overhead. Therefore, one can enjoy our approach up to reasonably many parties in the computing environment where the memory blow-up is not a big issue. With a normal laptop, we show our implementation results for up to 16 parties in the next section.

## B. IMPLEMENTATION

We provide a prototype implementation of our multikey bootstrapping key generation algorithm. We have also implemented the blind rotation algorithm of [13] instantiated with our bootstrapping keys. The prototype was done in Rust using the Concrete library (concrete-core version 1.0.0-beta) [14] and can be found at this GitHub repository. We include the Cargo.lock file of our project and the generated files from our benchmarking to ensure reproducibility.

## C. RESULTS AND RECOMMENDED PARAMETER SETS

To test our new key generation algorithm, we have computed a NAND gate using the blind rotation algorithm of [13] and our bootstrapping keys. We want to remark that the noise after bootstrapping is independent of the noise of the LWE ciphertext being bootstrapped. As long as the noise after bootstrapping is smaller than  $q/16$  it is possible to compute

a new gate. Therefore, our approach can be used for a more complex circuit than just a single NAND gate.

We have performed a search over a set of 26 possible parameter sets for decomposition base  $g$  and level  $\ell$  of the RGSW ciphertexts composing the keys, and we have selected the best parameters in terms of bootstrapping time and noise growth. This selection was done after computing 500 NAND gates per parameter set. Our experiments were done using a machine with an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz with 8GB of RAM and the results can be found in Table 2.

We use parameter  $N = 2048$ ,  $\log Q = 64$ ,  $\theta = 1.85 \cdot 2^4$  for the local public key (generated during the parties' setup phase) to achieve 110 bit security. The keys are used to generate ciphertexts and global keys which have higher security due to  $k$  aggregation.

We can see the trade-off between running time and the noise contained in the ciphertexts and keys, depending on the choice of  $B = \log_2 g$  and  $\ell$ , where  $g^\ell \leq Q$  in Table 2. Since  $\ell$  influences on the size of the bootstrapping key, it also affects the running time. Moreover, as we can see from Section VII, both  $\ell$  and  $g$  are important factors for noise growth. However, the noise grows in  $\text{poly}(g)$ . Therefore, our small choice of  $B$  increases bootstrapping running time but decreases the noise contained in the bootstrapping key and the final ciphertext after bootstrapping, significantly, which guarantees a much lower decryption failure probability. In Table 2, we give two versions of noise in logarithmic form, in the second column from the right. The first is the noise in  $R_Q$  and the second in the parentheses is the noise after modulus switching from  $Q$  to  $q$ . Since our message  $m \in \{0, 1\}$  is encoded as  $q/8 \cdot m$ , the final noise should be smaller than  $q/16$  to guarantee the correctness. Blind rotation with the global bootstrapping key works over  $R_Q$ , therefore, the correctness holds if the noise contained in the output of blind rotation is less than  $Q/16$ . Our result shows that we still have enough noise room to handle more operations before decryption failure occurs. Moreover, our bootstrapping key noise grows linearly in  $k$  as our analysis expected (for fixed parameters  $Q, \theta, N$  and similar choice of  $\ell$  and  $B$ ).

In Table 3, we show how much memory the server requires to store the bootstrapping keys, which increases linearly in  $k$ . We also show the time to generate the keys. We want to remind the reader that this key generation happens only once and can be done by the server alone. Therefore, the parties can produce their individual keys, send them to the server and go offline until the server generates the bootstrapping keys. For the single-key case, the original TFHE key generation takes around 1.5 seconds, while ours takes 6 seconds. Since bootstrapping keys are forms of RGSW ciphertext and the number of keys depends on the size of the secret key, the keys are the main factor of the server's memory overhead of FHE in general.

As we mentioned above (Section VII), Lee et al. [12]'s bootstrapping key generation would be very similar since the number of the dominant operation (internal product) of their

**TABLE 2.** Parameter sets recommended achieving at least 110-bit security based on LWE estimator [23] for different number parties  $k$ . We indicate by  $\log q$  and  $\log Q$  the LWE and RLWE modulus, respectively. The noise in a fresh RLWE ciphertext is indicated as  $\sigma_{rlwe}$ . The noise in a fresh LWE ciphertext is indicated as  $\sigma_{lwe}$ .  $B$  corresponds to  $\log(g)$ . The last column details the noise contained in the bootstrapping keys after running the homomorphic indicator algorithm. The values in the last three columns correspond to the average of 500 NAND operations, each performed with a freshly encrypted LWE ciphertext.

$k$	$N$	$n$	$\log q$	$\log Q$	$\sigma_{rlwe}(= \theta)$	$\sigma_{lwe}$	$B$	$\ell$	Time (in seconds)	Bootstrapping noise	Bsk noise
2	2048	530	32	64	$1.85 \cdot 2^{4.2}$	$2^{17}$	12	3	<b>0.20</b>	56.2 (24.2)	35.91
							6	8	0.48	<b>45.6 (13.6)</b>	30.37
4	2048	495	32	64	$1.85 \cdot 2^{4.2}$	$2^{17}$	11	4	<b>0.33</b>	56.12 (24.12)	36.95
							7	7	0.59	<b>48.97 (16.97)</b>	32.98
8	2048	495	32	64	$1.85 \cdot 2^{4.2}$	$2^{17}$	8	4	<b>0.46</b>	57.51 (57.51)	40.29
							7	6	0.70	<b>50.65 (18.65)</b>	33.85
16	2048	495	32	64	$1.85 \cdot 2^{4.2}$	$2^{17}$	10	5	<b>0.90</b>	58.37 (26.37)	38.02
							7	6	1.06	<b>52.79 (20.79)</b>	35.81

**TABLE 3.** Size in GB and time of generation (in seconds) of the bootstrapping keys produced by our approach with respect to the best parameters for time/noise error. We also provide the time of generation (in seconds) of the rests of the keys generated during setup. That is, the generation of all the users public and secret keys, the global public and secret key and the global key switching key. The values in the last two columns correspond to the average of 500 NAND operations, each performed with a freshly encrypted LWE ciphertext.

$k$	$N$	$n$	$\log q$	$\log Q$	$B$	$\ell$	Bsk size	Bsk generation time	Other keys generation time
2	2048	530	32	64	12	3	0.42	220	3.80
					6	8	0.45	620	3.70
4	2048	495	32	64	11	4	0.63	540	7.29
					7	7	0.66	970	7.26
8	2048	495	32	64	8	4	1.3	1330	15.20
					7	6	1.1	1600	14.94
16	2048	495	32	64	10	5	2.3	2900	31.29
					7	6	2.2	3570	37.76

algorithm is the same as our case. Since they don't provide experimental results, our results can be used as a reference for their case.

**IX. DISCUSSION AND CONCLUSION**

**A. APPLICATIONS OF TFHE-BASED MULTIPARTY FHE**

Determining whether a MKHE or a MPHE approach is better for a given application will come down to understanding two things. First, the character of the parties involved. If the application needs to provide parties with the capability of joining and leaving during the life cycle of the application, then multikey is the only choice. On the other hand, if the application has a static number of users from start to finish, then both approaches can be considered. The deciding factor in this case will be the bandwidth of the network and the storage capabilities of the parties. That is, the ciphertexts produced by a multikey scheme grow linearly in size with respect to the number of parties. The ciphertexts considered in a multiparty scheme have constant size. In fact, they have the same size as the single-party equivalent of the scheme. Another detail that needs to be considered in this discussion is the setup time. A multiparty-based application requires an expensive setup phase since the generation of the global bootstrapping key is computationally expensive. If the application only needs to run once, the multiparty approach might not be the best approach and the user should favor some other alternative. On the other hand, if the application is executed a good amount of times, the amortized cost compensates for the initial requirement of the setup phase.

In what follows, we give examples of applications that could benefit from the multiparty variant of TFHE that we present in this paper. We will try to provide the best explanation as of why we think that these applications would indeed need to use a multiparty variant of TFHE and not some other FHE-based solutions such as multikey or the use of some other scheme instead of TFHE.

- **k-NN learning.** k-NN is a well-known Machine Learning algorithm that given a distance  $\delta$ , a collection of vectors  $D$  (model vectors) and a source vector  $v$  returns the  $k$  vectors in  $D$  closest to  $v$  with respect to distance  $\delta$ . Using our new technique for generating bootstrapping keys for TFHE, it is possible to run k-NN homomorphically in the following scenario, as discussed in Fig. 1. The data owner sends its data to a server (the computing party), by encrypting  $D$  with the global public key. The client encrypts its source vector  $v$  using the same global public key and sends it to the server, which runs the k-NN algorithm homomorphically using the bootstrapping procedure of [10], together with our bootstrapping keys to evaluate the sign function as described in this work. The server sends back to the client the set of model vectors  $M$  closest to  $v$  together with the partial decryption interacting with the data owner. This allows the client to decrypt  $M$ . If another party wants to do the same, it simply interacts with the server to generate a new collection of global public key, bootstrapping keys and secret keys.
- **Privacy-Preserving Analysis on Medical Data.** A recent work [24] on privacy-preserving analysis of

medical data was done via multiparty homomorphic encryption. They are using a collaborative (interactive) version of bootstrapping as the original MPHE [4] proposed. It can use our protocol built upon [5] (non-interactive design) when the data analysis requires a non-linear function, or gate operation, which is more attractive in a cloud service scenario.

## B. DISCUSSION

Our homomorphic indicator can be of independent interest to homomorphically indicate where the desired position is in an array/vector. There is a similar technique which achieves the same functionality as ours, introduced in [11]. Their algorithm is called *homomorphic traversal* and outputs a unit vector where the desired component is set to 1, and 0 elsewhere. Their algorithm gives less noise in the output than the homomorphic indicator procedure since the multiplication depth is  $\log k$  instead of  $k$ , where  $k$  is the dimension of the output vector.

However, they need a bit representation form of  $i$  for inputs, which is an implementation bottleneck in our case. That is, we would need a bit representation of each  $S_j \leq k$  which is a master secret element to run their algorithm for all  $j \in \{0, \dots, n-1\}$ . Since  $S_j$ 's are not known to every user, the only way to do this is to homomorphically compute the binary addition of all the bit representations of  $s_{i,j}$ 's which were sent by all  $k$  users.

Moreover, handling a carry bit homomorphically is not well studied and not practical enough for now. This is the reason why we have designed a new way to output the same unit vector with only simple homomorphic operations. Therefore, each approach can be used in different applications, depending on the required input form.

## C. CONCLUSION AND FUTURE WORK

We propose a novel approach to construct bootstrapping keys for the TFHE scheme in the multiparty setting given a predefined set of parties. We compare two different TFHE bootstrapping designs which can handle multi-digit secret keys in the single-user setting and that can be extended to deal with multiple users. From our comparison, we have determined which of the two approaches provides a faster bootstrapping algorithm, and we have built an efficient global bootstrapping key compatible with it.

To this end, we introduce a novel algorithm called *homomorphic indicator* to obliviously compute an (encrypted) unit vector where the encryption of one is placed according to the input parameters. We believe that this construction can be of independent interest.

We have implemented our design as a proof-of-concept. Given a suitable set of parameters, we have been able to compute TFHE gate bootstrapping in less than a second for up to 16 parties.

As detailed in the paper, our method induces a memory blow-up in the bootstrapping keys when the number of parties grows. As future work, we will address this issue and provide

optimal parameters for as many parties as possible. We also leave as future work the instantiation and performance evaluation of a real-use case using our technique.

## REFERENCES

- [1] H.-S. Lee and J. Park, "On the security of multikey homomorphic encryption," in *Proc. 17th IMA Int. Conf. Cryptogr. Coding*, in Lecture Notes in Computer Science, vol. 11929, M. Albrecht, Ed. Heidelberg, Germany: Springer, Dec. 2019, pp. 236–251.
- [2] E. Kim, H.-S. Lee, and J. Park, "Towards round-optimal secure multiparty computations: Multikey FHE without a CRS," *Int. J. Found. Comput. Sci.*, vol. 31, no. 2, pp. 157–174, Feb. 2020, doi: 10.1142/S012905412050001X.
- [3] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proc. 44th Annu. ACM Symp. Theory Comput.*, May 2012, pp. 1219–1234.
- [4] C. Mouchet, J. Troncoso-Pastoriza, J.-P. Bossuat, and J.-P. Hubaux, "Multiparty homomorphic encryption from ring-learning-with-errors," *Proc. Privacy Enhancing Technol.*, vol. 2021, no. 4, pp. 291–311, Oct. 2021.
- [5] J. Park, "Homomorphic encryption for multiple users with less communications," *IEEE Access*, vol. 9, pp. 135915–135926, 2021.
- [6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. 3rd Innov. Theor. Comput. Sci. Conf.*, S. Goldwasser, Ed. New York, NY, USA: Association for Computing Machinery, Jan. 2012, pp. 309–325.
- [7] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptol. ePrint Arch.*, Tech. Rep. 2012/144, 2012. [Online]. Available: <https://eprint.iacr.org/2012/144>
- [8] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology—ASIACRYPT 2017* (Lecture Notes in Computer Science), vol. 10624, T. Takagi and T. Peyrin, Eds. Heidelberg, Germany: Springer, Dec. 2017, pp. 409–437.
- [9] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *Cyber Security Cryptography and Machine Learning*, S. Dolev, O. Margalit, B. Pinkas, and A. Schwarzmann, Eds. Cham, Switzerland: Springer, 2021, pp. 1–19.
- [10] M. Zuber and R. Sirdey, "Efficient homomorphic evaluation of k-NN classifiers," *Proc. Privacy Enhancing Technol.*, vol. 2021, no. 2, pp. 111–129, Apr. 2021.
- [11] K. Cong, D. Das, J. Park, and H. V. L. Pereira, "SortingHat: Efficient private decision tree evaluation via homomorphic encryption and transciphering," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 563–577, doi: 10.1145/3548606.3560702.
- [12] Y. Lee, D. Micciancio, A. Kim, R. Choi, M. Deryabin, J. Eom, and D. Yoo, "Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption," in *Advances in Cryptology—EUROCRYPT 2023*, C. Hazay and M. Stam, Eds. Cham, Switzerland: Springer, 2023, pp. 227–256.
- [13] M. Joye and P. Paillier, "Blind rotation in fully homomorphic encryption with extended keys," in *Cyber Security, Cryptology, and Machine Learning*, S. Dolev, J. Katz, and A. Meisels, Eds. Cham, Switzerland: Springer, 2022, pp. 1–18.
- [14] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending TfhE," in *Proc. 8th Workshop Encrypted Comput. Appl. Homomorphic Cryptogr. (WAHC)*, vol. 15, 2020, pp. 1–7.
- [15] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. New York, NY, USA: ACM Press, Nov. 2019, pp. 395–412.
- [16] P. Mukherjee and D. Wichs, "Two round multiparty computation via multi-key FHE," in *Advances in Cryptology—EUROCRYPT 2016* (Lecture Notes in Computer Science), vol. 9666, M. Fischlin and J.-S. Coron, Eds. Heidelberg, Germany: Springer, May 2016, pp. 735–763.
- [17] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. P. Fitzek, and N. Aaraj, "Survey on fully homomorphic encryption, theory, and applications," *Proc. IEEE*, vol. 110, no. 10, pp. 1572–1609, Oct. 2022.

- [18] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “TFHE: Fast fully homomorphic encryption over the torus,” *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, Jan. 2020.
- [19] G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs, “Multiparty computation with low communication, computation and interaction via threshold FHE,” in *Advances in Cryptology—EUROCRYPT 2012* (Lecture Notes in Computer Science), vol. 7237, D. Pointcheval and T. Johansson, Eds. Heidelberg, Germany: Springer, Apr. 2012, pp. 483–501.
- [20] M. Joye, “SoK: Fully homomorphic encryption over the [discretized] torus,” *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2022, no. 4, pp. 661–692, Aug. 2022. [Online]. Available: <https://tches.iacr.org/index.php/TCHES/article/view/9836>
- [21] S. Chatel, C. Mouchet, A. U. Sahin, A. Pyrgelis, C. Troncoso, and J.-P. Hubaux, “PELTA—Shielding multiparty-FHE against malicious adversaries,” *Cryptol. ePrint Arch., Tech. Rep. 2023/642*, 2023. [Online]. Available: <https://eprint.iacr.org/2023/642>
- [22] L. Bergerat et al., “Parameter optimization & larger precision for (T)FHE,” *J. Cryptol.*, vol. 36, no. 28, 2023, doi: [10.1007/s00145-023-09463-5](https://doi.org/10.1007/s00145-023-09463-5).
- [23] M. R. Albrecht, R. Player, and S. Scott, “On the concrete hardness of learning with errors,” *J. Math. Cryptol.*, vol. 9, no. 3, pp. 169–203, Oct. 2015, doi: [10.1515/jmc-2015-0016](https://doi.org/10.1515/jmc-2015-0016).
- [24] R. Geva et al., “Collaborative privacy-preserving analysis of oncological data using multiparty homomorphic encryption,” *Proc. Nat. Acad. Sci. USA*, vol. 120, no. 33, 2023, Art. no. e2304415120. [Online]. Available: <https://www.pnas.org/doi/abs/10.1073/pnas.2304415120>



**JEONGEUN PARK** received the B.S. degree in mathematics and the Ph.D. degree in mathematics with a specialization in cryptology from Ewha Womans University, Seoul, in 2015 and 2021, respectively. During the Ph.D. study, she studied the use cases of homomorphic encryption for a single user and multiple users. She is currently a Postdoctoral Researcher of applications of homomorphic encryption with Computer Security and Industrial Cryptography (COSIC), KU Leuven, Belgium.



**SERGI ROVIRA** received the B.Sc. degree in computer science and in mathematics from Universitat de Barcelona, in 2017, and the M.Sc. degree in advanced mathematics from Universitat de Barcelona. He is currently pursuing the Ph.D. degree in cryptography with Universitat Pompeu Fabra (UPF). His research is focused on fully homomorphic encryption and lattice based post-quantum cryptography.

• • •