Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping

Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler

Keywords: FHE · TFHE · functional bootstrapping

Abstract. In this work, we first propose a new functional bootstrapping with TFHE for evaluating any function of domain and codomain the real torus $\mathbb{T}$ by using a small number of bootstrappings. This result improves some aspects of previous approaches: like them, we allow for evaluating any functions, but with better precision. In addition, we develop more efficient multiplication and addition over ciphertexts building on the digit-decomposition approach of [GBA21]. As a practical application, our results lead to an efficient implementation of ReLU, one of the most used activation functions in deep learning. The paper is concluded by extensive experimental results comparing each building block as well as their practical relevance and trade-offs.

1 Introduction

Machine learning application to the analysis of private data, such as health or genomic data, has encouraged the use of homomorphic encryption for private inference or prediction with classification or regression algorithms where the ML models and/or their inputs are encrypted homomorphically [Xie+14; Cha+17; Cha+19; Bou+18; ZCS20; ISZ19a; ZS21]. Even training machine learning models with privacy guarantees on the training data has been investigated in the centralized [JA18; CKP19; Nan+19; Lou+20] and collaborative [Séb+21; Mad+21] settings. In practice, machine learning algorithms and especially neural networks require the computation of non-linear activation functions such as the sign, ReLU or sigmoid functions. Computing non-linear functions homomorphically remains challenging. For levelled homomorphic schemes such as BFV [Bra12; FV12] or CKKS [Che+17], non-linear functions have to be approximated by polynomials. However, the precision of this approximation differs with respect to the considered plaintext space (i.e., input range), approximation polynomial degree and its coefficients size, and has a direct impact on the multiplicative depth and parameters of the cryptosystem. The more precise is the approximation, the larger are the cryptosystem parameters and the slower is the computation. On the other hand, homomorphic encryption schemes implementing bootstrapping such

as TFHE [Chi+16; Chi+19] or FHEW [DM15] can be tweaked to encode functions via look-up table evaluations within their bootstrapping procedure. Hence, rather than being just used for refreshing ciphertexts (i.e., reducing their noise level), the boostrapping becomes *functional* [BST19] or *programmable* [CJP21] by allowing the evaluation of arbitrary functions as a bonus.

In this work, we further investigate the capabilities of TFHE functional bootstrapping. In 2016, the TFHE paper made a breakthrough by proposing an efficient bootstrapping for homomorphic gate computation. Then, Bourse et al., [Bou+17] and Izabachene et al., [ISZ19b] used the same bootstrapping algorithm for extracting the (encrypted) sign of an encrypted input. Boura et al., [Bou+19] showed later that TFHE bootstrapping could be extended to support a wider class of functionalities. Indeed, TFHE bootstrapping naturally allows to encode function evaluation via their representation as look-up tables (LUTs). Recently, different approaches have been investigated for functional bootstrapping improvement. In particular, Kluczniak and Schild [KS21] and Yang et al., [Yan+21] proposed two methods that take into consideration the negacyclicity of the cyclotomic polynomial used within the bootstrapping, for encoding look-up tables over the full real torus $\mathbb{T}$. Meanwhile, Guimarães et al., [GBA21] extended the ideas in Bourse et al., [BST19] to support the evaluation of certain activation functions such as the sigmoid.

Contributions – In this paper, we review, unify and extend the capabilities of TFHE functional bootstrapping. We strive to present the main existing methods as well as new variants. We compare their relative accuracy and performance as well as discuss their main pros and cons. Indeed, on top of the extensions that we present, we aim for this paper to be a complete reference for anyone looking to get a view of the state of functional bootstrapping. As such, several methods for LUTs evaluation using functional bootstrapping are presented: the usual method using one bit of padding (described clearly in [CJP21]), two methods coming from recent papers that work without padding [KS21; Yan+21], one novel approach also working without padding, and a method using digit decomposition of the inputs in order to get an arbitrary large plaintext space (presented initially by Bourse et al., [BST19] and generalized later by Guimarães et al. [GBA21]). The first method encodes the plaintext space in $[0, \frac{1}{2}[$, i.e., the segment of the real torus $\mathbb{T}$ corresponding to the positive numbers. Meanwhile, the other methods use the full torus for encoding the plaintext space and propose various solutions to cope with the negacyclicity of TFHE bootstrapping when used for evaluating LUTs. A novel way we present to achieve this is to use several bootstrappings one after the other to cancel the negacyclicity of a single bootstrapping. Finally, the decomposition method allows working with larger plaintext spaces. Its main idea is to decompose each plaintext into small digits which allows keeping TFHE parameters small enough to lead to performance improvements. We generalize the chaining method of [GBA21] in order to compute *any* function with *any* chosen precision.

Paper organization – The remainder of this paper is organized as follows. Section 2 reviews TFHE building blocks. Section 3 describes the functional bootstrapping idea coming from the TFHE gate bootstrapping. Sections 4 and 5 detail several methods, including ours, for the intricate Look-Up Tables (LUTs) encoding via the functional bootstrapping. Indeed, section 4 describes methods for LUTs evaluation when having a unique ciphertext as input. Meanwhile, section 5 considers the case where LUTs are evaluated over several ciphertexts encrypting separately the digits of a large plaintext. Finally, section 6 gives unitary results comparing these methods for LUTs evaluation over encrypted data.

2 TFHE

2.1 Notations

In the upcoming sections, we denote vectors by bold letters and so, each vector $\boldsymbol{x}$ of n elements is described as: $\boldsymbol{x} = (x_1, \ldots, x_n)$. $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ is the dot product between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. We denote matrices by capital letters, and the set of matrices with m rows and n columns with entries sampled in $\mathbb{K}$ by $\mathcal{M}_{m,n}(\mathbb{K})$. $x \xleftarrow{\$} \mathbb{K}$ denotes sampling x uniformly from $\mathbb{K}$, while $x \xleftarrow{\mathcal{N}(\mu,\sigma^2)} \mathbb{K}$ refers to sampling x from $\mathbb{K}$ following a Gaussian distribution of mean μ and variance σ^2.

We will refer to the real torus by $\mathbb{T} = \mathbb{R}/\mathbb{Z}$. $\mathbb{T}$ is the additive group of real numbers modulo 1 ($\mathbb{R}\ mod[1]$) and it is a $\mathbb{Z}$-module. That is, multiplication by scalars from $\mathbb{Z}$ is well-defined over $\mathbb{T}$. $\mathbb{T}_N[X]$ denotes the $\mathbb{Z}$-module $\mathbb{R}[X]/(X^N+1)\ mod[1]$ of torus polynomials, where N is a power of 2. $\mathcal{R}$ is the ring $\mathbb{Z}[X]/(X^N+1)$ and its subring of polynomials with binary coefficients is $\mathbb{B}_N[X] = \mathbb{B}[X]/(X^N+1)$ ($\mathbb{B} = \{0,1\}$). Finally, $[x]$ will denote the encryption of x over $\mathbb{T}$, $\mathbb{T}_N[X]$ or $\mathcal{R}$.

Given a function $f : \mathbb{T} \to \mathbb{T}$, we define $\mathrm{LUT}_N(f)$ to be Look-Up Table defined by the set of N pairs $\left(i, f\left(\frac{i}{N}\right)\right)$. We may write $\mathrm{LUT}(f)$ when the value N is implied. Given a function $f : \mathbb{T} \to \mathbb{T}$, we define a polynomial $P_{f,N} \in \mathbb{T}_N[X]$ of degree N by writing $P_{f,N} = \sum_{i=0}^{N-1} f\left(\frac{i}{2N}\right) \cdot X^i$. For simplicity sake, we may write P_f instead of $P_{f,N}$ when the value N is implied.

2.2 TFHE Structures

The TFHE encryption scheme was proposed in 2016 [Chi+16]. It improves the FHEW cryptosystem [DM15] and introduces the TLWE problem as an adaptation of the LWE problem to $\mathbb{T}$. It was updated later in [Chi+17] and both works were recently unified in [Chi+19]. The TFHE scheme is implemented as the TFHE library [Chi+]. TFHE relies on three structures to encrypt plaintexts defined over $\mathbb{T}$, $\mathbb{T}_N[X]$ or $\mathcal{R}$:

– **TLWE Sample:** $(\boldsymbol{a}, b)$ is a valid TLWE sample if $\boldsymbol{a} \xleftarrow{\$} \mathbb{T}^n$ and $b \in \mathbb{T}$ verifies $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e$, where $\boldsymbol{s} \xleftarrow{\$} \mathbb{B}^n$ is the secret key, and $e \xleftarrow{\mathcal{N}(0,\sigma^2)} \mathbb{T}$. In this case, $(\boldsymbol{a}, b)$ is a fresh encryption of 0.

– **TRLWE Sample:** a pair $(\boldsymbol{a}, b) \in \mathbb{T}_N[X]^k \times \mathbb{T}_N[X]$ is a valid TRLWE sample if $\boldsymbol{a} \xleftarrow{\$} \mathbb{T}_N[X]^k$, and $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e$, where $\boldsymbol{s} \xleftarrow{\$} \mathbb{B}_N[X]^k$ is a TRLWE secret key and $e \xleftarrow{\mathcal{N}(0,\sigma^2)} \mathbb{T}_N[X]$ is a noise polynomial. In this case, $(\boldsymbol{a}, b)$ is a fresh encryption of 0.

  The TRLWE decision problem consists of distinguishing TRLWE samples from random samples in $\mathbb{T}_N[X]^k \times \mathbb{T}_N[X]$. Meanwhile, the TRLWE search problem consists in finding the private polynomial $\boldsymbol{s}$ given arbitrarily many TRLWE samples. When $N = 1$ and k is large, the TRLWE decision and search problems become the TLWE decision and search problems, respectively.

  Let $\mathcal{M} \subset \mathbb{T}_N[X]$ (or $\mathcal{M} \subset \mathbb{T}$) be the discrete message space[1]. To encrypt a message $m \in \mathcal{M} \subset \mathbb{T}_N[X]$, we add $(\mathbf{0}, m) \in \mathbb{T}_N[X]^k \times \mathbb{T}_N[X]$ to a TRLWE sample encrypting 0 (or to a TLWE sample of 0 if $\mathcal{M} \subset \mathbb{T}$). In the following, we refer to an encryption of m with the secret key $\boldsymbol{s}$ as a T(R)LWE ciphertext noted $c \in \mathrm{T(R)LWE}_{\boldsymbol{s}}(m)$.

  To decrypt a sample $c \in \mathrm{T(R)LWE}_{\boldsymbol{s}}(m)$, we compute its *phase* $\phi(c) = b - \langle \boldsymbol{a}, \boldsymbol{s} \rangle = m + e$. Then, we round to it to the nearest element of $\mathcal{M}$. Therefore, if the error e was chosen to be small enough (and yet high enough to ensure security), the decryption will be accurate.

– **TRGSW Sample:** is a vector of l TRLWE samples encrypting 0. To encrypt a message $m \in \mathcal{R}$, we add $m.H$ to a TRGSW sample of 0, where H is a gadget matrix[2]. Chilotti et al., [Chi+19] defines an external product between a TRGSW sample A encrypting $m_a \in \mathcal{R}$ and a TRLWE sample $\boldsymbol{b}$ encrypting $m_b \in \mathbb{T}_N[X]$. This external product consists in multiplying A by the approximate decomposition of $\boldsymbol{b}$ with respect to H (Definition 3.12 in [Chi+19]). It yields an encryption of $m_a.m_b$ i.e., a TRLWE sample $c \in \mathrm{TRLWE}_{\boldsymbol{s}}(m_a.m_b)$. Otherwise, the external product allows also to compute a controlled MUX gate (CMUX) where the selector is $c_b \in \mathrm{TRGSW}_{\boldsymbol{s}}(b), b \in \{0, 1\}$, and the inputs are $c_0 \in \mathrm{TRLWE}_{\boldsymbol{s}}(m_0)$ and $c_1 \in \mathrm{TRLWE}_{\boldsymbol{s}}(m_1)$.

2.3 TFHE Bootstrapping

TFHE bootstrapping relies mainly on three building blocks:

[1] In practice, we discretise the Torus with respect to our plaintext modulus. For example, if we want to encrypt $m \in \mathbb{Z}_4 = \{0, 1, 2, 3\}$, we encode it in $\mathbb{T}$ as one of the following value $\{0, 0.25, 0.5, 0.75\}$.

[2] Refer to Definition 3.6 and Lemma 3.7 in TFHE paper [Chi+19] for more information about the gadget matrix H.

– **Blind Rotate:** rotates a plaintext polynomial encrypted as a TRLWE ciphertext by an encrypted position. It takes as inputs: a TRLWE ciphertext $c \in \text{TRLWE}_{\boldsymbol{k}}(m)$, a vector $(a_1, \ldots, a_p, a_{p+1} = b)$ where $\forall i,\ a_i \in \mathbb{Z}_{2N}$ and p TRGSW ciphertexts encrypting $(s_1, \ldots, s_p)$ where $\forall i,\ s_i \in \mathbb{B}$. It returns a TRLWE ciphertext $c' \in \text{TRLWE}_{\boldsymbol{k}}(X^{\langle \boldsymbol{a}, \boldsymbol{s} \rangle - b}.m)$. In this paper, we will refer to this algorithm by BlindRotate.

– **TLWE Sample Extract:** takes as inputs a ciphertext $c \in \text{TRLWE}_{\boldsymbol{k}}(m)$ and a position $p \in [\![0, N-1]\!]$, and returns a TLWE ciphertext $c' \in \text{TLWE}_{\boldsymbol{k}}(m_p)$ where m_p is the p^{th} coefficient of the polynomial m. In this paper, we will refer to this algorithm by SimpleExtract.

– **Public Functional Key Switching:** transforms a set of p ciphertexts $c_i \in \text{TLWE}_{\boldsymbol{k}}(m_i)$ into a ciphertext $c' \in \text{T(R)LWE}_{\boldsymbol{s}}(f(m_1, \ldots, m_p))$, where $f()$ is a public linear morphism from $\mathbb{T}^p$ to $\mathbb{T}_N[X]$. Note that functional key switching serves at changing encryption keys and parameters. In this paper, we will refer to this algorithm by KeySwitch.

TFHE comes with two bootstrapping algorithms. The first one is the gate bootstrapping. It aims at reducing the noise level of a TLWE sample that encrypts the result of a boolean gate evaluation on two ciphertexts, each of them encrypting a binary input. The binary nature of inputs/outputs of this algorithm is not due to inherent limitations of the TFHE scheme but rather to the fact that the authors of the paper were building a bitwise set of operators for which this bootstrapping operation was perfectly fitted.

TFHE gate bootstrapping steps are summarized in Algorithm 1. The step 1 consists in selecting a value $\hat{m} \in \mathbb{T}$ which will serve later for setting the coefficients of the test polynomial $testv$ (in step 3). The step 2 rescales the components of the input ciphertext $\boldsymbol{c}$ as elements of $\mathbb{Z}_{2N}$. The step 3 defines the test polynomial $testv$. Note that for all $p \in [\![0, 2N]\!]$, the constant term of $testv \cdot X^p$ is $\hat{m}$ if $p \in]\!]\frac{N}{2}, \frac{3N}{2}[\![$ and $-\hat{m}$ otherwise. The step 4 returns an accumulator $ACC \in \text{TRLWE}_{\boldsymbol{s'}}(testv.X^{\langle \bar{\boldsymbol{a}}, \boldsymbol{s} \rangle - \bar{b}})$. Indeed, the constant term of ACC is $-\hat{m}$ if $\boldsymbol{c}$ encrypts 0, or $\hat{m}$ if $\boldsymbol{c}$ encrypts 1. Then, step 5 creates a new ciphertext $\boldsymbol{c'}$ by extracting the constant term of ACC and adding to it $(\boldsymbol{0}, \hat{m})$. That is, $\boldsymbol{c'}$ either encrypts 0 if $\boldsymbol{c}$ encrypts 0, or m if $\boldsymbol{c}$ encrypts 1 (By choosing $m = \frac{1}{2}$, we get a fresh encryption of 1).

TFHE specifies a second type of bootstrapping called *circuit bootstrapping*. It converts TLWE samples into TRGSW samples, and serves mainly for TFHE use in a levelled manner.

3 TFHE Functional Bootstrapping

3.1 Encoding and Decoding

Our goal is to build an homomorphic LUT of any function $f : \mathcal{I} \to \mathcal{O}$ with varying precision and with input and output spaces $\mathcal{I}, \mathcal{O} \subset \mathbb{R}$.

Algorithm 1 TFHE gate bootstrapping [Chi+19]

Input: a constant $m \in \mathbb{T}$, a TLWE sample $\boldsymbol{c} = (\boldsymbol{a}, b) \in \mathrm{TLWE}_{\boldsymbol{s}}(x \cdot \frac{1}{2})$ with $x \in \mathbb{B}$, a bootstrapping key $BK_{\boldsymbol{s} \to \boldsymbol{s'}} = (BK_i \in \mathrm{TRGSW}_{S'}(s_i))_{i \in [\![1,n]\!]}$ where S' is the TRLWE interpretation of a secret key $\boldsymbol{s'}$

Output: a TLWE sample $\boldsymbol{c'} = (\boldsymbol{a'}, b') \in \mathrm{TLWE}_{\boldsymbol{s}}(x.m)$

1: Let $\hat{m} = \frac{1}{2}m \in \mathbb{T}$ (pick one of the two possible values)
2: Let $\bar{b} = \lfloor 2Nb \rceil$ and $\bar{a}_i = \lfloor 2Na_i \rceil \in \mathbb{Z}, \forall i \in [\![1, n]\!]$
3: Let $testv := (1 + X + \cdots + X^{N-1}) \cdot X^{\frac{N}{2}} \cdot \hat{m} \in \mathbb{T}_N[X]$
4: $ACC \leftarrow \mathsf{BlindRotate}((\mathbf{0}, testv), (\bar{a}_1, \ldots, \bar{a}_n, \bar{b}), (BK_1, \ldots, BK_n))$
5: $\boldsymbol{c'} = (\mathbf{0}, \hat{m}) + \mathsf{SampleExtract}(ACC)$
6: return $\mathsf{KeySwitch}_{\mathsf{s'} \to \mathsf{s}}(\boldsymbol{c'})$

Since we use TFHE as our homomorphic encryption scheme, every message from plaintext input or output space needs to be encoded in $\mathbb{T}$. Therefore, in order to build our function f, we need to create a torus-to-torus function $f_{\mathbb{T}}$ and appropriate encoding and decoding functions ι and ω.

$$\begin{array}{ccc} \mathcal{I} & \xrightarrow{f = \omega \circ f_{\mathbb{T}} \circ \iota} & \mathcal{O} \\ \iota \downarrow & & \uparrow \omega \\ \mathbb{T} & \underset{f_{\mathbb{T}}}{\longrightarrow} & \mathbb{T} \end{array}$$

In most cases, ι and ω are rescaling functions: a multiplication or a division by a single fixed value. In the following, we show several ways to build any Look-Up Table (LUT) evaluating function $f_{\mathbb{T}}$.

3.2 Functional Bootstrapping Idea

The original bootstrapping algorithm from [Chi+16] had already all the tools to implement a LUT of any negacyclic function[3]. In particular, TFHE is well-suited for $\frac{1}{2}$-antiperiodic function, as the plaintext space for TFHE is $\mathbb{T}$, where $[0, \frac{1}{2}[$ corresponds to positive values and $[\frac{1}{2}, 1[$ to negative ones, and the bootstrapping step 2 of the Algorithm 1 encodes elements from $\mathbb{T}$ into powers of X modulus $(X^N + 1)$. Note that $X^{\alpha+N} \equiv -X^{\alpha} mod[X^N + 1]$ and allows encoding negacyclic functions as explained in the upcoming sections.

Bourse et al., [Bou+19] were the first to use the term *functional bootstrapping* for TFHE. They describe how TFHE bootstrapping computes a sign function. In addition, they state that bootstrapping can be used to build a Rectified Linear

[3] Negacyclic functions are antiperiodic functions with period p, i.e., verifying $f(x) = -f(x + p)$. For example sine is antiperiodic with period π and of course, periodic with period 2π.

Unit (ReLU). However, they do not delve into the details of how to implement the ReLU in practice[4].

Algorithm 2 describes a sign computation with the TFHE bootstrapping. It returns μ if m is positive (i.e., $m \in [0, \frac{1}{2}[$), and $-\mu$ if m is negative.

Algorithm 2 Sign extraction with bootstrapping

Input: a constant $\mu \in \mathbb{T}$, a TLWE sample $\boldsymbol{c} = (\boldsymbol{a}, b) \in \mathrm{TLWE}_{\boldsymbol{s}}(m)$ with $m \in \mathbb{T}$, a bootstrapping key $BK_{\boldsymbol{s}\to\boldsymbol{s'}} = (BK_i \in \mathrm{TRGSW}_{S'}(s_i))_{i\in[\![1,n]\!]}$ where S' is the TRLWE interpretation of a secret key $\boldsymbol{s'}$

Output: a TLWE sample $\boldsymbol{c'} = (\boldsymbol{a'}, b') \in \mathrm{TLWE}_{\boldsymbol{s}}(\mu.sign(m))$

1: Let $\bar{b} = \lfloor 2Nb \rceil$ and $\bar{a}_i = \lfloor 2Na_i \rceil \in \mathbb{Z}, \forall i \in [\![1, n]\!]$
2: Let $testv := (1 + X + \cdots + X^{N-1}) \cdot \mu \in \mathbb{T}_N[X]$
3: $ACC \leftarrow \mathsf{BlindRotate}((\mathbf{0}, testv), (\bar{a}_1, \ldots, \bar{a}_n, \bar{b}), (BK_1, \ldots, BK_n))$
4: $\boldsymbol{c'} = \mathsf{SampleExtract}(ACC)$
5: return $\mathsf{KeySwitch}_{\mathsf{s'}\to\mathsf{s}}(\boldsymbol{c'})$

When we look at the building blocks of Algorithm 2, we notice that there is some leeway to build more complex functions just by changing the coefficients of the test polynomial $testv$.

Let $t = \sum_{i=0}^{N-1} t_i \cdot X^i$ where $t_i \in \mathbb{T}$ and $g_t(x)$ the function:

$$g_t : \begin{array}{ccc} [\![-N, N-1]\!] & \to & \mathbb{T} \\ i & \mapsto & \begin{cases} t_i & \text{if } i \in [\![0, N[\![\\ -t_{i+N} & \text{if } i \in [\![-N, 0[\![\end{cases} \end{array} \tag{1}$$

Proposition 1. *If we bootstrap a TLWE ciphertext $[x] = (\boldsymbol{a}, b)$ with the test polynomial $testv = t$, the output of the bootstrapping is $[g_t(\phi(\bar{\boldsymbol{a}}, \bar{b}))]$, where $(\bar{\boldsymbol{a}}, \bar{b})$ is the rescaled version of $(\boldsymbol{a}, b)$ in $\mathbb{Z}_{2N}$ (line 1 of Algorithm 2).*

Proof. First, we remind that for any positive integer i s.t. $0 \leq i < N$, we have:

$$testv.X^{-i} = t_i + \cdots - t_0 X^{N-i} - \cdots - t_{i-1} X^{N-1} \mod [X^N + 1] \tag{2}$$

Then, we notice that BlindRotate (line 3 of Algorithm 2) computes $testv \cdot X^{-\phi(\bar{\boldsymbol{a}}, \bar{b})}$. Therefore, we obtain the following results using equation (2):

- if $\phi(\bar{\boldsymbol{a}}, \bar{b}) \in [\![0, N[\![$, the constant term of $testv \cdot X^{-\phi(\bar{\boldsymbol{a}}, \bar{b})}$ is $t_{\phi(\bar{\boldsymbol{a}}, \bar{b})}$.
- if $\phi(\bar{\boldsymbol{a}}, \bar{b}) \in [\![-N, 0[\![$, we have:

  $testv \cdot X^{-\phi(\bar{\boldsymbol{a}}, \bar{b})} = -testv \cdot X^{-\phi(\bar{\boldsymbol{a}}, \bar{b})-N} mod [X^N + 1]$

[4] The article does only mention that the function 2×ReLU can be built from an absolute value function but does not explain how to divide by two to get the ReLU result.

with $(\phi(\bar{\boldsymbol{a}}, \bar{b}) + N) \in [\![0, N[\![$. So, the constant term of $testv \cdot X^{-\phi(\bar{\boldsymbol{a}}, \bar{b})}$ is $-t_{\phi(\bar{\boldsymbol{a}}, \bar{b})+N}$.

All that remains for the bootstrapping algorithm is extracting the previous constant term (in line 4) and keyswitching (in line 5) to get the TLWE sample $[g_t(\phi(\bar{\boldsymbol{a}}, \bar{b}))]$.

We can use the previous proposition to build a discretized function evaluation as follows. Let $h : [0, \frac{1}{2}[\to \mathbb{T}$ be any function, and g_h the well-defined function:

$$g_h : \begin{array}{ccc} [\![-N, N-1]\!] & \to & \mathbb{T} \\ x & \mapsto & \begin{cases} h(\frac{x}{2N}) & \text{if } x \in [\![0, N[\![\\ -h(\frac{x+N}{2N}) & \text{if } x \in [\![-N, 0[\![\end{cases} \end{array} \tag{3}$$

Let's call P_h the polynomial of degree N defined by: $P_h = \sum_{i=0}^{N-1} h\left(\frac{i}{2N}\right) \cdot X^i$. Now, if we apply the bootstrapping Algorithm 2 to a TLWE ciphertext $[x] = (\boldsymbol{a}, b)$ with $testv = P_h$, it outputs $[g_h(\phi(\bar{\boldsymbol{a}}, \bar{b}))]$ (by applying Proposition 1). That is, Algorithm 2 allows encoding a discretized negacyclic version of h of period $\frac{1}{2}$. By the way, it allows encoding a discretized version of any negacyclic function.

3.3 Private Functional Bootstrapping

The functional bootstrapping algorithm can be adapted to compute an encrypted negacyclic function. Indeed, given a function $f : \mathbb{T} \to \mathbb{T}$, we create $[P_f]$, a TRLWE ciphertext whose i^{th} coefficient is a TLWE ciphertext encrypting $f(\frac{i}{2N})$. Such a ciphertext can be created using the TFHE public functional key-switching operation (see Algorithm 2 of [Chi+19]) from N TLWE ciphertexts $\left[f(\frac{i}{2N})\right]$.

Let $[\mu] = (\boldsymbol{a}, b)$ be a ciphertext encrypting the message μ. Then, the Algorithm 3 outputs an encryption of $f(\frac{\phi(\bar{\boldsymbol{a}}, \bar{b})}{2N})$.

Algorithm 3 Encrypted LUT

Input: a TLWE sample $[\mu] = (\boldsymbol{a}, b) \in \text{TLWE}_{\boldsymbol{s}}(\mu)$ with $\mu \in \mathbb{T}$, a bootstrapping key $BK_{\boldsymbol{s} \to \boldsymbol{s'}} = (BK_i \in \text{TRGSW}_{S'}(s_i))_{i \in [\![1,n]\!]}$ where S' is the TRLWE interpretation of a secret key $\boldsymbol{s'}$

Output: a TLWE sample $\boldsymbol{c'} = (\boldsymbol{a'}, b') \in \text{TLWE}_{\boldsymbol{s}}(f(\frac{\phi(\bar{\boldsymbol{a}}, \bar{b})}{2N}))$

1: Let $\bar{b} = \lfloor 2Nb \rceil$ and $\bar{a}_i = \lfloor 2Na_i \rceil \in \mathbb{Z}, \forall i \in [\![1, n]\!]$
2: Let $testv := [P_f]$
3: $ACC \leftarrow \mathsf{BlindRotate}(testv, (\bar{a}_1, \ldots, \bar{a}_n, \bar{b}), (BK_1, \ldots, BK_n))$
4: $\boldsymbol{c'} = \mathsf{SampleExtract}(ACC)$
5: return $\mathsf{KeySwitch}_{\mathsf{s'} \to \mathsf{s}}(\boldsymbol{c'})$

3.4 Multi-Value Functional Bootstrapping

Carpov et al., [CIM19] introduced a nice method for evaluating k different LUTs using one bootstrapping. Indeed, they factor the test polynomial P_{f_i} associated to the function f_i into a product of two polynomials v_0 and v_i, where v_0 is a common factor to all P_{f_i}. In fact, they notice that:

$$(1 + X + \cdots + X^{N-1}) \cdot (1 - X) = 2 \mod [X^N + 1] \tag{4}$$

Let's write P_{f_i} as: $P_{f_i} = \sum_{j=0}^{N-1} \alpha_{i,j} X^j$ with $\alpha_{i,j} \in \mathbb{Z}$, we obtain using equation (4):

$$\begin{aligned} P_{f_i} &= \frac{1}{2} \cdot (1 + \cdots + X^{N-1}) \cdot (1 - X) \cdot P_{f_i} \mod [X^N + 1] \\ &= v_0 \cdot v_i \mod [X^N + 1] \end{aligned}$$

where:

$$\begin{aligned} v_0 &= \frac{1}{2} \cdot (1 + \cdots + X^{N-1}) \\ v_i &= \alpha_{i,0} - \alpha_{i,N-1} + (\alpha_{i,1} - \alpha_{i,0}) \cdot X + \cdots + (\alpha_{i,N-1} - \alpha_{i,N-2}) \cdot X^{N-1} \end{aligned} \tag{5}$$

Thanks to this factorization, we are able to compute many LUTs with one bootstrapping. Indeed, we just have to set the initial test polynomial to $testv = v_0$ during the bootstrapping. Then, after the BlindRotate, we multiply the obtained ACC by each v_i corresponding to LUT(f_i) to obtain ACC_i (for more details about multi-value bootstrapping, refer to the Algorithm 7 in the Appendix section A).

4 Look-Up-Tables over a Single Ciphertext

In section 3.2, we demonstrated that functional bootstrapping allows for the computation of LUT(h) for any negacyclic function h. In this section, we describe 4 different ways to build homomorphic LUTs using *any* function (i.e., not necessarily negacyclic ones). We present 3 solutions from the state of the art [CJP21; KS21; Yan+21] in sections 4.1, 4.2 and 4.3, and one that is novel to our work in section 4.4.

As in section 3.1, we call $f_{\mathbb{T}} : \mathbb{T} \to \mathbb{T}$ the function used to build our homomorphic LUT, and $f : \mathcal{I} \to \mathcal{O}$ its corresponding function over the actual input and output spaces.

4.1 Partial Domain Functional Bootstrapping

This method avoids the negacyclic restriction of functional bootstrapping by encrypting values from $[0, \frac{1}{2}[$ (i.e., half of the torus). Let's set the test polynomial to

be P_h, the output of the bootstrapping operation is given by Equation 3:

$$g_h : \begin{array}{ccc} [\![-N, N-1]\!] & \to & \mathbb{T} \\ x & \mapsto & \begin{cases} h(\frac{x}{2N}) & \text{if } x \in [\![0, N[\![\\ -h(\frac{x+N}{2N}) & \text{if } x \in [\![-N, 0[\![\end{cases} \end{array}$$

If we restrict g_h domain to $[\![0, N[\![$, we ensure that g_h is just a LUT based on function h (h is not necessarily negacyclic). That is, we obtain a method to evaluate a LUT in a *single* bootstrapping. However, we have to encode the plaintext space over a smaller portion of the torus $\mathbb{T}$, therefore increasing the relative noise introduced by the TFHE encryption process. The overall result will hence be less accurate.

4.2 Full Domain Functional Bootstrapping–FDFB

Kluczniak and Schild [KS21] proposed this method to evaluate encrypted LUTs of domain the whole torus $\mathbb{T}$. Let's consider a TLWE ciphertext $[m]$ encrypting the message m, and a function f of domain $\mathbb{T}$. We denote by g the function:

$$g : \begin{array}{ccc} \mathbb{T} & \to & \mathbb{T} \\ x & \mapsto & -f(x + \frac{1}{2}) \end{array}$$

We define the Heaviside function H as:

$$H : x \mapsto \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

H can be expressed using the sign function as follows: $H(x) = \frac{\text{sign}(x)+1}{2}$.

First, we compute $[H(m)]$ with only one bootstrapping (using Algorithm 2) and deduce $(1 - H)([m]) = [(1 - H)(m)]$. Then, we make a keyswitch to transform the TLWE sample $[(1-H)(m)]$ into a TRLWE sample. Finally, we define:

$$c_{\text{LUT}} = [(1 - H)(m)] \cdot (\mathbf{0}, P_g - P_f) + (\mathbf{0}, P_f)$$

$$c_{\text{LUT}} = \begin{cases} [P_f] & \text{if } \mu \geq 0 \\ [P_g] & \text{if } \mu < 0 \end{cases}$$

Note that depending on the sign of m, c_{LUT} is a TRLWE encryption of P_f or P_g, the test polynomials of f or g, respectively. Indeed, after a bootstrapping of $[m]$ using c_{LUT} as a test polynomial, we obtain $[f(m)]$. This functional bootstrapping requires 2 BlindRotate during the bootstrapping: one to compute the Heaviside function and the other to apply the encrypted LUT.

4.3 Full Domain Functional Bootstrapping–TOTA

Yan et al., [Yan+21] proposed this method to evaluate arbitrary functions over the torus using a functional bootstrapping. Let's consider a ciphertext $[m_1] = (\boldsymbol{a}, b = \boldsymbol{a}.\boldsymbol{s}+m_1+e)$. Then, by dividing each coefficient of this ciphertext by 2, we get a ciphertext $[m_2] = (\frac{\boldsymbol{a}}{2}, \frac{\boldsymbol{a}}{2}.\boldsymbol{s}+m_2+\frac{e}{2})$ where $m_2 = \frac{m_1}{2}+\frac{k}{2}$ with $k \in \{0,1\}$ and $\frac{m_1}{2} \in [0, \frac{1}{2}]$. Using the original bootstrapping algorithm, we compute $[\frac{\text{sign}(m_2)}{4}]$ an encryption of $\frac{\text{sign}(m_2)}{4} = \begin{cases} \frac{1}{4} & \text{if } k = 0 \\ -\frac{1}{4} & \text{if } k = 1 \end{cases}$. Then $[m_2] - [\frac{\text{sign}(m_2)}{4}] + (0, \frac{1}{4})$ is an encryption of $\frac{m_1}{2}$.

For any function f, let's define $f_{(2)}$ such that $f_{(2)}(x) = f(2x)$. Since $\frac{m_1}{2} \in [0, \frac{1}{2}]$, we can compute $f_{(2)}(\frac{m_1}{2})$ with a single bootstrapping using the partial domain solution from 4.1, and $f_{(2)}(\frac{m_1}{2}) = f(m_1)$.

Thus, this technique allows computing any function with only 2 bootstrappings. Keep in mind that the torus is actually discretized, so some noise and some loss of precision are introduced after dividing by 2 due to the rounding of the coefficients.

4.4 Full Domain Functional Bootstrapping with Composition

In this section, we present a novel method to compute any function using the full torus as plaintext space. In this regard, it uses the same plaintext space as solutions presented in Sections 4.2 and 4.3.

This solution can only work if computations are exact. We therefore assume that it is implemented with a parameter set large enough and an input space small enough that computing the phase of any ciphertext returns its equivalent plaintext. This means we are limited to exact computations whereas all other methods presented in this paper allow for approximate computations.

Odd functions Let's have $f_{\mathbb{T}} : \mathbb{T} \to \mathbb{T}$ be an odd function. This means that, $\forall x \in \mathbb{T},\ f_{\mathbb{T}}(-x) = -f_{\mathbb{T}}(x)$.

Let's set $h = \text{Id}$ to be the identity function. Then we can define a functional bootstrapping with an output function g_{Id} as such:

$$g_{\text{Id}} : x \mapsto \begin{cases} \frac{x}{2N} & \text{if } x \in [\![0, N[\![\\ -\frac{x+N}{2N} & \text{if } x \in [\![-N, 0[\![\end{cases}$$

Let's now set h to be the restriction of $f_{\mathbb{T}}$ over positive values $[0, \frac{1}{2}[$. Then we can define $g_{f_{\mathbb{T}^+}}$ as such:

$$g_{f_{\mathbb{T}^+}} : x \mapsto \begin{cases} f_{\mathbb{T}}\left(\frac{x}{2N}\right) & \text{if } x \in [\![0, N[\![\\ -f_{\mathbb{T}}\left(\frac{x+N}{2N}\right) & \text{if } x \in [\![-N, 0[\![\end{cases} \tag{6}$$

We now can compose $g_{f_{\mathbb{T}^+}}$ with g_{Id} (we assume the outputs of g_{Id} are rescaled up to $[\![-N, N[\![$).

$$g_{f_{\mathbb{T}^+}} \circ g_{\mathrm{Id}} : x \mapsto \begin{cases} f_{\mathbb{T}}\left(\frac{x}{2N}\right) & \text{if } x \in [\![0, N[\![\\ f_{\mathbb{T}}\left(\frac{x}{2N}\right) & \text{if } x \in]\!]-N, 0]\!] \\ f_{\mathbb{T}}(0) & \text{if } x = N \end{cases}$$

Therefore $g_{f_{\mathbb{T}^+}} \circ g_{\mathrm{Id}}$ evaluates a LUT based on $f_{\mathbb{T}}$ for the whole torus except for values around $\frac{1}{2}$ (when $x = N$). While this can be corrected for certain specific functions $f_{\mathbb{T}}$ (the identity function for instance) with specific tricks, we have yet to find a way to correct it in the general case.

Absolute value function Let's have $f_{\mathbb{T}} : \mathbb{T} \to \mathbb{T}$ be the absolute value function: $f_{\mathbb{T}}(x) = |x|$.

Then let's set $h_0 : x \mapsto x - \frac{1}{4}$. We can define a functional bootstrapping with an output function g_{h_0} as such:

$$g_{h_0} : x \mapsto \begin{cases} \frac{x}{2N} - \frac{1}{4} & \text{if } x \in [\![0, N[\![\\ -\frac{x}{2N} - \frac{1}{4} & \text{if } x \in [\![-N, 0[\![\end{cases}$$

Therefore, $g_{h_0} + \frac{1}{4}$ is a LUT based on the absolute value function over the whole torus.

Even functions We want to build an homomorphic LUT based on any even function over the whole torus. Let $f_{\mathbb{T}} : \mathbb{T} \to \mathbb{T}$ be an even function. This means that, $\forall x \in \mathbb{T},\ f_{\mathbb{T}}(-x) = f_{\mathbb{T}}(x)$. Most importantly, it means that $\forall x \in \mathbb{T}, f_{\mathbb{T}}(x) = f_{\mathbb{T}}(|x|)$. We can now compose two LUTs: the absolute value LUT built in Section 4.4 ($g_{h_0} + \frac{1}{4}$); and the LUT based on the restriction of $f_{\mathbb{T}}$ over positive values described in Equation 6 ($g_{f_{\mathbb{T}^+}}$). We have:

$$g_{f_{\mathbb{T}^+}} \circ \left(g_{h_0} + \frac{1}{4}\right) : x \mapsto \begin{cases} f_{\mathbb{T}}\left(\frac{x}{2N}\right) & \text{if } x \in [\![0, N[\![\\ f_{\mathbb{T}}\left(\frac{x}{2N}\right) & \text{if } x \in]\!]-N, 0]\!] \\ f_{\mathbb{T}}(0) & \text{if } x = N \end{cases}$$

As is the case with odd functions, this implementation of a LUT over any even function cannot compute a correct result for inputs around $\frac{1}{2}$.

Any function Any function $f_{\mathbb{T}}$ can be written as a sum of an even function and an odd function: $f_{\mathbb{T}}(x) = \frac{f_{\mathbb{T}}(x)+f_{\mathbb{T}}(-x)}{2} + \frac{f_{\mathbb{T}}(x)-f_{\mathbb{T}}(-x)}{2}$. Sections 4.4 and 4.4 showed we can build an homomorphic LUT based on any odd or even function with at most 2 functional bootstrapping operations. This means that we can build one over any kind of function with at most 4 functional bootstrapping operations. There are a host of useful functions however (sigmoid, monomial

functions, trigonometric functions, identity, ..) which can be computed using only 2 bootstrapping operations because they are one sum away from an odd or even function.

No approximate arithmetic allowed. As stated, this solution is only suitable for precise arithmetic. This means that the size of the input space and the the size of the parameters have to be chosen so that every functional bootstrapping operation is certain to return the correct result. If it were not to be so, input plaintext values around 0 can be computed correctly by the first bootstrapping operation but incorrectly by the second. This would mean a result translated exactly by $\frac{1}{2}$ from the correct result.

5 Look-Up-Tables over Multiple Ciphertexts

In section 4, we discussed several functional bootstrapping methods that take as input one ciphertext. These methods have a limited plaintext space and precision, and allow evaluating look-up tables with a size bounded by the degree of the used cyclotomic polynomial (N). In addition, these methods are not suited for computing a LUT for a multivariate function f that takes as inputs two or more ciphertexts. In order to overcome these issues, we describe in this section a method for computing functions using multiple ciphertexts as inputs.

Our proposed solution improves the results of Guimarães et al., [GBA21]. They, themselves, generalize the ideas of Boura et al. [BST19] and discuss two methods for homomorphic computation with digits: a tree-based approach and a chaining approach. We expand on the chaining method in order to obtain any function through its use as opposed to the subset of function previously allowed.

Subsequently, we use this method to apply a LUT to a *single* message decomposed over *multiple* ciphertexts. That is, we decompose each plaintext into several digits in a certain base B and encrypt these digits separately. Decomposition allows working with a larger plaintext space $\mathcal{I}$ while using an acceptable parameters set for an efficient computation.

In this section, we first *review* the tree-based method and then *improve* the chaining method to make it fit any function. We show how those methods can be used as building blocks in order to compute additions and multiplications of messages decomposed over multiple ciphertexts. We then show how to compute the ReLU function over a single, decomposed, plaintext. The choice of ReLU as a worthy application of our novel method was made because it is the most used activation function in modern convolutional neural networks.

5.1 Tree-based Method

We consider n TLWE ciphertexts $c_0, \ldots, c_{n-1}$ encrypting the messages $m_0, \ldots, m_{n-1}$ over half of the torus and $B \in \mathbb{N}$, such that each ciphertext c_i corresponds to

an encryption of $m_i \in [\![0, B-1]\!]$. We denote by $f : [\![0, B-1]\!]^n \to [\![0, B-1]\!]$ our target function and by g the bijection:

$$g : \begin{array}{l} [\![0, B-1]\!]^n \to [\![0, B^n-1]\!] \\ (a_0, ..., a_{n-1}) \mapsto \sum_{i=0}^{n-1} a_i.B^i \end{array}$$

We encode the LUT for f in B^{n-1} TRLWE ciphertexts. Each ciphertext encrypts a polynomial P_i where:

$$P_i(X) = \sum_{j=0}^{B-1} \sum_{k=0}^{\frac{N}{B}-1} f \circ g^{-1}(j.B^{n-1} + i) X^{j.\frac{N}{B}+k}$$

Then, we apply the BlindRotate algorithm to c_{n-1} and each $\mathsf{TRLWE}(P_i)$, and use the SampleExtract algorithm to extract the first coefficient of the result. We end up with B^{n-1} TLWE ciphertexts each encrypting a message $f \circ g^{-1}(m_{n-1}.B^{n-1} + i)$ for $i \in [\![0, B^{n-1}-1]\!]$. Thanks to TLWE to TRLWE keyswitching, we batch them into B^{n-2} TRLWE ciphertexts corresponding to the LUT of h where:

$$h : \begin{array}{l} [\![0, B-1]\!]^{n-1} \to [\![0, B-1]\!] \\ (a_0, ..., a_{n-2}) \mapsto f(a_0, ..., a_{n-2}, m_{n-1}) \end{array}$$

We iterate this operation until getting only one TLWE ciphertext encrypting $f(m_0, ..., m_{n-1})$. Since a function from $[\![0, B-1]\!]^n$ to $[\![0, B-1]\!]^k$ can be decomposed in k functions from $[\![0, B-1]\!]^n$ to $[\![0, B-1]\!]$, we can actually build any function between any inputs, once they are decomposed in base B then encrypted.

Note that the BlindRotate algorithm is costly and we have to recall it $\sum_{i=0}^{n-1} B^i = \frac{B^n-1}{B-1}$ times. Fortunately, we can make it faster by encoding the first LUTs in plaintext polynomials rather than TRLWE ciphertexts. Then, we use the multi-value bootstrapping given in [CIM19] to compute only one bootstrapping instead of B^{n-1} in the first step of the algorithm. Thus we end-up by running $1 + \sum_{i=0}^{n-2} B^i = 1 + \frac{B^{n-1}-1}{B-1}$ BlindRotate.

The order in which the blind rotations are performed can be represented by a tree of depth $n-1$. Even if we ignore the noise added by the keyswitching operations, we end up with a noise about n times bigger than a simple bootstrapping.

5.2 Chaining Method

The chaining method has a much lower complexity and lower error growth than the tree-based method but, as presented in [GBA21], works only for a more restricted set of functions.

Let's consider n TLWE ciphertexts $c_0, \ldots, c_{n-1}$ encrypting the messages $m_0, \ldots, m_{n-1}$ and denote by $LC(a, b)$ *any* linear combination of a and b. Given some functions $(f_i)_{i \in [\![0, n-1]\!]}$ so that $f_i : [\![0, B-1]\!] \to [\![0, B-1]\!]$, we can build a function

$f : [\![0, B-1]\!]^n \to [\![0, B-1]\!]$ following Algorithm 4. Each f_i can be implemented in the homomorphic domain using the partial-domain functional bootstrapping method described in Section 4.1. The result of this algorithm has the same noise as a simple functional bootstrapping, thus much less than the noise output of the tree method.

Algorithm 4 Chaining method

Input: A vector $(c_0, \ldots, c_{n-1})$ of TLWE ciphertexts encrypting the vector of messages $(m_0, \ldots, m_{n-1})$.

Output: A ciphertext encrypting $f(m_0, \ldots, m_{n-1})$. f is defined here by the linear combinations chosen at every step and the different single-input functions f_i.

$\overline{c_0} \leftarrow f_0(c_0)$
for $i \in [\![0, n-2]\!]$ **do**
  $\overline{c_{i+1}} \leftarrow f_{i+1}(LC(\overline{c_i}, c_{i+1}))$
return $\overline{c_{n-1}}$

Most functions cannot be computed in such a simplistic way, which greatly restricts its use even though it can be effective for functions with carry-like logic as stated in [GBA21].

Generalization. It is possible to build any function f using a similar method. We introduce the function g as such:

$$g : \begin{array}{ccc} [\![0, B-1]\!]^2 & \to & [\![0, B^2-1]\!] \\ (a_0, a_1) & \mapsto & a_0 + a_1.B \end{array}$$

That function is a bijection, which means that if a ciphertext can hold any message in $[\![0, B^2-1]\!]$, then we can compute any function of two ciphertexts c_1 and c_2 by applying one functional bootstrapping over $g(c_1, c_2)$.

Note that when using base 2, we can easily build any logic door with this method. We can then build a circuit with those doors to build any functions. The same idea works for any base B.

That generalization comes at the cost of multiple bits of padding and the conception of the proper circuit.

5.3 Addition

We expect additions of two messages to be computed in linear time with respect to the number of digits of each message. Thus the tree-based method is ill-suited for this operation, since the tree-based method computing time grows exponentially with the number of digits used as inputs. Meanwhile, the chaining method is not exactly adapted to this operation if applied directly. Nonetheless, we show that we can still use any of the two methods to compute the addition effectively.

Let $m_1 = \sum_{i=0}^{n} m_{1,i}.B^i$ and $m_2 = \sum_{i=0}^{n} m_{2,i}.B^i$ be two messages expressed in base B. For each pair (i, j), let $c_{i,j}$ be the ciphertext encrypting the message $m_{i,j}$. We define $c_i = (c_{i,0}, \ldots, c_{i,n})$ as the vector of ciphertexts encrypting m_i in base B. Finally, we denote by h the half adder function, and by f the full adder one:

$$h : \begin{array}{ccc} [\![0, B-1]\!]^2 & \to & [\![0, B-1]\!]^2 \\ (a, b) & \mapsto & ((a+b)[B], \lfloor (a+b)/B \rfloor) \end{array}$$

$$f : \begin{array}{ccc} [\![0, B-1]\!]^2 \times \{0, 1\} & \to & [\![0, B-1]\!]^2 \\ (a, b, c) & \mapsto & ((a+b+c)[B], \lfloor (a+b+c)/B \rfloor) \end{array}$$

These two functions are the only requirements to build the addition operation. But, in order to be able to create those two adders, we need to create the following sub-functions:

$$mod : \begin{array}{ccc} [\![0, 2B-1]\!] & \to & [\![0, B-1]\!] \\ x & \mapsto & x[B] \end{array}$$

$$carry : \begin{array}{ccc} [\![0, 2B-1]\!] & \to & \{0, 1\} \\ x & \mapsto & \lfloor x/B \rfloor \end{array}$$

Algorithm 5 Addition

Input: Two vectors of ciphertexts $c_1 = (c_{1,i})_{i \in [\![0,n-1]\!]}$ and $c_2 = (c_{2,i})_{i \in [\![0,m-1]\!]}$ encrypting two messages m_1 and m_2 written in base B. We suppose here that $n \geq m$.
Output: An encryption of $m_1 + m_2$ in base B.

$(\overline{c_{1,0}}, \overline{c_{2,0}}) \leftarrow h(c_{1,0}, c_{2,0})$
for $i \in [\![0, m-2]\!]$ **do**
  $(\overline{c_{1,i+1}}, \overline{c_{2,i+1}}) \leftarrow f(c_{1,i+1}, c_{2,i+1}, \overline{c_{2,i}})$
for $i \in [\![m-1, n-2]\!]$ **do**
  $(\overline{c_{1,i+1}}, \overline{c_{2,i+1}}) \leftarrow h(c_{1,i+1}, \overline{c_{2,i}})$
return $(\overline{c_{1,0}}, ..., \overline{c_{1,n-1}}, \overline{c_{2,n-1}})$

We can use either the tree-based method or the chaining method to compute *mod* or *carry* functions. The chaining method needs one bit of padding to work, while the tree-based method is slower, especially for the full adder which is a three inputs function. Finally, we present Algorithm 5 for computing addition between two vectors of ciphertexts.

The time complexity of Algorithm 5 is linear with respect to the number of digits of the entries. The noise of each output ciphertext is the same as the noise of a simple bootstrapping if we use the chaining method for computing the sub-functions *mod* and *carry*. Meanwhile, with the tree-based method, we end-up with the noise of a simple bootstrapping followed by two BlindRotate.

5.4 Multiplication

As we expected linear computation time to be achievable for the homomorphic addition, we expect to achieve quadratic time complexity for homomorphic multiplication. Let m_1 and m_2 be two messages and $c_1 = (c_{1,i})_{i\in[\![0,n-1]\!]}$ and $c_2 = (c_{2,i})_{i\in[\![0,m-1]\!]}$ be their encryption in base B. In order to evaluate $m_1.m_2$ in the encrypted domain, we first multiply each digit of m_1 by each digit of m_2. Then, we have just to add the obtained elements properly using half and full adders to get the final result.

Since we have already introduced homomorphic adders, we only need to describe how to multiply two digits. Given two messages a and b in $[\![0, B-1]\!]$, we need to compute $a.b[B]$ and $a.b/B$ in the encrypted domain. If we use the tree-base method, we can compute both functions with three LUTs since both functions will use the same selector in the first step. Otherwise, we can also use the generalized chaining method to compute both needed functions using two LUTs, but this method comes at the cost of using multiple bits of padding.

Let's denote by $\mathsf{MultDigits}(c_a, c_b)$ a method for computing $a.b[B]$ and by $\mathsf{CarryMult}(c_a, c_b)$ a method for computing $a.b/B$. Then the multiplication of m_1 and m_2 can be done with Algorithm 6.

Algorithm 6 Multiplication

Input: Two vectors of ciphertexts $c_1 = (c_{1,i})_{i\in[\![0,n-1]\!]}$ and $c_2 = (c_{2,i})_{i\in[\![0,m-1]\!]}$ encrypting two messages m_1 and m_2 written in base B.
Output: An encryption $\overline{c} = (\overline{c_i})_{i\in[\![0,n+m-1]\!]}$ of $m_1.m_2$ in base B.

```
for i ∈ [[0, n + m − 1]] do
    SubMul_i ← empty vector
for i ∈ [[0, n − 1]] do
    for j ∈ [[0, m − 1]] do
        Put MultDigits(c_{1,i}, c_{2,j}) in vector SubMul_{i+j}
        Put CarryMult(c_{1,i}, c_{2,j}) in vector SubMul_{i+j+1}
c̄_0 ← SubMul_0[0]
for i ∈ [[1, n + m − 1]] do
    c̄_i ← (Σ_{j=0}^{size(SubMul_i)−1} SubMul_i[j])[B] using adders
    Put the carries in SubMul_{i+1}
return (c̄_0, ..., c̄_{n+m−1})
```

The time complexity of Algorithm 6 is quadratic with respect to the number of digits of the entries. The noise of the outputs is similar to the noise of the adder sub-functions.

5.5 ReLU

In this section, we describe how to avoid using the tree-based method, as it is, for the implementation of the ReLU activation function. Let's consider $\mu =$

$\sum_{i=0}^{n} \mu_i.B^i$ a message written using radix complement representation in base B, and $(c_i)_{i\in[\![0,n]\!]} = (\mathrm{TLWE}_{\boldsymbol{s}}(\mu_i))_{i\in[\![0,n]\!]}$.

In order to use the tree-based method to evaluate intermediate functions on each encrypted digit, we use a functional bootstrapping to create a selector S from c_n that encrypts the torus element 0 if $0 \leq \mu_n < \frac{B}{2}$ and $\frac{1}{4}$ if $\frac{B}{2} \leq \mu_n < B$. Note that $(0 \leq \mu_n < \frac{B}{2}) \iff (\mu \geq 0)$, so the value of S depends on the sign of μ. Then, for each c_i, we create using keyswitching a TRLWE ciphertext $\mathrm{LUT}(c_i)$ so that for $j \in [\![0, \frac{N}{2} - 1]\!]$, $\mathsf{SampleExtract}(\mathrm{LUT}(c_i), j)$ is an encryption of μ_i, and for $j \in [\![\frac{N}{2}, N-1]\!]$, $\mathsf{SampleExtract}(\mathrm{LUT}(c_i), j)$ is an encryption of 0. Then, $\mathsf{SampleExtract}(\mathsf{BlindRotate}(S, \mathrm{LUT}(c_i)), 0)$ outputs:

$$\overline{c_i} = \begin{cases} \mathrm{TLWE}(0, s) & \text{if } \mu < 0 \\ \mathrm{TLWE}(\mu_i, s) & \text{if } \mu \geq 0 \end{cases}$$

Thus, $(\overline{c_i})_{i\in[\![0,n]\!]}$ is an encryption of $\mathsf{ReLU}(\mu)$ using radix complement representation in base B.

Otherwise, we can compute the ReLU function using the chaining method. In this case, each ciphertext has to encrypt a value in $[\![0, 2B-1]\!]$. First, let's compute a selector S from c_n such that:

$$S = \begin{cases} \mathrm{TLWE}(0, s) & \text{if } \mu \geq 0 \\ \mathrm{TLWE}(B, s) & \text{if } \mu < 0 \end{cases}$$

Then, let's define:

$$f : \begin{array}{ccc} [\![0, 2B-1]\!] & \to & [\![0, 2B-1]\!] \\ x & \mapsto & \begin{cases} x & \text{if } x < B \\ 0 & \text{if } x \geq B \end{cases} \end{array}$$

This function can be computed with one functional bootstrapping. For each c_i, we compute $\overline{c_i} = f(c_i + S)$. We obtain $(\overline{c_i})_{i\in[\![0,n]\!]}$ an encryption using radix complement representation in base B of $\mathsf{ReLU}(\mu)$.

6 Experimental Results

In this section, we compare unitary time and accuracy performances for all of the functional bootstrapping variations presented above. For this, we choose a precise set of parameters given in Table 1. These parameters allow us to have a security parameter of $\lambda = 120$ according to the latest iteration of the LWE estimator[5] [APS15; Pla18]. Note several things however about this security estimate. First, these parameters will probably need to change in time for the security to remain stable. Second, we estimate the security of our scheme using a standard, non-quantum cost model as defined by the developers of the TFHE library[6].

[5] `https://bitbucket.org/malb/lwe-estimator/` using commit a2a6e84

[6] `https://tfhe.github.io/tfhe/security_and_params.html`

In Table 1, N and α are used both for the initial encryption of the plaintext and for the encryption of the key when creating the different bootstrapping keys needed. In practice, any of those encryptions can be done with a different set of parameters if needed. Parameters B_g and l determine the precision of a bootstrapping operation and necessarily impact its running time. We choose them here so that we get a good accuracy at the expense of a slower running time than could be obtained otherwise.

Table 1. Parameters used for accuracy evaluations on all of the functional bootstrapping methods.

λ	N	α	l	B_g
120	1024	10^{-8}	13	2^2

Accuracy. Although most of the functional bootstrapping methods described in Sections 4 and 5 allow for approximate arithmetic (with the notable exception of the *composition solution* in Section 4.4), we choose to evaluate their precision using precise arithmetic. We also choose to evaluate the precision of the implementation of the ReLU function. Both of these choices are made for simplicity sake but our results can be generalized to approximate arithmetic: a method which is shown to be more accurate here will be so as well when using approximate arithmetic; the same is true for another function. The protocol for our accuracy evaluation is the following.

Start with a plaintext space size of $|\mathcal{I}| = 1$. Create encryption and evaluation keys. Create the plaintext space and encode it into the torus. Make $\frac{10000}{|\mathcal{I}|}$ encryptions of each value in $\mathcal{I}$ for a total of 10000 encryptions. Run all of the ciphertexts through an identity functional bootstrapping operation first. Indeed, we don't want to evaluate accuracy on fresh ciphertexts since most operations in real-world scenarios will not happen on fresh ciphertexts. This means accuracy will be lower than that presented in most other papers, but will more closely match the accuracy seen in real world scenarios. Then, run them all through a ReLU operation. Decipher and check the results. If all results are correct, increment the plaintext space size and repeat. Otherwise return the previous plaintext space size.

In the end, we return the biggest plaintext space size for which the operation returns correct results for the 10000 encryptions. Results are presented in Table 6.

This method measures the accuracy of all of the different methods in the case of their use for precise arithmetic. It is reasonable to expect that the most precise method for exact arithmetic will be the most precise for approximate arithmetic if we exclude the composition method which is only suited for precise operations. This however deserves more experimentation.

Section	4.1	4.2	4.3	4.4	5.5
Article	[CJP21]	[KS21]	[Yan+21]	us	us
$\lvert\mathcal{I}\rvert_{\max}$	23	22	14	27	2^{4n}
# of BlindRotate	1	2	2	3	$n+1$

Table 2. In this table we present the accuracy and time performance of all of the methods presented above. The accuracy is measured with the same parameters and is defined by the largest plaintext space size that allows for precise error-less arithmetic over a set of 10000 tests. The corresponding article is given for reference, though in the case of [CJP21], the article is not the one that introduced the method (as it was never formally introduced) but rather the one that formalized it best thus far.

Time performance. The *Blind Rotate* operation is by very far the most expensive of all those used in the functional bootstrapping algorithms. Extractions and polynomial multiplications are essentially free operations compared with the very heavy Blind Rotation. Therefore we will not talk in terms of precise time here but rather in terms of number of Blind Rotations to evaluate the time performance of each method. This is because depending on the parameters chosen, the time for a given Blind Rotation can change drastically. Know however that given our parameters, a single Blind Rotation takes approximately $0.19s$. The partial domain (or half torus) solution presented in Section 4.1 only takes 1 Blind Rotation for any operation. FDFB and TOTA, presented respectively in Sections 4.2 and 4.3 take 2 Blind Rotations and therefore twice the time. The composition solution we introduce in Section 4.4 has a variable number of Blind Rotations depending on the function: for instance, 1 for an absolute value, 2 for an identity, 3 for the ReLU (the one we evaluate here), and 4 for the more complex functions.

7 Conclusion

Through the use of several bootstrapping operations and - in some cases - additional operations, every full domain method (Sections 4.2, 4.3 and 4.4) adds some output noise when compared to the simpler and quicker partial domain method (Section 4.1). The question is: does a larger initial plaintext space make up for the added noise ? Table 6 shows us that the Yan et al., [Yan+21] (TOTA) method is both less accurate and twice as time-consuming than the partial domain method. Kluczniak and Schild [KS21] (FDFB) method, though closer in accuracy - is less accurate than the partial domain method and still twice as time-consuming. Our novel composition method (Section 4.4) is more accurate than any of the previously mentioned methods, however thrice as time consuming as the partial-domain method. As for our digit-decomposition method (Section 5), it allows for an arbitrary precision, though with a corresponding running time always much higher than the partial domain solution.

Given these experimental measures, our recommendations on the use of these functional bootstrapping methods are the following, given specific applicative scenarios:

- **Precise integer arithmetic above all else.** In some real-world applicative cases, precision is the only criteria that matters. In the case of offline computations for instance that can take hours, days or weeks, one wants to increase the precision of the result as much as possible. For this applicative scenario, **our generalized digit-decomposition functional bootstrapping method** is the appropriate choice. It is the *only* method with unbounded precision for functional bootstrapping computation of *any* function in the literature.
- **Efficient approximate or precise integer arithmetic.** In the case where we need either an approximate or a precise arithmetic computation in a limited amount of time, the **partial domain method** is an obvious choice. Its precision difference with other "no-decomposition" methods is too small to ever justify their use in this case.
- **Efficient precise modular arithmetic.** There is a case where one wishes to use modular arithmetic instead of integer arithmetic. In this case, the partial domain method cannot be used as plaintexts are encoded on only half of the torus which is not an additive group. In this case one of the full domain methods must be used. If the computation must be precise then **our novel composition method** is the most precise among the options.
- **Efficient approximate modular arithmetic.** In the case where the arithmetic is modular but the computation can be approximate, the composition method cannot be used as it only works with precise arithmetic. Therefore the preferred option becomes **FDFB** [KS21].

Furthermore, the operators presented in this paper provide key building blocks for enabling advanced deep learning functions over encrypted data.

A Multi-value Bootstrapping Algorithm

We remind that any test polynomial for a LUT(f_i) can be factorized as:

$$\begin{aligned}\text{LUT}(f_i) &= \sum_{i=0}^{N-1} \alpha_i X^i = v_0 \cdot v_i \ mod[X^N + 1] \\ v_0 &= \frac{1}{2}.(1 + \cdots + X^{N-1}) \\ v_i &= \alpha_0 - \alpha_{N-1} + (\alpha_1 - \alpha_0).X + \cdots + (\alpha_{N-1} - \alpha_0).X^{N-1}\end{aligned}$$

Algorithm 7 Multi-value bootstrapping

Input: a TLWE sample $\boldsymbol{c} = (\boldsymbol{a}, b) \in \text{TLWE}_{\boldsymbol{s}}(m)$ with $m \in \mathbb{T}$, a bootstrapping key $BK_{\boldsymbol{s}\to\boldsymbol{s'}} = (BK_i \in \text{TRGSW}_{S'}(s_i))_{i\in[\![1,n]\!]}$ where S' is the TRLWE interpretation of a secret key $\boldsymbol{s'}$, k LUTs s.t. $\text{LUT}(f_i) = v_0.v_i, \forall i \in [\![1,k]\!]$

Output: a list of k TLWE samples $\boldsymbol{c'_i} = (\boldsymbol{a'_i}, b'_i) \in \text{TLWE}_{\boldsymbol{s}}(f_i(\frac{\phi(\bar{\boldsymbol{a}},\bar{b})}{2N}))$

1: Let $\bar{b} = \lfloor 2Nb \rceil$ and $\bar{a}_i = \lfloor 2Na_i \rceil \in \mathbb{Z}, \forall i \in [\![1,n]\!]$
2: Let $testv := v_0$
3: ACC $\leftarrow$ $\mathsf{BlindRotate}((\mathbf{0}, testv), (\bar{a}_1, \ldots, \bar{a}_n, \bar{b}), (BK_1, \ldots, BK_n))$
4: **for** $i \leftarrow 1$ to k **do**
5: $\quad \text{ACC}_i := \text{ACC} \cdot v_i$
6: $\quad \boldsymbol{c'_i} = \mathsf{SampleExtract}(\text{ACC}_i)$
7: $\quad$ return $\mathsf{KeySwitch}_{\mathsf{s'}\to\mathsf{s}}(\boldsymbol{c'_i})$

References

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. *On the concrete hardness of Learning with Errors*. Cryptology ePrint Archive, Report 2015/046. 2015.

[Bou+19] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. "Simulating Homomorphic Evaluation of Deep Learning Predictions". In: *Cyber Security Cryptography and Machine Learning*. Ed. by Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung. Cham: Springer International Publishing, 2019, pp. 212–230.

[Bou+18] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. "Fast Homomorphic Evaluation of Deep Discretized Neural Networks". In: *Proceedings of CRYPTO 2018*. Springer, 2018.

[Bou+17] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. *Fast Homomorphic Evaluation of Deep Discretized Neural Networks*. Cryptology ePrint Archive, Report 2017/1114. `https://ia.cr/2017/1114`. 2017.

[BST19] Florian Bourse, Olivier Sanders, and Jacques Traoré. *Improved Secure Integer Comparison via Homomorphic Encryption*. Cryptology ePrint Archive, Report 2019/427. `https://ia.cr/2019/427`. 2019.

[Bra12] Zvika Brakerski. "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 868–886. ISBN: 978-3-642-32009-5.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. "New Techniques for Multi-value Input Homomorphic Evaluation and Applications". In: *Topics in Cryptology – CT-RSA 2019*. Ed. by Mitsuru Matsui. Cham: Springer International Publishing, 2019, pp. 106–126. ISBN: 978-3-030-12612-4.

[Cha+19] Herve Chabanne, Roch Lescuyer, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. "Recognition Over Encrypted Faces: 4th International Conference, MSPN 2018, Paris, France". In: 2019.

[Cha+17] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. *Privacy-Preserving Classification on Deep Neural Network*. Cryptology ePrint Archive, Report 2017/035. 2017.

[Che+17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: (2017). Ed. by Tsuyoshi Takagi and Thomas Peyrin.

[CKP19] Jung Hee Cheon, Duhyeong Kim, and Jai Hyun Park. "Towards a Practical Clustering Analysis over Encrypted Data". In: *IACR Cryptology ePrint Archive* (2019).

[Chi+16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds". In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 3–33. ISBN: 978-3-662-53887-6.

[Chi+17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE". In: *ASIACRYPT*. 2017.

[Chi+] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. *TFHE: Fast Fully Homomorphic Encryption Library*. URL: `https://tfhe.github.io/tfhe/`.

[Chi+19] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "TFHE: Fast Fully Homomorphic Encryption Over the Torus". In: *Journal of Cryptology* 33 (Apr. 2019). DOI: `10.1007/s00145-019-09319-x`.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. "Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks". In: *Cyber Security Cryptography and Machine Learning*. Ed. by Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann. Cham: Springer International Publishing, 2021, pp. 1–19. ISBN: 978-3-030-78086-9.

[DM15] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second". In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 617–640. ISBN: 978-3-662-46800-5.

[FV12] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2012/144. https://ia.cr/2012/144. 2012.

[GBA21] Antonio Guimarães, Edson Borin, and Diego F. Aranha. "Revisiting the functional bootstrap in TFHE". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.2 (Feb. 2021), pp. 229–253. DOI: 10.46586/tches.v2021.i2.229-253. URL: https://tches.iacr.org/index.php/TCHES/article/view/8793.

[ISZ19a] M. Izabachène, R. Sirdey, and M. Zuber. "Practical Fully Homomorphic Encryption for Fully Masked Neural Networks". In: *Cryptology and Network Security - 18th International Conference, CANS 2019, Proceedings*. Vol. 11829. Lecture Notes in Computer Science. Springer, 2019, pp. 24–36.

[ISZ19b] Malika Izabachène, Renaud Sirdey, and Martin Zuber. "Practical Fully Homomorphic Encryption for Fully Masked Neural Networks". In: *Cryptology and Network Security*. Ed. by Yi Mu, Robert H. Deng, and Xinyi Huang. Cham: Springer International Publishing, 2019, pp. 24–36. ISBN: 978-3-030-31578-8.

[JA18] Angela Jäschke and Frederik Armknecht. "Unsupervised Machine Learning on Encrypted Data". In: *IACR Cryptology ePrint Archive* (2018).

[KS21] Kamil Kluczniak and Leonard Schild. *FDFB: Full Domain Functional Bootstrapping Towards Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2021/1135. https://ia.cr/2021/1135. 2021.

[Lou+20] Qian Lou, Bo Feng, Geoffrey Charles Fox, and Lei Jiang. "Glyph: Fast and Accurately Training Deep Neural Networks on Encrypted Data". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 9193–9202. URL: https://proceedings.neurips.cc/paper/2020/file/685ac8cadc1be5ac98da9556bc1c8d9e-Paper.pdf.

[Mad+21] Abbass Madi, Oana Stan, Aurélien Mayoue, Arnaud Grivet-Sébert, Cédric Gouy-Pailler, and Renaud Sirdey. "A Secure Federated Learning framework using Homomorphic Encryption and Verifiable Computing". In: *2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*. 2021, pp. 1–8. DOI: 10.1109/RDAAPS48126.2021.9452005.

[Nan+19] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. "Towards Deep Neural Network Training on Encrypted Data". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2019, pp. 40–48. DOI: 10.1109/CVPRW.2019.00011.

[Pla18] Rachel Player. "Parameter selection in lattice-based cryptography". PhD thesis. University of London, Oct. 2018.

[Séb+21] Arnaud Grivet Sébert, Rafael Pinot, Martin Zuber, Cédric Gouy-Pailler, and Renaud Sirdey. "SPEED: secure, PrivatE, and efficient deep learning". In: *Mach. Learn.* 110.4 (2021), pp. 675–694. DOI: 10.1007/s10994-021-05970-3. URL: https://doi.org/10.1007/s10994-021-05970-3.

[Xie+14] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin E. Lauter, and Michael Naehrig. "Crypto-Nets: Neural Networks over Encrypted Data". In: *CoRR* (2014).

[Yan+21] Zhaomin Yang, Xiang Xie, Huajie Shen, Shiying Chen, and Jun Zhou. *TOTA: Fully Homomorphic Encryption with Smaller Parameters and Stronger Security*. Cryptology ePrint Archive, Report 2021/1347. https://ia.cr/2021/1347. 2021.

[ZCS20] Martin Zuber, Sergiu Carpov, and Renaud Sirdey. "Towards real-time hidden speaker recognition by means of fully homomorphic encryption". In: *International Conference on Information and Communications Security*. Springer. 2020, pp. 403–421.

[ZS21] Martin Zuber and Renaud Sirdey. "Efficient homomorphic evaluation of k-NN classifiers". In: *Proc. Priv. Enhancing Technol.* 2021.2 (2021), pp. 111–129. DOI: 10.2478/popets-2021-0020. URL: https://doi.org/10.2478/popets-2021-0020.