Efficient TFHE Bootstrapping in the Multiparty Setting

Jeongeun Park[1] and Sergi Rovira[2]

[1] imec-COSIC, KU Leuven
[2] WiSeCom, Universitat Pompeu Fabra

Abstract. In this paper, we introduce a new approach to efficiently compute 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 in what parameter sets, based on our noise analysis. 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.

Keywords: TFHE, Homomorphic Encryption, Multiparty TFHE

1 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 [16,15].

If numerous users want to compute a common function keeping their input privacy, multi-key homomorphic encryption (MKHE) [18] and multiparty homomorphic encryption (MPHE) (a.k.a. threshold-multikey FHE) [19] are the best solutions for non-interactive protocols[1]. 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, only partial information of it, so decrypting other's ciphertexts becomes impossible. Decryption is possible only when all the participants (users) agree to do so.

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 distinct advantages and there are different applications where each can have better performance than the other, as analyzed in [21]. 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.

[1] Each user can protect its input from other parties in both designs since the secret key is never shared.

The main technical bottleneck of designing MPHE is to create the common evaluation key efficiently. Mouchet et al. [19], introduce a MPHE construction requiring two rounds between participants. In [21], the author improved upon [19] 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 evaluation 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 [5], FV [12], and CKKS [8]).

MPHE designs on TFHE has not studied well even though TFHE is widely used in privacy preserving protocols due to its advantages[2]. Recently, Lee et al. [17] introduced a (theoretical version of) MPHE protocol based on TFHE. They first design a single key TFHE with larger coefficients for the secret key and then, they naturally extend it to the MPHE version, introducing a simple (global) evaluation key generation algorithm which requires all parties' local evaluation keys. This makes sense because the common secret key of MPHE corresponds to the sum $s := s_1 + \cdots + s_k$, 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 evaluation 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 single key setting. In other words, we must take the noise contained in the global keys into account, when it comes to choosing parameters.

A concurrent paper [14] gives a protocol for TFHE with larger key size with a different technique. We observe that their technique requires only k multiplication in FFT domain among bootstrapping keys in the main loop of the boostrapping algorithm. Whereas Lee et al. [17] 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.

In this work, we extend [14]'s scheme to a version of MPHE, introducing an efficient evaluation key generation algorithm. To do this, we introduce an algorithm called *homomorphic indicator*, which homomorphically computes a unit vector of dimension m. 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 evaluation key generation to indicate which key is selected during bootstrapping. The server can therefore run the bootstrapping algorithm of [14] with the indicated bootstrapping key, without revealing the corresponding secret value because the output of the algorithm is the same as their bootstrapping key. We believe that our technique for constructing the evaluation key can be of independent interest.

Furthermore, we compare the complexity of each bootstrapping technique and we show which scheme performs better in what parameter sets, based on our noise analysis. Our approach suffers from high memory overhead as the number of parties grows, whereas the other existing approach has constant memory overhead. We provide a prototype implementation of our key generation algorithm

[2] TFHE outperforms the rest (BGV,FV,CKKS) when the computation is over bits and for non-linear functions such as ReLU, max/min functions.

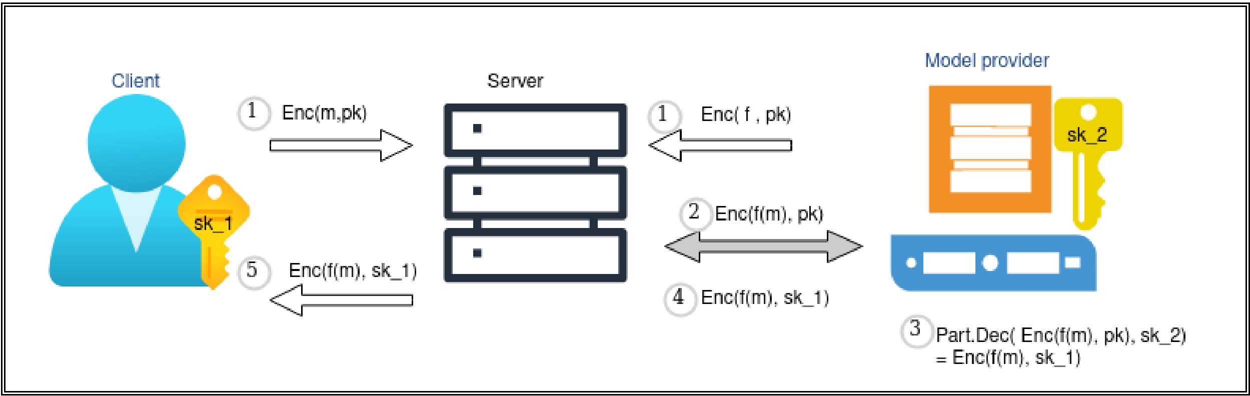

Fig. 1: Illustration of an application scenario of MPHE (two-party case). (1) Once a client and a service provider generates a common key (pk) during setup phase, both can send their item (m : client's data, f : machine learning model) encrypted under the key to the server. (2) After the computation is over, server sends the output ($\mathsf{Enc}(f(m), \mathsf{pk})$) to the model provider so that (3) he can run partial decryption with his own secret key $\mathsf{sk_2}$ and (4) sends the output which is an encryption of $\mathsf{Enc}(f(m), \mathsf{sk_1})$) 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 $\mathsf{sk_1}$.

using the Concrete library [10] 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.

1.1 A brief note on applications

The direct application of MPHE is privacy preserving machine learning [7,23,11] 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 Figure 1). At the end of the server's computation and partial decryption [20] 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, so that the client can go offline after query phase, until the computation is over. More applications are discussed in Section 8.1 in detail.

2 Preliminaries

2.1 Notations.

We denote the security parameter of the FHE scheme by λ. 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 $\mathbf{x}_i$ 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 A is denoted by $\mathsf{A}[i]$. 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 θ, θ' 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 χ according to a suitable standard deviation. Let β, ϵ, $\ell = O(\log q)$ be the parameters of the TFHE scheme [9]. We denote the number of users

by k. The LWE (resp. RLWE) secret key of each user is denoted by $\mathbf{s}_i = (s_{i,0}, \ldots, s_{i,n-1})$, where $s_{i,j} \in \{0,1\}$ and $n \geq 1$ (resp. $s_i \in R$). The (master) secret key corresponding to the common public key for an LWE ciphertext is denoted by $S = (S_0, S_1, \ldots, S_{N-1})$, where $S_j = \sum_{i=1}^{k} s_{i,j} \leq k$ for all $j \in \{0, \ldots, n-1\}$. Similarly, $s := s_1 + s_2 + \cdots + s_k$ is the master secret key for an RLWE/RGSW ciphertext. Given a base g and $\ell = O(\log q)$, we define a gadget vector $\mathbf{g} = (1, g, \ldots, g^{\ell-1})^t$. An RGSW ciphertext is of the form $\mathsf{RGSW}_s(m) := (\mathbf{a}, \mathbf{b}) \in \mathcal{R}_q^{2\ell \times 2}$. 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 $\mathbf{G}^{-1}(a) \cdot \mathbf{G} = a \in \mathcal{R}_q^2$. We define $\mathbf{G}^{-1}(\mathbf{c}) := (\mathbf{G}^{-1}(c_1), \ldots, \mathbf{G}^{-1}(c_\delta))$, for a vector $\mathbf{c} = (c_1, \ldots, c_\delta) \in \mathcal{R}_q^{\delta \times 2}$. In this paper, we use the terms *parties* and *users* interchangeably to refer to the entities who join the homomorphic computation.

2.2 TFHE Ciphertexts.

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

- An LWE ciphertext is defined as $\mathsf{LWE}_{\mathbf{s}}(m) := (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$, where $\mathbf{a}$ is a random vector, $b = \langle \mathbf{a}, \mathbf{s} \rangle + \Delta \cdot m + e$ 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.
- A RLWE ciphertext is defined as $\mathsf{RLWE}_s(m) := (a, b) \in \mathcal{R}_q^2$, where $b = a \cdot s + \Delta \cdot m + e'$ 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.
- Fix a base B_g and $\ell = O(\log q)$, and the gadget vector $\mathbf{g} = (1, g, \ldots, g^{\ell-1})^t$. An RGSW ciphertext is of the form $\mathsf{RGSW}_s(m) := (\mathbf{a}, \mathbf{b}) \in \mathcal{R}_q^{2\ell \times 2}$, where $\mathbf{b} = \mathbf{H} + m \cdot \mathbf{G}$, each row of $\mathbf{H}$ is a $\mathsf{RLWE}_s(0)$ and $\mathbf{G}$ is a gadget matrix defined as $\mathbf{G} := \mathbf{I}_2 \otimes \mathbf{g}$.
- $\mathsf{TrivialNoiseless}(\mathsf{tag}, \mu)$: it takes as input a tag $\mathsf{tag} \in \{\mathsf{LWE}, \mathsf{RLWE}, \mathsf{RGSW}\}$ which indicates the ciphertext form, and a message μ, and outputs a ciphertext with the form according to tag with a noiseless mask.

2.3 External and internal products.

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

$$\boxdot : \mathsf{RGSW} \times \mathsf{RLWE} \to \mathsf{RLWE}, \ (\mathbf{C}_1, \mathbf{c}_2) \mapsto \mathbf{C}_1 \boxdot \mathbf{c}_2 = \mathbf{G}^{-1}(\mathbf{c}_2) \cdot \mathbf{C}_1.$$

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 2ℓ RLWE rows $(\mathbf{c}_1, \ldots, \mathbf{c}_{2\ell})^t$ of $\mathbf{C}_2$. It is defined as follows:

$$\boxtimes : \mathsf{RGSW} \times \mathsf{RGSW} \to \mathsf{RGSW}, \ (\mathbf{C}_1, \mathbf{C}_2) \mapsto (\mathbf{C}_1 \boxdot \mathbf{c}_1, \ldots, \mathbf{C}_1 \boxdot \mathbf{c}_{2\ell}).$$

2.4 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:

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

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:

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

2.5 Blind rotation.

The most expensive procedure of TFHE bootstrapping is called *blind rotation*. This algorithm converts a ciphertext $\mathsf{LWE}_{\mathbf{s}}(m) = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ encrypted using the secret key $\mathbf{s}$ into another ciphertext $\mathsf{RLWE}_{s'}(X^{-\overline{m}} \cdot v) \in \mathcal{R}_q^2$ via computing $\mathsf{CMUX}_{\boxdot}$ n times, where $\overline{m}$ is a rounded approximation of $b - \langle \mathbf{a}, \mathbf{s} \rangle$ and v is a polynomial, and n is the dimension of the input LWE ciphertext.

2.6 Multiparty FHE

Let $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$ be a set of k parties. Informally, a *multiparty homomorphic encryption scheme* (MPHE) over $\mathcal{P}$ is an homomorphic encryption scheme where the secret key is a function of the k individual secret keys of each party $\mathbf{s}_1, \ldots, \mathbf{s}_k$. In this paper, we are interested in the case where the secret key $\mathbf{s}$ of the MPHE is computed as $\mathbf{s} = \mathbf{s}_1 + \cdots + \mathbf{s}_k$. 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 [2,21,19]. For example, following the same notation as in subsection 2.2, the ciphertext $\mathsf{LWE}_{\mathbf{s}_1}(0) + \cdots + \mathsf{LWE}_{\mathbf{s}_k}(0)$ can be decrypted using the secret key $\mathbf{s}$ provided that the same $\mathbf{a}$ was used for all parties in the masks of these k LWE ciphertexts[3] . We follow the most communication efficient MPHE protocol [21], 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) evaluation 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 which 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 step 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).

[3] However, due to security issues, this technique is not directly used for generating ciphertexts, only for generating keys.

3 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.

3.1 Bootstrapping of Lee et al. [17]

The work of Lee et al. [17] introduces a new efficient blind rotation algorithm that works with arbitrary secret key distributions and requires small evaluation 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 $\mathrm{acc} = \mathsf{RLWE}_s(f'(X))$ and a collection of blind rotation keys $\mathrm{bsk}_j := \mathsf{RGSW}_s(X^{s_j})$ for $j \in J$, for some indexing set J, their blind rotation algorithm proceeds by multiplying acc by all bsk_j, obtaining $\mathrm{acc} = \mathsf{RLWE}_s(f'(X)X^{\sum_{j\in J} s_j})$. Then, applying an automorphism they obtain $\mathrm{acc} = \mathsf{RLWE}_s(f'(X^g)X^{g\cdot\sum_{j\in J} s_j})$, for a suitable integer g depending on the polynomial degree N. After repeating this process with other indexing sets they finally obtain $\mathrm{acc} = \mathsf{RLWE}_s(f(X) \cdot X^{\beta+\langle\alpha,s\rangle})$.

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

To use this blind rotation algorithm in the threshold 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 $\mathbf{s}_1, \ldots, \mathbf{s}_k$ be the secret keys of a set of k parties. Each party j computes $\mathrm{bsk}_{j,i} = \mathsf{RGSW}_s(X^{s_{j,i}})$ where $s_{j,i}$ is the i-th component of $\mathbf{s}_j$ and sends this to the computing party. Then, the computing party obtains $\mathrm{bsk}_i = \boxtimes_{\prod_{j\in J}} \mathrm{bsk}_{j,i} = \mathsf{RGSW}_s(X^{s_{*,i}})$.

3.2 Bootstrapping of Joye and Paillier's [14]

As mentioned in the introduction, each loop of the blind rotation in [14] 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 $\{\mathsf{key}_{j,i}\}_{i\in[k]}$, per secret key component S_j ($j \in \{0, \ldots, n\}$) such that all of them are encryptions of 0 or only one encrypts 1 (the rest encrypt 0). Then, the computing party runs a selection algorithm to choose the corresponding $X^{a_j \cdot S_j}$, as follows:

$$1 + (X^{a_j} - 1) \cdot \mathsf{key}_{j,1} + (X^{2\cdot a_j} - 1) \cdot \mathsf{key}_{j,2} + \cdots + (X^{k\cdot a_j} - 1) \cdot \mathsf{key}_{j,k} \, .$$

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

4 The common bootstrapping key generation

After each user sends its locally generated bootstrapping key encrypting the coefficients of its secret key, the computing party (a server) creates a global bootstrapping key to run single key TFHE bootstrapping with multi digit secret elements. 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 from the locally generated keys sent by all users. To do this, we introduce our novel algorithm called *homomorphic indicator* below.

4.1 Homomorphic Indicator

Given an array 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 A, that is, the output array will have an encryption of 1 in the $W_H(A)$-th position and encryptions of 0 everywhere else.

Let us start by explaining its plaintext version. The algorithm takes as input an array A of k slots[4] filled with 0's and another array 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 C as follows. If the first controller bit is 1, the first slot of A becomes 1, and the rest of A is untouched. If the second controller bit is also 1, 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 A will have a 1 at the position corresponding to the number of 1's in C.

In the ciphertext case, we start with two arrays of $k+1$ slots, denoted by A^{old} and A^{new}, where the 0-th slot of 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 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, A^{new} is updated based on the corresponding controller bit and the values in the slots of A^{old}, as follows. For each $j \in [k]$, $\mathsf{A}^{new}[j] \leftarrow \mathsf{A}^{old}[j-1]$ if $\mathbf{C}_i$ is an encryption of 1, otherwise, $\mathsf{A}^{new}[j] \leftarrow \mathsf{A}^{old}[j]$. This can be instantiated as

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

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 A^{new}. The 0-th slot should also be updated to 0 or remain 1. That is, $\mathsf{A}^{new}[0]$ is updated to $\mathsf{A}^{old}[0]$ if $\mathbf{C}_i$ is an encryption of 0, otherwise, $\mathsf{A}^{new}[0]$ becomes an encryption of 0. This is instantiated as

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

After all the slots of A^{new} are updated, A^{old} is set as 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 A^{new} as the desired result.

Algorithm 1 Homomorphic Indicator ($\mathsf{Hom.Indicator}$)

Input: $\{\mathbf{C}_i\}_{i\in[m]}, \mathsf{A}^{new}$ and A^{old}.
Output: A^{old}.
  for $i \leftarrow 1$ to k **do**
    for $j \leftarrow 1$ to k **do**
      $\mathsf{A}^{new}[j] := \mathsf{CMUX}_{\boxtimes}(\mathbf{C}_i, \mathsf{A}^{old}[j], \mathsf{A}^{old}[j-1])$
    end for
    $\mathsf{A}^{new}[0] := \mathsf{A}^{old}[0] \boxtimes (1 - \mathbf{C}_i)$
    for $j \leftarrow 0$ to k **do**
      $\mathsf{A}^{old}[j] := \mathsf{A}^{new}[j]$
    end for
  end for

[4] A slot of an array refers to an element of an array in this paper, we use these terms interchangeably.

Note that we use internal product to instantiate CMUX gate and multiplication between two ciphertexts for a direct application to our bootstrapping key generation. However, one can use internal product or any type of homomorphic multiplication. Since the multiplicative depth of this algorithm is linear in the size of the output unit vector, internal/external product is highly recommended if the size is large to manage noise issue.

4.2 Instantiation of bootstrapping key generation

Local bootstrapping key generation Generating local bootstrapping key is similar to [21], however, the instantiation is a bit different in order to minimize the noise generated during this process. 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 global public key via one interaction. We specifically follow the way of [13] 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 $b_i = a \cdot s_i + e_i$, 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 $s := (s_1 + s_2 + \ldots + s_k)$.

Every time all k users generate ciphertexts/ local evaluation 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 of this procedure below. In practice, we set $m = \lceil 3 \cdot \log q \rceil$ as [5] analyzed.

Algorithm 2 Local encryption ($\mathsf{Local.Enc}$)

Input: A vector of dimension m consisting of ciphertexts (all of which can be one of LWE/RLWE/RGSW forms), denoted by V, and a message μ, a tag $\mathsf{tag} \in \{\mathsf{LWE}, \mathsf{RLWE}, \mathsf{RGSW}\}$.
Output: an LWE/RLWE/RGSW ciphertext c encrypting a message μ.
Sample a random vector $\mathsf{R} \leftarrow \{0,1\}^m$
Computes $\bar{\mu} := \mathsf{TrivialNoiseless}(\mathsf{tag}, \mu)$.
Computes $c := \langle \mathsf{V}, \mathsf{R} \rangle + \bar{\mu}$

Global evaluation key generation The main goal of this algorithm is to create the global bootstrapping key of [14] *on the fly* using all clients' keys which have already been given at the beginning of the protocol. As we explained in Section 2, the global bootstrapping key is an array of k RGSW ciphertexts per coefficient. We denote this key as $\widehat{\mathsf{bsk}} := [\widehat{\mathsf{bsk}}_0, \ldots, \widehat{\mathsf{bsk}}_{n-1}]$, where each component consists of k RGSW ciphertexts, that is $\widehat{\mathsf{bsk}}_i = [\mathsf{RGSW}_s(b_{i,1}), \ldots, \mathsf{RGSW}_s(b_{i,k})]$, where $b_{i,j} \in \{0,1\}$ for and $j \in [k]$ and $i \in [0, \ldots n-1]$.

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

Each user. Each client i sends its (local) bootstrapping key $\mathsf{bsk}_i := (\mathsf{RGSW}_s(s_{i,0}), \mathsf{RGSW}_s(s_{i,1}), \ldots, \mathsf{RGSW}_s(s_{i,n-1}))$ to a server.

Server.

- Initialization: The Server creates two arrays of $k+1$ slots per coefficient, denoted by A^{old} and A^{new}, respectively, where the first slot of A^{old} is 1 and the rest are zeros.
- As soon as it receives the (local) bootstrapping keys $\mathsf{bsk}_1, \ldots, \mathsf{bsk}_k$ from the predefined k users, it executes the algorithm 3, running $\mathsf{Hom.Indicator}$ as a subroutine.

Algorithm 3 Global bootstrapping key generation

Input: $\{\mathsf{bsk}_i\}_{i\in[k]}, \mathsf{A}^{new}$ and A^{old}.
Output: $\widehat{\mathsf{bsk}}$.

for $t \leftarrow 0$ to $n-1$ **do**
  for $i \leftarrow 1$ to k **do**
    Parse $\mathbf{C}_{i,t} := \mathsf{bsk}_i[t]$
  end for
  $\mathsf{A} := \mathsf{Hom.Indicator}(\{\mathbf{C}_{i,t}\}_{i\in[k]}, \mathsf{A}^{new}, \mathsf{A}^{old})$
  $\widehat{\mathsf{bsk}}[t] := [\mathsf{A}[1], \ldots, \mathsf{A}[k]]$
  Refresh A^{new} and A^{old}
end for

4.3 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 $\mathsf{A}^{old}, \mathsf{A}^{new}$. The array A^{old} has a RGSW encryption of 1 at the first position and a RGSW encryption of 0 in the rest while A^{new} consists of RGSW encryptions of 0.

In the first iteration, since $s_{1,0} = 1$, we will set $\mathsf{A}^{new}[1] = \mathrm{RGSW}(1)$ and $\mathsf{A}^{new}[j] = \mathrm{RGSW}(0)$ for all $j \neq 1$. We have then 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 exactly correspond to the bootstrapping keys needed for 4 parties in the blind rotation protocol of [14]. This example is depicted in Figure 2.

5 TFHE bootstrapping in the multiparty setting

Once the predefined k users (parties) generate their own local bootstrapping key as discussed in Section 4.2 and upload them to server, the server runs Algorithm 3 to create the common/global

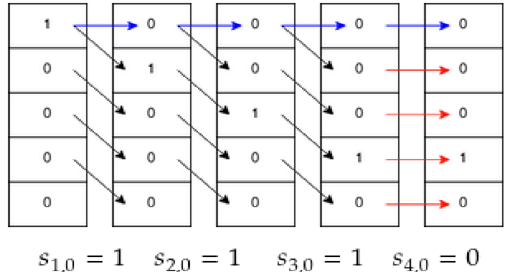

Fig. 2: Example for k = 4 of our bootstarpping key generation. The blue arrow shows the direction of moving the first slot. The black arrows show that the elements in the new array comes from the previous slots in the old array when the corresponding secret key component is 1. The red arrow shows that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

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 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 a solution, the original TFHE [9] switches the ciphertext format from a 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.

5.1 Key Switching in the multiparty setting

The key switching algorithm for our case is the same as in the single key TFHE setting. Since all the secret keys correspond to the sum of all the parties' secrets, the key switching homomorphically from $\mathsf{LWE}_z(m)$ to $\mathsf{LWE}_{z'}(m)$ where both z and z' are the sum of all parties' secrets with different size. Let us define $z = z_1 + z_2 + \cdots + z_k$ and $z' = z'_1 + z'_2 + \cdots + z'_k$, where each z_i and z'_i are the secrets of the i-th party. Then each user produces $\mathsf{ksk}_i := \mathsf{LWE}_{z'}(z_i)$ as their key switching keys by running $\mathsf{Local.Enc}$ function. And the server adds up all ksk_i to generate the global key switching key.

5.2 Overall description of our bootstrapping

Each i-th user:

- **Setup for public key:** 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 $V_{\mathsf{LWE}}, V_{\mathsf{RLWE}}, V_{\mathsf{RGSW}}$.
- **Generating local evaluation keys:** generates local bootstrapping key bsk_i by running $\mathsf{Local.Enc}(V_{\mathsf{RGSW}}, \mathsf{RGSW}, s_{i,j})$ for all $j \in [n]$. Similarly, generates local key switching key ksk_i by running $\mathsf{Local.Enc}(V_{\mathsf{LWE}}, \mathsf{LWE}, s_i)$.

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

Server:

- **Setup for server:** runs Algorithm 3 to generate a global bootstrapping key $\widehat{\mathsf{bsk}}$. Generates a global key switching key denoted by $\widehat{\mathsf{ksk}}$ by summing up all given ksk_i given by k users.
- **After a gate operation:** run Joye and Paillier's bootstrapping with $\widehat{\mathsf{bsk}}$ and $\widehat{\mathsf{ksk}}$.

6 Noise analysis

In this section, we analyze the noise growth of the setup phase and the final noise after bootstrapping. Let e be the initial noise in each user's (local) public key, that is, $\mathsf{Var}(e) = \theta^2$ as we defined in Section 2. 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 $\iota = \lceil 3 \cdot \log q \rceil$ to guarantee LWE security [5] (as we mentioned in Section 4.2). Therefore, $\mathsf{Var}(\mathsf{Err}(\mathsf{bsk}_i)) \leq k \cdot \iota \cdot \theta^2$.

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 $\mathbf{A}_i$ of one homomorphic indicator operation as

$$\mathsf{Var}(\mathsf{Err}(\mathbf{A}_i)) \leq k \cdot \ell \cdot N \cdot g^2 \cdot \mathsf{Var}(\mathsf{Err}(\mathsf{bsk}_i)) + k \cdot (1 + N) \cdot \epsilon^2,$$

based on the analysis of [9]. Each element of $\hat{\mathsf{bsk}}$ is a vector of RGSW ciphertexts (denoted by $[\mathbf{A}_0, \ldots, \mathbf{A}_{n-1}]$). Each component of such vectors is created via homomorphic indicator which consists of k consecutive CMUX gates (internal products). We define the variance of $\hat{\mathsf{bsk}}$ to be the maximum variance of $\mathbf{A}_i$'s. Hence, the noise contained in $\hat{\mathsf{bsk}}$ can be bounded as follows:

$$\begin{aligned}\mathsf{Var}(\mathsf{Err}(\hat{\mathsf{bsk}})) &= \max_i(\mathsf{Var}(\mathsf{Err}(\mathbf{A}_0)), \ldots, \mathsf{Var}(\mathsf{Err}(\mathbf{A}_{n-1}))) \\ &\leq k \cdot \ell \cdot N \cdot g^2 \cdot \mathsf{Var}(\mathsf{Err}(\mathsf{bsk}_i)) + k \cdot (1 + N) \cdot \epsilon^2.\end{aligned}$$

Now, we run bootstrapping as specified in [14] with the constructed $\hat{\mathsf{bsk}}$. In the blind rotation algorithm of [14], there are k additions among the corresponding $\hat{\mathsf{bsk}}$ components in every loop. After the additions, the output, say $\mathbf{C}_{add}$, contains the noise of which the variance is

$$\mathsf{Var}(\mathsf{Err}(\mathbf{C}_{add})) \leq 2 \cdot k \cdot \mathsf{Var}(\mathsf{Err}(\hat{\mathsf{bsk}})).$$

In terms of noise growth, the blind rotation of [14] is viewed as TFHE bootstrapping with $\mathbf{C}_{add}$ as the bootstrapping key. After blind rotation, the output RLWE ciphertext c has a noise that can be bounded as follows:

$$\mathsf{Var}(\mathsf{Err}(c)) \leq n \cdot \ell \cdot N \cdot g^2 \cdot \mathsf{Var}(\mathsf{Err}(\mathbf{C}_{add})) + n \cdot (1 + N)\epsilon^2.$$

Performing a key switching to convert c to an LWE ciphertext, denoted by $\mathbf{c}$, only adds a small noise $N \cdot \ell \cdot g^2 \cdot \mathsf{Var}(\mathsf{Err}(\mathsf{evk}_i))$ 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 [17] 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:

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

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,

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

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.

7 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 [17] since they did not implement their multi-key extension. Therefore, for fair comparison, we analyze the noise growth of bootstrapping key of both designs in terms of crucial parameters such as k, N, n, ℓ, and provide benchmarks with respect to the running time of dominant operation of both cases on the same machine.

7.1 Complexity Comparison

Bootstrapping running time

Scheme	blind rotation
[17]	$(1.5n + w) \cdot T_{mult}$
[14]	$n \cdot T_{mult} + k \cdot n \cdot (4 \cdot \ell \cdot T_{PM})$

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.

In Table 1 we provide the computational cost of the blind rotation algorithms of [17] and [14] 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 [14] outperforms that of [17]. For this, we can use Table 1 and upper bound k as follows:

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

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 6, we obtain the following relation among parameters

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

to guarantee the 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 (2). However, the practical bound for k depends on r and l, as described by (1). For example, let us fix $l = 3$. 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 [17] 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 [17] for $k \geq 4$. When $r \approx 64$ in our implementation, we can expect the upper bound of k which guarantee 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} [4]. In this case, r will increase with respect to the case $N = 2^{11}$, resulting in higher values of k for which our approach provides faster bootstrapping than that of [17].

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 as being part of the setup phase. The setup phase of Lee et al. consists of $n \cdot k$ 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 $k \cdot n$ 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.

Memory blowup However, the bootstrapping key size increases as k grows in our extension of [14], whereas the memory overhead of Lee et al. does not rely on k once the bootstrapping key is generated. 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 result up to 16 parties in the next section.

7.2 Implementation

We provide a prototype implementation of our multikey bootstrapping key generation algorithm. We have also implemented the blind rotation algorithm of [14] instantiated with our bootsrapping keys. The prototype was done in Rust using the Concrete library (concrete-core version 1.0.0-beta) [10].

For some operations such as encryption/decryption and generation of key-switching keys, the default code that Concrete provides is not sufficient since it is written to work with binary secret keys. This has required us to include new functions in concrete-core to account for non-binary secret keys. Therefore, to ensure reproducible results, we include in our GitHub repository a modified version of concrete-core together with the Cargo.lock file of our project. We also include the files generated from the benchmark.

7.3 Results and recommended parameter sets

To test our new key generation algorithm, we have computed a NAND gate using the blind rotation algorithm of [14] 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 l 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 $N = 2048, \log Q = 64, \theta = 1.85 * 2^4$ for local public key (generated during parties' setup phase) to achieve 110 bit security. The keys are used to generate ciphertexts and global keys which has higher security due to k aggregation.

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

Table 2: Parameter sets recommended achieving at least 110-bit security based on LWE estimator [1] 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 σ_{rlwe}. The noise in a fresh LWE ciphertext is indicated as σ_{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.

We can see that 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 ℓ, where $g^\ell \leq Q$ in Table 2. Since ℓ influences on the size of bootstrapping key, it also affects the running time. Moreover, as we can see from Section 6, both ℓ and g are important factors for the noise growth. However, the noise grows in poly(g). Therefore, our small choice of B increases bootstrapping running time but decreases the noise contained in bootstrapping key and the final ciphertext after bootstrapping, significantly,

which guarantees 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 one is the noise in R_Q and the second one 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, θ, N and similar choice of ℓ and B).

In Table 3, we show how much the server consumes memory to store bootstrapping keys, which increases linearly in k. We also show the time to generate the keys. We want to remain to 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 produces 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 are depending on the size of secret key, the keys are the main factor of server's memory overhead of FHE in general.

k	N	n	$\log q$	$\log Q$	B	l	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

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.

As we mentioned above (Section 6), Lee et al.[17]'s bootstrapping key generation would be so much similar since the number of the dominant operation (internal product) of their algorithm is same as our case. Since they don't provide experimental result, our result can be used for a reference for their case.

8 Discussion and Conclusion

8.1 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 need 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 very 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 will execute a good amount of times, the amortized cost well compensates 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 solution 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 δ, 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 δ. The inner workings of k-NN are beyond the scope of this work, we will just point out that the main function to homorphically evaluate k-NN is the sign function:

$$\begin{aligned}\mathrm{Sign}_l : &\mathbb{N} \to \{0,1\}\\ &x \mapsto 0 \text{ if } x \leq m-l\\ &x \mapsto 1 \text{ if } x > m-l.\end{aligned}$$

  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 Figure 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 [23], 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.
- **Deep Neural Networks.** A similar concept can be applied in the world of image processing by means of Neural Networks. That is, very recent works [22,3] shows how to do Deep Neural Network inference using TFHE. Only the single-key variant of TFHE is considered. The new bootstrapping method proposed in this work can be used to perform Deep Neural Network inference over encrypted data using TFHE. That is, a single client can use an already trained NN by sending its encrypted query to the server homomorphically encrypted. We believe that if the number of parties can be fixed in advanced (for example, in the case of centralized NN, see the survey in [6] for concrete constructions) then our multiparty equivalent of TFHE could be used to train a more accurate model using data from different parties while preserving their privacy. Depending on the size of the databases and the number of users, a multikey approach could quickly become impractical.

- **Inter-bank data sharing for fraud prevention.** Imagine that the top five banks of a given country want to improve their fraud detection and prevention capabilities. Their idea is to share their data between the banks in order to understand how fraud is happening and to study how it can be prevented. Of course, the banks can not share the data in the clear between them, since it is highly sensible customer information. For this, they can use an FHE approach which allows each bank to query the databases of the other banks in a privacy preserving manner in order to learn what they can improve in their end. In this scenario, the size of the databases of the banks make the use of MPHE more favourable than MKHE since the number of banks are already fixed. Moreover, the nature of the data and the consequences of errors in the encrypted computations make the use of a TFHE-based approach better than, say, a CKKS approach where some error is inherently present in the resulting plaintext. This scenario is not theoretical, the idea of using FHE for inter-bank data sharing has been already studied, explored and implemented in the UUEE.

8.2 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 work 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 $j \in \{0, \ldots, n-1\}$. Since S_j's are not known to every user, the only way to do this is to homomorphically compute binary addition of all the bit representations of $s_{i,j}$'s which were sent by all k users.

However, 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.

8.3 Conclusion and Future works

We propose a novel approach to construct blind rotation 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 blind rotation 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 it is detailed in the paper, our method induces a memory blow-up in the blind rotation 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.

Acknowledgement

This work was partially done when the second-listed author visited imec-COSIC, KU Leuven. The first-listed author has been supported by CyberSecurity Research Flanders with reference number VR20192203. The second-listed author has been supported by an FPI grant with reference number PRE2019-088830.

References

1. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology 9(3), 169–203 (2015), https://doi.org/10.1515/jmc-2015-0016
2. Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., Wichs, D.: Multiparty computation with low communication, computation and interaction via threshold FHE. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 483–501. Springer, Heidelberg (Apr 2012)
3. Benamira, A., Guérand, T., Peyrin, T., Saha, S.: Tt-tfhe: a torus fully homomorphic encryption-friendly neural network architecture (2023)
4. Bergerat, L., Boudi, A., Bourgerie, Q., Chillotti, I., Ligier, D., Orfila, J.B., Tap, S.: Parameter optimization & larger precision for (t)fhe. Cryptology ePrint Archive, Paper 2022/704 (2022), https://eprint.iacr.org/2022/704, https://eprint.iacr.org/2022/704
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012. pp. 309–325. ACM (Jan 2012)
6. Cabrero-Holgueras, J., Pastrana, S.: Sok: Privacy-preserving computation techniques for deep learning. Proceedings on Privacy Enhancing Technologies 2021, 139 – 162 (2021)
7. Chen, H., Dai, W., Kim, M., Song, Y.: Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 395–412. ACM Press (Nov 2019)
8. Cheon, J.H., Kim, A., Kim, M., Song, Y.S.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part I. LNCS, vol. 10624, pp. 409–437. Springer, Heidelberg (Dec 2017)
9. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast fully homomorphic encryption over the torus. Journal of Cryptology 33(1), 34–91 (Jan 2020)
10. Chillotti, I., Joye, M., Ligier, D., Orfila, J.B., Tap, S.: Concrete: Concrete operates on ciphertexts rapidly by extending tfhe. In: WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. vol. 15 (2020)
11. Cong, K., Das, D., Park, J., Pereira, H.V.: Sortinghat: Efficient private decision tree evaluation via homomorphic encryption and transciphering. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 563–577. CCS '22, Association for Computing Machinery, New York, NY, USA (2022), https://doi.org/10.1145/3548606.3560702
12. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), https://eprint.iacr.org/2012/144
13. Joye, M.: Sok: Fully homomorphic encryption over the [discretized] torus. IACR Transactions on Cryptographic Hardware and Embedded Systems 2022(4), 661–692 (Aug 2022), https://tches.iacr.org/index.php/TCHES/article/view/9836
14. Joye, M., Paillier, P.: Blind rotation in fully homomorphic encryption with extended keys. In: Dolev, S., Katz, J., Meisels, A. (eds.) Cyber Security, Cryptology, and Machine Learning. pp. 1–18. Springer International Publishing, Cham (2022)
15. Kim, E., Lee, H.S., Park, J.: Towards round-optimal secure multiparty computations: Multikey fhe without a crs. International Journal of Foundations of Computer Science 31(02), 157–174 (2020), https://doi.org/10.1142/S012905412050001X

16. Lee, H.S., Park, J.: On the security of multikey homomorphic encryption. In: Albrecht, M. (ed.) 17th IMA International Conference on Cryptography and Coding. LNCS, vol. 11929, pp. 236–251. Springer, Heidelberg (Dec 2019)
17. Lee, Y., Micciancio, D., Kim, A., Choi, R., Deryabin, M., Eom, J., Yoo, D.: Efficient fhew bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In: Hazay, C., Stam, M. (eds.) Advances in Cryptology – EUROCRYPT 2023. pp. 227–256. Springer Nature Switzerland, Cham (2023)
18. LópezAlt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Proceedings of the forty-fourth annual ACM symposium on Theory of computing. pp. 1219–1234. ACM (2012)
19. Mouchet, C., Troncoso-Pastoriza, J., Bossuat, J.P., Hubaux, J.P.: Multiparty homomorphic encryption from ring-learning-with-errors. Proceedings on Privacy Enhancing Technologies 2021(4), 291–311 (2021)
20. Mukherjee, P., Wichs, D.: Two round multiparty computation via multi-key FHE. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 735–763. Springer, Heidelberg (May 2016)
21. Park, J.: Homomorphic encryption for multiple users with less communications. IEEE Access 9, 135915–135926 (2021)
22. Stoian, A., Frery, J., Bredehoft, R., Montero, L., Kherfallah, C., Chevallier-Mames, B.: Deep neural networks for encrypted inference with tfhe. Cryptology ePrint Archive, Paper 2023/257 (2023), https://eprint.iacr.org/2023/257, https://eprint.iacr.org/2023/257
23. Zuber, M., Sirdey, R.: Efficient homomorphic evaluation of k-nn classifiers. Proceedings on Privacy Enhancing Technologies 2021, 111 – 129 (2021)