A Practical TFHE-Based Multi-Key Homomorphic Encryption with Linear Complexity and Low Noise Growth*

Yavuz Akın[1], Jakub Klemsa[1,2(✉)], and Melek Önen[1]

[1] EURECOM
Sophia-Antipolis, France
{yavuz.akin,jakub.klemsa,melek.onen}@eurecom.fr
[2] Czech Technical University in Prague
Prague, Czech Republic

Abstract. Fully Homomorphic Encryption enables arbitrary computations over encrypted data and it has a multitude of applications, e.g., secure cloud computing in healthcare or finance. Multi-Key Homomorphic Encryption (MKHE) further allows to process encrypted data from multiple sources: the data can be encrypted with keys owned by different parties. In this paper, we propose a new variant of MKHE instantiated with the TFHE scheme. Compared to previous attempts by Chen et al. and by Kwak et al., our scheme achieves computation runtime that is linear in the number of involved parties and it outperforms the faster scheme by a factor of 4.5-6.9×, at the cost of a slightly extended pre-computation. In addition, for our scheme, we propose and practically evaluate parameters for up to 128 parties, which enjoy the same estimated security as parameters suggested for the previous schemes (100 bits). It is also worth noting that our scheme—unlike the previous schemes—did not experience *any* error in any of our seven setups, each running 1 000 trials.

Keywords: Multi-key homomorphic encryption · TFHE scheme · Secure cloud computing

1 Introduction

Fully Homomorphic Encryption (FHE) refers to a cryptosystem that allows for an evaluation of an arbitrary computable function over encrypted data (first-ever scheme in [14], find a survey in [1]). With FHE, a secure cloud-aided computation, between a user (**U**) and a semi-trusted cloud (**C**), may proceed as follows:

- **U** generates secret keys sk, and evaluation keys ek, which she sends to **C**;
- **U** encrypts her sensitive data d with sk, and sends the encrypted data to **C**;
- **C** employs ek to evaluate function f, homomorphically, over the encrypted data, yielding an encryption of $f(d)$, which it sends back to **U**;
- **U** decrypts the message from **C** with sk, obtaining the result: $f(d)$ in plain.

* This work was supported by the MESRI-BMBF French-German joint project UPCARE (ANR-20-CYAL-0003-01). Find the full version at https://ia.cr/2023/065.

In such a setup, there is one party that holds all the secret keying material. In case the data originate from multiple sources, *Multi-Key (Fully) Homomorphic Encryption* (MKHE) comes into play. First proposed by López-Alt et al. [20], MKHE is a primitive that enables the homomorphic evaluation over data encrypted with multiple different, unrelated keys. This allows to relax the intrinsic restriction of a standard FHE, which demands a single data owner.

Previous Work. Following the seminal work of López-Alt et al. [20], different approaches to design an MKHE scheme have emerged: first attempts require a fixed list of parties at the beginning of the protocol [12, 25], others allow parties to join dynamically [5, 27], Chen et al. [8] extend the plaintext space from a single bit to a ring. Later, Chen et al. [6] propose an MKHE scheme based on the TFHE scheme [11], and they claim to be the first to practically implement an MKHE scheme; in this paper, we refer to their scheme as CCS. The evaluation complexity of their scheme is quadratic in the number of parties and authors only run experiments with up to 8 parties. The CCS scheme is improved in recent work by Kwak et al. [19], who achieve quasi-linear complexity (actually quadratic, but with a very low coefficient at the quadratic term); in this paper, we refer to their scheme as KMS. Parallel to CCS and KMS, which are both based on TFHE, there exist other promising schemes: e.g., [7], defined for BFV [4, 13] and CKKS [10], improved in [16] to achieve linear complexity, or [23], implemented in the Lattigo Library [24], which requires to first construct a common public key; also referred to as the *Multi-Party HE* (MPHE). The capabilities/use-cases of TFHE and other schemes are fairly different, therefore we solely focus on the comparison of TFHE-based MKHE.

Our Contributions. We propose a new TFHE-based MKHE scheme with a linear evaluation complexity and with a sufficiently low error rate, which allows for a practical instantiation with an order of hundreds of parties while achieving evaluation times proportional to those of plain TFHE. More concretely, our scheme builds upon the following technical ideas (k is the number of parties):

Summation of RLWE keys: Instead of *concatenation* of RLWE keys (in certain sense proposed in both CCS and KMS), our scheme works with RLWE encryptions under the *sum* of individual RLWE keys. As a result, this particular improvement decreases the evaluation complexity from quadratic to linear.

Ternary distribution for RLWE keys: Widely adopted by existing FHE implementations [15, 24, 22, 29], zero-centered ternary distribution $\zeta\colon (-1, 0, 1) \to (p, 1-2p, p)$ works well as a distribution of the coefficients of RLWE keys; we suggest $p \approx 0.1135$. It helps reduce the growth of a certain noise term by a factor of k, which in turn helps find more efficient TFHE parameters.

Avoid FFT in pre-computations: In our experiments, we notice an unexpected error growth for higher numbers of parties and we verify that the source of these errors is Fast Fourier Transform (FFT), which is used for fast polynomial multiplication. To keep the evaluation times low and to decrease the number of errors at the same time, we suggest replacing FFT with an exact method just in the pre-computation phase. We also show that FFT

causes a considerable amount of errors in KMS, however, replacing FFT in its pre-computations is unfortunately not sufficient.

We provide two variants of our scheme:

Static variant: the list of parties is fixed – the evaluation cost is independent of the number of participating parties, and the result is encrypted with all keys;

Dynamic variant: the computation cost is proportional to the number of participating parties, and the result is only encrypted with their keys (i.e., any subset of parties can go offline).

The variants only differ in pre-computation algorithms – performance-wise, given a fixed number of parties, the variants are equivalent (it only depends on the parameters of TFHE) and the evaluation complexity is linear in the number of involved parties. The construction of our scheme remains similar to that of plain TFHE, making it possible to adopt prospective advances of TFHE (or its implementation) to our scheme. In addition to the design of a new MKHE scheme:

- We support our scheme by a theoretical noise-growth & security analysis. Thanks to the low noise growth, we instantiate our scheme with as many as 128 parties. We show that our scheme is secure in the semi-honest model;
- We design and evaluate a deep experimental study, which may help evaluate future schemes. In particular, we suggest simulating the NAND gate to measure errors more realistically. Compared to KMS, we achieve 4.5-6.9× better bootstrapping times, while using the same implementation of TFHE and parameters with the same estimated security (100 bits). The bootstrapping times are around 140 ms per party (experimental implementation);
- We extend previous work by providing an experimental evaluation of the probability of errors. For our scheme, the measured noises fall within the expected bounds, which are designed to satisfy the rule of 4σ (1 in 15 787); we indeed do not encounter *any* error in any of our 9 000 trials in total.

Paper Outline. We briefly recall the TFHE scheme in Section 2 and we present our scheme in Section 3. We analyze the security, correctness & noise growth, and performance of our scheme in Section 4, which is followed by a thorough experimental evaluation in Section 5. We conclude our paper in Section 6.

2 Preliminaries

In this section, we briefly recall the original TFHE scheme [11]. First, let us provide a list of symbols & notation that we use throughout the paper:

- $\mathbb{B}$: the set of binary coefficients $\{0, 1\} \subset \mathbb{Z}$,
- $\mathbb{T}$: the additive group $\mathbb{R}/\mathbb{Z}$ referred to as the *torus* (i.e., real numbers $\bmod 1$),
- $\mathbb{Z}_n$: the quotient ring $\mathbb{Z}/n\mathbb{Z}$ (or its additive group),
- $M^{(N)}[X]$: the set of polynomials $\bmod X^N + 1$, with coefficients from M,
- \$: the uniform distribution,
- $a \xleftarrow{\alpha} M$: the draw of random variable a from M with distribution α (for $\alpha \in \mathbb{R}$, we consider the /discrete/ normal distribution $N(0, \alpha)$),
- $\mathrm{E}[X]$, $\mathsf{Var}[X]$: the expectation and the variance of random variable X.

2.1 TFHE Scheme

The TFHE scheme is based on the *Learning With Errors* (LWE) encryption scheme introduced by Regev [28]. TFHE employs two variants, originally referred to as T(R)LWE, which stands for *(Ring)* LWE *over the Torus*. The ring variant (shortly RLWE; introduced in [21]) is defined by polynomial degree $N = 2^\nu$ (with $\nu \in \mathbb{N}$), dimension $n \in \mathbb{N}$, noise distribution ξ over the torus, and key distribution ζ over the integers (generalized to respective polynomials $\bmod X^N + 1$). Informally, to encrypt torus polynomial $m \in \mathbb{T}^{(N)}[X]$, RLWE outputs the pair $(b = m - \langle \mathbf{z}, \mathbf{a} \rangle + e, \mathbf{a})$, referred to as the RLWE *sample*, where $\mathbf{z} \xleftarrow{\zeta} (\mathbb{Z}^{(N)}[X])^n$ is a secret key, $e \xleftarrow{\xi} \mathbb{T}^{(N)}[X]$ is an error term (aka. *noise*), and $\mathbf{a} \xleftarrow{\$} (\mathbb{T}^{(N)}[X])^n$ is a random mask. To decrypt, evaluate $\varphi_{\mathbf{z}}(b, \mathbf{a}) = b + \langle \mathbf{z}, \mathbf{a} \rangle = m + e$, also referred to as the *phase*. Internally, RLWE samples are further used to build so-called RGSW *samples*, which encrypt integer polynomials, and which allow for homomorphic multiplication of integer-torus polynomials. It is widely believed that RLWE sample $(b, \mathbf{a})$ is computationally indistinguishable from a random element of $(\mathbb{T}^{(N)}[X])^{1+n}$ (shortly random-like), provided that adequate parameters are chosen. If $\mathbf{a} = \mathbf{0}$ and $e = 0$, we talk about a *trivial sample*. The plain variant (shortly LWE) operates with plain torus elements instead of polynomials.

Bootstrapping. By its construction, (R)LWE is additively homomorphic: the sum of samples encrypts the sum of plaintexts. However, the error terms also add up, i.e., the average noise of the result grows. To deal with this issue, TFHE (as well as other fully homomorphic schemes) defines a routine referred to as *bootstrapping*. In addition to refreshing the noise of a noisy sample, TFHE bootstrapping is capable of evaluating a custom *Look-Up Table* (LUT), which makes TFHE fully homomorphic. Find an illustration of the operation flow in Figure 1. For a comprehensive technical description of TFHE, we refer to Appendix A.

In this paper, we focus on the basic variant of TFHE with a Boolean message space: true and false are encoded into $\mathbb{T} \sim [-1/2, 1/2)$ as $-1/8$ and $1/8$, respectively. To homomorphically evaluate the NAND gate over input samples $\mathbf{c}_{1,2}$, the sum $(1/8, \mathbf{0}) - \mathbf{c}_1 - \mathbf{c}_2$ is bootstrapped with a LUT, which holds $1/8$ and $-1/8$ for the positive and for the negative half of $\mathbb{T}$, respectively.

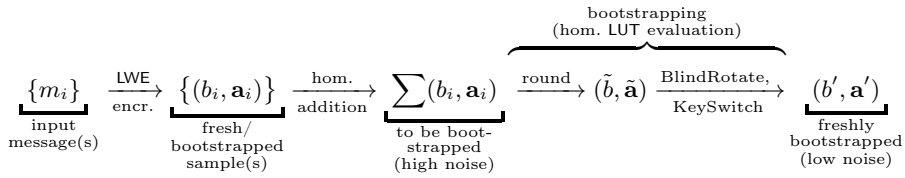

Fig. 1. The flow of TFHE: homomorphic addition and bootstrapping, which is composed of other operations. The output sample $(b', \mathbf{a}')$ may proceed to another homomorphic addition, or to the output and decryption.

3 The AKÖ Scheme

In this section, we recall the notion of Multi-Key Homomorphic Encryption (MKHE) and we propose two variants of MKHE. We outline changes that lead from the basic TFHE [11] towards our proposal of MKHE – we outline the format of multi-key bootstrapping keys, and we comment on a distribution for RLWE keys. We provide a technical description of our scheme, which we denote AKÖ.

3.1 Towards the AKÖ Scheme

In addition to the capabilities of a standard FHE scheme, an MKHE scheme:

(i) runs a homomorphic evaluation over ciphertexts encrypted with unrelated keys of multiple parties (accompanied by corresponding evaluation keys);
(ii) requires the collaboration of all involved parties, holding the individual keys, to decrypt the result.

Note that there exist multiple approaches to reveal the result: e.g., one outlined in [6], referred to as *Distributed Decryption*, or one described in [23], referred to as *Collective Public-Key Switching*.

We propose our scheme in two variants:

Static variant: the list of parties is fixed at the beginning, then evaluation keys are jointly calculated – no matter how many parties join a computation, the evaluation time is also fixed and the result is encrypted with all the keys;
Dynamic variant: after a "global" list of parties is fixed, evaluation keys are jointly calculated, however, only a subset of parties may join a computation – the evaluation cost is proportional to the size of the subset and the result is only encrypted with respective keys (i.e., the remaining parties can go offline). If a party joins later, a part of the joint pre-calculation of evaluation keys needs to be executed in addition, as opposed to CCS [6] and KMS [19].

Note that in many practical use cases—in particular, if we require semi-honest parties—the (global) list of parties is fixed, e.g., hospitals may constitute the parties. In addition, the pre-calculation protocol is indeed lightweight.

As already outlined, our scheme is based on the three following ideas:

(i) create RLWE samples encrypted under the sum of individual RLWE keys,
(ii) use a ternary (zero-centered) distribution for individual RLWE keys, and
(iii) avoid Fast Fourier Transform (FFT) in pre-computations.

Below, we discuss (i) and (ii), leaving (iii) for the experimental part (Section 5). Note that the following lines might require an in-depth knowledge of TFHE.

(R)LWE Keys & Bootstrapping Keys. First, let us emphasize that secret keys of individual parties are *never* revealed to any other party, however, the description of AKÖ involves all of them. The underlying (and never reconstructed) LWE key is the *concatenation* of individual keys, i.e., $\mathbf{s} \coloneqq \left(\mathbf{s}^{(1)}, \mathbf{s}^{(2)}, \ldots, \mathbf{s}^{(k)}\right) \in$

$\mathbb{B}^{kn}$, where $\mathbf{s}^{(p)} \in \mathbb{B}^n$ are secret LWE keys of individual parties. We refer to $\mathbf{s}$ as the *common* LWE *key*. For RLWE keys, we consider their *summation*, i.e., $Z \coloneqq \sum_p z^{(p)}$, which we refer to as the *common* RLWE *key*. This particular improvement decreases the computational complexity from $O(k^2)$ to $O(k)$.

For bootstrapping keys, we follow the original construction of TFHE, where we use the common (R)LWE keys. For *blind-rotate keys*, we generate an RGSW sample of each bit of the common LWE key $\mathbf{s} = \big(\mathbf{s}^{(1)}, \dots, \mathbf{s}^{(k)}\big)$, under the common RLWE key $Z = \sum_p z^{(p)}$. In addition, any party shall neither leak its own secrets nor require the secrets of others. Hence, we employ RLWE public key encryption [21]. Let us outline the desired form of a blind-rotate key for bit s:

$$\mathsf{BK}_s = \begin{pmatrix} \mathbf{b}^{\Delta} + s \cdot \mathbf{g} & \mathbf{a}^{\Delta} \\ \mathbf{b}^{\square} & \mathbf{a}^{\square} + s \cdot \mathbf{g} \end{pmatrix}, \quad \mathsf{BK}_s \in \big(\mathbb{T}^{(N)}[X]\big)^{2d \times 2}, \tag{1}$$

where $(\mathbf{b}^{\Delta}, \mathbf{a}^{\Delta})$ and $(\mathbf{b}^{\square}, \mathbf{a}^{\square})$ hold $d + d$ RLWE encryptions of zero under the key Z; cf. `TFHE.RgswEncr`. For *key-switching keys*, we need to generate an LWE sample of the sum of j-th coefficients of individual RLWE secret keys $z^{(p)}$, under the common LWE key $\mathbf{s}$, for $j \in [0, N-1]$. Here a simple concatenation of masks (values $\mathbf{a}$) and a summation of masked values (values b) do the job. With such keys, bootstrapping itself is identical to that of the original TFHE.

Ternary Distribution for RLWE Keys. For individual RLWE keys, we suggest to use zero-centered ternary distribution $\zeta_p \colon (-1, 0, 1) \to (p, 1-2p, p)$, parameterized by $p \in (0, 1/2)$, which is widely adopted by the main FHE libraries like HElib [15], Lattigo [24], SEAL [22], or HEAAN [29]. Although not adopted in CCS nor in KMS, in our scheme, a zero-centered distribution for RLWE keys is particularly useful, since we sum the keys into a common key, which is then also zero-centered. This helps reduce the blind-rotate noise from $O(k^3)$ to $O(k^2)$, which in turn helps find more efficient TFHE parameters.

It is worth noting that for "small" values of p, such keys are also referred to as *sparse keys* (in particular with a fixed/limited Hamming weight), and there exist specially tailored attacks [9, 31]. At this point, we motivate the choice of p solely by keeping the information entropy of ζ_p equal to 1 bit, however, there is no intuition—let alone a proof—that the estimated security would be at least similar (more in Section 5.1). For the information entropy of ζ_p, we have

$$H(\zeta_p) = -2p \log(p) - (1-2p) \log(1-2p) \stackrel{!}{=} 1, \tag{2}$$

which gives $p \approx 0.1135$. For $z_i \sim \zeta_p$, we have $\mathsf{Var}[z_i] = 2p \approx 0.227$.

3.2 Technical Description of AKÖ

Algorithms with index q are executed locally at the respective party, encryption algorithms naturally generalize to vector inputs.

Static Variant of AKÖ. Below, we provide algorithms for the static variant:

∘ $\underline{\texttt{AKÖ.Setup}}(1^\lambda, k)$: Given security parameter λ and the number of parties k, generate & distribute parameters for:

- LWE encryption: dimension n, standard deviation $\alpha > 0$ (of the noise);
- LWE decomposition: base B', depth d';
- set up LWE gadget vector: $\mathbf{g}' \leftarrow (1/B', 1/B'^2, \ldots, 1/B'^{d'})$;
- RLWE encryption: polynomial degree N (a power of two), std-dev $\beta > 0$;
- RLWE decomposition: base B, depth d;
- set up RLWE gadget vector: $\mathbf{g} \leftarrow (1/B, 1/B^2, \ldots, 1/B^d)$;
- generate a *common random polynomial* (CRP) $\underline{a} \xleftarrow{\$} \mathbb{T}^{(N)}[X]$.

∘ $\underline{\texttt{AKÖ.SecKeyGen}_q()}$: Generate secret keys $\mathbf{s}^{(q)} \xleftarrow{\$} \mathbb{B}^n$ and $z^{(q)} \in \mathbb{Z}^{(N)}[X]$, s.t. $z_i^{(q)} \xleftarrow{\zeta_p} \{-1, 0, 1\}$.

∘ $\underline{\texttt{AKÖ...}}$: Algorithms for (R)LWE en/decryption and bootstrapping (including `BlindRotate`, `KeySwitch`, etc.) are the same as in TFHE; cf. Appendix A.

∘ $\underline{\texttt{AKÖ.RLwePubEncr}\big(m, (b,a)\big)}$: Given message $m \in \mathbb{T}^{(N)}[X]$ and public key $(b,a) \in \mathbb{T}^{(N)}[X]^2$ (an RLWE sample of $0 \in \mathbb{T}^{(N)}[X]$ under key $z \in \mathbb{Z}^{(N)}[X]$), generate temporary RLWE key $r^{(q)}$, s.t. $r_i^{(q)} \xleftarrow{\zeta} \{-1, 0, 1\}$. Evaluate $b' \leftarrow \texttt{RLweSymEncr}_q(m, b, r^{(q)})$ and $a' \leftarrow \texttt{RLweSymEncr}_q(0, a, r^{(q)})$. Output (b', a'), which is an RLWE sample of m under the key z.

∘ $\underline{\texttt{AKÖ.RLweRevPubEncr}\big(m, (b,a)\big)}$: Proceed as `RLwePubEncr`, with a difference in the evaluation of $b' \leftarrow \texttt{RLweSymEncr}_q(0, b, r^{(q)})$ and $a' \leftarrow \texttt{RLweSymEncr}_q(m, a, r^{(q)})$, where only m and 0 are swapped, i.e., m is added to the right-hand side instead of the left-hand side.

∘ $\underline{\texttt{AKÖ.BlindRotKeyGen}_q()}$: Calculate and broadcast public key $b^{(q)} \leftarrow \texttt{RLweSymEncr}_q(0, \underline{a})$, using the CRP $\underline{a}$ as the mask. Evaluate $B = \sum_{p=1}^{k} b^{(p)}$ (n.b., $(B, \underline{a}) = \mathsf{RLWE}_Z(0)$, hence it may serve as a common public key). Finally, for $j \in [1, n]$, output the *blind-rotate key* (related to $s_j^{(q)}$ and Z):

$$\mathsf{BK}_j^{(q)} \leftarrow \begin{pmatrix} \texttt{RLwePubEncr}_q\big(\mathbf{s}_j^{(q)} \cdot \mathbf{g}, (B, \underline{a})\big) \\ \\ \texttt{RLweRevPubEncr}_q\big(\mathbf{s}_j^{(q)} \cdot \mathbf{g}, (B, \underline{a})\big) \end{pmatrix}, \tag{3}$$

which is an RGSW sample of the j-th bit of $\mathbf{s}^{(q)}$, under the common RLWE key Z.

∘ $\underline{\texttt{AKÖ.KeySwitchKeyGen}_q()}$: For $i \in [1, N]$, broadcast $[\mathbf{b}_i^{(q)} | \mathbf{A}_i^{(q)}] \leftarrow \texttt{LweSymEncr}_q\big(\mathbf{z}_i^{(q)*} \cdot \mathbf{g}'\big)$, where $\mathbf{z}^{(q)*} \leftarrow \texttt{KeyExtract}(z^{(q)})$. Aggregate and for $i \in [1, N]$, output the *key-switching key* (for $Z_i = \sum_p z_i^{(p)}$ and $\mathbf{s} = (\mathbf{s}^{(1)}, \ldots, \mathbf{s}^{(k)})$):

$$\mathsf{KS}_i = \Big[\underbrace{\sum_{p=1}^{k} \mathbf{b}_i^{(p)}}_{\mathbf{b}_i} \;\Big|\; \underbrace{\mathbf{A}_i^{(1)}, \mathbf{A}_i^{(2)}, \ldots, \mathbf{A}_i^{(k)}}_{\mathbf{A}_i} \Big], \tag{4}$$

which is a d'-tuple of LWE samples of $\mathbf{g}'$-respective fractions of $\mathbf{Z}_i^*$ under the common LWE key $\mathbf{s}$, where $\mathbf{Z}_i^*$ is the i-th element of the extraction of the common RLWE key $Z = \sum_p z^{(p)}$.

Changes to AKÖ towards the Dynamic Variant. For the dynamic variant, we provide modified versions of `BlindRotKeyGen` and `KeySwitchKeyGen`; other algorithms are the same as in the static variant. Note that, in case we allow a party to join later, all temporary keys need to be stored permanently and both algorithms need to be (partially) repeated. This causes a slight pre-computation overhead over CCS and KMS.

∘ $\underline{\texttt{AKÖ.BlindRotKeyGen\_dyn}_q()}$: Calculate and broadcast public key $b^{(q)}$ as described in the $\texttt{AKÖ.BlindRotKeyGen}_q()$ algorithm. Then, for $j \in [1, n]$:

1: generate two vectors of d temporary RLWE keys $\mathbf{r}_j^{(q)}$ and $\mathbf{r'}_j^{(q)}$
2: for $p \in [1, k]$, $p \neq q$, output $\mathbf{b}_{q,j}^{\Delta(p)} \leftarrow \texttt{RLweSymEncr}_q(0, b^{(p)}, \mathbf{r}_j^{(q)})$
3: output $\mathbf{b}_{q,j}^{\Delta(q)} \leftarrow \texttt{RLweSymEncr}_q(\mathbf{s}_j^{(q)} \cdot \mathbf{g}, b^{(q)}, \mathbf{r}_j^{(q)})$
4: output $\mathbf{a}_{q,j}^{\Delta} \leftarrow \texttt{RLweSymEncr}_q(0, \underline{a}, \mathbf{r}_j^{(q)})$
5: for $p \in [1, k]$, output $\mathbf{b}_{q,j}^{\square(p)} \leftarrow \texttt{RLweSymEncr}_q(0, b^{(p)}, \mathbf{r'}_j^{(q)})$
6: output $\mathbf{a}_{q,j}^{\square} \leftarrow \texttt{RLweSymEncr}_q(\mathbf{s}_j^{(q)} \cdot \mathbf{g}, \underline{a}, \mathbf{r'}_j^{(q)})$

To construct the j-th blind-rotate key of party q, related to subset of parties $\mathcal{S} \ni q$, evaluate

$$\mathsf{BK}_{j,\mathcal{S}}^{(q)} \leftarrow \begin{pmatrix} \sum_{p\in\mathcal{S}} \mathbf{b}_{q,j}^{\Delta(p)} & \mathbf{a}_{q,j}^{\Delta} \\ \sum_{p\in\mathcal{S}} \mathbf{b}_{q,j}^{\square(p)} & \mathbf{a}_{q,j}^{\square} \end{pmatrix}, \tag{5}$$

which is an RGSW sample of $\mathbf{s}_j^{(q)}$ under the subset RLWE key $Z_{\mathcal{S}} = \sum_{p\in\mathcal{S}} z^{(p)}$. N.b., $\mathsf{BK}_{j,\mathcal{S}}^{(q)}$ is only calculated at runtime, once $\mathcal{S}$ is known.

∘ $\underline{\texttt{AKÖ.KeySwitchKeyGen\_dyn}_q()}$: Proceed as $\texttt{AKÖ.KeySwitchKeyGen}_q()$, while instead of outputting aggregated KS_i's, aggregate relevant parts once $\mathcal{S}$ is known:

$$\mathsf{KS}_{i,\mathcal{S}} = \Big[\sum_{p\in\mathcal{S}} \mathbf{b}_i^{(p)} \;\Big|\; \left(\mathbf{A}_i^{(p)}\right)_{p\in\mathcal{S}}\Big]. \tag{6}$$

Possible Improvements. In [6], authors suggest an improvement that decreases the noise growth of key-switching, which can also be applied in our scheme; we provide more details in the full version of this paper [18].

4 Theoretical Analysis of AKÖ

In this section, we provide a theoretical analysis of our AKÖ scheme with respect to *security*, *correctness* (noise growth), and *performance*. For a detailed technical description of some of the involved algorithms, we refer to Appendix A – in particular, for those shared by AKÖ and TFHE; cf. AKÖ... in Section 3.2.

4.1 Security

We assume that all parties follow the protocol *honestly-but-curiously* (aka. the *semi-honest model*). First, let us recall what *is* secure and what *is not* in LWE (selected methods; also holds for RLWE):

- ✓ re-use secret key $\mathbf{s}$ with fresh mask $\mathbf{a}$ and fresh noise e;
- ✓ re-use common random mask $\underline{\mathbf{a}}$ with multiple distinct secret keys $\mathbf{s}^{(p)}$ and fresh noises $e^{(p)}$;
- ✗ publish $\langle \mathbf{s}, \mathbf{a} \rangle$ in any form (e.g., release the phase φ or the noise e);
- ✗ re-use the pair $(\mathbf{s}, \mathbf{a})$ with fresh noises e_i.

Below, we show that if all parties act semi-honestly, our scheme is secure in both of its variants. Note that rather than formal proofs, we provide informal sketches.

Public Key Encryption. In AKÖ, there are two algorithms for public key encryption: `RLwe(Rev)PubEncr`$\big(m, (b, a)\big)$. They re-use a common random mask (the public key pair (b, a)) with fresh temporary key $r^{(q)}$. Provided that b and a are indistinguishable from random (random-like), it does not play a role to which part the message m is added/encrypted, i.e., both variants are secure.

Blind-Rotate Key Generation (static variant). Provided that CRP $\underline{a}$ is random-like, which is trivial to achieve in the random oracle model, we can assume that (our) $b^{(q)}$ is random-like. Assuming that other parties act honestly, also their $b^{(p)}$'s are random-like, hence the sum B is random-like, too. With $(B, \underline{a})$ random-like, public key encryption algorithms are secure, hence `AKÖ.BlindRotKeyGen`$_q$ is secure, too.

Blind-Rotate Key Generation (dynamic variant). In this variant, party q re-uses temporary secret key $r^{(q)}$ for encryption of zeros using public keys $b^{(p)}$ of other parties, and for encryption of own secret key $\mathbf{s}^{(q)}$. This is secure provided that $b^{(p)}$'s are random-like, which is true if generated honestly.

Key-Switching Key Generation (both variants). The `AKÖ.KeySwitchKeyGen(_dyn)`$_q$ algorithms employ the standard LWE encryption, hence they are both secure.

4.2 Correctness & Noise Growth

The most challenging part of all LWE-based schemes is to estimate the noise growth across various operations. First, we provide estimates of the noise growth of blind-rotate and key-switching, next, we combine them into an estimate of the noise of a freshly bootstrapped sample. Finally, we identify the maximum of error, which may cause incorrect bootstrapping. We evaluate all noises for the static variant, while for the dynamic variant, we provide more comments below respective theorems. All proofs can be found in the full version of this paper [18].

Theorem 1 (Noise Growth of Blind-Rotate). *The* `AKÖ.BlindRotate` *algorithm returns a sample with noise variance given by*

$$\mathsf{Var}[\langle \bar{\mathbf{Z}}, \mathsf{ACC} \rangle] \approx \underbrace{knNdV_B\beta^2(3+6pkN)}_{\mathsf{BK}\ error} + \underbrace{1/2 \cdot kn\varepsilon^2(1+2pkN)}_{decomp.\ error} + \underbrace{\mathsf{Var}[tv]}_{usually\ 0}. \quad (7)$$

For the dynamic variant, we have $(3 + k \cdot 6pN) \to \big(1 + k(2 + 6pN)\big)$ in the BK error term, which we consider practically negligible as $6pN \approx 700$.

Theorem 2 (Noise Growth of Key-Switching). *The* `AKÖ.KeySwitch` *algorithm returns a sample that encrypts the same message as the input sample, while changing the key from* $\mathbf{Z}^*$ *to* $\mathbf{s}$*, with additional noise* e_{KS}*, given by* $\langle \bar{\mathbf{s}}, \bar{\mathbf{c}}'' \rangle = \langle \bar{\mathbf{Z}}^*, \bar{\mathbf{c}}' \rangle + e_{\mathsf{KS}}$*, for which*

$$\mathsf{Var}[e_{\mathsf{KS}}] \approx \underbrace{Nkd' V_{B'} \beta'^2}_{\mathsf{KS}\ error} + \underbrace{2pkN\varepsilon'^2}_{decomp.\ error} . \tag{8}$$

For the dynamic variant, key-switching keys are structurally equivalent, hence this estimate holds in the same form.

Corollary 1 (Noise of a Freshly Bootstrapped Sample). *The* `AKÖ.Bootstrap` *algorithm returns a sample with noise variance given by*

$$V_0 \approx \underbrace{3knNdV_B\beta^2(1+2pkN)}_{\mathsf{BK}\ error} + \underbrace{1/2kn\varepsilon^2(1+2pkN)}_{b.\text{-}r.\ decomp.} + \underbrace{Nkd' V_{B'}\beta'^2}_{\mathsf{KS}\ error} + \underbrace{2pkN\varepsilon'^2}_{k.\text{-}s.\ decomp.} . \tag{9}$$

For the dynamic variant, the BK *error term is changed according to Theorem 1.*

Maximum of Error. During homomorphic evaluations, freshly bootstrapped samples get homomorphically added/subtracted, before being possibly bootstrapped again; cf. Figure 1. Before a noisy sample gets blindly rotated, it gets scaled and rounded to Z_{2N}, which induces an additional rounding error.

Lemma 1 (Rounding Error). *The rounding step before* `BlindRotate` *induces an additional error with variance (in the torus scale) given by*

$$\mathsf{Var}\big[\langle \bar{\mathbf{s}}, 1/2N \cdot (\tilde{b}, \tilde{\mathbf{a}}) - (b, \mathbf{a}) \rangle\big] = \frac{1 + kn/2}{48N^2} =: V_{round}(N, n, k). \tag{10}$$

After rounding, the noise gets refreshed inside the `BlindRotate` algorithm, which "blindly-rotates" a torus polynomial, referred to as the *test vector*, which encodes a LUT. I.e., the rounding step is where the maximum of errors across the whole computation appears. We focus on this error in the experimental part, since it may cause incorrect blind-rotation, in turn, incorrect LUT evaluation. In the following corollary, we evaluate the variance of the maximal error and we define quantity κ, which is a scaling factor of normal distribution $N(0, 1)$.

Corollary 2 (Maximum of Error). *The maximum average error throughout homomorphic computation is achieved inside* `AKÖ.Bootstrap` *by the rounded sample* $1/2N \cdot (\tilde{b}, \tilde{\mathbf{a}})$ *with variance*

$$V_{\max} \approx \max\Big\{\sum k_i^2\Big\} \cdot V_0 + V_{round}, \tag{11}$$

where k_i *are coefficients of linear combinations of independent, freshly bootstrapped samples, which are evaluated during homomorphic calculations, before being bootstrapped (e.g.,* $\sum k_i^2 = 2$ *for the NAND gate evaluation). We denote*

$$\kappa := \frac{\delta/2}{\sqrt{V_{\max}}} = \frac{\delta}{2\sigma_{\max}}, \tag{12}$$

where δ is the distance of encodings that are to be distinguished (e.g., $1/4$ for encoding of bools).

We use κ to estimate the probability of *correct blind rotation* (CBRot). E.g., for $\kappa = 3$, we have $\Pr[\text{CBRot}] \approx 99.73\% \approx 1/370$ (aka. rule of 3σ), however, we rather lean to $\kappa = 4$ with $\Pr[\text{CBRot}] \approx 1/15\,787$. Since the maximum of error is achieved within blind-rotate, it dominates the overall probability of *correct bootstrapping* (CBStrap), i.e., we assume $\Pr[\text{CBStrap}] \approx \Pr[\text{CBRot}]$.

4.3 Performance

Since the structure of all components in both variants of AKÖ is equivalent to that of plain TFHE with only $n \to kn$ (due to LWE key concatenation), we evaluate the performance characteristics very briefly: `AKÖ.BlindRotate` is dominated by $4d \cdot kn$ degree-N polynomial multiplications, whereas `AKÖ.KeySwitch` is dominated by $Nd' \cdot (1 + kn)$ torus multiplications, followed by $1 + kn$ summations of Nd' elements. Using FFT for polynomial multiplication, for bootstrapping, we have the complexity of $O(N \log N \cdot 4dkn) + O(Nd' \cdot (1 + kn))$.

For key sizes, we have $|\mathsf{BK}| = 4dNkn \cdot |\mathbb{T}_{\mathsf{RLWE}}|$ and $|\mathsf{KS}| = d'N(1+kn) \cdot |\mathbb{T}_{\mathsf{LWE}}|$, where $|\mathbb{T}_{(\mathsf{R})\mathsf{LWE}}|$ denotes the size of respective torus representation.

5 Experimental Evaluation

For a fair comparison, we implement our AKÖ scheme[3] side by side with previous schemes CCS [6] and KMS [19]. These are implemented in a fork [30] of a library[4] [26] that implements TFHE in Julia. For the sake of simplicity, we implement only the static variant on AKÖ – recall that performance-wise, the two variants are equivalent, for noise growth, the differences are negligible.

In this section, we first comment on errors induced by existing TFHE implementations. Then, we introduce type-1 and type-2 decryption errors that one may encounter during TFHE-based homomorphic evaluations. Finally, we provide three kinds of results of our experiments:

1. for all the three schemes (CCS, KMS, and AKÖ) and selected parameter sets, we measure the *performance*, the *noise variances*, and the *amount of decryption errors* of the two types,
2. we demonstrate the *effect of FFT* during the pre-computation phase of AKÖ,
3. we compare the performance of all the three schemes with a *fixed parameter set* tailored for 16 parties, with different numbers of actually participating parties (i.e., the setup of the dynamic variant).

We run our experiments on a machine with an Intel Core i7-7800X processor and 128 GB of RAM.

[3] Available at `https://gitlab.eurecom.fr/fakub/3-gen-mk-tfhe` as `3gen`.
[4] As noted by the authors, the code serves solely as a proof-of-concept.

Implementation Errors. The major source of errors that stem from a particular implementation of the TFHE scheme is Fast Fourier Transform (FFT), which is used for fast modular polynomial multiplication in RLWE; find a study on FFT errors in [17]. Also, the finite representation of the torus (e.g., 64-bit integers) changes the errors slightly, however, we neglect this contribution as long as the precision (e.g., 2^{-64}) is smaller than the standard deviation of the (R)LWE noise. Note that these kinds of errors are not taken into account in Section 4.2, which solely focuses on the theoretical noise growth of the scheme itself.

Due to the excessive noise that we observe for higher numbers of parties with our scheme, we suggest replacing FFT in pre-computations (i.e., in blind-rotate key generation) with an exact method. This leads to an increase of the pre-computation costs (n.b., it has no effect on the bootstrapping time), however, in Section 5.2, we show that the benefit is worth it.

Types of Decryption Errors. The ultimate goal of noise analysis is to keep the probability of obtaining an incorrect result reasonably low. Below, we describe two types of decryption errors, which originate from bootstrapping, and which we measure in our experiments. N.b., the principle of `BlindRotate` is the same across the three schemes, hence it is well-defined for all of them.

Note 1. For the notion of *correct decryption*, we always assume symmetric intervals around encodings. E.g., for the Boolean variant of TFHE, which encodes true and false as $\pm 1/8$, we only consider the "correct" interval for true as $(0, 1/4)$, although $(0, 1/2)$ would work, too. Hence in the Boolean variant, actual incorrect decryption & decoding would be half less likely than what we actually measure.

Fresh Bootstrap Error. We bootstrap noiseless sample $\mathbf{c}$ of μ, i.e., `BlindRotate` rotates the test vector "correctly", meaning that $\bar{\varphi}/2N = \mu$ selects the correct position from the encoded LUT. Then, we evaluate the probability of the resulting phase φ' falling outside the correct interval. We refer to this error as the *type-1 error*, denoted Err_1. This probability relates to the noise of a correctly blind-rotated, freshly bootstrapped sample. It can be estimated from V_0; see (9).

Blind Rotate Error. Let us consider a homomorphic sum of two independent, freshly bootstrapped samples (cf. Figure 1). We evaluate the probability that the sum, after the rounding step inside bootstrapping, selects a value at an *incorrect* position from the test vector, which encodes the LUT (as discussed in Section 4.2). We refer to this error as the *type-2 error*, denoted Err_2. It can be estimated from $V_{\max}$; see (11). We evaluate Err_2 by simulating the NAND gate:

$$\left.\begin{array}{l}\text{fresh } \mathbf{c}_1 \xrightarrow{\texttt{Bootstr.}} \mathbf{c}_1' \\ \text{fresh } \mathbf{c}_2 \xrightarrow{\texttt{Bootstr.}} \mathbf{c}_2'\end{array}\right\} \; (1/8 - \mathbf{c}_1' - \mathbf{c}_2') \to \text{get rounded } \tilde{\varphi} \to \text{check } \bar{\varphi}/2N \overset{?}{\in} (0, 1/4). \tag{13}$$

5.1 Experiment #1: Comparison of Performance & Errors

For the three schemes—CCS, KMS and AKÖ—we measure the main quantities: the bootstrapping time (median), the variance V_0 of a freshly bootstrapped sample (defined in (9)), the scaling factor κ (defined in (12)), and the number of

errors of both types. We extend the previous work – there is no experimental evaluation of noises/errors in CCS nor in KMS. In all experiments, we replace FFT in pre-computations with an exact method. For CCS and KMS, we employ the parameters suggested by the original authors, and we estimate their security with the `lattice-estimator` by Albrecht et al. [2, 3]. We obtain an estimate of about 100 bits, therefore for our scheme, we also suggest parameters with estimated 100-bit security. We provide more details on concrete security estimates of the parameters of CCS, KMS and AKÖ in the full version of this paper [18]. The results for CCS, KMS and AKÖ can be found in Table 1, 2 and 3, respectively.

In the results for CCS, we may notice that for 2 to 8 parties, the measured value of κ, denoted $\kappa^{(m)}$, agrees with the calculated value $\kappa^{(c)}$, whereas for 16 parties (n.b., parameters added in KMS [19]), the measured value $\kappa^{(m)}$ drops significantly, which indicates an unexpected error growth.

In the results for KMS, we may notice a similar drop of κ – here it occurs for all numbers of parties – we suppose that this is caused by FFT in bootstrapping (more on FFT later in Section 5.2). For both experiments, we further use $\kappa^{(m)}$ and Z-values of the normal distribution to evaluate the expected rate of Err_2, which is in perfect accordance with the measured one.

For our AKÖ scheme, the results do not show *any* error of any type. Regarding the values of κ (also V_0), we measure lower noise than expected – this we suppose to be caused by a certain statistical dependency of variables – indeed, our estimates of noise variances are based on an assumption that variables are independent, which is not always fully satisfied. We are able to run AKÖ with up to 128 parties, while the only limitation for 256 parties appears to be the size of RAM. We believe that with more RAM ($> 128\,\mathrm{GB}$) or with a more optimized implementation, it would be possible to practically instantiate the scheme with even more parties.

Table 1. Key sizes (taken from [19]), bootstrapping times (t_B; median), noises and errors of the CCS scheme [6], with original parameters and *without* FFT in pre-computations (i.e., using precise calculations). *Parameters for $k = 16$ added by [19]. Labels $^{(c)}$ and $^{(m)}$ refer to calculated and measured values, respectively. Running 1 000 trials, i.e., evaluating 2 000 bootstraps; cf. (13). N.b., the actual error rate of a NAND gate would be approximately half of Err_2; cf. Note 1.

k	$\lvert\mathsf{keys}\rvert$ [MB]	t_B [s]	$V_0^{(c)}$ [10^{-4}]	$V_0^{(m)}$ [10^{-4}]	$\kappa^{(c)}$	$\kappa^{(m)}$	$\mathsf{Err}_{1,2}$ [‰]		Exp. Err_2
2	95	.58	16.2	14.6	2.19	2.30	1	24	21
4	108	2.4	19.1	18.6	2.01	2.04	3	41	41
8	121	10	6.36	6.27	3.39	3.41	0	0	.65
*16	214	86	2.15	34.5	5.07	1.49	29	128	136

Table 2. Key sizes (taken from [19]), bootstrapping times (t_B; median), noises and errors of the KMS scheme [19], with original parameters and without FFT in pre-computations. Running 1 000 trials.

k	$\lvert\mathsf{keys}\rvert$ [MB]	t_B [s]	$V_0^{(c)}$ [10^{-4}]	$V_0^{(m)}$ [10^{-4}]	$\kappa^{(c)}$	$\kappa^{(m)}$	$\mathsf{Err}_{1,2}$ [‰]		Exp. Err_2
2	215	.61	.458	11.5	12.7	2.60	1.5	12	9.3
4	286	2.1	.915	15.3	8.97	2.26	4	29	24
8	251	5.4	1.83	17.1	6.34	2.13	3	35	33
16	286	15	3.66	32.0	4.49	1.56	22.5	122	119
32	322	35	7.32	30.1	3.17	1.60	23	109	110

Table 3. Parameters, key sizes (calculated), bootstrapping times (t_B; median), noises, and errors of the static variant of AKÖ, without FFT in pre-computations. Running 1 000 trials, no errors of type Err_2 (let alone Err_1) experienced.

k	LWE				RLWE				$\lvert\mathsf{keys}\rvert$ [GB]	t_B [s]	$V_0^{(c)}$ [10^{-4}]	$V_0^{(m)}$ [10^{-4}]	$\kappa^{(c)}$	$\kappa^{(m)}$
	n	$\log_2(\alpha)$	B'	d'	N	$\log_2(\beta)$	B	d						
2	520	−13.52	2^3	3			2^7	2	.08	.19	4.69	4.18	4.04	4.27
4	510	−13.26	2^2	5	1 024	−30.70	2^6	3	.24	.56	3.96	2.02	4.33	5.93
8	540	−14.04	2^2	5			2^4	4	.66	1.2	4.43	4.20	4.01	4.11
16	590	−15.34	2^3	4			2^{26}	1	.93	1.8	4.56	1.02	4.04	7.90
32	620	−16.12	2^3	4	2 048	−62.00	2^{26}	1	2.0	4.3	3.58	1.21	4.38	6.78
64	650	−16.90	2^3	4			2^{25}	1	4.1	8.6	3.41	1.80	4.20	5.25
128	670	−17.42	2^3	5			2^{24}	1	9.1	18	2.40	.486	4.15	5.47

5.2 Experiment #2: The Effect of FFT in Pre-Computations

As outlined, polynomial multiplication in RLWE, when implemented using FFT, introduces additional error, on top of the standard RLWE noise. In this experiment, we compare noises of freshly bootstrapped samples: once *with* FFT in blind-rotate key generation (induces additional errors), once *without* FFT (we use an exact method instead). We choose our AKÖ scheme with 32 parties.

We observe a tremendous growth of the noise of a freshly bootstrapped sample in case FFT is employed for blind-rotate key generation: in almost 4% of such cases, even a freshly bootstrapped sample gets decrypted incorrectly (i.e., $\mathsf{Err}_1 \approx 4\%$). On the other hand, such a growth does not occur for lower numbers of parties, hence we suggest verifying whether in the particular case, the effect of FFT is remarkable, or negligible, and then decide accordingly. Recall that pre-computations with FFT are much faster (e.g., for 64 parties, we have 33 s vs. 212 s of the total pre-computation time).

Unexpected Error Growth in KMS. For the KMS scheme, we observe an unexpected error growth (cf. Table 2), which we suppose to be caused by FFT in bootstrapping. We replace *all* FFTs in the entire computation of KMS—including bootstrapping—with an exact method, and we re-run Experiment #1 with the KMS scheme with (only) 2 parties – due to a $\sim 40\times$ slower evaluation.

We obtain $V_0^{(m)} \approx 5.58 \cdot 10^{-4}$, which is still much more than the expected value $V_0^{(c)} \approx 0.458 \cdot 10^{-4}$, but the value of $\kappa^{(m)}$ increases from 2.60 to 3.73 and it results in no type-2 errors. At least partially, this confirms our hypothesis that the unexpected error growth in KMS is caused by FFT in evaluation.

Supporting evidence can be found in the design of KMS: in its blind-rotate, we observe that there are (up to) 4 nested FFTs: one in the circled $\star$ product, followed by three inside `ExtProd`: one in the $\odot$ product and two in `NewHbProd`. Compared with `AKÖ`, where there is just one level of FFT inside blind-rotate in `Prod`, this is likely the most significant practical improvement over KMS.

5.3 Experiment #3: Performance Comparison

We extend the performance comparison of CCS and KMS, presented in Figure 2 of KMS [19] (which we re-run on our machine), by the performances of our `AKÖ` scheme. Note that the setup of that experiment corresponds to the dynamic variant – recall that performance-wise, the dynamic variant is equivalent to the static variant, which is implemented in our experimental library. For each scheme, we employ its own parameter set tailored for 16 parties, while we instantiate it with different numbers of actually participating parties; find the results in Figure 2.

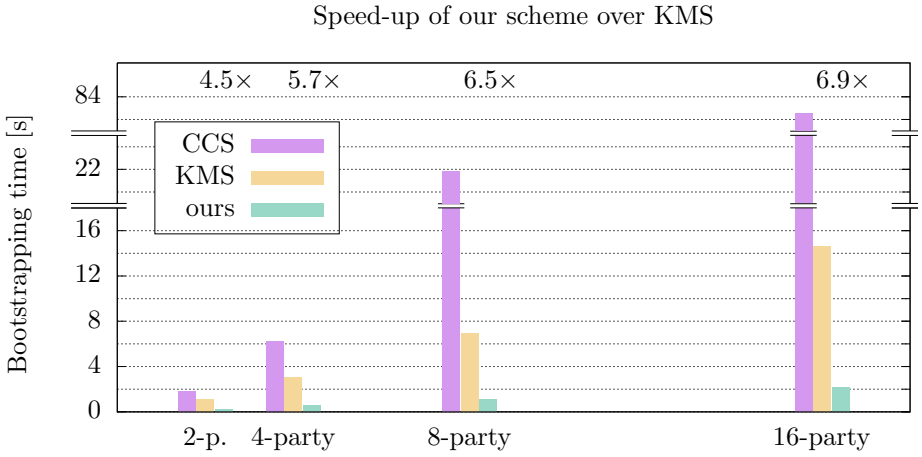

Fig. 2. Comparison of median bootstrapping times of the CCS scheme [6], the KMS scheme [19], and our `AKÖ` scheme. 100 runs with respective parameters for 16 parties were executed. N.b., FFT in pre-computations does not affect performance.

5.4 Discussion

The goal of our experiments is to show the practical usability of our AKÖ scheme: we compare its performance as well as the probability of errors with previous schemes – CCS [6] and KMS [19].

In terms of bootstrapping time, AKÖ runs faster than both previous attempts (cf. Figure 2). Also, the theoretical complexity of AKÖ is linear in the number of parties (cf. Section 4.3), as opposed to quadratic and quasi-linear for CCS and KMS, respectively.

To evaluate the number of errors that may occur during bootstrapping, we propose a new method that simulates the rounding step of `BlindRotate` (cf. (13)), which is the same across all the three schemes. Our experiments show that both CCS and KMS suffer from a considerably high error rate (cf. Table 1 and 2, respectively): for CCS, the original parameters are rather poor; for KMS, it seems that there are too many nested FFT's in bootstrapping – we show that FFT in evaluation—at least partially—causes the unexpected error growth.

To sum up, AKÖ significantly outperforms both CCS & KMS in terms of bootstrapping time and/or error rate. The major practical limitation of the CCS scheme is the quadratic growth of the bootstrapping time, whereas the KMS scheme suffers from the additional error growth in implementation. A disadvantage of AKÖ is that it requires (a small amount of) additional pre-computations if a new party decides to join the computation in the dynamic variant. Also AKÖ does not enable parallelization, as opposed to KMS.

6 Conclusion

We propose a new TFHE-based MKHE scheme named AKÖ in two variants, depending on whether only a subset of parties is desired to take part in a homomorphic computation. We implement AKÖ side-by-side with other similar schemes CCS and KMS, and we show its practical usability in thorough experimentation, where we also suggest secure & reliable parameters. Thanks to its low noise growth, AKÖ can be instantiated with hundreds of parties; namely, we tested up to 128 parties. Compared to previous schemes, AKÖ achieves much faster bootstrapping times, however, a slight overhead of pre-computations is induced. For KMS, we show that FFT errors are prohibitive for its practical deployment – unfortunately, replacing FFT in pre-computations is not enough.

Besides benchmarking, we suggest emulating (a part of) the NAND gate to achieve a more realistic error analysis: the measured amount of errors shows to be in perfect accordance with the expected amount. This method may help future schemes to evaluate their practical reliability.

Future Work. We plan to extend the threat model to assume malicious parties.

References

1. Acar, A., Aksu, H., Uluagac, A.S., Conti, M.: A survey on homomorphic encryption schemes: Theory and implementation. ACM Computing Surveys (Csur) **51**(4), 1–35 (2018)
2. Albrecht, M.R., contributors: Security Estimates for Lattice Problems. `https://github.com/malb/lattice-estimator` (2022)
3. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology **9**(3), 169–203 (2015)
4. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: Annual Cryptology Conference. pp. 868–886. Springer (2012)
5. Brakerski, Z., Perlman, R.: Lattice-based fully dynamic multi-key fhe with short ciphertexts. In: Annual International Cryptology Conference. pp. 190–213. Springer (2016)
6. Chen, H., Chillotti, I., Song, Y.: Multi-key homomorphic encryption from tfhe. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 446–472. Springer (2019)
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: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 395–412 (2019)
8. Chen, L., Zhang, Z., Wang, X.: Batched multi-hop multi-key fhe from ring-lwe with compact ciphertext extension. In: Theory of Cryptography Conference. pp. 597–627. Springer (2017)
9. Cheon, J.H., Hhan, M., Hong, S., Son, Y.: A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret lwe. IEEE Access **7**, 89497–89506 (2019)
10. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 409–437. Springer (2017)
11. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)
12. Clear, M., McGoldrick, C.: Multi-identity and multi-key leveled fhe from learning with errors. In: Annual Cryptology Conference. pp. 630–656. Springer (2015)
13. Fan, J., Vercauteren, F.: Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2012/144 (2012), `https://ia.cr/2012/144`
14. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 169–178 (2009)
15. Halevi, S., Shoup, V.: Design and implementation of a homomorphic-encryption library. IBM Research (Manuscript) **6**(12-15), 8–36 (2013)
16. Kim, T., Kwak, H., Lee, D., Seo, J., Song, Y.: Asymptotically Faster Multi-Key Homomorphic Encryption from Homomorphic Gadget Decomposition. Cryptology ePrint Archive, Paper 2022/347 (2022), `https://ia.cr/2022/347`
17. Klemsa, J.: Fast and error-free negacyclic integer convolution using extended fourier transform. In: Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings. pp. 282–300. Springer (2021)
18. Klemsa, J., Önen, M., Akın, Y.: A practical tfhe-based multi-key homomorphic encryption with linear complexity and low noise growth. Cryptology ePrint Archive, Paper 2023/065 (2023), `https://eprint.iacr.org/2023/065`, `https://eprint.iacr.org/2023/065`

19. Kwak, H., Min, S., Song, Y.: Towards Practical Multi-key TFHE: Parallelizable, Key-Compatible, Quasi-linear Complexity. Cryptology ePrint Archive, Paper 2022/1460 (2022), https://ia.cr/2022/1460
20. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Proceedings of the forty-fourth annual ACM symposium on Theory of computing. pp. 1219–1234 (2012)
21. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 1–23. Springer (2010)
22. Microsoft: SEAL (release 4.1). https://github.com/Microsoft/SEAL (Jan 2023)
23. Mouchet, C., Troncoso-Pastoriza, J., Bossuat, J.P., Hubaux, J.P.: Multiparty homomorphic encryption from ring-learning-with-errors. Proceedings on Privacy Enhancing Technologies pp. 291–311 (2021)
24. Mouchet, C.V., Bossuat, J.P., Troncoso-Pastoriza, J.R., Hubaux, J.P.: Lattigo: A multiparty homomorphic encryption library in go. In: Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography. pp. 64–70. No. CONF (2020)
25. Mukherjee, P., Wichs, D.: Two round multiparty computation via multi-key fhe. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 735–763. Springer (2016)
26. NuCypher: TFHE.jl. https://github.com/nucypher/TFHE.jl (2022)
27. Peikert, C., Shiehian, S.: Multi-key fhe from lwe, revisited. In: Theory of cryptography conference. pp. 217–238. Springer (2016)
28. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing. pp. 84–93 (2005)
29. SNUCrypto: HEAAN (release 1.1). https://github.com/snucrypto/HEAAN (2018)
30. SNUPrivacy: MK-TFHE. https://github.com/SNUPrivacy/MKTFHE (2022)
31. Son, Y., Cheon, J.H.: Revisiting the hybrid attack on sparse secret lwe and application to he parameters. In: Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 11–20 (2019)

A Technical Description of TFHE

We provide a technical description of the TFHE scheme in a form of self-descriptive algorithms. Parameters and secret keys are considered implicit inputs.

◦ TFHE.Setup(1^λ): Given security parameter λ, generate parameters for:

- LWE encryption: dimension n, standard deviation $\alpha > 0$ (of the noise);
- LWE decomposition: base B', depth d';
- set up LWE gadget vector: $\mathbf{g}' \leftarrow (1/B', 1/B'^2, \ldots, 1/B'^{d'})$;
- RLWE encryption: polynomial degree N (a power of two), standard deviation $\beta > 0$;
- RLWE decomposition: base B, depth d;
- set up RLWE gadget vector: $\mathbf{g} \leftarrow (1/B, 1/B^2, \ldots, 1/B^d)$.

Other input parameters of the `Setup` algorithm may include the maximal allowed probability of error, or the plaintext space size (for other than Boolean circuits).

∘ $\underline{\mathsf{TFHE.SecKeyGen}()}$: Generate secret keys for:

- LWE encryption: $\mathbf{s} \xleftarrow{\$} \mathbb{B}^n$;
- RLWE encryption: $z \xleftarrow{\$} \mathbb{B}^{(N)}[X]$, (alternatively $z_i \xleftarrow{\zeta} \{-1, 0, 1\}$ for some distribution ζ).

For LWE key $\mathbf{s} \in \mathbb{B}^n$, we denote $\bar{\mathbf{s}} \coloneqq (1, \mathbf{s}) \in \mathbb{B}^{1+n}$ the extended secret key, similarly for an RLWE key $z \in \mathbb{Z}^{(N)}[X]$, we denote $\bar{\mathbf{z}} \coloneqq (1, z) \in \mathbb{Z}^{(N)}[X]^2$.

∘ $\underline{\mathsf{TFHE.LweSymEncr}(\mu)}$: Given message $\mu \in \mathbb{T}$, sample fresh mask $\mathbf{a} \xleftarrow{\$} \mathbb{T}^n$ and noise $e \xleftarrow{\alpha} \mathbb{T}$. Evaluate $b \leftarrow -\langle \mathbf{s}, \mathbf{a} \rangle + \mu + e$ and output $\bar{\mathbf{c}} = (b, \mathbf{a}) \in \mathbb{T}^{1+n}$, an LWE encryption of μ. This algorithm is used as the main encryption algorithm of the scheme. We generalize this as well as subsequent algorithms to input vectors and proceed element-by-element.

∘ $\underline{\mathsf{TFHE.RLweSymEncr}(m, a = \emptyset, z_{in} = z)}$: Given message $m \in \mathbb{T}^{(N)}[X]$, sample fresh mask $a \xleftarrow{\$} \mathbb{T}^{(N)}[X]$, unless explicitly given. If the pair (a, z_{in}) has been used before, output $\perp$. Otherwise, sample fresh noise $e \in \mathbb{T}^{(N)}[X]$, $e_i \xleftarrow{\beta} \mathbb{T}$, and evaluate $b \leftarrow -z_{in} \cdot a + m + e$. Output $\bar{\mathbf{c}} = (b, a) \in \mathbb{T}^{(N)}[X]^2$, an RLWE encryption of m. In case a is given, we may limit the output to only b.

∘ $\underline{\mathsf{TFHE.(R)LwePhase}(\bar{\mathbf{c}})}$: Given (R)LWE sample $\bar{\mathbf{c}}$, evaluate and output $\varphi \leftarrow \langle \bar{\mathbf{s}}, \bar{\mathbf{c}} \rangle$, where $\bar{\mathbf{s}}$ is respective (R)LWE extended secret key.

∘ $\underline{\mathsf{TFHE.EncrBool}(b)}$: Set $\mu = \pm 1/8$ for b true or false, respectively. Output $\mathtt{LweSymEncr}(\mu)$.

∘ $\underline{\mathsf{TFHE.DecrBool}(\bar{\mathbf{c}})}$: Output $\mathtt{LwePhase}(\bar{\mathbf{c}}) > 0$, assuming $\mathbb{T} \sim [-1/2, 1/2)$.

∘ $\underline{\mathsf{TFHE.RgswEncr}(m)}$: Given $m \in \mathbb{Z}^{(N)}[X]$, evaluate $\mathbf{Z} \leftarrow \mathtt{RLweSymEncr}(\mathbf{0})$, where $\mathbf{0}$ is a vector of $2d$ zero polynomials (i.e., $\mathbf{Z} \in (\mathbb{T}^{(N)}[X])^{2d \times 2}$). Output $\mathbf{Z} + m \cdot \mathbf{G}$, an RGSW sample of m.

∘ $\underline{\mathsf{TFHE.Prod}\big(\mathsf{BK}, (b, a)\big)}$: Given RGSW sample BK of $s \in \mathbb{Z}^{(N)}[X]$, and RLWE sample (b, a) of $m \in \mathbb{T}^{(N)}[X]$, evaluate and output:

$$(b', a') \leftarrow \begin{pmatrix} \mathbf{g}^{-1}(b) \\ \mathbf{g}^{-1}(a) \end{pmatrix}^T \cdot \mathsf{BK} =: \mathsf{BK} \boxdot (b, a), \tag{14}$$

which is an RLWE sample of $s \cdot m \in \mathbb{T}^{(N)}[X]$; in TFHE also referred to as the *external product*.

∘ $\underline{\mathsf{TFHE.BlindRotate}\big(\bar{\mathbf{c}}, \{\mathsf{BK}_i\}_{i=1}^n, tv\big)}$: Given $\bar{\mathbf{c}} = (b, a_1, \ldots, a_n) \in \mathbb{T}^{1+n}$, an LWE sample of $\mu \in \mathbb{T}$ under key $\mathbf{s} \in \mathbb{B}^n$; $(\mathsf{BK}_i)_{i=1}^n$, RGSW samples of $\mathbf{s}_i$ under RLWE key z (aka. *blind-rotate keys*); and $\mathsf{RLWE}_z(tv) \in \mathbb{T}^{(N)}[X]^2$, (usually trivial) RLWE sample of $tv \in \mathbb{T}^{(N)}[X]$ (aka. *test vector*), evaluate:

1: $\tilde{b} \leftarrow \lfloor 2Nb \rceil, \quad \tilde{a}_i \leftarrow \lfloor 2Na_i \rceil$ for $1 \leq i \leq n$
2: $\mathsf{ACC} \leftarrow X^{\tilde{b}} \cdot \mathsf{RLWE}(tv)$
3: **for** $i = 1, \ldots, n$ **do**
4: $\quad \mathsf{ACC} \leftarrow \mathsf{ACC} + \mathtt{Prod}\big(\mathsf{BK}_i, X^{\tilde{a}_i} \cdot \mathsf{ACC} - \mathsf{ACC}\big)$ ▷ ACC or $X^{\tilde{a}_i} \cdot \mathsf{ACC}$ if $\mathbf{s}_i = 0$ or $\mathbf{s}_i = 1$, resp.

Output $\mathsf{ACC} = \mathsf{RLWE}_z(X^{\tilde{\varphi}} \cdot tv)$, an RLWE encryption of test vector "rotated" by $\tilde{\varphi}$, where $\tilde{\varphi} = \lfloor 2Nb \rceil + s_1 \lfloor 2Na_1 \rceil + \ldots + s_n \lfloor 2Na_n \rceil \approx 2N(\bar{\mathbf{s}} \cdot \bar{\mathbf{c}}) \approx 2N\mu$.

∘ $\underline{\mathsf{TFHE}.\mathtt{KeyExtract}(z)}$: Given RLWE key $z \in \mathbb{Z}^{(N)}[X]$, output $\mathbf{z}^* \leftarrow (z_0, -z_{N-1}, \ldots, -z_1)$.

∘ $\underline{\mathsf{TFHE}.\mathtt{SampleExtract}(b, a)}$: Given RLWE sample $(b, a) \in \mathbb{T}^{(N)}[X]^2$ of $m \in \mathbb{T}^{(N)}[X]$ under RLWE key $z \in \mathbb{Z}^{(N)}[X]$, output LWE sample $(b', \mathbf{a}') \leftarrow (b_0, a_0, \ldots, a_{N-1}) \in \mathbb{T}^{1+N}$ of $m_0 \in \mathbb{T}$ (the constant term of m) under the extracted LWE key $\mathbf{z}^* = \mathtt{KeyExtract}(z)$.

∘ $\underline{\mathsf{TFHE}.\mathtt{KeySwitchKeyGen}()}$: For $j \in [1, N]$, evaluate and output a key-switching key for z_j and $\mathbf{s}$: $\mathsf{KS}_j \leftarrow \mathtt{LweSymEncr}\big(\mathbf{z}_j^* \cdot \mathbf{g}'\big)$, where $\mathbf{z}^* \leftarrow \mathtt{KeyExtract}(z)$. KS_j is a d'-tuple of LWE samples of $\mathbf{g}'$-respective fractions of $\mathbf{z}_j^*$ under the key $\mathbf{s}$.

∘ $\underline{\mathsf{TFHE}.\mathtt{KeySwitch}\big(\bar{\mathbf{c}}', \{\mathsf{KS}_j\}_{j=1}^N\big)}$: Given LWE sample $\bar{\mathbf{c}}' = (b', a'_1, \ldots, a'_N) \in \mathbb{T}^{1+N}$ (extraction of an RLWE sample), which encrypts $\mu \in \mathbb{T}$ under the extraction of an RLWE key $\mathbf{z}^* = \mathtt{KeyExtract}(z)$, and a set of key-switching keys for z and $\mathbf{s}$, evaluate and output

$$\bar{\mathbf{c}}'' \leftarrow (b', \mathbf{0}) + \sum_{j=1}^{N} \mathbf{g}'^{-1}(a'_j)^T \cdot \mathsf{KS}_j, \tag{15}$$

which is an LWE sample of the same $\mu \in \mathbb{T}$ under the LWE key $\mathbf{s}$.

∘ $\underline{\mathsf{TFHE}.\mathtt{Bootstrap}\big(\bar{\mathbf{c}}, tv, \{\mathsf{BK}_i\}_{i=1}^n, \{\mathsf{KS}_j\}_{j=1}^N\big)}$: Given LWE sample $\bar{\mathbf{c}}$ of $\mu \in \mathbb{T}$ under LWE key $\mathbf{s}$, test vector $tv \in \mathbb{T}^{(N)}[X]$ that encodes a LUT, and two sets of keys for blind-rotate and for key-switching (aka. *bootstrapping keys* – the evaluation keys of TFHE), evaluate:

1: $\bar{\mathbf{c}}' \leftarrow \mathtt{BlindRotate}\big(\bar{\mathbf{c}}, \{\mathsf{BK}_i\}_{i=1}^n, tv\big)$;
2: $\bar{\mathbf{c}}'' \leftarrow \mathtt{KeySwitch}\big(\mathtt{SampleExtract}(\bar{\mathbf{c}}'), \{\mathsf{KS}_j\}_{j=1}^N\big)$.

Output $\bar{\mathbf{c}}''$, which is an LWE sample of—vaguely speaking—"evaluation of the LUT at μ", under the key $\mathbf{s}$, with a refreshed noise. Details on the encoding of the LUT are out of the scope of this paper.

∘ $\underline{\mathsf{TFHE}.\mathtt{Add}(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2)}$: Output $\bar{\mathbf{c}}_1 + \bar{\mathbf{c}}_2$, which encrypts the sum of input plaintexts. Using just "+".

∘ $\underline{\mathsf{TFHE}.\mathtt{NAND}\big(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2, \{\mathsf{BK}_i\}_{i=1}^n, \{\mathsf{KS}_j\}_{j=1}^N\big)}$: Given encryptions of bools b_1 and b_2 under LWE key $\mathbf{s}$, and bootstrapping keys for $\mathbf{s}$ and z, set the test vector as $tv \leftarrow 1/8 \cdot (1 + X + X^2 + \ldots + X^{N-1})$. Output $\bar{\mathbf{c}}'' \leftarrow \mathtt{Bootstrap}\big(1/8 - \bar{\mathbf{c}}_1 - \bar{\mathbf{c}}_2, tv, \{\mathsf{BK}_i\}_{i=1}^n, \{\mathsf{KS}_j\}_{j=1}^N\big)$, which is an encryption of $\neg(b_1 \wedge b_2)$ under the key $\mathbf{s}$.