IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 2, pp. 229–253. DOI:10.46586/tches.v2021.i2.229-253

Revisiting the functional bootstrap in TFHE

Antonio Guimarães[1], Edson Borin[1] and Diego F. Aranha[2]

[1] Institute of Computing, University of Campinas, Brazil
{antonio.guimaraes,edson}@ic.unicamp.br
[2] Department of Computer Science, Aarhus University, Denmark
dfaranha@cs.au.dk

Abstract. The FHEW cryptosystem introduced the idea that an arbitrary function can be evaluated within the bootstrap procedure as a table lookup. The faster bootstraps of TFHE strengthened this approach, which was later named *Functional Bootstrap* (Boura *et al.*, CSCML'19). From then on, little effort has been made towards defining efficient ways of using it to implement functions with high precision. In this paper, we introduce two methods to combine multiple functional bootstraps to accelerate the evaluation of reasonably large look-up tables and highly precise functions. We thoroughly analyze and experimentally validate the error propagation in both methods, as well as in the functional bootstrap itself. We leverage the multi-value bootstrap of Carpov *et al.* (CT-RSA'19) to accelerate (single) lookup table evaluation, and we improve it by lowering the complexity of its error variance growth from quadratic to linear in the value of the output base. Compared to previous literature using TFHE's functional bootstrap, our methods are up to 2.49 times faster than the lookup table evaluation of Carpov *et al.* (CT-RSA'19) and up to 3.19 times faster than the 32-bit integer comparison of Bourse *et al.* (CT-RSA'20). Compared to works using logic gates, we achieved speedups of up to 6.98, 8.74, and 3.55 times over 8-bit implementations of the functions ReLU, Addition, and Maximum, respectively.

Keywords: Functional Bootstrap · TFHE · Lookup Table · Homomorphic Encryption.

1 Introduction

The efficient evaluation of non-linear functions with high precision is a challenge for homomorphic encryption schemes and many of them rely on arithmetic approximations, such as Taylor, Fourier, and Chebyshev series [BGGJ19, CCS19, KWN20]. They allow them to work with packed messages in a SIMD[1] manner [BGH13], which greatly reduces the amortized cost of operations. However, the cost of implementing such approximations grows exponentially with the desired precision [LJ19], which makes this approach unfit for many applications. Other schemes implement circuits using binary gates [CGGI20], which are a versatile and straightforward way of achieving good precision. Their main disadvantage is the low throughput of operations, which leads to scalability problems.

All currently known fully homomorphic encryption (FHE) schemes rely on noisy ciphertexts for security, *i.e.*, the encryption process adds a small error (noise) to the message. This error grows when performing arithmetic, and, eventually, it might affect

We would like to thank the São Paulo Research Foundation (grants 2013/08293-7 and 2019/12783-6), and the National Council for Scientific and Technological Development (grant 313012/2017-2) for supporting this research. We also would like to thank Tancrède Lepoint for serving as the shepherd for this paper.

[1]Single instruction, multiple data

Received: 2020-10-15 Accepted: 2020-12-15 Published: 2021-02-23

significant bits of the message. To preserve the message, FHE schemes reset the error from time to time during the circuit evaluation by using a *bootstrap procedure*. It is usually an expensive process, but schemes implementing circuits using logic gates may enable very fast bootstraps at the cost of performing one at every gate. In some of them, it is also possible to implement the evaluation of arbitrary functions within the bootstrap, which usually results in more efficient implementations [CJP20].

A bootstrap procedure that evaluates a function (other than an ordinary reset of error) is called *Functional Booststrap* [BGGJ19]. In FHEW-like cryptosystems [DM15], the bootstrap is just a rounding function evaluated using a lookup table (LUT). Thus, to evaluate an arbitrary function, we only need to replace the bootstrap LUT with one that encodes it. This idea was introduced with the FHEW cryptosystem [DM15, MP20] and has been used by a few applications in the literature. We can refer, for example, to FHE-DiNN [BMMP18] and the work of Izabachène *et al.* [ISZ19], which both implement the sign function using the functional bootstrap of the TFHE cryptosystem [CGGI20]. These applications can perform a functional bootstrap in just tens of milliseconds thanks to the low precision (1 bit) required by the sign function. Applications requiring higher precision, on the other hand, need to increase the parameters of the cryptosystem to keep the evaluation correct, which leads to deteriorated performance. For example, a 6-bit-to-6-bit LUT takes 1.5 seconds to be evaluated using TFHE's functional bootstrap, as implemented by Carpov *et al.* [CIM19].

Contributions. We show how to evaluate functions with high precision using multiple functional bootstraps, but without increasing (too much) the parameters of the cryptosystem. This approach results in a much smaller impact on performance. The following contributions are presented in this work.

- We introduce two new methods to combine multiple functional bootstraps in TFHE:
  - A tree-based one that allows the easy implementation of arbitrary functions, as well as a tree optimization based on particular properties of each function. We leverage the multi-value bootstrap of Carpov *et al.* [CIM19] to lower the number of bootstraps in this method (asymptotically) from exponential to linear in the size of the input.
  - A chaining one that presents a better error rate growth behavior, but which is more intricate to implement depending on the target function.
- We perform an error variance analysis, including experimental validation, and a comparison between the aforementioned methods.
- We present optimizations to the building blocks used in our methods, which are also contributions of independent interest.
  - We introduce a *multi-value extract* procedure that produces multiple LWE samples encrypting the same value with independent errors. It enables improving the error growth on ciphertext scaling from quadratic to linear with little performance overhead. It also improves the error variance growth in the multi-value bootstrap [CIM19] from quadratic to linear in the output base.
  - We introduce a *"base-aware"* Key Switching to pack $B < N$ LWE samples in an RLWE sample, where N is the polynomial size. In this work, B is the base of our integer encoding (thus, "base-aware"), but the technique enables gains of up to $\lfloor \frac{N}{B} \rfloor$ times for any $B < N$.
- We present implementations[2] of several relevant functions and compare their performance with state-of-the-art implementations from the literature.

[2]The source code is available at https://github.com/antoniocgj/FBT-TFHE.

Lookup tables are an important tool in the implementation of arbitrary functions in homomorphic circuits. They are as versatile as logic gates while capable of providing better throughput of operations. Our methods speed up their evaluation in up to 2.49 times, and, for specific functions, we also show possibilities of optimizations over the generic LUT evaluation. Compared to implementations using logic gates, we achieve speedups of up to 8.74 times in simple and useful functions, such as integer addition.

This paper is organized as follows: Section 2 reviews the basics of the TFHE cryptosystem with a focus on its functional bootstrap; Section 3 introduces the methods to combine functional bootstraps, analyzes their error variance behavior, and presents optimizations in their building blocks; Section 4 presents a performance analysis of our methods, including comparison with the literature; Section 5 summarizes the related work; Section 6 concludes the paper.

2 The TFHE Cryptosystem

TFHE is a fully homomorphic cryptosystem with security based on the (Ring) Learning With Errors problem [Reg09]. It is based on the FHEW cryptosystem [DM15], but it features much faster bootstraps thanks to the use of binary secrets [MP20]. In this section, we review the concepts of TFHE necessary for the understanding of this paper.

Let $\mathbb{A}$ be a set, we denote by $\mathbb{A}_q^n$ the set of vectors with n elements in $\mathbb{A}$ modulo q and by $\mathbb{A}_N[X]^n$ the set of vectors of n polynomials modulo $(X^N + 1)$. If omitted, $n = 1$ and $q = \infty$. The Real Torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ is the set of real numbers modulo 1 and $\mathbb{B} = \mathbb{Z}_2$ is the set of binary numbers $\{0, 1\}$. TFHE defines three types of ciphertexts, which we summarize below as samples of zero.

- **TLWE Sample:** A pair $(a, b) \in \mathbb{T}^{n+1}$, where $b = \langle a, s\rangle + e$. The vector a is uniformly sampled from $\mathbb{T}^n$, the secret key s is uniformly sampled from $\mathbb{B}^n$, the error $e \in \mathbb{T}$ is sampled from a Gaussian distribution with mean 0 and standard deviation σ, and $\langle\ ,\ \rangle$ denotes the inner product.
- **TRLWE Sample:** A pair $(a, b) \in \mathbb{T}_N[X]^{k+1}$, where $b = a \cdot S + e$. The vector a is uniformly sampled from $\mathbb{T}_N[X]^k$, the secret key S is uniformly sampled from $\mathbb{B}_N[X]^k$, and the error $e \in \mathbb{T}_N[X]$ is a polynomial with random coefficients sampled from a Gaussian distribution with mean 0 and standard deviation σ.
- **TRGSW Sample:** A vector of ℓ TRLWE samples.

Encryption To encrypt a message $m \in \mathbb{T}$ (TLWE) or $m \in \mathbb{T}_\mathbb{N}[\mathbb{X}]$ (TRLWE), we simply add $(0, m)$ to a fresh sample of zero. We denote by $c \in \text{T(R)LWE}_s(m)$ the T(R)LWE sample c that encrypts m with key s. To ease the notation, we consider each key has its attached set of parameters. A message $m \in \mathbb{T}_\mathbb{N}[\mathbb{X}]$ can also be encrypted in TRGSW samples by adding $m \cdot H$ to a TRGSW sample of zero, where H is a gadget decomposition matrix. We do not use TRGSW samples in our algorithms and, therefore, we will get into further details about it only when necessary.

Decryption To decrypt a sample, we first calculate its phase (message + error): $\phi(c) = b - \langle a, s\rangle$. Considering approximate computing, the phase might be a good enough approximation for the message (depending on the error variance). For exact computing, we need to remove the error, and we do so by rounding the phase to the nearest valid value for messages. This requires us to define a set of valid messages over the Torus. The rounding procedure fails if the error is greater than half the distance (in the Torus) between two consecutive messages.

Arithmetic We add two ciphertext (TLWE or TRLWE) samples $c_1 = (a_1, b_1)$ and $c_2 = (a_2, b_2)$ by simply adding their terms: $c_1 + c_2 = (a_1 + a_2, b_1 + b_2)$. The multiplication between a ciphertext $c_1 = (a_1, b_1)$ and a scalar cleartext $z \in \mathbb{Z}$ (or $z \in \mathbb{Z}_N[X]$ for TRLWE) is a direct result from the addition: $c_1 \cdot z = (a_1 \cdot z, b_1 \cdot z)$. TFHE does not support multiplications between T(R)LWE samples. To be fully homomorphic, it relies on external products between TRGSW and TRLWE samples. In this case, we first decompose the TRLWE sample in ℓ TRLWE samples using a Gadget decomposition algorithm [GMP19]. Then, we perform an inner product between the decomposed TRLWE and the TRGSW sample (which already is a vector of ℓ TRLWE samples).

2.1 Building Blocks of TFHE

TFHE has three algorithms as its main building blocks, which we briefly review in this section. Following the notation defined by Chillotti *et al.* [CGGI20], we use underbars and overbars to indicate input and output variables, respectively. The first building block is the *Public Functional Key Switching*, shown in Algorithm 1. It allows the switching of keys and parameters from TLWE to T(R)LWE samples, such as the packing of TLWE samples in a TRLWE sample. It also evaluates a linear function f over the input TLWEs. Chillotti *et al.* [CGGI20] define this algorithm with binary decomposition, as Algorithm 1 shows, but TFHE supports other power of 2 bases. We analyze the impact of changing the decomposition base on the error growth in Section 3.3.

Algorithm 1: TFHE's TLWE-to-T(R)LWE Public Functional Key Switching [CGGI20]

```
  Input  : $p$ TLWE samples $\underline{c}^{(z)} = (\underline{a}^{(z)}, \underline{b}^{(z)}) \in \mathsf{TLWE}_{\underline{s}}(\mu_z)$, $z \in [\![1, p]\!]$
  Input  : a public R-Lipschitz linear function $f : \mathbb{T}^T \mapsto \mathbb{T}_N[X]$
  Input  : a precision parameter $t \in \mathbb{Z}$
  Input  : a Key Switching key $\mathsf{KS}_{i,j} \in \mathsf{T(R)LWE}_{\overline{s}}(\frac{s_i}{2^j})$, for $i \in [\![1, n]\!]$ and $j \in [\![1, t]\!]$.
  Output : a T(R)LWE sample $\overline{c} \in \mathsf{T(R)LWE}_{\overline{s}}(f(\mu_z))$, for $z \in [\![1, p]\!]$.
1 for $i \in [\![1, n]\!]$ do
2     $a_i \leftarrow f(\underline{a}_i^{(1)}, \underline{a}_i^{(2)}, ..., \underline{a}_i^{(p)})$
3     Let $\tilde{a}_i = \lceil a_i \rfloor_{\frac{1}{2^t}}$ be the closest multiple of $\frac{1}{2^t}$ to $a_i$.
4     Decompose each $\tilde{a}_i = \sum_{j=1}^{t} \tilde{a}_{i,j} \cdot 2^{-j}$, where $\tilde{a}_{i,j} \in \mathbb{B}_N[X]$.
5 Return $(0, f(\underline{b}_i^{(1)}, \underline{b}_i^{(2)}, ..., \underline{b}_i^{(p)})) - \sum_{i=1}^{n} \sum_{j=1}^{t} \tilde{a}_{i,j} \cdot \mathsf{KS}_{i,j}$
```

The second building block is the $\mathrm{SampleExtract}_p(c)$ algorithm, which receives a TRLWE sample $c \in \mathrm{TRLWE}_S(m)$ and a position $p \in [\![0, N-1]\!]$. It returns a LWE sample $\overline{c} \in \mathrm{TLWE}_{\overline{S}}(m_p)$, where $m_p \in \mathbb{T}$ is the p-th coefficient of $m \in \mathbb{T}_N[X]$, and $\overline{S} \in \mathbb{B}^N$ is a vector (TLWE) interpretation of $S \in \mathbb{B}_N[X]$.

The last building block is the *BlindRotate* algorithm, shown in Algorithm 2. It rotates the polynomial encrypted in a TRLWE sample by an encrypted number. This is the core procedure for the evaluation of look-up tables (LUTs) in TFHE, introduced in the vertical packing by Chillotti *et al.* [CGGI20]. Suppose we want to look up the number $s \in \mathbb{Z}$ in the LUT A, both encrypted. To use *BlindRotate* directly, s needs to be binary decomposed ($s = \sum_{i=0}^{n-1} s_i \cdot 2^i$) and encrypted in $C_i \in \mathrm{TRGSW}_S(s_i)$, for $i \in [\![1, n]\!]$. Each position of A is packed in a coefficient of a Torus polynomial in $\mathbb{T}_N[X]$ and encrypted in a TRLWE sample. By calling *BlindRotate*$(A, (2^0, 2^1, ..., 2^n, 0), (C_0, C_1, C_n))$, the position we want to look up in A is moved to the constant term of the TRLWE sample, and we can use $\mathrm{SampleExtract}_0(\overline{c})$ to obtain its LWE encryption. It is important to note that the multiplication $X^{a_i} \cdot \mathrm{ACC}$ occurs modulo the cyclotomic polynomial $\Phi_{2N} = (X^N + 1)$ and, hence, presents a negacyclic property, *e.g.* $X^N \cdot \mathrm{ACC}$ *mod* $\Phi_{2N} = -\mathrm{ACC}$.

Algorithm 2: TFHE's BlindRotate Algorithm [CGGI20]

```
Input  : a sample $c \in \mathsf{TRLWE}_S(m)$
Input  : a list of integers $(a_1, ..., a_n, b) \in \mathbb{Z}_{2N}^{n+1}$
Input  : a list of samples $C_i \in \mathsf{TRGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Output : a TRLWE sample of $\overline{c} \in \mathsf{TRLWE}_S(X^{-\rho} \cdot m)$, where $\rho = b - \sum_{i=1}^{n} s_i \times a_i$ mod $2N$
1 ACC ← $X^{-b} \cdot c$
2 for $i = 1$ to $p$ do
3   | ACC ← CMUX($C_i, X^{a_i} \cdot$ ACC, ACC)
4 return ACC

1 Procedure CMUX($C, A, B$)
2   | return $C \cdot (B - A) + B$
```

2.2 Bootstrap

There are two bootstraps algorithm in TFHE. Algorithm 3 shows the *Gate Bootstrap*, which was introduced for the implementation of logic gates and is supposed to be used on samples representing binary numbers. Torus values are in the interval $(-0.5, 0.5]$ in the signed representation. Zero is usually represented by $-\frac{1}{4}$ while one is represented by $+\frac{1}{4}$. The arithmetic of each logic gate implementation change these values, but it keeps their signal. Then, TFHE uses the Gate Bootstrap with $\mu = \frac{1}{4}$ to set the value back to $+\frac{1}{4}$ or $-\frac{1}{4}$ (plus a small bootstrap error), depending on the bit value. For example, the `NAND` gate between TLWE samples $c_1 \in \{-\frac{1}{4}, +\frac{1}{4}\}$ and $c_2 \in \{-\frac{1}{4}, +\frac{1}{4}\}$ is implemented as $Bootstrap((0, \frac{5}{8}) - c_1 - c_2)$.

Algorithm 3: TFHE's Gate Bootstrap algorithm [CGGI20]

```
Input  : a TLWE sample $\underline{c} = (\underline{a}, \underline{b}) \in \mathsf{TLWE}_s(m)$
Input  : a constant $\mu \in \mathbb{T}$
Input  : a bootstrapping key $\mathsf{BK}_i \in \mathsf{TRGSW}_S(s_i)$, for $i \in [\![1, n]\!]$.
Output : $\overline{c} \in TLWE_{\overline{S}}(\overline{m} \cdot \mu)$, where $\overline{S} \in \mathbb{B}^N$ is a vector (TLWE) interpretation of
         $S \in \mathbb{B}_N[X]$ and $\overline{m} = \begin{cases} 1, & \text{if } m < 0.5. \\ -1, & \text{otherwise.} \end{cases}$
1 $b \leftarrow \lfloor 2N\underline{b} \rceil$ and $a_i \leftarrow \lfloor 2N\underline{a}_i \rceil \in \mathbb{Z}_{2N}$ for each $i \in [\![1, n]\!]$
2 $v \leftarrow (1, X, X^2, ..., X^{N-1}) \cdot \mu \in \mathbb{T}_N[X]$
3 ACC ← BlindRotate($(0, v), (a_1, ..., a_n, b), (\mathsf{BK}_1, ..., \mathsf{BK}_n)$)
4 return SampleExtract$_0$(ACC)
```

The second bootstrap algorithm in TFHE is the *Circuit Bootstrap*, which converts TLWE samples to TRGSW samples. This bootstrap is 10 times more expensive than the Gate Bootstrap, but it enables the possibility of fully composable CMUX circuits. Since it requires a 64-bit Torus precision, it is implemented only in the experimental branch of TFHE, which we do not use in this work.

2.3 Functional Bootstrap in TFHE

Similarly to the *Functional Key Switching*, the bootstrap can be said to be *"Functional"* if it maps a set of inputs to a codomain in a programmatical manner, *i. e.* it can evaluate functions. In TFHE, the bootstrap uses the BlindRotate algorithm to perform a lookup table (LUT) evaluation. LUTs are a very simple yet efficient way of evaluating discretized functions. They encode functions by storing discretized elements of their images, and the evaluation is performed by selecting a table position based on the function parameter, which is the lookup *Selector*. The gate bootstrap of TFHE is the evaluation of a 1-bit lookup table (LUT) encoding the rounding function. The *test vector* $(0, v)$ (line 2 of Algorithm 3) encodes the LUT with the value of μ and $-\mu$ (thanks to the negacyclic property), and $\underline{c}$ is the *selector*. We can evaluate some other functions by just adjusting the value of μ. For example, the `sign` function, as used in FHE-DiNN [BMMP18] and by Izabachène *et al.* [ISZ19]. To evaluate an arbitrary LUT (with more than one value and its negative), we need to increase the size of the LUT, to discretize the function according to this size, and to work only with the positive half of the Torus, since the negacyclic property can only be explored by anti-symmetric functions.

We represent integers in the Torus by partitioning its positive half in slices. We define the *Torus base*, B, as the number of integers mapped, and the *Torus slice* as the distance between two consecutive integers in the Torus. For example, with $B = 4$, the size of each Torus slice is 0.125 and the map of integers is $(0, 1, 2, 3) \mapsto (0, 0.125, 0.25, 0.375)$. This approach has been extensively used to represent bounded integers in the Torus [BMMP18, CIM19, ISZ19]. To represent unbounded integers, we decompose them in *digits* of the base B and encrypt each digit in a TLWE sample. This approach is more common for applications using binary digits, but it has also been used with arbitrary bases [BST20].

2.3.1 Encoding a LUT in a TRLWE sample

A single TRLWE sample $c \in \mathbb{T}_N[X]^{k+1}$ can encode at most N entries of a lookup table (LUT). However, when using the bootstrap to evaluate the LUT, only a small fraction of N will actually be available if the goal is perfect computation. The first step of the bootstrap is to scale each element of the ciphertext to $2N$ and round it to an integer. This process introduces a significant error variance, which is additive with the variance of the Gaussian error. To prevent these lookup errors, we map each position of a LUT to a sequence of many consecutive coefficients of the *test vector* and we adjust the selector to lookup positions in the middle of these sequences. For example, with $B = 4$ and $N = 1024$, an integer LUT $L = [l_1, l_2, l_3, l_4] \in \mathbb{Z}_B^4$ is mapped to $\sum_{i=0}^{255} \frac{l_1}{2B} X^i + \sum_{i=256}^{511} \frac{l_2}{2B} X^i + \sum_{i=512}^{767} \frac{l_3}{2B} X^i + \sum_{i=768}^{1023} \frac{l_4}{2B} X^i$ and we add a *precision offset* of $\frac{1}{4B} = \frac{1}{16}$ to the selector $\underline{c}$. Algorithm 4 shows the functional bootstrap in TFHE already considering the integer and LUT encoding we defined.

Algorithm 4: Functional Bootstrap in TFHE

```
Input : a TLWE sample $\underline{c} = (\underline{a}, \underline{b}) \in \mathsf{TLWE}_s(\frac{m}{2B})$, for $m \in Z_B$
Input : an integer LUT $L = [l_0, l_1, ..., l_{B-1}] \in \mathbb{Z}_B^B$
Input : a bootstrapping key $\mathsf{BK}_i \in \mathsf{TRGSW}_S(s_i)$, for $i \in [\![1, n]\!]$.
Output : $\overline{c} \in TLWE_{\overline{S}}(\frac{L[m]}{2B})$, where $\overline{S} \in \mathbb{B}^N$ is a vector (TLWE) interpretation of
         $S \in \mathbb{B}_N[X]$
1 $b \leftarrow \lfloor 2N\underline{b} \rceil$ and $a_i \leftarrow \lfloor 2N\underline{a}_i \rceil \in \mathbb{Z}_{2N}$ for each $i \in [\![1, n]\!]$
2 $v \leftarrow \sum_{i=0}^{N-1} \frac{1}{2B} \cdot l_{\lfloor \frac{iB}{N} \rfloor} X^i \in \mathbb{T}_N[X]$
3 ACC $\leftarrow$ BlindRotate$((0, v), (a_1, ..., a_n, b + \frac{1}{4B}), (\mathsf{BK}_1, ..., \mathsf{BK}_n))$
4 return SampleExtract$_0$(ACC)
```

2.3.2 Multi Value Bootstrap

The most expensive procedure in the LUT evaluation using the functional bootstrap of TFHE is the BlindRotate. The multi-value bootstrap is a technique that allows the evaluation of multiple LUTs with the same selector using just one BlindRotate. Suppose we want to evaluate q LUTs $(L_0, L_1, ..., L_q)$ with the same selector $\underline{c} \in \text{TLWE}_s(m)$ and we want to minimize the number of blind rotations. A simple way of doing so is to perform a (single-value) functional bootstrap with *test vector* $(0, v) = (0, 1) \in T_N[X]^{k+1}$

and selector $\underline{c}$. Its output would be $\overline{c} = TLWE_s(X^{-2N\cdot\phi(\underline{c}) \bmod 2N})$. Then, to evaluate each LUT, we represent it as an integer polynomial in $\mathbb{Z}_N[X]$, multiply by $\overline{c}$, and extract the constant term. This approach requires just one BlindRotate, but the error variance grows quadratically with the square norm of each LUT.

Carpov *et al.* [CIM19] introduced a multi-value bootstrap scheme that allows for a much smaller error growth. As Algorithm 5 shows, the test vector is set to $(0, \tau\cdot\frac{1}{2}\sum_{i=0}^{N-1} X^i)$, where τ is a scaling factor usually set as the `gcd` among the coefficients of all LUTs. After the blind rotation, each LUT (represented as a polynomial in $TV_{F_i} \in \mathbb{Z}_N[X]$) is divided by v, before being multiplied by the accumulator (ACC). This division greatly reduces the square norm of the LUTs and, hence, the error growth. Carpov *et al.* also introduced a very efficient way of calculating $\frac{TV_{F_i}}{v}$ that only requires a subtraction between each pair of consecutive coefficients of each LUT.

Algorithm 5: Carpov *et al.* multi-value bootstrapping algorithm [CIM19]

```
Input : a TLWE sample $\underline{c} = (\underline{a}, \underline{b}) \in \mathsf{TLWE}_s(\frac{m}{2N})$, $m \in \mathbb{Z}_{2N}$
Input : a scale factor $\tau$
Input : $q$ LUTs encoded in polynomials $TV_{F_i} \in \mathbb{Z}_N[X]$, for $i \in [\![1, q]\!]$
Input : a bootstrapping key $\mathsf{BK}_i \in \mathsf{TRGSW}_S(s_i)$, for $i \in [\![1, n]\!]$.
Output: An array of TLWE samples $\overline{c_i} \in \mathsf{TLWE}_{\overline{S}}(\frac{F_i(m)}{2N})$ for $i = 1, ..., q$, where $\overline{S} \in \mathbb{B}^N$ is a
        vector interpretation of $S \in \mathbb{B}_N[X]$.
1 $b \leftarrow \lfloor 2N\underline{b} \rceil, a_i \leftarrow \lfloor 2N\underline{a}_i \rceil \in \mathbb{Z}_{2N}$ for each $i \in [\![1, n]\!]$
2 $v \leftarrow \frac{1}{2}\sum_{i=0}^{N-1} X^i \cdot \frac{1}{2N} \cdot \tau \in \mathbb{T}_N[X]$
3 ACC $\leftarrow$ BlindRotate$((0, v), (a_1, ..., a_n, b), (\mathsf{BK}_1, ..., \mathsf{BK}_n))$
4 $\overline{c_i}$ = SampleExtract$_0(\frac{TV_{F_i}}{v} \cdot$ ACC$)$, for each $i \in [\![1, q]\!]$
5 Return $\overline{c}$
```

3 Combining Functional Bootstraps

To evaluate large LUTs considering the limitations we describe in Section 2.3.1, we either need to increase N (which increases the bootstrap time superlinearly [BST20]) or to lookup in multiple TRLWE samples. In this section, we follow this latter approach and introduce two methods to combine multiple functional bootstraps to evaluate a single large LUT. Figure 1 illustrates the evaluation of an 8-bit parity function using them. In both methods, we use the encoding for unbounded integers we described in Section 2.3 and represent them in base B with d digits.

3.1 Tree-based method

Our first method to evaluate functions using multiple functional bootstraps is structured as a (convergence) tree and uses the output of a lookup to construct a new LUT. Algorithm 6 shows its final version. Let L be the B^d-sized LUT encoded in B^{d-1} TRLWE samples (following the encoding we described in Section 2.3.1) and $\underline{c}_i \in \mathrm{TLWE}_s(\frac{m_i}{2B})$ for $i \in [\![0, d)$ be the selector, such that $\sum_{i=0}^{d-1} m_i B^i = m$ encodes the integer m in base B with d digits. Our first step is to perform a functional bootstrap using the same selector c_0 on each of the B^{d-1} TRLWE samples. This process results in B^{d-1} TLWE samples. If $d = 1$, the evaluation is finished. Otherwise, we use a TLWE-to-TRLWE key switching to pack them in B^{d-2} TRLWE samples. With that, we reduced our problem to the evaluation of a B^{d-1}-sized LUT encoded in B^{d-2} TRLWE samples with selector $\underline{c}_i \in \mathrm{TLWE}_s(\frac{m_i}{2B})$ for $i \in [\![1, d)$ and we can recursively repeat this process until we have a single TRLWE sample (as in $d = 1$). Figure 2a illustrates it for $B = 4$ and a LUT encoding the sigmoid function

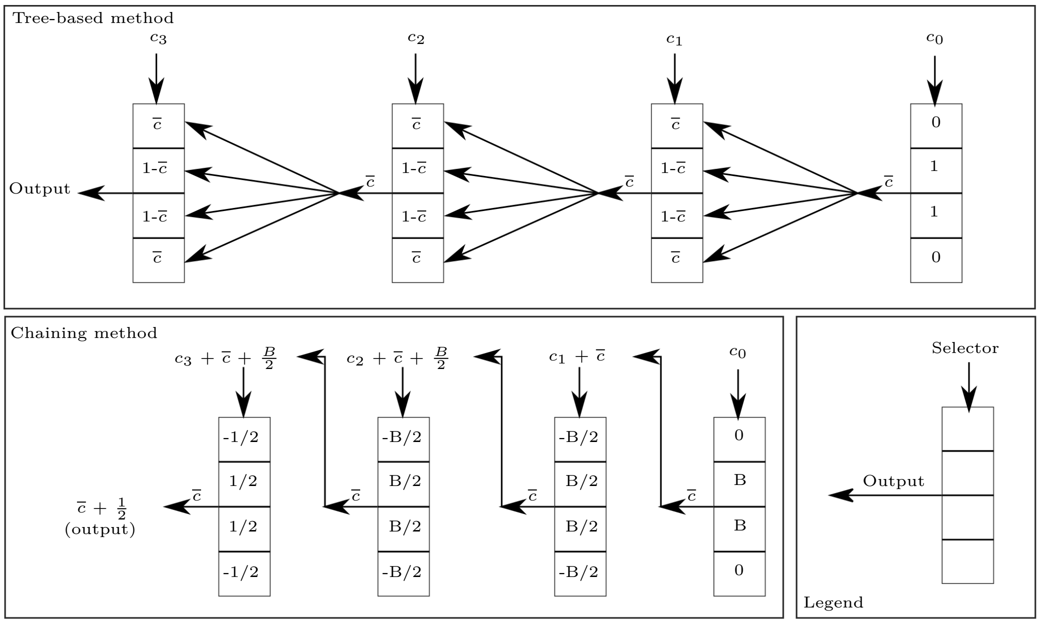


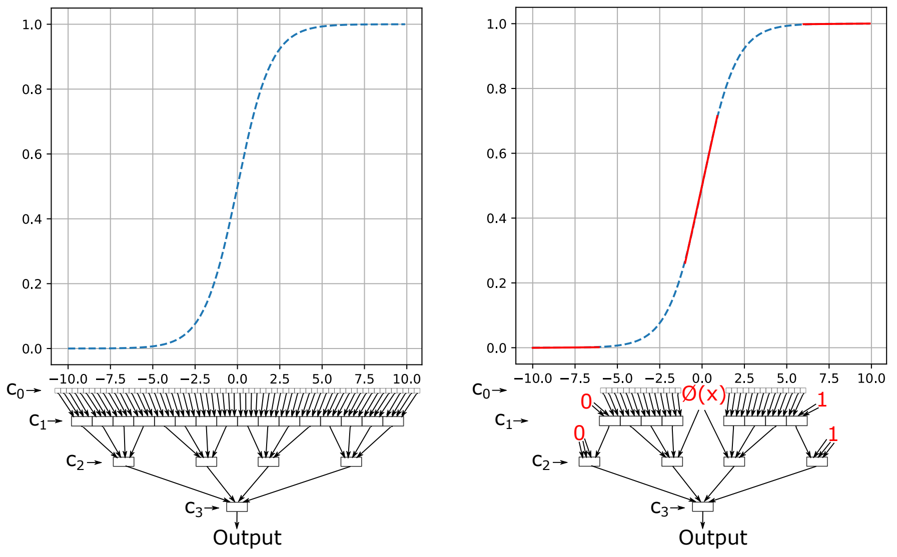
Figure 1: Evaluation of an 8-bit parity function using the tree-based and the chaining methods using base $B = 4$.

with 8 bits of precision (thus, $d = 4$). Each rectangle represents a LUT, the arrows indicate the flow of data and c_i for $i \in [\![0, 3]\!]$ is the selector.

Algorithm 6: Tree-based method for combining functional bootstraps

```
Input  : a set of TLWE samples $\underline{c}_i \in \mathsf{TLWE}_s(\frac{m_i}{2B})$, such that $\sum_{i=0}^{d-1} m_i B^i = m$ encodes the
         integer $m$ in base $B$ with $d$ digits.
Input  : a set L of $B^d$ polynomials $\in \mathbb{Z}_N[X]$ encoding the lookup table of an arbitrary
         function F.
Input  : a bootstrapping key $\mathsf{BK}_i \in \mathsf{TRGSW}_S(s_i)$, for $i \in [\![1, n]\!]$.
Input  : a Key Switching key $\mathsf{KS}_{i,j} \in \mathsf{T(R)LWE}_{\overline{s}}(\frac{s_i}{2^j})$, for $i \in [\![1, n]\!]$ and $j \in [\![1, t]\!]$.
Output : A TLWE sample $\overline{c} \in \mathsf{TLWE}_{\overline{S}}(\frac{F(m)}{2B})$, where $\overline{S} \in \mathbb{B}^N$ is a vector (TLWE)
         interpretation of $S \in \mathbb{B}_N[X]$.
1 TV ← L
2 $f : \mathbb{T}^B \mapsto \mathbb{T}_N[X] = (a_1, ..., a_B) \mapsto a_1 X^{N-1} + \ ... \ + a_B$
3 for $i \leftarrow 0$ to $d - 1$ do
4     $\overline{c} \leftarrow$ MultiValueBoostrap($\underline{c}_i$, TV, BK)
5     for $j \leftarrow 1$ to $B^{d-i-2}$ do
6         $\mathsf{TV}_{j-1} \leftarrow$ PublicKeySwitch(($\overline{c}_{(j-1)\times B}, ..., \overline{c}_{j \times B}$), $f$, KS)
7 return $\overline{c}_0$
```

Without the multi-value bootstrap, the complexity of this process (measured in the number of functional bootstraps) would be exponential in the number of digits d. However, each level of the tree performs B^{d-1-i} bootstraps using the same selector c_i, thus allowing us to replace them with a single multi-value bootstrap, which reduces the complexity to linear in the number of digits. This complexity improvement results in similar performance gain asymptotically, but it faces practical problems for not so large LUTs. As we described in Section 2.3.2, the multi-value bootstrap relies on multiplications between the accumulator (ACC in line 4 of Algorithm 5) and the LUTs encoded as polynomials in $\mathbb{Z}_N[X]$. In the



(a) Full-tree **(b)** Optimized tree

Figure 2: Evaluation of an 8-bit sigmoid function using the tree-based with base $B = 4$.

first level of our tree, this is not a problem since the LUTs are cleartext and can be encoded as polynomials in $\mathbb{Z}_N[X]$. Starting from level 1, the LUTs are now encrypted as TRLWE samples. Arithmetically, this is not an obstacle, but the multiplication between ACC and the LUT is now a multiplication between two TRLWE samples, which is not directly performed in TFHE. We would need to convert ACC from TRLWE to TRGSW using a Circuit Bootstrap, which would only be worth it if the number of Functional Bootstraps we are replacing is very large. In this work, most of our examples are 8-bit functions, and, therefore, we only use the multi-value bootstrap in the first level of the tree.

TFHE presents an experimental version of a TLWE-to-TRGSW circuit bootstrap implemented with 64-bit precision that costs around 130 ms (or 10 times a gate bootstrap). It does not present a TRLWE-to-TRGSW bootstrap. Naively, we could adapt TFHE's algorithm to implement a TRLWE-to-TRGSW circuit bootstrap, which would be N times more expensive. However, we cannot consider such implementation a representative of TRLWE-to-TRGSW conversion performance. Practical implementations on this are mostly an open problem and there are many recent developments in the literature [MS18, BMMP18, CDKS20] that can be used to achieve much more efficient conversions. The original (32-bit) implementation of TFHE does not support the circuit bootstrap, and, if we worked on another implementation of TFHE, it would be hard to compare with results from the literature. We instead prefer to leave the implementation of a TRLWE-to-TRGSW conversion as future work, especially considering that this paper is focused more on algorithmic than implementation aspects.

3.1.1 Optimizing the Tree

One of the main advantages of this method is the versatility. We can not only evaluate any function but also optimize the tree considering the particular characteristics of each function. The sigmoid function, for example, has three intervals in its domain that could be linearly evaluated or approximated: $[\![-\infty, -6]\!] \approx 0$, $[\![6, \infty]\!] \approx 1$, and $[\![-1, +1]\!] \approx 0.24x + 0.5$. Figure 2b illustrates this optimization. $\varnothing(x)$ is the linear combination to calculate $f(x) = 0.24x + 0.5$ using integers (fixed-point representation). The output error of the optimized tree is not increased by approximations using cleartext literals (intervals $[\![-\infty,-6]\!] \approx 0$, $[\![6,\infty]\!] \approx 1$). The linear combination $\varnothing(x)$ increases the error if its results are not bootstrapped, which is something to consider when composing functions to make an application. Considering security, the implications of optimizing the tree are limited to secondary aspects that are not on our scope. For example, the full-tree design enables us to easily achieve circuit privacy by simply encrypting the initial LUT, whereas tree optimizations could give partial information about the function.

3.2 Chaining Method

This second method is a generalized version of the integer comparison algorithm of Bourse *et al.* [BST20]. It is much more functionally restricted than the first method, but it usually presents a smaller error growth. Its main characteristic is that the output of a lookup is used to construct the selector of the next lookup, whereas, in the tree-based approach, the output is used to construct the next LUT. This difference has deep implications on the error propagation and overall functionality. Consider a selector $\underline{c}_i \in \text{TLWE}_s(\frac{m_i}{2B})$ for $i \in [\![0,d)$ with d digits. The first functional bootstrap uses $\underline{c}_0$ as the selector and outputs $\overline{c}_0$. From then on, each evaluation uses the selector $\oslash(\underline{c}_i, \overline{c}_{i-1})$ for $i \in [\![1,d)$, where $\oslash$ is a linear combination. We can define this method as being functionally capable of evaluating any function that can be encoded in LUTs such that, for each digit, either the output of $\oslash(\underline{c}_i, \overline{c}_{i-1})$ is smaller than the size of the LUT, B, or the function being encoded follows a B-anti-cyclic [BGGJ19] logic. Although it is hard to generically define functions with such restrictions, this method seems to be especially good for functions that require carry-like logics, such as additions and multiplications.

3.3 Error analysis

In this section, we analyze the error variance growth of each algorithm used to perform a functional bootstrap and we calculate the overall probability of error based on the final error variance. We start by reproducing two equations from Chillotti *et al.* [CGGI20]. Equation 1 and 2 show the variance resultant of the key switching and the gate bootstrap procedures, respectively. The underbar indicates input variables, ϑ_{KS} and ϑ_{BK} are, respectively, the variance of the bootstrap and key switching keys, and $\epsilon = \frac{1}{2B_g^\ell}$ and B_g are the gadget decomposition quality and base, also respectively. All other variables were introduced at the beginning of Section 2.

$$Var(Err(c)) \leq R^2 Var(Err(\underline{c})) + \underline{n}tN\vartheta_{KS} + \underline{n}2^{-2(t+1)} \tag{1}$$

$$Var(Err(c)) \leq \underline{n}(k+1)\ell N(\frac{B_g}{2})^2\vartheta_{BK} + \underline{n}(1+kN)\epsilon^2 \tag{2}$$

In the chaining method, functional bootstraps have the same output error variance as the gate bootstrap (Equation 2) since both use noiseless *test vectors*. In the tree-based method, the *test vector* is a TRLWE sample that might be encrypting the result of previous table lookups and, hence, we need to consider its error variance, which is additive with the one introduced by the bootstrap, giving us Equation 3. Considering the multi-value bootstrap, Carpov *et al.* presents Equation 4. Up to line 3 of algorithm 5 the variance is the same as the single value bootstrap, but the multiplication $\frac{TV_{F_i}}{v} \cdot \text{ACC}$ in line 4 multiples the variance of the bootstrap by $\|TV_f\|_2^2 \leq s(q-1)^2$, where s and q are the input and output bases, respectively.

$$Var(Err(c)) \leq Var(Err(TV)) + \underline{n}(k+1)\ell N(\frac{B_g}{2})^2\vartheta_{BK} + \underline{n}(1+kN)\epsilon^2 \tag{3}$$

$$Var(Err(c)) \leq \|TV_f\|_2^2(\underline{n}(k+1)\ell N(\frac{B_g}{2})^2\vartheta_{BK} + \underline{n}(1+kN)\epsilon^2) \tag{4}$$

As for the key switching algorithm, Chillotti *et al.* [CGGI20] only analyzes it using binary decomposition and TFHE only implements the TLWE-to-TLWE key switching. In this work, we use both the TLWE-to-TLWE key switching of TFHE and a TLWE-to-TRLWE key switching to pack TLWE samples in a TRLWE sample. This TLWE-to-TRLWE key switching is a core algorithm of our tree-based approach and, to improve efficiency, we need to use greater bases for decomposition. Considering that, we reanalyzed the key switching error variance introduction. Equation 1 presents three terms. The first term, $R^2Var(Err(\underline{c}))$, comes from possible scalings performed by the linear function f (we only use 1-Lipschitz functions, therefore $R^2 = 1$). The second term, $\underline{n}tN\vartheta_{KS}$, comes from the addition of ($\underline{n}tN$) T(R)LWE samples (the summation in line 5 of Algorithm 1), each of them with variance ϑ_{KS}. The third term, $\underline{n}2^{-2(t+1)}$, is the variance introduced by the rounding of the binary decomposition. Chillotti *et al.* defines the error variance starting from the error amplitude: Each of the $\underline{n}$ elements of the vector $\underline{a}$ of the TLWE input $\underline{c}$ are rounded to the closest multiple of 2^{-t}, which introduces an error of at most $|\frac{2^{-t}}{2}| = 2^{-(t+1)}$. The variance is then calculated by squaring this amplitude and multiplying it by $\underline{n}$, *i.e.*, $|\frac{2^{-t}}{2}|^2 \cdot \underline{n} = \underline{n}2^{-2(t+1)}$. To change the decomposition base, we can replace 2, which gives us $\frac{1}{4}\underline{n}base^{-2t}$. Albeit correct, this arithmetic approach is not as tight as desirable.

The vector $\underline{a}$ of the input TLWE sample is generated from a uniform distribution. Hence, the error inserted by rounding each of its elements to 2^{-t} is also uniform and varies from $-\frac{base^{-t}}{2}$ to $+\frac{base^{-t}}{2}$. The variance of an uniform distribution varying from a to b is $\frac{1}{12}(a-b)^2$ and the sum of $\underline{n}$ uniformly distributed variables are a (scaled) Irwin-Hall distribution with variance $\underline{n} \cdot \frac{1}{12}(a-b)^2$. Applying these equations to our case, we have that the error variance introduced by the decomposition is $\underline{n} \cdot \frac{1}{12}(-\frac{base^{-t}}{2} - \frac{base^{-t}}{2})^2 = \frac{1}{12}\underline{n}base^{-2t}$. Irwin-Hall distributions are very good approximations for Gaussian distributions and we can just replace the third term of the Equation 1, which results in Equation 5.

$$Var(Err(c)) \leq R^2Var(Err(\underline{c})) + \underline{n}tN\vartheta_{KS} + \frac{1}{12}\underline{n}base^{-2t} \tag{5}$$

3.3.1 Error Rate

Knowing the error variance is useful for approximate computation, but, for perfect execution, we need to calculate the probability of such error affecting significant bits of the message. In the decryption, a failure occurs if the rounding procedure rounds to the wrong integer, which will happen if the absolute value of the error is greater than half of the least significant bit of the message in the Torus representation. Equation 6 gives us the probability of this error occurring for a TLWE sample x, with standard deviation σ_x and the least significant bit encoded in the torus with value $(2 \cdot interval)$. erf is the Gaussian error function.

$$P(|err(x)| > interval) = 1 - erf(\frac{interval}{\sigma_x\sqrt{2}}) \tag{6}$$

The bootstrap may also fail since the error affects the accumulator blind rotation amount. The failure occurs if the accumulator is rotated to a polynomial in which the coefficient of the constant term is different from the desired one. In this way, we need the error to be within the *interval* of half the Torus slice size we are working with, which is $\frac{1}{2} \cdot \frac{0.5}{B}$. Scaling by 2^{32}, we have $interval = \frac{2^{31}}{2B}$. Besides the TLWE error variance, we also need to consider the rounding error introduced when selecting the $log_2(2N)$ most significant bits of each position of a scaled to $2N$. The discarded bits of each position are uniformly distributed with value ranging from $-\frac{2^{31}}{2N}$ to $+\frac{2^{31}}{2N}$. The variance of a uniform distribution is $\frac{1}{12}$ times the square of its amplitude, thus $\sigma^2_{a_i} = \frac{1}{12} \cdot \frac{2^{32}}{2N}^2$, for each $a_i \in a$. In the worst case, n positions will be added to calculate the phase, leading to variance $\sigma^2_{\sum_i^n a_i} = n \cdot \sigma^2_{a_i}$. This variance is additive with the one of the Gaussian error, and we can obtain the error rate using Equation 6.

The bootstrap is much more susceptible to failures than the decryption. Therefore, the error variance of TLWE samples that are input for bootstraps ends up being the main component to control the error rate. Given a function that can be evaluated by both of our methods, we can define which one presents the better error rate in function of such variance. Let $\oslash$ be the linear operation of the chaining method and s be its error variance scaling, *i.e.* how many times the error variance of the output of $\oslash$ is greater than the one of the inputs. Figure 3 shows the error rate in function of the TLWE input error variance for 8-bit functions in base 4 for $s \in \{2, 4, 6, 10\}$. We found few practical cases with $s > 2$, but, in all of them, the tree-based approach required more bootstraps, and s could be lowered to 2 by increasing the number of bootstraps of the chaining one. In this way, we conclude that the chaining method is usually the better choice for the functions it can evaluate. However, as we noted before, its functionality is very restricted. Table 1 summarizes the main characteristics of our two methods.

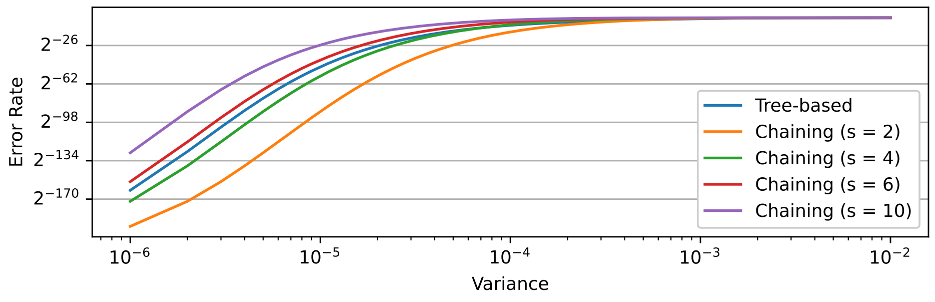


Figure 3: Comparison between the error rates of the tree-based and chaining methods.

3.4 Improving building blocks

3.4.1 Base-aware TLWE-to-TRLWE Key Switching

A regular TLWE-to-TRLWE Key Switching packs N TLWE samples in one TRLWE sample in $\mathbb{T}_N[X]^{k+1}$. However, as we described in Section 2.3.1, we want to pack B TLWE samples with each one being mapped to a sequence of consecutive coefficients in the TRLWE sample. In Algorithm 7, we exploit the fact that $B < N$ to both accelerate the key switching and to reduce the error variance growth in $\frac{N}{B}$ times compared to the regular TLWE-to-TRLWE Key Switching. Comparing with Algorithm 1, we obtain speedups by replacing multiplications between N-sized polynomials (line 5 of Algorithm 1) by inner products between a B-sized vector of binary digits and a B-sized vector of TRLWE samples (line 6 of Algorithm 7). The Key Switching key, however, is B times bigger.

3.4.2 Multi-Value Extract

The error variance growth of adding two variables x and y is defined as $\sigma^2_{x+y} = \sigma^2_x + \sigma^2_y + 2\rho\sigma_x\sigma_y$, where σ^2 is the variance and ρ is the correlation between the normally distributed variables. When this correlation is linear, ρ may be defined as its degree. If the variables are completely independent, then $\rho = 0$ and $\sigma^2_{x+y} = \sigma^2_x + \sigma^2_y$. If they are the same variable, then $\rho = 1$ and $\sigma^2_{x+x} = \sigma^2_x + \sigma^2_x + 2\sigma_x\sigma_x = 4\sigma^2_x$. Defining the multiplication as a sequence of additions of the same variable, we have that $\sigma^2_{n\times x} = n^2 \times \sigma^2_x$, for $n \in \mathbb{Z}$.

Table 1: Comparison between the chaining and tree-based methods. We use overbars to indicate output variables, $\oslash$ to represent linear transforms, and ":" to denote digit slicing. The function *prob* is given by Equation 6, d is the number of digits, s is the scaling factor of the error variance in a linear transform (i.e. how much the linear transform increases the error variance), and σ^2 represents error variances (specifically, σ^2_{BT} and σ^2_{KS} are the output error variance of the bootstrap and TRLWE Key switching, respectively).

	Chaining	Tree-Based
Supported Functions	(recursive definition) Let f be a function over an operand x with d digits. For each digit x_i for $i \in [0, d)$, there shall exist a linear combination $\oslash_i$ and a LUT L_i, s. t.: if $i = 0$, $\overline{x}_0 = L_i(x_0) = \oslash_f(f(x_0))$, if $i \geq 1$, $\overline{x}_i = L_i(\oslash_{i-1}(\overline{x}_i - 1, x_i))$ $= \oslash_f(f(x_i : x_0))$.	Any.
Suitable Functions	Most functions following carry-like (e.g. addition) or test (e.g. sign) logics.	All other functions.
Error var. σ^2_{out}	$s_{\oslash_f} \cdot \sigma^2_{BT}$	$(d-1) \cdot \sigma^2_{KS} + d \cdot \sigma^2_{BT}$
Success Rate	$\prod_{i=0}^{d-1} prob(\sigma^2_{in} \cdot s_{\oslash_i})$	$prob(\sigma^2_{out}) \cdot prob(\sigma^2_{in})^d$
Possible improvements and tradeoffs	The scaling factors, s, can be lowered by bootstrapping intermediate results of the linear combinations.	Optimizations in intervals of the function domain which are mapped to constant values or linear combinations.
Complexity in number of bootstraps	Linear in d.	Linear in d (Assymtoptically).

To avoid the quadratic variance growth at multiplications, we could implement them as sequences of additions of independent TLWE samples encrypting the same number. We obtain these independent encryptions by extracting multiple coefficients from the accumulator (ACC) at the end of a bootstrap procedure. We call this process *Multi-Value Extract.* Recall that, in our LUT encoding (Section 2.3.1), each LUT position is mapped to a sequence of $\frac{N}{B}$ coefficients. Therefore, after the bootstrap, we should have $\frac{N}{B}$ independent encryptions of each number, which we can use to perform the multiplication as shown in Algorithm 8. The additional extracts on the ACC reduce the *interval* used to calculate the error rate in Equation 6 from $\frac{1}{4B}$ to $(\frac{1}{4B} - \frac{b}{4N})$. However, we find this reduction to have a negligible impact on the error rate for the values we tested. We sustain the independence between coefficients of the ACC on the Independence Heuristic [CGGI20] (Definition 1).

Definition 1 (Independence Heuristic,[CGGI20])**.** The error of the coefficients of TRLWE samples (including TRGSW samples) and all linear combinations of them considered in TFHE are independent and concentrated.

We tried to experimentally validate the error variance of the multiplication using the multi-value extract, and we obtained the results in Figure 4a. We noticed that the error variance is still growing quadratically, which indicates that the coefficients are not independent. To obtain formal guarantees of independence (instead of a heuristic), Chillotti *et al.* [CGGI20] points out that we could perform the gadget decomposition in a probabilistic way. We implemented the probabilistic gadget decomposition proposed by Genise *et al.* [GMP19], but we obtained no improvements over the deterministic algorithm. We were only able to obtain the linear growth by lowering the error variance introduced by the gadget decomposition. We increased the size of the decomposition base $log_2(B_g)$ from 4 to 5, which improves its precision $\ell log_2(B_g)$ from 20 to 25 bits, a reasonably high value for a 32-bit implementation. Figure 4b shows the results.

From this experiment, we can conclude that, although the independence heuristic holds

Algorithm 7: Base-aware TLWE-to-TRLWE Public Functional Key Switching

```
Input  : B TLWE samples $\underline{c}^{(z)} = (\underline{a}^{(z)}, \underline{b}^{(z)}) \in \mathsf{TLWE}_{\underline{s}}(\mu_z)$, $z \in [\![1, B]\!]$
Input  : a precision parameter $t \in \mathbb{Z}$
Input  : a Key Switching key $\mathsf{KS}_{i,j,b} \in \mathsf{TRLWE}_{\overline{S}}(\frac{s_i}{base^j} \cdot \sum_{q=bN/B}^{(b+1)N/B-1} X^q)$, for $i \in [\![1, n]\!]$,
         $j \in [\![1, t]\!]$, and $b \in [\![0, B)$.
Output : a TRLWE sample $\overline{c} \in \mathsf{TRLWE}_{\overline{s}}(f(\mu_z))$, for $z \in [\![1, p]\!]$.
1 $f : \mathbb{T}^B \mapsto \mathbb{T}_N[X] = (a_1, ..., a_B) \mapsto a_1 X^{N-1} + \ ... \ + a_B$
2 for $i \in [\![1, n]\!]$ do
3     for $b \in [\![1, B]\!]$ do
4         $\tilde{a}_{i,b} \leftarrow \lceil \underline{a}_i^{(b)} \rfloor_{\frac{1}{2^t}}$, i.e., the closest multiple of $\frac{1}{2^t}$ to $\underline{a}_i^{(b)}$.
5     Decompose each $\tilde{a}_{i,b} = \sum_{j=1}^{t} \tilde{a}_{i,j,b} \cdot 2^{-j}$, where $\tilde{a}_{i,j,b} \in \mathbb{B}^B$.
6 Return $(0, f(\underline{b}_i^{(1)}, \underline{b}_i^{(2)}, ..., \underline{b}_i^{(p)})) - \sum_{i=1}^{n} \sum_{j=1}^{t} \langle \tilde{a}_{i,j}, \mathsf{KS}_{i,j} \rangle$
```

Algorithm 8: Multiplication (scaling) using the multi-value extract

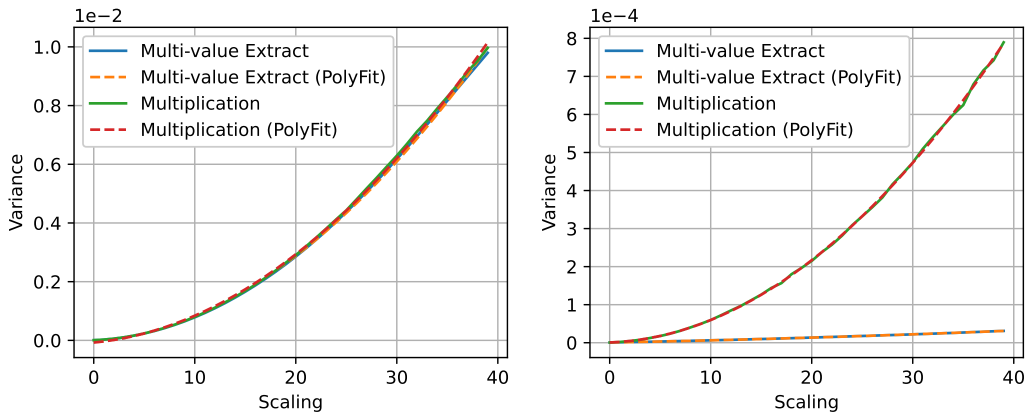
```
Input  : a TRLWE sample $\underline{c} \in \mathsf{TRLWE}_{\underline{S}}(\mathsf{p})$, which is the accumulator (ACC) of a previous
         functional bootstrap, and a cleartext scalar $b \in \mathbb{Z}$.
Output : a TLWE sample $\overline{c} \in \mathsf{TLWE}_{\overline{S}}(b \cdot p_0)$, where $p_0$ is the constant term of $p$, and
         $\overline{S} \in \mathbb{B}^N$ is a vector interpretation of $\underline{S} \in \mathbb{B}_N[X]$.
1 $\overline{c} \leftarrow \mathsf{TLWE}_{\overline{S}}(0)$
2 $\overline{c} \leftarrow \overline{c} + \mathsf{SampleExtract}_i(p)$, for each $i \in [\![0, \lceil \frac{b}{2} \rceil - 1]\!]$
3 $\overline{c} \leftarrow \overline{c} - \mathsf{SampleExtract}_i(p)$, for each $i \in [\![N - \lfloor \frac{b}{2} \rfloor, N - 1]\!]$
4 Return $\overline{c}$
```

for the Gaussian error, the same cannot be said about the error introduced by the gadget decomposition. We also measured that the error variance introduced by the decomposition is bigger than the estimations using the right term of Equation 2, $\underline{n}(1+kN)\epsilon^2$. We consider it as an indication that this dependency between coefficients might affect not only our multi-value extract but also the bootstrap itself (although further research is certainly necessary to support that). The use of a probabilistic gadget decomposition with higher entropy might be an alternative solution to the increase of ℓ or B_g. However, in our case, it would require us to increase other parameters, which would impact performance.

Impact on the multi-value bootstrap The multi-value bootstrap of Carpov *et al.* [CIM19] increases the bootstrap error variance in $\|TV_f\|_2^2 \leq s(q-1)^2$ times, where s and q are the input and output bases, respectively. Carpov *et al.* uses $q = 2$ (the binary base) to lower the error, but it needs to convert from base q to s to use the output in another function. It does so by using a key switching at the beginning of each functional bootstrap to perform a base composition, which is s-Lipschitz, and, therefore, introduces the previously avoided quadratic error. In summary, composable circuits often need the input and output bases to be the same, and, in these cases, solely outputting in the binary base would just transfer the complexity to the base composition. However, with the introduction of the multi-value extract, we can perform the scaling required by the base composition linearly and, thus, reduce the complexity of error variance from $s(q-1)^2$ to $s(q-1)$.

3.5 Practical parameters and experimental error variance measument

Our first step to define practical parameters was to survey the literature on non-linear functions that are usually implemented using perfect computation with TFHE. Then, based on the required precision, we manually searched for parameters aiming at an error rate of at least 2^{-30}. Table 2 shows two of the most promising sets we found. A more



(a) 20-bit precision ($\ell = 5$ and $log_2(B_g) = 4$)) **(b)** 25-bit precision ($\ell = 5$ and $log_2(B_g) = 5$)

Figure 4: Comparison between the variance of scaling using the multi-value extract and direct multiplication.

methodological search could likely yield parameters with a better error rate or efficiency. The security level is defined by the (R)LWE dimensions (n and kN) and the standard deviations of the errors. The LWE Estimator [APS15] reports a cost of $2^{127.1}$ on the primal attack via uSVP and $2^{139.6}$ on the dual-lattice attack, both using the `BKZ.sieve` cost model. Although we could increase the security level by increasing the parameters, we use these values since they are the same used by TFHE and most of the previous literature.

Table 2: Sets of parameters used to evaluate the performance of our implementations.

Name	Security Level	B	LWE		RLWE			Bootstrap		Key Switch	
			n	σ	N	k	σ	ℓ	$log_2(B_g)$	Base	t
5_5_6_2	127	4	630	2^{-15}	1024	1	2^{-25}	5	5	64	2
6_4_6_3								6	4		3

We estimated the error rate for parameters using the equations of Section 3.3, and we experimented to validate their results. We find this validation to be necessary mainly because we based our error analysis on equations designed for TFHE to work with binary digits and logic gates. Once we introduced larger bases with arbitrary LUT evaluation and tightened some of the variance estimations, we could no longer support our statistics on the experimental validation provided by Chillotti *et al.* [CGGI20]. Table 3 shows the results of our experiments. We measure the variance of 2^{14} samples and calculated a 95%-confidence interval using the Chi-square distribution. Due to computational limits, we could only validate the error rate for higher variances. We performed 15,438,720 bootstraps over LWE samples with $\sigma^2 = 1.70\text{E-}04$. The output was wrong in only 39 of them. In both cases, the experiments showed that our estimations are reasonably close upper bounds for the actual values. Although we cannot extrapolate the results for different variances, the experiments provide some evidence of correctness for our equations on the considered parameters.

Table 3: Experimental validation for the variance and error rate for two sets of parameters.

Params.	Measurement		Estimative	Experiment		
				Measured	95% Conf. Interval	
5_5_6_2	Variance	Bootstrap	1.47E-06	4.94E-07	4.84E-07	5.05E-07
		TRLWE KeySwitch	5.09E-06	2.53E-06	2.47E-06	2.58E-06
	Error rate (log_2)	σ^2 = 1.70E-04	-18.001	-18.595		
6_4_6_3	Variance	Bootstrap	4.41E-07	1.70E-07	1.67E-07	1.74E-07
		TRLWE KeySwitch	1.25E-09	6.38E-10	6.25E-10	6.52E-10

4 Performance Results

We benchmarked our methods by using them to implement a set of relevant functions from the literature. We selected two functions from previous literature on the functional bootstrap of TFHE and three functions that are building blocks for neural network algorithms. We set the precision of our implementations to match the ones we are comparing to. We executed our experiments using an Intel i7-7700 processor at 4.20 GHz running Ubuntu 18.04. We compiled our code using GCC 7.3.0 with flags `-O3 -std=c++11 -funroll-all-loops -march=native -ltfhe-spqlios-fma -lm`, and we used the *"optim"* build of TFHE. Each result is the average of 100 executions. We tried to compile and execute previous implementations on the same environment. When we could not reproduce them (either because the authors did not provide the source code or because their source code did not compile in our environment), we report the results presented by the authors and add an observation about the differences between machines. Fortunately, most authors report the execution time for the default gate bootstrap of TFHE, which we can use as a "TFHE benchmark score" and adjust the speedup values accordingly. In our machine, the default gate bootstrap of TFHE runs in 9ms and 13ms using the old (80-bit security) and the new (127-bit security) set of parameters, respectively.

4.1 Lookup Table evaluation

Lookup tables are very commonly used in homomorphic circuits, and the implementation of Carpov *et al.* [CIM19] is one of the most recent and efficient of them. It introduces the multi-value bootstrap of TFHE, which we explore in our tree-based method. Table 4 compares their implementation for a 6-bit-to-6-bit LUT with one using our tree-based method (Algorithm 6). Both implementations are based on the functional bootstrap of TFHE. The difference is that the implementation of Carpov *et al.* [CIM19] performs a single functional bootstrap with base $B = 2^6$, whereas our implementations perform several functional bootstraps with base $B = 2^2$ and combine them using the tree-based method.

Table 4: Comparison between implementations of a 6-bit-to-6-bit LUT.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[CIM19]	≥128	≈ 8 GB	-26.94	1570[a]	1.00
5_5_6_2	127	≈ 4.3 GB	-59.59	378.2	2.49
6_4_6_3	127	≈ 6.5 GB	-134.84	457.9	2.06

[a] Result provided by the authors, who executed experiments on a machine 1.67 times slower than ours. The speedup was adjusted accordingly.

4.2 32-bit Integer Comparison

Integer comparison is an extremely useful function in computing, but it presents some challenges to be implemented in homomorphic circuits, especially for unbounded integers. Bourse *et al.* [BST20] presented a very efficient implementation using the functional bootstrap of TFHE. The chaining method we presented in Section 3.2 is a generalization of their technique. To compare with them, we implemented a unary tree to evaluate the integer comparison. Algorithm 9 describes our implementation. Table 5 compares them with the integer comparison implemented by Zhou *et al.* [ZLPL20] using logic gates.

We calculated the speedups using as reference the slowest implementation, which, in this case, is the second implementation of Bourse *et al.* [BST20]. However, adjusting the speedups to consider the difference in speed between machines, we can see that the implementation of Zhou *et al.* [ZLPL20] using logic gates is up to 1.75 times slower than the one of Bourse *et al.* [BST20] and up to 5.6 times slower than ours. The chaining and tree-based methods perform the same number of bootstraps and should present a very

Table 5: Comparison between implementations of a 32-bit integer comparison.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[BST20]	90	≈ 1.2 GB	-50[b]	2232[a]	1.75
	109	≈ 3.4 GB	-47[b]	3902[a]	1.00
	211	≈ 4.6 GB	-89[b]	3840[a]	1.02
[ZLPL20]	80	≈ 0.3 GB	Negligible	1143.2	0.93
	127	≈ 0.3 GB	Negligible	1867.2	0.57
5_5_6_2	127	≈ 4.3 GB	-26.51	334.1	3.19
6_4_6_3	127	≈ 6.5 GB	-129.58	396.4	2.68

[a] Execution time provided by the authors, who executed experiments on a machine 3.67 times slower than ours. The speedup was adjusted accordingly.
[b] Error Rate provided by the authors. We speculatively estimate it to be much lower, but we do not have sufficient data to calculate.

similar performance. Nonetheless, we were able to achieve significant speedups thanks to our tighter variance and error rate estimations, which enabled a better choice of parameters.

4.3 Neural Network Functions

Neural network inference algorithms are currently one of the major use cases for homomorphic encryption [BGGJ19, LJ19, ZLPL20]. They achieve great performance levels with approximate computation in some cryptosystems [CKKS17], but perfect computation still seems to be necessary to achieve state-of-the-art inference accuracy for deep neural networks [BGGJ19, LJ19]. We implemented three functions that are building blocks for them and that are provided by SHE [LJ19], an implementation of secure neural network inference based on TFHE that achieves state-of-the-art inference accuracy. SHE presents very fast arithmetic (thanks to the use of Lognet), but it relies on logic gates to implement non-linearities. We also compare our results with the implementations of Zhou *et al.* [ZLPL20], which are generally faster than SHE but present worse inference accuracy.

4.3.1 ReLU

The Rectified Linear Unit (ReLU) is a neural network activation function broadly used due to its simple implementation and non-linear properties. Algorithm 10 shows our implementation using the Functional Bootstrap. The logic is similar to a multiplexer. Table 6 compares it with implementations using logic gates. For the 127-bit security level, our implementations are up to 6.98 times faster than the one of Lou and Jiang [LJ19] and 1.19 times faster than the one of Zhou *et al.* [ZLPL20]. Although the logic gates of TFHE generally introduce error rates much smaller than the functional bootstrap, the error rate of our implementation is 2^{-137}, which is also negligible compared to the security level.

Table 6: Comparison between implementations of an 8-bit ReLU function.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[LJ19]	80	≈ 0.3 GB	Negligible	380	1.59
	127	≈ 0.3 GB	Negligible	603.1	1.00
[ZLPL20]	80	≈ 0.3 GB	Negligible	64.8	9.31
	127	≈ 0.3 GB	Negligible	103.1	5.85
5_5_6_2	127	≈ 4.3 GB	-137.092	86.4	6.98
6_4_6_3	127	≈ 6.5 GB	-180.973	103.6	5.82

4.3.2 Maximum

Our implementation of the maximum function is, at first, similar to the integer comparison followed by a multiplexer. However, in this context, we have to also consider signed numbers, which leads us to Algorithm 11. Table 7 shows performance results.

Table 7: Comparison between implementations of an 8-bit max function.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[LJ19]	80	≈ 0.3 GB	Negligible	379.3	2.14
	127	≈ 0.3 GB	Negligible	593.1	1.37
[ZLPL20]	80	≈ 0.3 GB	Negligible	483.1	1.68
	127	≈ 0.3 GB	Negligible	810.6	1.00
5__5__6__2	127	≈ 4.3 GB	-50.0874	228.3	3.55
6__4__6__3	127	≈ 6.5 GB	-156.744	276.8	2.93

4.3.3 Addition

We implemented this function using the chaining method since it presents a carry-like logic. Our implementations using the tree-based approach were more expensive since, although we can produce a linear combination for which the carry follows a B-anti-cyclic logic, we could not do the same for the addition itself. Algorithm 12 describes our implementation and Table 8 compares it with implementations using logic gates. We used the multi-value extract to perform the scaling in line 6 of Algorithm 12.

Table 8: Comparison between implementations of an 8-bit addition function.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[LJ19]	80	≈ 0.3 GB	Negligible	585	1.21
	127	≈ 0.3 GB	Negligible	708.9	1.00
[ZLPL20]	80	≈ 0.3 GB	Negligible	338	2.10
	127	≈ 0.3 GB	Negligible	548.7	1.29
5__5__6__2	127	≈ 4.3 GB	-124.7	81.1	8.74
6__4__6__3	127	≈ 6.5 GB	-176.139	94.8	7.48

4.4 Additional estimations

Using the results of Section 4.3, we can estimate the performance gains our methods could bring to full applications. Let us take, for example, the binarized convolutional neural network (CNN) of Zhou *et al.* [ZLPL20], which can be implemented using the functions of Section 4.3. This CNN classifies images in the MNIST dataset and is composed of 3 binarized convolutional layers, 3 max-pooling layers, and two fully connected layers. We counted the number of operations on each one of them and estimated their execution time using the results of Section 4.3. Table 9 shows the results. Although this is a basic estimation, our execution times are reasonably close to the ones reported by Zhou *et al.* [ZLPL20], especially considering the differences in execution environments. A real implementation would likely present better performance for our methods since they allow for further optimizations, such as using the multi-value bootstrap to batch multiple operations. Nonetheless, the current estimation indicates a speedup of up to 4.9 times, which is within the expected considering the results of Section 4.3, where the speedup over basic operations for this same implementation ranged from 1.19 (ReLU) to 6.77 (Addition) times.

We can also estimate the performance impact of changing the size of the keys. The main reason we use large keys (compared to implementations using logic gates) is the use of decomposition bases greater than 2 in the TLWE-to-TRLWE key switching. Using base 64, the Key Switching keys take 4.0 GiB and 6.0 GiB for the parameters **5__5__6__2** and **6__4__6__3**, respectively. Decreasing the base would also linearly decrease the size of these keys, but, to avoid increasing the error, we would need to logarithmically increase the value of t and, consequently, the key-switching execution time. For example, using base 16

Table 9: Estimation of an inference on the Binarized CNN of Zhou *et al.* [ZLPL20].

	Security Level	Execution time per layer (h)			Total (h)	Speedup
		Bin. Conv.	Max-Pool.	Fully Conn.		
[ZLPL20] (reported)	80	19.20	0.67	21.35	41.22	1.88
[ZLPL20]	80	20.18	0.96	26.87	48.01	1.62
	127	32.46	1.56	43.62	77.64	1.00
5_5_6_2	127	7.39	0.53	7.91	15.83	4.90
6_4_6_3	127	8.19	0.61	9.20	18.00	4.31

instead of 64, the parameter **5_5_6_2** would need a 1 GiB key, but t would need to be increased from 2 to 3, which would increase the execution time of the key switching from 6 ms to 9 ms. At first, this tradeoff seems promising for many functions, especially the ones which make little use of the TLWE-to-TRLWE key switching. However, increasing the value of t also increases the second term of Equation 5 and, hence, might affect the output error variance negatively. For simplicity, in this paper, we chose two sets of parameters that fit all functions we implemented. A more targeted search for parameters would likely yield better results, and the methods we introduced allow for easily changing parameters even within a single function evaluation.

5 Related Work

The literature on using lookup tables (LUT) in homomorphic circuits dates back to the first fully homomorphic encryption schemes presented and has been used with most of the modern FHE schemes [CGH$^+$18, LFFJ19, NRPH19]. LUTs are a simple and powerful technique to represent arbitrary functions, but problems with latency and precision had also been reported [LFFJ19, NRPH19]. The introduction of LUT evaluations within the bootstrap by FHEW [DM15] started a new line of work on this topic. Bonnoron *et al.* [BDF18] introduced techniques to evaluate gates with a large number of input bits using a single bootstrap of FHEW and implemented a 6-bit-to-6-bit LUT that runs in less than 10 seconds. Based on their work, Carpov *et al.* [CIM19] presented the multi-value bootstrap of TFHE and lowered the execution time of the 6-bit-to-6-bit LUT to only 1.5 seconds. Our work makes extensive use of the multi-value bootstrap, but we focus more on accelerating the evaluation of non-linear functions than improving the multi-value bootstrap itself. Nonetheless, we did present some contributions towards it, such as the reduction of the complexity of its error growth. Moreover, our combination methods also introduce two new ideas to this line of work: how to use the multi-value bootstrap to accelerate single (instead of multiple) LUT evaluations; and how to improve the LUT evaluation based on particular properties of the encoded function.

Tree-based approach The most similar strategy to our tree-based evaluation is the vertical packing of Chillotti *et al.* [CGGI20], which suggests the use of a CMUX tree to choose among multiple TRLWE samples and, then, uses a single BlindRotate to perform a final lookup. Similarly to ours, their method also allows some optimizations based on the encoded function (although they did not present nor explore this idea itself). On the other hand, our method constructs LUTs on-the-fly using results of previous lookups, which allows optimizations even within a single LUT. The main difference between them, however, is in their use of TFHE's building blocks. The vertical packing is directly based on CMUX gates and, hence, requires the selector to be encrypted in the binary base in TRGSW samples. This makes it a very good choice in the leveled setting, but it requires one circuit bootstrap per bit in the fully homomorphic one. Chillotti *et al.* [CGGI20] reports an execution time of around 800ms to evaluate a 6-bit-to-6-bit LUT in the fully homomorphic setting with

80 bits of security. Correcting for the security level and difference between machines, the implementation of Carpov *et al.* is already slightly faster. Bouse *et al.* [BST20] also presents a "tree-based" technique for integer comparison using the functional bootstrap, but, besides the name, by our definitions, it is equivalent to the chaining method. More specifically, it rearranges the chaining evaluation in a tree-like fashion (specific to the integer comparison logic) so that it can be parallelized in multiple threads. The output of each LUT is still used to construct the selector of the next one (which defines the chaining) and the technique has no further similarities with our "tree-based" approach.

Evaluation of neural network inference using the Functional Bootstrap Using the functional bootstrap of TFHE in neural network inference seems to be a recent (and promising) trend. As previously cited, FHE-DiNN [BMMP18] and Izabachène *et al.* [ISZ19] implemented the sign activation function using it. Boura *et al.* [BGGJ19] simulated the error propagation of several activation functions and introduced the term *Functional bootstrap*, which we adopt in this paper. Klemsa *et al.* [KN20] presented a Ruby version of TFHE, which includes the functional bootstrap, targeted at neural network implementations. From all previous literature, the only one we found to combine multiple functional bootstraps to evaluate a single function is the integer comparison of Bourse *et al.* [BST20].

6 Conclusion

In this paper, we presented two methods to combine multiple functional bootstraps in TFHE; we performed a thorough error variance and error rate analysis on our methods and on the functional bootstrap itself, including experimental validation; we introduced a multi-value extract procedure to improve the error behavior on scalings and, especially, on the multi-value bootstrap; we introduced a "base-aware" TLWE-to-TRLWE Key Switching to speedup the LWE packing; and, finally, we selected practical parameters and benchmarked our methods using relevant functions from the literature. We achieved speedups of up to 3.19 times compared to previous literature on the functional bootstrap of TFHE, and of up to 8.7 times compared to implementations using logic gates.

Arbitrary LUTs are inherently exponential to evaluate, which gives more importance to the possibility of optimizing them based on the function particularities, a feature that our methods introduce. In our practical experiments, we limited ourselves to work with the original implementation of TFHE and with precision levels that were previously defined in the literature. Our results demonstrate efficient evaluations with the precision of up to 6 bits for completely arbitrary functions, and up to 32 bits for functions with enough opportunities for optimizations. The techniques themselves can be easily extended to work with higher parameters and are already defined to efficiently exploit the circuit bootstrap. To test it in practice, however, we would need to move to more optimized versions of TFHE (with at least 64-bit Torus precision) and to implement an efficient version of the circuit bootstrap, which is mostly a practical open problem. This process would bring contributions of its own and, thus, we leave it as future work. Nonetheless, the speedups we achieved are certainly a good measurement of improvement over previous literature, and some of our contributions, such as the multi-value extract, go beyond the context of functional bootstrap implementations and are useful even for pure arithmetic.

As future work, we also intend: to implement our algorithms for the functional bootstrap using more optimized versions of TFHE; to pursue implementation optimizations for our techniques themselves; the research on efficient ways of implementing the TRLWE-to-TRGSW circuit bootstrap; and to accelerate the Functional Bootstrap and TRLWE Key Switching exploiting techniques such as proposed by Bourse *et al.* [BMMP18], Chen *et al.*[CDKS20], and Micciancio and Sorrell [MS18]. Ultimately, we intend to create a library for the functional bootstrap and to test it in frameworks of automatic code generation.

References

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3):169 – 203, 01 Oct. 2015.

[BDF18] Guillaume Bonnoron, Léo Ducas, and Max Fillinger. Large FHE Gates from Tensored Homomorphic Accumulator. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2018*, pages 217–251, Cham, 2018. Springer International Publishing.

[BGGJ19] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Simulating Homomorphic Evaluation of Deep Learning Predictions. In Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung, editors, *Cyber Security Cryptography and Machine Learning*, pages 212–230, Cham, 2019. Springer International Publishing.

[BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed Ciphertexts in LWE-Based Homomorphic Encryption. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography – PKC 2013*, pages 1–13, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 483–512, Cham, 2018. Springer International Publishing.

[BST20] Florian Bourse, Olivier Sanders, and Jacques Traoré. Improved Secure Integer Comparison via Homomorphic Encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, pages 391–416, Cham, 2020. Springer International Publishing.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved Bootstrapping for Approximate Homomorphic Encryption. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 34–54, Cham, 2019. Springer International Publishing.

[CDKS20] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts. Cryptology ePrint Archive, Report 2020/015, 2020. https://eprint.iacr.org/2020/015.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CGH$^+$18] Jack L. H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. Doing Real Work with FHE: The Case of Logistic Regression. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '18, page 1–12, New York, NY, USA, 2018. Association for Computing Machinery.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New Techniques for Multi-value Input Homomorphic Evaluation and Applications. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, pages 106–126, Cham, 2019. Springer International Publishing.

[CJP20] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. Technical report, Zama, 2020.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic Encryption for Arithmetic of Approximate Numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[GMP19] Nicholas Genise, Daniele Micciancio, and Yuriy Polyakov. Building an Efficient Lattice Gadget Toolkit: Subgaussian Sampling and More. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 655–684, Cham, 2019. Springer International Publishing.

[ISZ19] Malika Izabachène, Renaud Sirdey, and Martin Zuber. Practical Fully Homomorphic Encryption for Fully Masked Neural Networks. In Yi Mu, Robert H. Deng, and Xinyi Huang, editors, *Cryptology and Network Security*, pages 24–36, Cham, 2019. Springer International Publishing.

[KN20] J. Klemsa and M. Novotný. WTFHE: neural-netWork-ready Torus Fully Homomorphic Encryption. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–5, June 2020.

[KWN20] Delaram Kahrobaei, Alexander Wood, and Kayvan Najarian. Homomorphic Encryption for Machine Learning in Medicine and Bioinformatics. *ACM Comput. Surv.*, 2020.

[LFFJ19] Qian Lou, Bo Feng, Geoffrey C. Fox, and Lei Jiang. Glyph: Fast and Accurately Training Deep Neural Networks on Encrypted Data, 2019.

[LJ19] Qian Lou and Lei Jiang. SHE: A Fast and Accurate Deep Neural Network for Encrypted Data. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 10035–10043. Curran Associates, Inc., 2019.

[MP20] Daniele Micciancio and Yuriy Polyakov. Bootstrapping in FHEW-like Cryptosystems. Cryptology ePrint Archive, Report 2020/086, 2020. https://eprint.iacr.org/2020/086.

[MS18] Daniele Micciancio and Jessica Sorrell. Ring Packing and Amortized FHEW Bootstrapping. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 100:1–100:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[NRPH19] K. Nandakumar, N. Ratha, S. Pankanti, and S. Halevi. Towards Deep Neural Network Training on Encrypted Data. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 40–48, June 2019.

[Reg09] Oded Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. *J. ACM*, 56(6), September 2009.

[ZLPL20] J. Zhou, J. Li, E. Panaousis, and K. Liang. Deep Binarized Convolutional Neural Network Inferences over Encrypted Data. In *2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pages 160–167, Aug 2020.

A Algorithms of functions presented in Section 4

For simplicity, we define two functions to use in the algorithms presented in this section:

- **FunctionalBoostrap:** It receives a selector c encrypted in a TLWE sample, a LUT encrypted in a TRLWE sample, and the bootstrap key BK. If necessary, it performs a TLWE key switching in the sample c to switch from the key $\overline{S}$ (with $n = 1024$) to s (with $n = 630$). Then, it executes the functional bootstrap and returns a TLWE sample with the result.
- **TRLWEKeySwitch:** It receives a vector of $B = 4$ TLWE samples and a Key Switching key. It performs a base-aware TLWE-to-TRLWE key switching using our default packing technique.

Algorithm 9: Integer comparison algorithm using the tree-based method.

```
Input  : two sets of TLWE samples $\underline{c}_{j,i} \in \text{TLWE}_s(\frac{m_{j,i}}{2B})$, such that $\sum_{i=0}^{d-1} m_{j,i}B^i = m_j$
         encodes each number $m_j$ for $j \in \{0,1\}$ and base $B = 4$ with $d$ digits.
Input  : a bootstrapping key BK and a TRLWE Key Switching key KS
Output : A TLWE sample $\overline{c} \in \text{TLWE}_{\overline{S}}(\frac{\overline{m}}{2B})$, where $\overline{S} \in \mathbb{B}^N$ is a vector (TLWE)
         interpretation of $S \in \mathbb{B}_N[X]$ and $\overline{m} = \begin{cases} 1, & \text{if } m_0 > m_1, \\ 0, & \text{if } m_0 = m_1, \\ -1, & \text{otherwise.} \end{cases}$
1 TV ← $[0,1,1,1]$
2 for $i \leftarrow 0$ to $d-1$ do
3     $\underline{c}_i \leftarrow \underline{c}_{0,i} - \underline{c}_{1,i}$
4     $\overline{c} \leftarrow$ FunctionalBoostrap($\underline{c}_i$, TV, BK)
5     TV ← TRLWEKeySwitch(($\overline{c}, 1, 1, 1$), KS)
6 return $\overline{c}$
```

Algorithm 10: ReLU implementation using the tree-based method.

```
Input  : a set of TLWE samples $\underline{c}_i \in \mathsf{TLWE}_s(\frac{m_i}{2B})$, such that $\sum_{i=0}^{d-1} m_i B^i = m$ encodes the
         integer $m$ in base $B = 4$ with $d$ digits.
Input  : a bootstrapping key BK and a TRLWE Key Switching key KS
Output : A set of TLWE sample $\bar{c}_i \in \mathsf{TLWE}_{\overline{S}}(\frac{\overline{m}_i}{2B})$, where $\overline{S} \in \mathbb{B}^N$ is a vector (TLWE)
         interpretation of $S \in \mathbb{B}_N[X]$ and $\sum_{i=0}^{d-1} \overline{m}_i B^i = \overline{m}$ encodes the integer
         $\overline{m} = \begin{cases} m, & \text{if } m > 0, \\ 0, & \text{otherwise.} \end{cases}$
1 for $i \leftarrow 0$ to $d-1$ do
2     TV ← TRLWEKeySwitch($(\underline{c}_i, \underline{c}_i, 0, 0)$, KS)
3     $\bar{c}_i$ ← FunctionalBoostrap($\underline{c}_{d-1}$, TV, BK)
4 return $\bar{c}$
```

Algorithm 11: Maximum algorithm using the tree-based method.

```
Input  : two sets of TLWE samples $\underline{c}_{j,i} \in \mathsf{TLWE}_s(\frac{m_{j,i}}{2B})$, such that $\sum_{i=0}^{d-1} m_{j,i} B^i = m_j$
         encodes each number $m_j$ for $j \in \{0,1\}$ and base $B = 4$ with $d$ digits.
Input  : a bootstrapping key BK and a TRLWE Key Switching key KS
Output : A TLWE sample $\bar{c} \in \mathsf{TLWE}_{\overline{S}}(\frac{\overline{m}}{2B})$, where $\overline{S} \in \mathbb{B}^N$ is a vector (TLWE)
         interpretation of $S \in \mathbb{B}_N[X]$ and $\sum_{i=0}^{d-1} \overline{m}_i B^i = \overline{m}$ encodes the integer
         $\overline{m} = \begin{cases} m_0, & \text{if } m_0 > m_1, \\ m_1, & \text{otherwise.} \end{cases}$
1  TV ← $[0, 1, 1, 1]$
2  for $i \leftarrow 0$ to $d-1$ do
3      $\underline{c}_i \leftarrow \underline{c}_{0,i} - \underline{c}_{1,i}$
4      $\tilde{c}$ ← FunctionalBoostrap($\underline{c}_i$, TV, BK)
5      TV ← TRLWEKeySwitch($(\tilde{c}, 1, 1, 1)$, KS)
6  TV ← TRLWEKeySwitch($(\tilde{c}, \tilde{c}, -\tilde{c}, -\tilde{c})$, KS)
7  $\tilde{c}$ ← FunctionalBoostrap($\underline{c}_{0,d-1}$, TV, BK)
8  TV ← TRLWEKeySwitch($(\tilde{c}, \tilde{c}, -\tilde{c}, -\tilde{c})$, KS)
9  $\tilde{c}$ ← FunctionalBoostrap($\underline{c}_{1,d-1}$, TV, BK)
10 $\tilde{c} \leftarrow \tilde{c} + (0, \frac{1}{2B})$
11 for $i \leftarrow 0$ to $d-1$ do
12     TV ← TRLWEKeySwitch($(\underline{c}_{1,i}, \underline{c}_{1,i}, \underline{c}_{0,i}, 0)$, KS)
13     $\bar{c}_i$ ← FunctionalBoostrap($\tilde{c}$, TV, BK)
14 return $\bar{c}$
```

Algorithm 12: Addition algorithm using the chaining method.

```
Input   : two sets of TLWE samples $\underline{c}_{j,i} \in \mathsf{TLWE}_s(\frac{m_{j,i}}{2B})$, such that $\sum_{i=0}^{d-1} m_{j,i}B^i = m_j$
          encodes each number $m_j$ for $j \in \{0, 1\}$ and base $B = 4$ with $d$ digits.
Input   : a bootstrapping key BK and a TRLWE Key Switching key KS
Output  : A TLWE sample $\overline{c} \in \mathsf{TLWE}_{\overline{S}}(\frac{\overline{m}}{2B})$, where $\overline{S} \in \mathbb{B}^N$ is a vector (TLWE)
          interpretation of $S \in \mathbb{B}_N[X]$ and $\sum_{i=0}^{d-1} \overline{m}_i B^i = \overline{m}$ encodes the integer
          $\overline{m} = m_0 + m_1 \bmod B^d$
1  TV $\leftarrow [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$
2  $\underline{c}_0 \leftarrow 0$
3  for $i \leftarrow 0$ to $d - 1$ do
4      $\underline{c}_i \leftarrow \underline{c}_{0,i} + \underline{c}_{1,i}$
5      $\tilde{c} \leftarrow$ FunctionalBoostrap($\underline{c}_i$, TV, BK)
6      $\underline{c}_i \leftarrow \underline{c}_i - B \cdot \tilde{c}$
7      $\underline{c}_i \leftarrow \underline{c}_i - (0, \frac{2}{2B})$
8      if $i < d - 1$ then
9          $\underline{c}_{i+1} \leftarrow (0, \frac{1}{4B})$
10         $\underline{c}_{i+1} \leftarrow \underline{c}_{i+1} + \tilde{c}$
11 return $\overline{c}$
```